OpenWISP

version dev

OpenWISP Community

December 20, 2024

Contents

OpenWISP *Documentation*

Everything you need to know about OpenWISP is here!

Note

For a complete overview of this documentation, refer to the **`Full Table of Contents`**.

Important

Are you looking for a quick overview of the OpenWISP application?

Try the OpenWISP Demo.

First Steps

Quick Start Guide

Try the Demo

Before installing OpenWISP, we recommend trying out the OpenWISP Demo system. This will give you a great overview of how the system works.

Once you have explored the demo, you can install your own instance by following the instructions below.

Install OpenWISP

For production usage, we recommend Deploying OpenWISP with the Ansible OpenWISP role.

Alternatively, you can use Docker OpenWISP.

Make Sure OpenWISP Can Reach Your Devices

For smooth operations, please Setup a Management Network.

Configure Your OpenWrt Devices

Follow the guide to Configure Your OpenWrt Devices.

If you don't have a physical OpenWrt-compatible device, you can install OpenWrt in a VirtualBox VM. The guide above covers how to do this.

Learn More

Once you have everything set up, we recommend exploring other sections of this documentation to make the most out of OpenWISP.

Depending on your use case, you might be interested in different features:

- **Automating Configuration Provisioning**: If your primary interest is automating the provisioning of configurations for OpenWrt devices, check out the Controller module.
- **Device Monitoring**: For those who need monitoring information from their devices, the Monitoring module will be particularly useful.
- **WiFi Connectivity and Security**: If you're focused on providing WiFi Hotspot connectivity or WPA Enterprise WiFi, take a look at the RADIUS and WiFi Login Pages modules.

Additionally, we offer tutorials for the most common scenarios:

- Open and/or WPA Protected WiFi Access Point SSID
- WiFi Hotspot, Captive Portal (Public WiFi), Social Login
- Setting Up WPA Enterprise (EAP-TTLS-PAP) Authentication
- Setting Up a Wireless Mesh Network

Explore these resources to fully leverage the capabilities of OpenWISP!

Seek Help

Reach out to the Community Support Channels.

Setting Up the Management Network

In this section, we will explain how to ensure that your OpenWISP instance can reach your network devices.

Why OpenWISP Needs to Reach Your Devices

For OpenWISP to perform tasks such as push operations, shell commands, firmware upgrades, and periodically run active checks, it **needs to be able to reach the network devices**.

There are two main deployment scenarios for OpenWISP:

- Public Internet Deployment
- Private Network

Public Internet Deployment

This is the most common scenario:

- The OpenWISP server is deployed in a data center exposed to the public internet. Thus, the server has a public IPv4 (and IPv6) address and usually a valid SSL certificate provided by Let's Encrypt or another commercial SSL provider.
- The network devices are geographically distributed across different locations (different cities, regions, or countries).

In this scenario, the OpenWISP application will not be able to reach the devices unless a management tunnel is used.

Therefore, **having a management VPN solution is crucial**, not only to allow OpenWISP to work properly but also to **perform debugging and troubleshooting** when needed.

Requirements for this scenario:

- A VPN server must be installed so that the OpenWISP server can reach the VPN peers. For more information on how to do this via OpenWISP, please refer to the following sections:
  - Wireguard
  - Wireguard over VXLAN
  - Zerotier
  - OpenVPN

  If you prefer to use other tunneling solutions (L2TP, Softether, etc.) and know how to configure those solutions on your own, that's fine as well.

  If the OpenWISP server is connected to a network infrastructure that allows it to reach the devices via preexisting tunneling or Intranet solutions (e.g., MPLS, SD-WAN), then setting up a VPN server is not needed, as long as there's a dedicated interface on OpenWrt with an assigned IP address that is reachable from the OpenWISP server.
- The devices must be configured to join the management tunnel automatically, either via a preexisting configuration in the firmware or via a Default Templates.
- The OpenWISP Config Agent running on the network devices must be configured to specify the `management_interface` option, which must be set to the interface name assigned by the VPN tunnel. The agent will communicate the IP of the management interface to the OpenWISP Server, and OpenWISP will use the management IP to reach the device.

  For example, if the *management interface* is named `tun0`, the openwisp-config configuration should look like the following:

```
# In /etc/config/openwisp on the device

config controller 'http'
    # ... other configuration directives ...
    option management_interface 'tun0'
```

Private Network

In some cases, the OpenWISP instance is directly connected to the same network where the devices it manages are operating.

Real-world examples:

- An office LAN where the OpenWISP instance and the network devices are in the same Layer 2 domain.

- A Layer 3 routed network, like that operated by an ISP, where each device already has an internal IP address that can be reached from the rest of the network.

In these cases, OpenWISP should be configured to accept requests using its private IP address and should be configured to use the **Last IP** field of the devices to reach them.

In this scenario, it's necessary to set the "OPENWISP_CONTROLLER_MANAGEMENT_IP_ONLY" setting to `False`.

Configure Your OpenWrt Device

This page will guide you through installing the OpenWISP agents on a device that supports OpenWrt.

Hint

No physical device? No problem! You can try OpenWISP using a Virtual Machine.

Prerequisites

Ensure you have already Installed the OpenWISP Server Application and Configured a Management Network.

Flash OpenWrt on Your Device

If you have a compatible network device, follow the official OpenWrt flashing guide.

If you don't have a physical device, you can install OpenWrt on a VirtualBox Virtual Machine.

Note

Enable SSH access and connect the device or VM to the internet.

When using VirtualBox, both Adapter1 and Adapter2 should use "Adapter Type: Intel PRO/1000 MT Desktop". Use a different IP address for the OpenWrt device than the one used for the local OpenWISP website (e.g., if your OpenWISP site uses 192.168.56.2, use 192.168.56.3 for the OpenWrt device).

Install the OpenWISP OpenWrt Agents

We recommend installing the latest versions of the OpenWISP packages. Download them onto your device from downloads.openwisp.io and then install them as follows:

```
cd /tmp

# WARNING: the URL may change over time, so verify the correct URL
# from downloads.openwisp.io

wget https://downloads.openwisp.io/openwisp-config/latest/openwisp-config_1.1.0-1_all.ipk
wget https://downloads.openwisp.io/openwisp-monitoring/latest/netjson-monitoring_0.2.1-1_all
wget https://downloads.openwisp.io/openwisp-monitoring/latest/openwisp-monitoring_0.2.1-1_al
```

```
opkg install openwisp-config_1.1.0a-1_all.ipk
opkg install netjson-monitoring_0.2.0a-1_all.ipk
opkg install openwisp-monitoring_0.2.0a-1_all.ipk
```

Note

If `wget` doesn't work (e.g., SSL issues), you can use `curl` or alternatively download the packages onto your machine and upload them to your device via `scp`.

Once the agents are installed on your OpenWrt device, let's ensure they can connect to OpenWISP successfully.

Edit the config file located at `/etc/config/openwisp`, which should look like the following sample:

```
# For more information about the config options, please see the README
# or https://github.com/openwisp/openwisp-config#configuration-options

config controller 'http'
    #option url 'https://openwisp2.mynetwork.com'
    #option interval '120'
    #option verify_ssl '1'
    #option shared_secret ''
    #option consistent_key '1'
    #option mac_interface 'eth0'
    #option management_interface 'tun0'
    #option merge_config '1'
    #option test_config '1'
    #option test_script '/usr/sbin/mytest'
    #option hardware_id_script '/usr/sbin/read_hw_id'
    #option hardware_id_key '1'
    option uuid ''
    option key ''
    # curl options
    #option connect_timeout '15'
    #option max_time '30'
    #option capath '/etc/ssl/certs'
    #option cacert '/etc/ssl/certs/ca-certificates.crt'
    # hooks
    #option pre_reload_hook '/usr/sbin/my_pre_reload_hook'
    #option post_reload_hook '/usr/sbin/my_post_reload_hook'
```

Uncomment and update the following lines:

- `url`: Set this to the hostname of your OpenWISP instance (e.g., if your OpenWISP server is at "192.168.56.2", set the URL to `https://192.168.56.2`).
- `verify_ssl`: Set to `'0'` if your controller's SSL certificate is self-signed; in production, use a valid SSL certificate to ensure security.
- `shared_secret`: Retrieve this from the OpenWISP dashboard in the Organization settings. The list of organizations is available at `/admin/openwisp_users/organization/`.
- `management_interface`: Refer to Setting Up the Management Network.

Hint

For more details on the configuration options, refer to OpenWrt Config Agent Settings.

Note

When testing or developing using the Django development server directly from your computer, make sure the server listens on all interfaces (`./manage.py runserver 0.0.0.0:8000`) and then point OpenWISP to use your local IP address (e.g. `http://192.168.1.34:8000`).

Save the file and restart the agent:

```
/etc/init.d/openwisp_config restart
```

Note

No changes are needed for the monitoring agent at this stage. The default settings work for most cases, and the agent restarts itself when the config agent is restarted.

For more details on its configuration options, refer to OpenWrt Monitoring Agent Settings.

Your OpenWrt device should now be able to register with OpenWISP.

If not, refer to the following **troubleshooting** guides:

- Troubleshooting issues with the OpenWrt Config Agent
- Troubleshooting issues with the OpenWrt Monitoring Agent
- Troubleshooting issues with the OpenWISP Server (Ansible role)

Seealso

- Config Agent Quick Start Guide
- OpenWrt Config Agent Settings
- Monitoring Agent Quick Start Guide
- OpenWrt Monitoring Agent Settings

Compiling Your Own OpenWrt Image

Warning

This section is for advanced users.

Compiling a custom OpenWrt image can save time when configuring new devices. By doing this, you can preinstall the agents and include your configurations (e.g., `url` and `shared_secret`) in the default image.

This way, you won't have to configure each new device manually, which is particularly useful if you provision and manage many devices.

Refer to the guide on compiling a custom OpenWrt image for more information.

How to Edit Django Settings

Table of Contents:

What is an OpenWISP Module?

The OpenWISP server application is composed of a number of modules called Django apps.

Django is the underlying Python web framework on top of which OpenWISP is built.

Some of the Django apps used by OpenWISP are developed and maintained by OpenWISP, other apps are developed and maintained by either Django or third party organizations, but most of these apps are configurable and customizable in different shapes or forms.

The most common way to modify the behavior of a Django app is by editing the project settings.py file, a file which holds all the global configuration of the application.

The Django based modules of OpenWISP are highly configurable and over time you may need to edit their settings, these settings are documented in the respective section of each module on this website, a reference is also provided for convenience at the end of this page.

If you are looking for a reference which lists and describes all the OpenWISP modules please refer to Architecture, Modules, Technologies.

Editing Settings with Ansible-OpenWISP2

The official ansible OpenWISP role provides many role variables which offer a convenient way to edit the most widely used settings of OpenWISP.

However, not all the possible settings have a corresponding variable because doing so would be very costly to maintain and make the code more complicated, for that reason the role provides a way to add any python instruction to define and manipulate settings via the `openwisp2_extra_django_settings_instructions` variable, e.g.:

```
# in the playbook variables add:
openwisp2_extra_django_settings_instructions:
  - |
    OPENWISP_NETWORK_TOPOLOGY_NODE_EXPIRATION = 14

    OPENWISP_MONITORING_METRICS = {
        'ping': {
            'alert_settings': {'tolerance': 60}
        },
        'config_applied': {
            'alert_settings': {'tolerance': 60}
        },
        'disk': {
            'alert_settings': {'tolerance': 60}
        },
        'memory': {
            'alert_settings': {'tolerance': 60}
        },
        'cpu': {
            'alert_settings': {
                'threshold': 95,
                'tolerance': 60
            }
```

```
        },
    }
```

This allows for great flexibility in configuring and extending OpenWISP: the possibility of running python code in the settings allows for limitless adaptation and customization.

Editing Settings with Docker-OpenWISP

Similarly to the ansible role, the dockerized version of OpenWISP provides mainly two ways of changing settings:

1. The most widely used setting have a dedicated environment variable.
2. For more advanced use cases, it's possible to provide an entirely custom django settings file.

OpenWISP Settings Reference

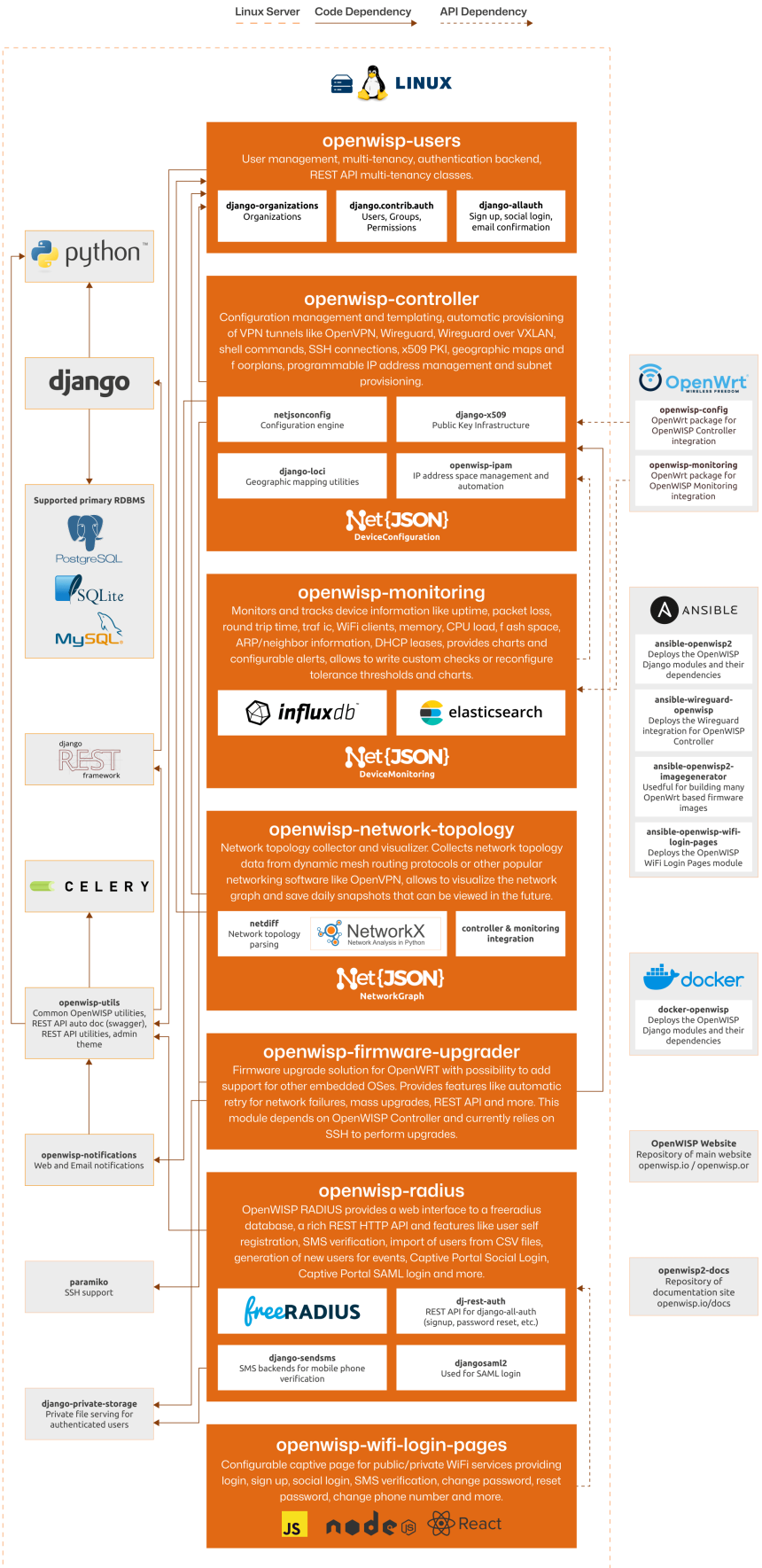
- OpenWISP Controller Settings
- OpenWISP Monitoring Settings
- OpenWISP Firmware Upgrader Settings
- OpenWISP Network Topology Settings
- OpenWISP Users Settings
- OpenWISP Notifications Settings
- OpenWISP Utils Settings

Project Overview

Architecture, Modules, Technologies

Architecture Overview

Modules, dependencies and technologies used.



The diagram above provides an overview of the OpenWISP architecture. It highlights the key technologies used, the structure of the OpenWISP modules, their major dependencies, and their interactions.

> **Important**
>
> For an enhanced viewing experience, open the image in a new browser tab.

Table of Contents:

OpenWISP Modules

> **Note**
>
> For more insights into the motivations and philosophy behind the modular architecture of OpenWISP, refer to Applying the Unix Philosophy to Django projects: a report from the real world.

Deployment

- Ansible OpenWISP2: Recommended method to deploy OpenWISP on virtual machines.

- Docker OpenWISP: Enables deployment of OpenWISP on Dockerized cloud infrastructure. While still under active development, the basic features of OpenWISP are functional.
- Ansible OpenWISP WiFi Login Pages: Ansible role for deploying the WiFi Login Pages module.
- Ansible OpenWISP2 Image Generator: Useful for generating multiple OpenWrt firmware images for different organizations with the OpenWISP packages preinstalled.
- Ansible Wireguard OpenWISP: Ansible role that enables deployment of Wireguard integration for OpenWISP Controller.

Server Side

- OpenWISP Users: Manages user authentication, multi-tenancy, and provides REST API utilities and classes for implementing multi-tenancy.
- OpenWISP Controller: Handles configuration management, VPN provisioning (OpenVPN, Wireguard, Wireguard over VXLAN), shell commands, SSH connections, x509 PKI management, geographic maps, floor plans, programmable IP address management, and subnet provisioning.

  This module depends on several Django apps and Python libraries developed or maintained by OpenWISP:

  - netjsonconfig: For configuration generation, validation, and parsing.
  - django-x509: Manages Public Key Infrastructure (certification authorities and x509 certificates).
  - django-loci: Provides geographic and indoor mapping features.
  - openwisp-ipam: Administers IP and subnet management.
  - django-rest-framework-gis: Adds GIS capabilities to Django REST Framework.
- OpenWISP Monitoring: Monitors and tracks device metrics like ping success rate, packet loss, round trip time, traffic, WiFi clients, memory, CPU load, flash space, ARP/neighbor information, DHCP leases, and provides charts and configurable alerts. It also allows custom checks and tolerance threshold configurations.
- OpenWISP Network Topology: Collects and visualizes network topology data from dynamic mesh routing protocols and other popular networking software like OpenVPN. It can visualize network graphs and save daily snapshots for future viewing.

  This module relies on two libraries developed and maintained by OpenWISP:

  - netdiff: Parses network topology.
  - netjsongraph.js: A JavaScript library for visualizing network graphs.
- OpenWISP Firmware Upgrader: Provides a firmware upgrade solution for OpenWrt and potentially other embedded OSes. Features include automatic retry for network failures, mass upgrades, a REST API, and more.
- OpenWISP RADIUS: Offers a web interface to a FreeRADIUS database, a rich REST HTTP API, and features like user self-registration, SMS verification, user import from CSV files, event-based user generation, Captive Portal Social Login, and Captive Portal SAML login.
- OpenWISP Notifications: Provides email and web notifications for OpenWISP, enabling modules to notify users about significant events in their network.
- OpenWISP Utils: Common utilities and classes shared by all OpenWISP Python modules. Includes many utilities for QA checks and automated testing, heavily used in continuous integration builds of most OpenWISP GitHub repositories.
- OpenWISP WiFi Login Pages: A configurable login page and self registration app for WiFi Hotspot services, offering features like login, sign up, social login, SMS verification, password reset and more. It is a frontend for the OpenWISP RADIUS REST API, designed for end users of a WiFi Hotspot service.

Network Device Side

- OpenWISP OpenWrt Config Agent: An OpenWrt package that integrates with OpenWISP Controller.

- OpenWISP OpenWrt Monitoring Agent: An OpenWrt package that integrates with OpenWISP Monitoring.

Website and Documentation

- openwisp-docs: Repository for the OpenWISP documentation, hosted on openwisp.io/docs.
- OpenWISP-Website: Repository for the OpenWISP website, hosted on openwisp.org.

Main Technologies Used

Python

Python is the primary programming language used for the server-side application (web admin, API, controller, workers).

Originally, OpenWISP was built on Ruby On Rails, but we later switched to Python due to its suitability for networking and a larger pool of potential contributors.

Find out more on why OpenWISP chose Python as its main language.

Django

Django is one of the most popular web frameworks for Python.

It is used extensively in our modules, allowing rapid development and access to a rich ecosystem.

It's the base framework used in most of the server-side modules of OpenWISP.

Find out more on why OpenWISP chose Django as its main web framework.

Django REST Framework

Django REST framework is a powerful and flexible toolkit for building Web APIs based on Django, widely used in most of the Django and web-based OpenWISP modules.

Find out more on why OpenWISP chose Django REST Framework to build its REST API.

Celery

Celery is a Python implementation of a distributed task queue. It is heavily used in OpenWISP to execute background tasks, perform network operations like monitoring checks, configuration updates, firmware upgrades, and more.

OpenWrt

OpenWrt is a Linux distribution designed for embedded systems, routers, and networking in general.

It has a very skilled community and is used as a base by many hardware vendors (Technicolor, Ubiquiti Networks, Linksys, Teltonika, and many others).

Lua

Lua is a lightweight, multi-paradigm programming language designed primarily for embedded systems and clients.

Lua is cross-platform, since the interpreter is written in ANSI C, and has a relatively simple C API.

It is the official scripting language of OpenWrt and is used heavily in the OpenWrt packages of OpenWISP: openwisp-config and openwisp-monitoring.

Node.js and React JS

NodeJS is a JavaScript runtime for building JS-based applications.

In OpenWISP, it's used as a base for frontend applications along with React, like the WiFi Login Pages module.

Ansible

Ansible is a popular software automation tool written in Python, generally used for automating software provisioning, configuration management, and application deployment.

We use Ansible to provide automated procedures to deploy OpenWISP, to compile custom OpenWrt images for different organizations, to deploy OpenWISP WiFi Login Pages, and to deploy the Wireguard integration for OpenWISP Controller.

Docker

We use Docker in docker-openwisp, which aims to ease the deployment of OpenWISP in a containerized infrastructure.

NetJSON

NetJSON is a data interchange format based on JSON designed to ease the development of software tools for computer networks.

RADIUS

RADIUS (Remote Authentication Dial-In User Service) is a networking protocol used for centralized Authentication, Authorization, and Accounting management of network services.

FreeRADIUS

FreeRADIUS is the most popular open-source implementation of the RADIUS protocol and is extensively relied upon in OpenWISP RADIUS.

Mesh Networking

A mesh network is a local network topology where infrastructure nodes connect directly, dynamically, and non-hierarchically to as many other nodes as possible. They cooperate to efficiently route data to and from clients.

OpenWrt supports the standard mesh mode (802.11s), which OpenWISP supports out of the box. Additionally, OpenWrt can support other popular dynamic open-source routing protocols such as OLSRd2, BATMAN-advanced, Babel, BMX, etc.

For more information on how to set up a mesh network with OpenWISP, refer to: How to Set Up a Wireless Mesh Network.

InfluxDB

InfluxDB is the default open-source time-series database used in OpenWISP Monitoring.

Elasticsearch

Elasticsearch is an alternative option that can be used in OpenWISP Monitoring as a time-series database. It excels in storing and retrieving data quickly and efficiently.

Networkx

Networkx is a network graph analysis library written in Python and used under the hood by netdiff and the OpenWISP Network Topology module.

Relational Databases

Django supports several Relational Database Management Systems.

The most notable ones are:

- PostgreSQL
- MySQL
- SQLite

For production usage, we recommend PostgreSQL.

For development, we recommend SQLite for its simplicity.

Other Notable Dependencies

- Paramiko (used in OpenWISP Controller and Firmware Upgrader).
- Django-allauth (used in OpenWISP Users).
- Django-organizations (used in OpenWISP Users).
- Django-swappable-models (used in all major Django modules).
- Django-private-storage (used in OpenWISP RADIUS and Firmware Upgrader).
- Dj-rest-auth (used in OpenWISP RADIUS).
- Django-sendsms (used in OpenWISP RADIUS).
- Django-saml2 (used in OpenWISP RADIUS).

Values and Goals of OpenWISP

Table of Contents:

What is OpenWISP?

OpenWISP is a robust and versatile software platform designed to simplify and automate network management, with a strong emphasis on wireless networks. It's widely used in various scenarios, including public WiFi hotspots, mesh networks, community networks, and IoT applications.

In December 2016, OpenWISP 2 was launched, marking the next generation of our software. This version, built with Python and Django, replaced the original version developed with Ruby on Rails. The OpenWISP community has

since cultivated an ecosystem of applications and tools that empower developers to create custom networking solutions. Our mission is to drive innovation and promote freedom in the realm of network infrastructure automation.

History

Refer to History of OpenWISP.

Core Values

1. Communication through Electronic Means is a Human Right

We believe that **communication through electronic means is a fundamental human right**.

According to Mozilla, 4 billion people live without internet access today.

Having witnessed the significant progress the internet has brought to our society, we are deeply convinced that addressing the issue of internet connectivity will help to alleviate the economic disparity that is so evident at the beginning of the 21st century.

For these reasons, **fighting the digital divide, both primary (lack of infrastructure) and secondary (lack of know-how), is our utmost priority**.

2. Net Neutrality

We believe Net Neutrality is beneficial to the internet because it ensures fair treatment (non-discrimination) of private communications.

The very first public WiFi networks built with OpenWISP in Italy adhere strictly to this principle: no content filtering of any type is allowed on these networks, and no special privileges are given to any private entities.

For this reason, we are opposed to including in our ecosystem and documentation any software tools or tutorials that aim to implement solutions contrary to Net Neutrality.

3. Privacy

Privacy is fundamental to a healthy and functional society.

The initial public WiFi networks built with OpenWISP in Italy adhere strictly to this principle: traffic logs are stored only for the duration mandated by law, and personal data is never sold to third parties.

Therefore, we oppose the inclusion in our ecosystem and documentation of any software tool or tutorial that aims to intrude upon user privacy by collecting and selling their data to third parties for profit.

4. Open Source, Licenses, and Collaboration

We release all our software under Open Source licenses on GitHub.

We primarily use two types of licenses:

- **GPLv3**: Used for software modules we consider to have significant commercial value for ISPs and private companies. This license aims to prevent these tools from being included in proprietary closed-source solutions, ensuring that private entities do not profit from our community's work without contributing back.
- **BSD3** and **MIT**: These highly permissive licenses are used for experimental and innovative software modules that are valuable but less monetizable. By allowing these modules to be included in proprietary solutions, we aim to reduce duplication of effort and encourage contributions from organizations and individuals.

We advocate for transparency and a community-driven approach, welcoming all new participants, contributors, and users.

Our community values support, friendliness, and collaboration, aiming to make our software as useful as possible to a wide audience, **while upholding our core values**.

We encourage those who share our values to reach out to us through our support channels and contribute to the project in any way they can, according to their means and available time.

5. Software Reusability for Long-Term Sustainability

Long-time contributors to **OpenWISP have firsthand experience with the pitfalls of dealing with inflexible monolithic applications** that are difficult to reuse beyond their original design scope.

We've witnessed numerous projects emerge with great promise, only to develop their code from scratch and eventually fade into obscurity. This recurring cycle represents a tremendous waste of human effort, energy, and resources.

For this reason, **OpenWISP 2 places a strong emphasis on modularity and reusability**, drawing inspiration from **best practices established in the Unix world** as outlined in The Art of Unix Programming by Eric S. Raymond.

The core modules of OpenWISP 2 are licensed and designed to facilitate inclusion by developers outside the OpenWISP community in their own applications (subject to licensing terms).

This approach fosters an ecosystem of modern networking software tools that attracts developers from around the globe.

The shared interest of users, modifiers, sharers, resellers, and contributors of these modules forms the bedrock of **long-term sustainability**.

Goals

- Help solve the problem of lack of internet connectivity by simplifying the deployment and management of low-cost network infrastructure worldwide.
- Drive innovation in the networking software realm through automation, modularity, reusability, flexibility, extensibility, and collaboration.
- Foster an ecosystem of software tools capable of generating numerous OpenWISP derivatives, enhancing the accessibility and affordability of electronic communication.
- Mitigate vendor lock-in by striving to support multiple operating systems and hardware vendors. While our official support is currently limited to OpenWrt derivatives, we have experimental configuration backends for Raspbian and AirOS, demonstrating feasibility for supporting multiple systems.
- Provide comprehensive documentation for both users and developers.
- Develop user-friendly web interfaces accessible to a broad audience.

Installers

Ansible OpenWISP

Seealso

Source code: github.com/openwisp/ansible-openwisp2.

This ansible role allows deploying the OpenWISP Server Application.

Recommended minimum ansible core version: 2.13.

Tested on **Debian (Bookworm/Bullseye)**, **Ubuntu (24/22/20 LTS)**.

The following diagram illustrates the role of the Ansible OpenWISP role within the OpenWISP architecture.

OpenWISP Architecture: highlighted Ansible OpenWISP role

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

System Requirements

The following specifications will run a new, *empty* instance of OpenWISP. Please ensure you account for the amount of disk space your use case will require, e.g. allocate enough space for users to upload floor plan images.

Hardware Requirements (Recommended)

- 2 CPUs
- 2 GB Memory
- Disk space - depends on the projected size of your database and uploaded photo images

Keep in mind that increasing the number of celery workers will require more memory and CPU. You will need to increase the amount of celery workers as the number of devices you manage grows.

For more information about how to increase concurrency, look for the variables which end with `_concurrency` or `_autoscale` in the Role Variables section.

Software

A fresh installation of one of the supported operating systems is generally sufficient, with no preconfiguration required. The Ansible Playbook will handle the installation and configuration of all dependencies, providing you with a fully operational OpenWISP setup.

Important

Ensure the hostname of your target machine matches what is in your Ansible configuration file. Also, please ensure that Ansible can access your target machine by SSH, be it either with a key or password. For more information see the Ansible Getting Started Documentation.

Supported Operating Systems

- Debian 12
- Debian 11
- Ubuntu 24 LTS
- Ubuntu 22 LTS
- Ubuntu 20 LTS

Deploying OpenWISP Using Ansible

Introduction & Prerequisites

Note

If you want to use the latest features of OpenWISP, refer to Deploying the Development Version of OpenWISP.

If you don't know how to use ansible, don't panic, this procedure will guide you towards a fully working basic OpenWISP installation.

If you already know how to use ansible, you can skip this tutorial.

First of all you need to understand two key concepts:

- for **“production server”** we mean a server (**not a laptop or a desktop computer!**) with public IpV4 / IPv6 which is used to host OpenWISP
- for **“local machine”** we mean the host from which you launch ansible, e.g.: your own laptop

Ansible is a configuration management tool that works by entering production servers via SSH, **so you need to install it and configure it on the machine where you launch the deployment** and this machine must be able to SSH into the production server.

Ansible will be run on your local machine and from there it will connect to the production server to install OpenWISP.

Note

It is recommended to use this procedure on clean virtual machines or linux containers.

If you are trying to install OpenWISP on your laptop or desktop PC just for testing purposes, please read Install OpenWISP for testing in a VirtualBox VM.

Install Ansible

Install ansible (minimum recommended version 2.13) **on your local machine** (not the production server!) if you haven't done already.

We suggest following the ansible installation guide. to install ansible. It is recommended to install ansible through a virtual environment to avoid dependency issues.

Please ensure that you have the correct version of Jinja installed in your Python environment: `pip install Jinja2>=2.11`

Install This Role

For the sake of simplicity, the easiest thing is to install this role **on your local machine** via `ansible-galaxy` (which was installed when installing ansible), therefore run:

```
ansible-galaxy install openwisp.openwisp2
```

Ensure that you have the community.general and ansible.posix collections installed and up to date:

```
ansible-galaxy collection install "community.general:>=3.6.0"
ansible-galaxy collection install "ansible.posix"
```

Choose a Working Directory

Choose a working directory **on your local machine** where to put the configuration of OpenWISP.

This will be useful when you will need to upgrade OpenWISP.

E.g.:

```
mkdir ~/openwisp2-ansible-playbook
cd ~/openwisp2-ansible-playbook
```

Create Inventory File

The inventory file is where group of servers are defined. In our simple case we will define just one group in which we will put just one server.

Create a new file called `hosts` in the working directory **on your local machine** (the directory just created in the previous step), with the following contents:

```
[openwisp2]
openwisp2.mydomain.com
```

Substitute `openwisp2.mydomain.com` with your **production server**'s hostname - **DO NOT REPLACE openwisp2.mydomain.com WITH AN IP ADDRESS**, otherwise email sending through postfix will break, causing 500 internal server errors on some operations.

Create Playbook File

Create a new playbook file `playbook.yml` **on your local machine** with the following contents:

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
    - openwisp.openwisp2
  vars:
    openwisp2_default_from_email: "openwisp2@openwisp2.mydomain.com"
```

The line `become: "{{ become | default('yes') }}"` means ansible will use the `sudo` program to run each command. You may remove this line if you don't need it (e.g.: if you are `root` user on the production server).

You may replace `openwisp2` on the `hosts` field with your production server's hostname if you desire.

Substitute `openwisp2@openwisp2.mydomain.com` with what you deem most appropriate as default sender for emails sent by OpenWISP 2.

Run the Playbook

Now is time to **deploy OpenWISP to the production server**.

Run the playbook **from your local machine** with:

```
ansible-playbook -i hosts playbook.yml -u <user> -k --become -K
```

Substitute `<user>` with your **production server**'s username.

The `-k` argument will need the `sshpass` program.

You can remove `-k`, `--become` and `-K` if your public SSH key is installed on the server.

Tip

- If you have an error like `Authentication or permission failure` then try to use *root* user `ansible-playbook -i hosts playbook.yml -u root -k`
- If you have an error about adding the host's fingerprint to the `known_hosts` file, you can simply connect to the host via SSH and answer yes when prompted; then you can run `ansible-playbook` again.

When the playbook is done running, if you got no errors you can login at `https://openwisp2.mydomain.com/admin` with the following credentials:

```
username: admin
password: admin
```

Substitute `openwisp2.mydomain.com` with your production server's hostname.

Now proceed with the following steps:

1. change the password (and the username if you like) of the superuser as soon as possible
2. update the `name` field of the default `Site` object to accurately display site name in email notifications
3. edit the information of the default organization
4. in the default organization you just updated, note down the automatically generated *shared secret* option, you will need it to use the auto-registration feature of openwisp-config
5. this Ansible role creates a default template to update `authorized_keys` on networking devices using the default access credentials. The role will either use an existing SSH key pair or create a new one if no SSH key pair exists on the host machine.

Now you are ready to start configuring your network! **If you need help** you can ask questions on one of the official OpenWISP Support Channels.

Upgrading OpenWISP

Important

It is strongly recommended to back up your current instance before upgrading.

Update this ansible-role via `ansible-galaxy`:

```
ansible-galaxy install --force openwisp.openwisp2
```

Run `ansible-playbook` again **from your local machine**:

```
ansible-playbook -i hosts playbook.yml
```

You may also run the playbook automatically periodically or when a new release of OpenWISP2, for example, by setting up a continuous integration system.

Deploying the Development Version of OpenWISP

The following steps will help you set up and install the development version of OpenWISP which is not released yet, but ships new features and improvements.

Create a directory for organizing your playbook, roles and collections. In this example, `openwisp-dev` is used. Create `roles` and `collections` directories in `~/openwisp-dev`.

```
mkdir -p ~/openwisp-dev/roles
mkdir -p ~/openwisp-dev/collections
```

Change directory to `~/openwisp-dev/` in terminal and create configuration and requirement files for Ansible.

```
cd ~/openwisp-dev/
touch ansible.cfg
touch requirements.yml
```

Setup `roles_path` and `collections_paths` variables in `ansible.cfg` as follows:

```
[defaults]
roles_path=~/openwisp-dev/roles
collections_paths=~/openwisp-dev/collections
```

Ensure your `requirements.yml` contains following content:

```
---
roles:
  - src: https://github.com/openwisp/ansible-openwisp2.git
    version: master
    name: openwisp.openwisp2-dev
collections:
  - name: community.general
    version: ">=3.6.0"
```

Install requirements from the `requirements.yml` as follows

```
ansible-galaxy install -r requirements.yml
```

Now, create hosts file and `playbook.yml`:

```
touch hosts
touch playbook.yml
```

Follow instructions in Create Inventory File section to configure `hosts` file.

You can reference the example playbook below (tested on Debian 11) for installing a fully-featured version of OpenWISP.

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
    - openwisp.openwisp2-dev
  vars:
    openwisp2_network_topology: true
    openwisp2_firmware_upgrader: true
    openwisp2_radius: true
    openwisp2_monitoring: true # monitoring is enabled by default
```

Read Role Variables section to learn about available configuration variables.

Follow instructions in Run the Playbook section to run above playbook.

Using Let's Encrypt SSL Certificate

This section explains how to **automatically install and renew a valid SSL certificate** signed by Let's Encrypt.

The first thing you have to do is to setup a valid domain for your OpenWISP instance, this means your inventory file (hosts) should look like the following:

```
[openwisp2]
openwisp2.yourdomain.com
```

You must be able to add a DNS record for `openwisp2.yourdomain.com`, you cannot use an ip address in place of `openwisp2.yourdomain.com`.

Once your domain is set up and the DNS record is propagated, proceed by installing the ansible role geerlingguy.certbot:

```
ansible-galaxy install geerlingguy.certbot
```

Then proceed to edit your `playbook.yml` so that it will look similar to the following example:

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
    - geerlingguy.certbot
    - openwisp.openwisp2
  vars:
    # SSL certificates
    openwisp2_ssl_cert: "/etc/letsencrypt/live/{{ inventory_hostname }}/fullchain.pem"
    openwisp2_ssl_key: "/etc/letsencrypt/live/{{ inventory_hostname }}/privkey.pem"

    # certbot configuration
    certbot_auto_renew_minute: "20"
    certbot_auto_renew_hour: "5"
    certbot_create_if_missing: true
    certbot_auto_renew_user: "<privileged-users-to-renew-certs>"
    certbot_certs:
      - email: "<paste-your-email>"
        domains:
          - "{{ inventory_hostname }}"
  pre_tasks:
    - name: Update APT package cache
      apt:
        update_cache: true
        changed_when: false
        retries: 5
        delay: 10
        register: result
        until: result is success
```

Read the documentation of geerlingguy.certbot to learn more about configuration of certbot role.

Once you have set up all the variables correctly, run the playbook again.

Enabling OpenWISP Modules

Enabling the Monitoring Module

The Monitoring module is enabled by default, it can be disabled by setting `openwisp2_monitoring` to `false`.

Enabling the Firmware Upgrader Module

It is encouraged that you read the quick-start guide of openwisp-firmware-upgrader before going ahead.

To enable the Firmware Upgrader module you need to set `openwisp2_firmware_upgrader` to `true` in your `playbook.yml` file. Here's a short summary of how to do this:

Step 1: Install ansible

Step 2: Install this role

Step 3: Create inventory file

Step 4: Create a playbook file with following contents:

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
    - openwisp.openwisp2
  vars:
    openwisp2_firmware_upgrader: true
```

Step 5: Run the playbook

When the playbook is done running, if you got no errors you can login at https://openwisp2.mydomain.com/admin with the following credentials:

```
username: admin
password: admin
```

You can configure openwisp-firmware-upgrader specific settings using the `openwisp2_extra_django_settings` or `openwisp2_extra_django_settings_instructions`.

E.g:

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
    - openwisp.openwisp2
  vars:
    openwisp2_firmware_upgrader: true
    openwisp2_extra_django_settings_instructions:
      - |
        OPENWISP_CUSTOM_OPENWRT_IMAGES = (
            ('my-custom-image-squashfs-sysupgrade.bin', {
                'label': 'My Custom Image',
                'boards': ('MyCustomImage',)
            }),
        )
```

Refer the Role Variables section of the documentation for a complete list of available role variables.

Enabling the Network Topology Module

To enable the Network Topology module you need to set `openwisp2_network_topology` to `true` in your `playbook.yml` file. Here's a short summary of how to do this:

Step 1: Install ansible

Step 2: Install this role

Step 3: Create inventory file

Step 4: Create a playbook file with following contents:

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
```

```
    - openwisp.openwisp2
  vars:
    openwisp2_network_topology: true
```

Step 5: Run the playbook

When the playbook is done running, if you got no errors you can login at https://openwisp2.mydomain.com/admin with the following credentials:

```
username: admin
password: admin
```

Enabling the RADIUS Module

To enable the RADIUS module you need to set `openwisp2_radius` to `true` in your `playbook.yml` file. Here's a short summary of how to do this:

Step 1: Install ansible

Step 2: Install this role

Step 3: Create inventory file

Step 4: Create a playbook file with following contents:

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
    - openwisp.openwisp2
  vars:
    openwisp2_radius: true
    openwisp2_freeradius_install: true
    # set to false when you don't want to register openwisp-radius
    # API endpoints.
    openwisp2_radius_urls: true
```

Note

`openwisp2_freeradius_install` option provides a basic configuration of freeradius for OpenIWSP, it sets up the radius user token mechanism if you want to use another mechanism or manage your freeradius separately, please disable this option by setting it to `false`.

Step 5: Run the playbook

When the playbook is done running, if you got no errors you can login at:

```
https://openwisp2.mydomain.com/admin
username: admin
password: admin
```

Note: for more information regarding radius configuration options, look for the word “radius” in the Role Variables section of this document.

Configuring FreeRADIUS for WPA Enterprise (EAP-TTLS-PAP)

You can use OpenWISP RADIUS for setting up WPA Enterprise (EAP-TTLS-PAP) authentication. This allows to authenticate on WiFi networks using Django user credentials. Prior to proceeding, ensure you've reviewed the tutorial on How to Set Up WPA Enterprise (EAP-TTLS-PAP) Authentication. This documentation section complements the tutorial and focuses solely on demonstrating the ansible role's capabilities to configure FreeRADIUS.

Important

The ansible role supports OpenWISP's multi-tenancy by creating individual FreeRADIUS sites for each organization. You must include configuration details for **each organization** that will use WPA Enterprise.

Here's an example playbook which enables OpenWISP RADIUS module, installs FreeRADIUS, and configures it for WPA Enterprise (EAP-TTLS-PAP):

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
    - openwisp.openwisp2
  vars:
    openwisp2_radius: true
    openwisp2_freeradius_install: true
    # Define a list of dictionaries detailing each organization's
    # name, UUID, RADIUS token, and ports for authentication,
    # accounting, and the inner tunnel. These details will be used
    # to create FreeRADIUS sites tailored for WPA Enterprise
    # (EAP-TTLS-PAP) authentication per organization.
    freeradius_eap_orgs:
        # A reference name for the organization,
        # used in FreeRADIUS configurations.
        # Don't use spaces or special characters.
      - name: openwisp
        # UUID of the organization.
        # You can retrieve this from the organization admin
        # in the OpenWISP web interface.
        uuid: 00000000-0000-0000-0000-000000000000
        # Radius token of the organization.
        # You can retrieve this from the organization admin
        # in the OpenWISP web interface.
        radius_token: secret-radius-token
        # Port used by the authentication service for
        # this FreeRADIUS site
        auth_port: 1822
        # Port used by the accounting service for this FreeRADIUS site
        acct_port: 1823
        # Port used by the authentication service of inner tunnel
        # for this FreeRADIUS site
        inner_tunnel_auth_port: 18230
        # If you want to use a custom certificate for FreeRADIUS
        # EAP module, you can specify the path to the CA, server
        # certificate, and private key, and DH key as follows.
        # Ensure that these files can be read by the "freerad" user.
        cert: /etc/freeradius/certs/cert.pem
        private_key: /etc/freeradius/certs/key.pem
        ca: /etc/freeradius/certs/ca.crt
        dh: /etc/freeradius/certs/dh
        tls_config_extra: |
          private_key_password = whatever
          ecdh_curve = "prime256v1"
      # You can add as many organizations as you want
      - name: demo
        uuid: 00000000-0000-0000-0000-000000000001
        radius_secret: demo-radius-token
        auth_port: 1832
        acct_port: 1833
```

```
    inner_tunnel_auth_port: 18330
    # If you omit the certificate fields,
    # the FreeRADIUS site will use the default certificates
    # located in /etc/freeradius/certs.
```

In the example above, custom ports 1822, 1823, and 18230 are utilized for FreeRADIUS authentication, accounting, and inner tunnel authentication, respectively. These custom ports are specified because the Ansible role creates a common FreeRADIUS site for all organizations, which also supports captive portal functionality. This common site is configured to listen on the default FreeRADIUS ports 1812, 1813, and 18120. Therefore, when configuring WPA Enterprise authentication for each organization, unique ports must be provided to ensure proper isolation and functionality.

Using Let's Encrypt Certificate for WPA Enterprise (EAP-TTLS-PAP)

In this section, we demonstrate how to utilize Let's Encrypt certificates for WPA Enterprise (EAP-TTLS-PAP) authentication. Similar to the Using Let's Encrypt SSL Certificate, we use geerlingguy.certbot role to automatically install and renew a valid SSL certificate.

The following example playbook achieves the following goals:

- Provision a separate Let's Encrypt certificate for the *freeradius.yourdomain.com* hostname. This certificate will be utilized by the FreeRADIUS site for WPA Enterprise authentication.
- Create a renewal hook to set permissions on the generated certificate so the FreeRADIUS server can read it.

Note

You can also use the same SSL certificate for both Nginx and FreeRADIUS, but it's crucial to understand the security implications. Please exercise caution and refer to the example playbook comments for guidance.

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
    - geerlingguy.certbot
    - openwisp.openwisp2
  vars:
    # certbot configuration
    certbot_auto_renew_minute: "20"
    certbot_auto_renew_hour: "5"
    certbot_create_if_missing: true
    certbot_auto_renew_user: "<privileged-users-to-renew-certs>"
    certbot_certs:
      - email: "<paste-your-email>"
        domains:
          - "{{ inventory_hostname }}"
      # If you choose to re-use the same certificate for both services,
      # you can omit the following item in your playbook.
      - email: "<paste-your-email>"
        domains:
          - "freeradius.yourdomain.com"
    # Configuration to use Let's Encrypt certificate for OpenWISP server (Nnginx)
    openwisp2_ssl_cert: "/etc/letsencrypt/live/{{ inventory_hostname }}/fullchain.pem"
    openwisp2_ssl_key: "/etc/letsencrypt/live/{{ inventory_hostname }}/privkey.pem"
    # Configuration for openwisp-radius
    openwisp2_radius: true
    openwisp2_freeradius_install: true
    freeradius_eap_orgs:
      - name: demo
```

```
        uuid: 00000000-0000-0000-0000-000000000001
        radius_secret: demo-radius-token
        auth_port: 1832
        acct_port: 1833
        inner_tunnel_auth_port: 18330
        # Update the cert_file and private_key paths to point to the
        # Let's Encrypt certificate.
        cert: /etc/letsencrypt/live/freeradius.yourdomain.com/fullchain.pem
        private_key: /etc/letsencrypt/live/freeradius.yourdomain.com/privkey.pem
        # If you choose to re-use the same certificate for both services,
        # your configuration would look like this
        # cert: /etc/letsencrypt/live/{{ inventory_hostname }}/fullchain.pem
        # private_key: /etc/letsencrypt/live/{{ inventory_hostname }}/privkey.pem
    tasks:
      # Tasks to ensure the Let's Encrypt certificate can be read by the FreeRADIUS server.
      # If you are using the same certificate for both services, you need to
      # replace "freeradius.yourdomain.com" with "{{ inventory_hostname }}"
      # in the following task.
      - name: "Create a renewal hook for setting permissions on /etc/letsencrypt/live/freeradi
        copy:
          content: |
            #!/bin/bash
            chown -R root:freerad /etc/letsencrypt/live/ /etc/letsencrypt/archive/
            chmod 0750 /etc/letsencrypt/live/ /etc/letsencrypt/archive/
            chmod -R 0640 /etc/letsencrypt/archive/freeradius.yourdomain.com/
            chmod 0750 /etc/letsencrypt/archive/freeradius.yourdomain.com/
          dest: /etc/letsencrypt/renewal-hooks/post/chown_freerad
          owner: root
          group: root
          mode: '0700'
        register: chown_freerad_result
      - name: Change the ownership of the certificate files
        when: chown_freerad_result.changed
        command: /etc/letsencrypt/renewal-hooks/post/chown_freerad
```

Deploying Custom Static Content

For deploying custom static content (HTML files, etc.) add all the static content in `files/ow2_static` directory. The files inside `files/ow2_static` will be uploaded to a directory named `static_custom` in `openwisp2_path`.

This is helpful for customizing OpenWISP's theme.

E.g., if you added a custom CSS file in `files/ow2_static/css/custom.css`, the file location to use in OPENWISP_ADMIN_THEME_LINKS setting will be `css/custom.css`.

Configuring CORS Headers

While integrating OpenWISP with external services, you can run into issues related to CORS (Cross-Origin Resource Sharing). This role allows users to configure the CORS headers with the help of django-cors-headers package. Here's a short summary of how to do this:

Step 1: Install ansible

Step 2: Install this role

Step 3: Create inventory file

Step 4: Create a playbook file with following contents:

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
```

```
    - openwisp.openwisp2
  vars:
    # Cross-Origin Resource Sharing (CORS) settings
    openwisp2_django_cors:
      enabled: true
      allowed_origins_list:
        - https://frontend.openwisp.org
        - https://logs.openwisp.org
```

Note: to learn about the supported fields of the `openwisp2_django_cors` variable, look for the word *"openwisp2_django_cors"* in the Role Variables section of this document.

Step 5: Run the playbook

When the playbook is done running, if you got no errors you can login at https://openwisp2.mydomain.com/admin, with the following credentials:

```
username: admin
password: admin
```

The ansible-openwisp2 only provides abstraction (variables) for handful of settings available in django-cors-headers module. Use the `openwisp2_extra_django_settings_instructions` or `openwisp2_extra_django_settings` variable to configure additional setting of `django-cors-headers` as shown in the following example:

```
- hosts: openwisp2
  become: "{{ become | default('yes') }}"
  roles:
    - openwisp.openwisp2
  vars:
    openwisp2_django_cors:
      enabled: true
      allowed_origins_list:
        - https://frontend.openwisp.org
        - https://logs.openwisp.org
    # Configuring additional settings for django-cors-headers
    openwisp2_extra_django_settings_instructions:
      - |
        CORS_ALLOW_CREDENTIALS = True
        CORS_ALLOW_ALL_ORIGINS = True
```

Install OpenWISP for Testing in a VirtualBox VM

If you want to try out OpenWISP in your own development environment, the safest way is to use a VirtualBox Virtual Machine (from here on VM).

Using Vagrant

Since August 2018 there's a new fast and easy way to install OpenWISP for testing purposes leveraging Vagrant, a popular open source tool for building and maintaining portable virtual software development environments.

To use this new way, clone the repository vagrant-openwisp2, it contains the instructions (in the `README.md`) and the vagrant configuration to perform the automatic installation.

Alternatively, you can read on to learn how to install *VirtualBox* and run *ansible-openwisp2* manually, this is useful if you need to test advanced customizations of *OpenWISP*.

Installing Debian 11 on VirtualBox

Install VirtualBox and create a new Virtual Machine running Debian 11. A step-by-step guide is available here, however we need to change a few things to get ansible working.

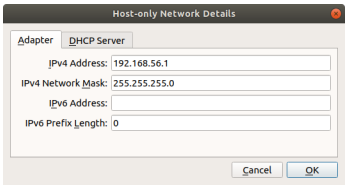
VM Configuration

Proceed with the installation as shown in the guide linked above, and come back here when you see this screen:

We're only running this as a server, so you can uncheck `Debian desktop environment`. Make sure `SSH server` and `standard system utilities` are checked.

Next, add a Host-only Network Adapter and assign an IP address to the VM.

- On the Main VirtualBox page, Go to `File > Host Network Manager`
- Click the + icon to create a new adapter
- Set the IPv4 address to `192.168.56.1` and the IPv4 Network Mask to `255.255.255.0`. You may need to select `Configure Adapter Manually` to do this. The IPv6 settings can be ignored



- Shut off your VM
- In your VM settings, in the Network section, click Adapter 2 and Enable this Adapter
- Select Host-only adapter and the name of the adapter you created
- Boot up your VM, run `su`, and type in your superuser password
- Run `ls /sys/class/net` and take note of the output
- Run `nano /etc/network/interfaces` and add the following at the end of the file:

```
auto enp0s8
iface enp0s8 inet static
    address 192.168.56.2
    netmask 255.255.255.0
    network 192.168.56.0
    broadcast 192.168.56.255
```

  Replace `enp0s8` with the network interface not present in the file but is shown when running `ls /sys/class/net`.

- Save the file with *CTRL+O* then Enter, and exit with *CTRL+X*.
- Restart the machine by running `reboot`.

Make sure you can access your VM via ssh:

```
ssh 192.168.56.2
```

Back to your local machine

Proceed with these steps in your **local machine**, not the VM.

Step 1: Install ansible

Step 2: Install the OpenWISP2 role for Ansible

Step 3: Set up a working directory

Step 4: Create the `hosts` file

Create an ansible inventory file named `hosts` **in your working directory** (i.e. not in the VM) with the following contents:

```
[openwisp2]
192.168.56.2
```

Step 5: Create the ansible playbook

In the same directory where you created the `host` file, create a file named `playbook.yml` which contains the following:

```
- hosts: openwisp2
  roles:
    - openwisp.openwisp2
  # the following line is needed only when an IP address is used as the inventory hostname
  vars:
      postfix_myhostname: localhost
```

Step 6: Run the playbook

```
ansible-playbook -i hosts playbook.yml -b -k -K --become-method=su
```

When the playbook ran successfully, you can log in at `https://192.168.56.2/admin` with the following credentials:

```
username: admin
password: admin
```

Troubleshooting

OpenWISP is deployed using **uWSGI** and also uses **daphne** for WebSockets and **celery** as a task queue.

All these services are run by **supervisor**.

```
sudo service supervisor start|stop|status
```

You can view each individual process run by supervisor with the following command:

```
sudo supervisorctl status
```

For more information about Supervisord, refer to Running supervisorctl.

The **nginx** web server sits in front of the **uWSGI** application server. You can control nginx with the following commands:

```
service nginx status start|stop|status
```

OpenWISP is installed in `/opt/openwisp2` (unless you changed the `openwisp2_path` variable in the Ansible playbook configuration). These are some useful directories to check when experiencing issues.

Location	Description
/opt/openwisp2	The OpenWISP 2 root directory.
/opt/openwisp2/log	Log files
/opt/openwisp2/env	Python virtual environment
/opt/openwisp2/db.sqlite3	OpenWISP 2 SQLite database

All processes are running as the `www-data` user.

If you need to copy or edit files, you can switch to the `www-data` user with the following commands:

```
sudo su www-data -s /bin/bash
cd /opt/openwisp2
source env/bin/activate
```

SSL Certificate Gotchas

When you access the admin website, you will receive an SSL certificate warning because the playbook creates a self-signed (untrusted) SSL certificate. You can get rid of the warning by installing your own trusted certificate and setting the `openwisp2_ssl_cert` and `openwisp2_ssl_key` variables accordingly or by following the instructions explained in the section Using Let's Encrypt SSL Certificate.

If you keep the untrusted certificate, you will also need to disable SSL verification on devices using openwisp-config by setting `verify_ssl` to `0`, although we advise against using this kind of setup in a production environment.

Role Variables

This role has many variables values that can be changed to best suit your needs.

Below are listed all the variables you can customize (you may also want to take a look at the default values of these variables).

```
- hosts: yourhost
  roles:
  # you can add other roles here
    - openwisp.openwisp2
  vars:
    # Enable the modules you want to use
    openwisp2_network_topology: false
    openwisp2_firmware_upgrader: false
    openwisp2_monitoring: true
    # you may replace the values of these variables with any value or URL
    # supported by pip (the python package installer)
    # use these to install forks, branches or development versions
    # WARNING: only do this if you know what you are doing; disruption
    # of service is very likely to occur if these variables are changed
    # without careful analysis and testing
    openwisp2_controller_version: "openwisp-controller~=1.0.0"
    openwisp2_network_topology_version: "openwisp-network-topology~=1.0.0"
    openwisp2_firmware_upgrader_version: "openwisp-firmware-upgrader~=1.0.0"
    openwisp2_monitoring_version: "openwisp-monitoring~=1.0.0"
    openwisp2_radius_version: "openwisp-radius~=1.0.0"
    openwisp2_django_version: "django~=3.2.13"
    # Setting this to true will enable subnet division feature of
    # openwisp-controller. Refer openwisp-controller documentation
    # for more information. https://github.com/openwisp/openwisp-controller#subnet-division-
    # By default, it is set to false.
    openwisp2_controller_subnet_division: true
    # when openwisp2_radius_urls is set to false, the radius module
    # is setup but it's urls are not added, which means API and social
    # views cannot be used, this is helpful if you have an external
    # radius instance.
    openwisp2_radius_urls: "{{ openwisp2_radius }}"
    openwisp2_path: /opt/openwisp2
    # It is recommended that you change the value of this variable if you intend to use
    # OpenWISP2 in production, as a misconfiguration may result in emails not being sent
    openwisp2_default_from_email: "openwisp2@yourhostname.com"
    # Email backend used by Django for sending emails. By default, the role
    # uses "CeleryEmailBackend" from django-celery-email.
    # (https://github.com/pmclanahan/django-celery-email)
    openwisp2_email_backend: "djcelery_email.backends.CeleryEmailBackend"
```

```
    # Email timeout in seconds used by Django for blocking operations
    # like connection attempts. For more info read the Django documentation,
    # https://docs.djangoproject.com/en/4.2/ref/settings/#email-timeout.
    # Defaults to 10 seconds.
    openwisp2_email_timeout: 5
    # edit database settings only if you are not using sqlite
    # eg, for deploying with PostgreSQL (recommended for production usage)
    # you will need the PostGIS spatial extension, find more info at:
    # https://docs.djangoproject.com/en/4.1/ref/contrib/gis/tutorial/
    openwisp2_database:
        engine: django.contrib.gis.db.backends.postgis
        name: "{{ DB_NAME }}"
        user: "{{ DB_USER }}"
        host: "{{ DB_HOST }}"
        password: "{{ DB_PASSWORD }}"
        port: 5432
    # SPATIALITE_LIBRARY_PATH django setting
    # The role will attempt determining the right mod-spatialite path automatically
    # But you can use this variable to customize the path or fix future arising issues
    openwisp2_spatialite_path: "mod_spatialite.so"
    # customize other django settings:
    openwisp2_language_code: en-gb
    openwisp2_time_zone: UTC
    # openwisp-controller context
    openwisp2_context: {}
    # additional allowed hosts
    openwisp2_allowed_hosts:
        - myadditionalhost.openwisp.org
    # geographic map settings
    openwisp2_leaflet_config:
        DEFAULT_CENTER: [42.06775, 12.62011]
        DEFAULT_ZOOM: 6
    # enable/disable geocoding check
    openwisp2_geocoding_check: true
    # specify path to a valid SSL certificate and key
    # (a self-signed SSL cert will be generated if omitted)
    openwisp2_ssl_cert: "/etc/nginx/ssl/server.crt"
    openwisp2_ssl_key: "/etc/nginx/ssl/server.key"
    # customize the self-signed SSL certificate info if needed
    openwisp2_ssl_country: "US"
    openwisp2_ssl_state: "California"
    openwisp2_ssl_locality: "San Francisco"
    openwisp2_ssl_organization: "IT dep."
    # the following setting controls which ip address range
    # is allowed to access the controller via unencrypted HTTP
    # (this feature is disabled by default)
    openwisp2_http_allowed_ip: "10.8.0.0/16"
    # additional python packages that will be installed with pip
    openwisp2_extra_python_packages:
        - bpython
        - django-owm-legacy
    # additional django apps that will be added to settings.INSTALLED_APPS
    # (if the app needs to be installed, the name its python package
    # must be also added to the openwisp2_extra_python_packages var)
    openwisp2_extra_django_apps:
        - owm_legacy
    # additional django settings example
    openwisp2_extra_django_settings:
        CSRF_COOKIE_AGE: 2620800.0
    # in case you need to add python instructions to the django settings file
```

```
openwisp2_extra_django_settings_instructions:
    - TEMPLATES[0]['OPTIONS']['loaders'].insert(0, 'apptemplates.Loader')
# extra URL settings for django
openwisp2_extra_urls:
  - "path(r'', include('my_custom_app.urls'))"
# allows to specify imports that are used in the websocket routes, e.g.:
openwisp2_websocket_extra_imports:
  - from my_custom_app.websockets.routing import get_routes as get_custom_app_routes
# allows to specify extra websocket routes, e.g.:
openwisp2_websocket_extra_routes:
  # Callable that returns a list of routes
  - get_custom_app_routes()
  # List of routes
  - "[path('ws/custom-app/', consumer.CustomAppConsumer.as_asgi())]"
# controller URL are enabled by default
# but can be disabled in multi-VM installations if needed
openwisp2_controller_urls: true
# The default retention policy that applies to the timeseries data
# https://github.com/openwisp/openwisp-monitoring#openwisp-monitoring-default-retention-
openwisp2_monitoring_default_retention_policy: "26280h0m0s" # 3 years
# whether NGINX should be installed
openwisp2_nginx_install: true
# spdy protocol support (disabled by default)
openwisp2_nginx_spdy: false
# HTTP2 protocol support (disabled by default)
openwisp2_nginx_http2: false
# ipv6 must be enabled explicitly to avoid errors
openwisp2_nginx_ipv6: false
# nginx client_max_body_size setting
openwisp2_nginx_client_max_body_size: 10M
# list of upstream servers for OpenWISP
openwisp2_nginx_openwisp_server:
  - "localhost:8000"
# dictionary containing more nginx settings for
# the 443 section of the openwisp2 nginx configuration
# IMPORTANT: 1. you can add more nginx settings in this dictionary
#            2. here we list the default values used
openwisp2_nginx_ssl_config:
    gzip: "on"
    gzip_comp_level: "6"
    gzip_proxied: "any"
    gzip_min_length: "1000"
    gzip_types:
        - "text/plain"
        - "text/html"
        - "image/svg+xml"
        - "application/json"
        - "application/javascript"
        - "text/xml"
        - "text/css"
        - "application/xml"
        - "application/x-font-ttf"
        - "font/opentype"
# nginx error log configuration
openwisp2_nginx_access_log: "{{ openwisp2_path }}/log/nginx.access.log"
openwisp2_nginx_error_log: "{{ openwisp2_path }}/log/nginx.error.log error"
# nginx Content Security Policy header, customize if needed
openwisp2_nginx_csp: >
  CUSTOM_NGINX_SECURITY_POLICY
# uwsgi gid, omitted by default
```

```
openwisp2_uwsgi_gid: null
# number of uWSGI process to spawn. Default value is 1.
openwisp2_uwsgi_processes: 1
# number of threads each uWSGI process will have. Default value is 1.
openwisp2_uwsgi_threads: 2
# value of the listen queue of uWSGI
openwisp2_uwsgi_listen: 100
# socket on which uwsgi should listen. Defaults to UNIX socket
# at "{{ openwisp2_path }}/uwsgi.sock"
openwisp2_uwsgi_socket: 127.0.0.1:8000
# extra uwsgi configuration parameters that cannot be
# configured using dedicated ansible variables
openwisp2_uwsgi_extra_conf: |
  single-interpreter=True
  log-4xx=True
  log-5xx=True
  disable-logging=True
  auto-procname=True
# whether daphne should be installed
# must be enabled for serving websocket requests
openwisp2_daphne_install: true
# number of daphne process to spawn. Default value is 1
openwisp2_daphne_processes: 2
# maximum time to allow a websocket to be connected (in seconds)
openwisp2_daphne_websocket_timeout: 1800
# the following setting controls which ip address range
# is allowed to access the openwisp2 admin web interface
# (by default any IP is allowed)
openwisp2_admin_allowed_network: null
# install ntp client (enabled by default)
openwisp2_install_ntp: true
# if you have any custom supervisor service, you can
# configure it to restart along with other supervisor services
openwisp2_extra_supervisor_restart:
    - name: my_custom_service
      when: my_custom_service_enabled
# Disable usage metric collection. It is enabled by default.
# Read more about it at
# https://openwisp.io/docs/user/usage-metric-collection.html
openwisp2_usage_metric_collection: false
# enable sentry example
openwisp2_sentry:
    dsn: "https://7d2e3cd61acc32eca1fb2a390f7b55e1:bf82aab5ddn4422688e34a486c7426e3@gets
openwisp2_default_cert_validity: 1825
openwisp2_default_ca_validity: 3650
# the following options for redis allow to configure an external redis instance if neede
openwisp2_redis_install: true
openwisp2_redis_host: localhost
openwisp2_redis_port: 6379
openwisp2_redis_cache_url: "redis://{{ openwisp2_redis_host }}:{{ openwisp2_redis_port }}
# the following options are required to configure influxdb which is used in openwisp-mon
openwisp2_influxdb_install: true
openwisp2_timeseries_database:
    backend: "openwisp_monitoring.db.backends.influxdb"
    user: "openwisp"
    password: "openwisp"
    name: "openwisp2"
    host: "localhost"
    port: 8086
# celery concurrency for the default queue, by default the number of CPUs is used
```

```
# celery concurrency for the default queue, by default it is set to 1
# Setting it to "null" will make concurrency equal to number of CPUs if autoscaling is n
openwisp2_celery_concurrency: null
# alternative to the previous option, the celery autoscale option can be set if needed
# for more info, consult the documentation of celery regarding "autoscaling"
# by default it is set to "null" (no autoscaling)
openwisp2_celery_autoscale: 4,1
# prefetch multiplier for the default queue,
# the default value is calculated automatically by celery
openwisp2_celery_prefetch_multiplier: null
# celery queuing mode for the default queue,
# leaving the default will work for most cases
openwisp2_celery_optimization: default
# whether the dedicated celerybeat worker is enabled which is
# responsible for triggering periodic tasks
# must be turned on unless there's another server running celerybeat
openwisp2_celerybeat: true
# whether the dedicated worker for the celery "network" queue is enabled
# must be turned on unless there's another server running a worker for this queue
openwisp2_celery_network: true
# concurrency option for the "network" queue (a worker is dedicated solely to network op
# the default is 1. Setting it to "null" will make concurrency equal to number of CPUs i
openwisp2_celery_network_concurrency: null
# alternative to the previous option, the celery autoscale option can be set if needed
# for more info, consult the documentation of celery regarding "autoscaling"
# by default it is set to "null" (no autoscaling)
openwisp2_celery_network_autoscale: 8,4
# prefetch multiplier for the "network" queue,
# the default is 1, which mean no prefetching,
# because the network tasks are long running and is better
# to distribute the tasks to multiple processes
openwisp2_celery_network_prefetch_multiplier: 1
# celery queuing mode for the "network" queue,
# fair mode is used in this case, which means
# tasks will be equally distributed among workers
openwisp2_celery_network_optimization: fair
# whether the dedicated worker for the celery "firmware_upgrader" queue is enabled
# must be turned on unless there's another server running a worker for this queue
openwisp2_celery_firmware_upgrader: true
# concurrency option for the "firmware_upgrader" queue (a worker is dedicated solely to
# the default is 1. Setting it to "null" will make concurrency equal to number of CPUs i
openwisp2_celery_firmware_upgrader_concurrency: null
# alternative to the previous option, the celery autoscale option can be set if needed
# for more info, consult the documentation of celery regarding "autoscaling"
# by default it is set to "null" (no autoscaling)
openwisp2_celery_firmware_upgrader_autoscale: 8,4
# prefetch multiplier for the "firmware_upgrader" queue,
# the default is 1, which mean no prefetching,
# because the firmware upgrade tasks are long running and is better
# to distribute the tasks to multiple processes
openwisp2_celery_firmware_upgrader_prefetch_multiplier: 1
# celery queuing mode for the "firmware_upgrader" queue,
# fair mode is used in this case, which means
# tasks will be equally distributed among workers
openwisp2_celery_firmware_upgrader_optimization: fair
# whether the dedicated worker for the celery "monitoring" queue is enabled
# must be turned on unless there's another server running a worker for this queue
openwisp2_celery_monitoring: true
# concurrency option for the "monitoring" queue (a worker is dedicated solely to monitor
# the default is 2. Setting it to "null" will make concurrency equal to number of CPUs
```

```
    # if autoscaling is not used.
    openwisp2_celery_monitoring_concurrency: null
    # alternative to the previous option, the celery autoscale option can be set if needed
    # for more info, consult the documentation of celery regarding "autoscaling"
    # by default it is set to "null" (no autoscaling)
    openwisp2_celery_monitoring_autoscale: 4,8
    # prefetch multiplier for the "monitoring" queue,
    # the default is 1, which mean no prefetching,
    # because the monitoring tasks can be long running and is better
    # to distribute the tasks to multiple processes
    openwisp2_celery_monitoring_prefetch_multiplier: 1
    # celery queuing mode for the "monitoring" queue,
    # fair mode is used in this case, which means
    # tasks will be equally distributed among workers
    openwisp2_celery_monitoring_optimization: fair
    # whether the default celery task routes should be written to the settings.py file
    # turn this off if you're defining custom task routing rules
    openwisp2_celery_task_routes_defaults: true
    # celery settings
    openwisp2_celery_broker_url: redis://{{ openwisp2_redis_host }}:{{ openwisp2_redis_port
    openwisp2_celery_task_acks_late: true
    # maximum number of retries by celery before giving up when broker is unreachable
    openwisp2_celery_broker_max_tries: 10
    # whether to activate the django logging configuration in celery
    # if set to true, will log all the celery events in the same log stream used by django
    # which will cause log lines to be written to "{{ openwisp2_path }}/log/openwisp2.log"
    # instead of "{{ openwisp2_path }}/log/celery.log" and "{{ openwisp2_path }}/log/celeryb
    openwisp2_django_celery_logging: false
    # postfix is installed by default, set to false if you don't need it
    openwisp2_postfix_install: true
    # allow overriding default `postfix_smtp_sasl_auth_enable` variable
    postfix_smtp_sasl_auth_enable_override: true
    # allow overriding postfix_smtpd_relay_restrictions
    postfix_smtpd_relay_restrictions_override: permit_mynetworks
    # allows overriding the default duration for keeping notifications
    openwisp2_notifications_delete_old_notifications: 10
    # Expiration time limit (in seconds) of magic sign-in links.
    # Magic sign-in links are used only when OpenWISP RADIUS is enabled.
    openwisp2_django_sesame_max_age: 1800 # 30 minutes
    # Maximum file size(in bytes) allowed to be uploaded as firmware image.
    # It overrides "openwisp2_nginx_client_max_body_size" setting
    # and updates nginx configuration accordingly.
    openwisp2_firmware_upgrader_max_file_size: 41943040 # 40MB
    # to add multi-language support
    openwisp2_internationalization: true
    openwisp2_users_auth_api: true
    # Allows setting OPENWISP_USERS_USER_PASSWORD_EXPIRATION setting.
    # Read https://github.com/openwisp/openwisp-users#openwisp_users_user_password_expiratio
    openwisp2_users_user_password_expiration: 30
      # Allows setting OPENWISP_USERS_STAFF_USER_PASSWORD_EXPIRATION setting.
    # Read https://github.com/openwisp/openwisp-users#openwisp_users_staff_user_password_exp
    openwisp2_users_staff_user_password_expiration: 30
    # used for SMS verification, the default is a dummy SMS backend
    # which prints to standard output and hence does nothing
    # one of the available providers from django-sendsms can be
    # used or alternatively, you can write a backend class for your
    # favorite SMS API gateway
    openwisp2_radius_sms_backend: "sendsms.backends.console.SmsBackend"
    openwisp2_radius_sms_token_max_ip_daily: 25
    openwisp2_radius_delete_old_radiusbatch_users: 365
```

```
openwisp2_radius_cleanup_stale_radacct: 1
openwisp2_radius_delete_old_postauth: 365
# days for which the radius accounting sessions (radacct) are retained,
# 0 means sessions are kept forever.
# we highly suggest to set this number according
# to the privacy regulation of your jurisdiction
openwisp2_radius_delete_old_radacct: 365
# days after which inactive users will flagged as unverified
# Read https://openwisp.io/docs/stable/radius/user/settings.html#openwisp-radius-unverif
openwisp2_radius_unverify_inactive_users: 540
# days after which inactive users will be deleted
# Read Read https://openwisp.io/docs/stable/radius/user/settings.html#openwisp-radius-de
openwisp2_radius_delete_inactive_users: 540
openwisp2_radius_allowed_hosts: ["127.0.0.1"]
# allow disabling celery beat tasks if needed
openwisp2_monitoring_periodic_tasks: true
openwisp2_radius_periodic_tasks: true
openwisp2_usage_metric_collection_periodic_tasks: true
# this role provides a default configuration of freeradius
# if you manage freeradius on a different machine or you need different configurations
# you can disable this default behavior
openwisp2_freeradius_install: true
# Set an account to expire T seconds after first login.
# This variable sets the value of T.
freeradius_expire_attr_after_seconds: 86400
freeradius_dir: /etc/freeradius/3.0
freeradius_mods_available_dir: "{{ freeradius_dir }}/mods-available"
freeradius_mods_enabled_dir: "{{ freeradius_dir }}/mods-enabled"
freeradius_sites_available_dir: "{{ freeradius_dir }}/sites-available"
freeradius_sites_enabled_dir: "{{ freeradius_dir }}/sites-enabled"
freeradius_rest:
    url: "https://{{ inventory_hostname }}/api/v1/freeradius"
freeradius_safe_characters: "+@abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234
# Sets the source path of the template that contains freeradius site configuration.
# Defaults to "templates/freeradius/openwisp_site.j2" shipped in the role.
freeradius_openwisp_site_template_src: custom_freeradius_site.j2
# Whether to deploy the default openwisp_site for FreeRADIUS.
# Defaults to true.
freeradius_deploy_openwisp_site: false
# FreeRADIUS listen address for the openwisp_site.
# Defaults to "*", i.e. listen on all interfaces.
freeradius_openwisp_site_listen_ipaddr: "10.8.0.1"
# A list of dict that includes organization's name, UUID, RADIUS token,
# TLS configuration, and ports for authentication, accounting, and inner tunnel.
# This list of dict is used to generate FreeRADIUS sites that support
# WPA Enterprise (EAP-TTLS-PAP) authentication.
# Defaults to an empty list.
freeradius_eap_orgs:
    # The name should not contain spaces or special characters
  - name: openwisp
    # UUID of the organization can be retrieved from the OpenWISP admin
    uuid: 00000000-0000-0000-0000-000000000000
    # Radius token of the organization can be retrieved from the OpenWISP admin
    radius_token: secret-radius-token
    # Port used by the authentication service for this FreeRADIUS site
    auth_port: 1832
    # Port used by the accounting service for this FreeRADIUS site
    acct_port: 1833
    # Port used by the authentication service of inner tunnel for this FreeRADIUS site
    inner_tunnel_auth_port: 18330
```

```
    # CA certificate for the FreeRADIUS site
    ca: /etc/freeradius/certs/ca.crt
    # TLS certificate for the FreeRADIUS site
    cert: /etc/freeradius/certs/cert.pem
    # TLS private key for the FreeRADIUS site
    private_key: /etc/freeradius/certs/key.pem
    # Diffie-Hellman key for the FreeRADIUS site
    dh: /etc/freeradius/certs/dh
    # Extra instructions for the "tls-config" section of the EAP module
    # for the FreeRADIUS site
    tls_config_extra: |
      private_key_password = whatever
      ecdh_curve = "prime256v1"
# Sets the source path of the template that contains freeradius site configuration
# for WPA Enterprise (EAP-TTLS-PAP) authentication.
# Defaults to "templates/freeradius/eap/openwisp_site.j2" shipped in the role.
freeradius_eap_openwisp_site_template_src: custom_eap_openwisp_site.j2
# Sets the source path of the template that contains freeradius inner tunnel
# configuration for WPA Enterprise (EAP-TTLS-PAP) authentication.
# Defaults to "templates/freeradius/eap/inner_tunnel.j2" shipped in the role.
freeradius_eap_inner_tunnel_template_src: custom_eap_inner_tunnel.j2
# Sets the source path of the template that contains freeradius EAP configuration
# for WPA Enterprise (EAP-TTLS-PAP) authentication.
# Defaults to "templates/freeradius/eap/eap.j2" shipped in the role.
freeradius_eap_template_src: custom_eap.j2
cron_delete_old_notifications: "'hour': 0, 'minute': 0"
cron_deactivate_expired_users: "'hour': 0, 'minute': 5"
cron_delete_old_radiusbatch_users: "'hour': 0, 'minute': 10"
cron_cleanup_stale_radacct: "'hour': 0, 'minute': 20"
cron_delete_old_postauth: "'hour': 0, 'minute': 30"
cron_delete_old_radacct: "'hour': 1, 'minute': 30"
cron_password_expiration_email: "'hour': 1, 'minute': 0"
cron_unverify_inactive_users: "'hour': 1, 'minute': 45"
cron_delete_inactive_users: "'hour': 1, 'minute': 55"
# cross-origin resource sharing (CORS) settings
# https://pypi.org/project/django-cors-headers/
openwisp2_django_cors:
  # Setting this to "true" will install the django-cors-headers package
  # and configure the Django middleware setting to support CORS.
  # By default, it is set to false.
  enabled: true
  # Configures "CORS_ALLOWED_ORIGINS" setting of the django-cors-headers
  # package. A list of origins that are authorized to make cross-site
  # HTTP requests. Read https://github.com/adamchainz/django-cors-headers#cors_allowed_o
  # for detail. By default, it is set to an empty list.
  allowed_origins_list: ["https://log.openwisp.org"]
```

Note

The default settings for controlling the number of processes and threads in uWSGI and Daphne are set conservatively. Users are encouraged to adjust these settings to match the scale of their project. The same applies to the concurrency and auto-scaling settings for Celery workers.

Developer Installation instructions

Note

This page is for developers who want to customize or extend the Ansible role of OpenWISP, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- Ansible OpenWISP2 User Docs

Installing for Development

First of all, create the directory where you want to place the repositories of the ansible roles and create directory roles.

```
mkdir -p ~/openwisp-dev/roles
cd ~/openwisp-dev/roles
```

Clone `ansible-openwisp2` and `Stouts.postfix` as follows:

```
git clone https://github.com/openwisp/ansible-openwisp2.git openwisp.openwisp2
git clone https://github.com/Stouts/Stouts.postfix
git clone https://github.com/openwisp/ansible-ow-influxdb openwisp.influxdb
```

Now, go to the parent directory & create hosts file and `playbook.yml`:

```
cd ../
touch hosts
touch playbook.yml
```

From here on you can follow the instructions available at the following sections:

- Install Ansible
- Create Inventory File
- Create Playbook File
- Run the Playbook

All done!

How to Run Tests

If you want to contribute to `ansible-openwisp2` you should run tests in your development environment to ensure your changes are not breaking anything.

To do that, proceed with the following steps:

Step 1: Clone `ansible-openwisp2`

Clone repository by:

```
git clone https://github.com/<your_fork>/ansible-openwisp2.git openwisp.openwisp2
cd openwisp.openwisp2
```

Step 2: Install docker

If you haven't installed docker yet, you need to install it (example for linux debian/ubuntu systems):

```
sudo apt install docker.io
```

Step 3: Install molecule and dependencies

```
pip install molecule[docker] molecule-plugins yamllint ansible-lint docker
```

Step 4: Download docker images

```
docker pull geerlingguy/docker-ubuntu2204-ansible:latest
docker pull geerlingguy/docker-ubuntu2004-ansible:latest
docker pull geerlingguy/docker-debian11-ansible:latest
```

Step 5: Run molecule test

```
molecule test -s local
```

If you don't get any error message it means that the tests ran successfully without errors.

Tip

Use `molecule test --destroy=never` to speed up subsequent test runs.

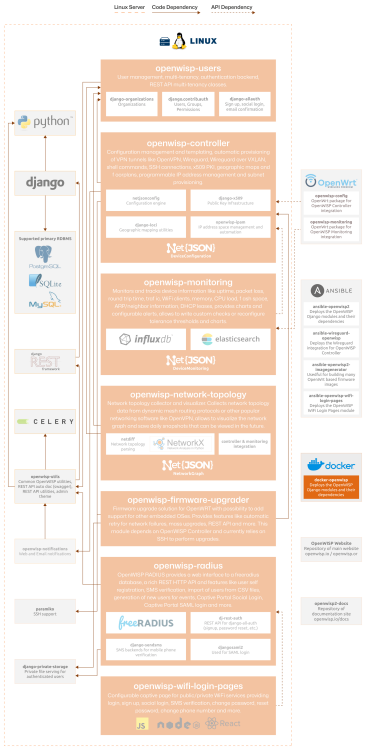
Docker OpenWISP

Seealso

Source code: github.com/openwisp/docker-openwisp.

Docker-OpenWISP makes it possible to set up isolated and reproducible OpenWISP environments, simplifying the deployment and scaling process.

The following diagram illustrates the role of Docker OpenWISP within the OpenWISP architecture.



OpenWISP Architecture: highlighted Docker OpenWISP

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

Quick Start Guide

This page explains how to deploy OpenWISP using the docker images provided by Docker OpenWISP.

Available Images

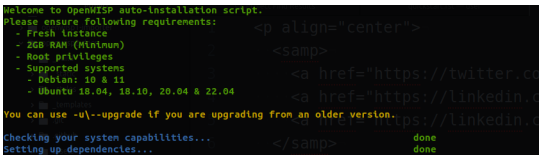
The images are hosted on Docker Hub and GitLab Container Registry.

Image Tags

All images are tagged using the following convention:

Tag	Software Version
latest	This is the most recent official release of OpenWISP. On Github, this corresponds to the latest tagged release.
edge	This is the development version of OpenWISP. On Github, this corresponds to the current master branch.

Auto Install Script



The auto-install script can be used to quickly install an OpenWISP instance on your server.

It will install the required system dependencies and start the docker containers.

This script prompts the user for basic configuration parameters required to set up OpenWISP. Below are the prompts and their descriptions:

- **OpenWISP Version:** Version of OpenWISP you want to install. If you leave this blank, the latest released version will be installed.
- **.env File Path:** Path to an existing ".env" file file if you have one. If you leave this blank, the script will continue prompting for additional configuration.
- **Domains:** The fully qualified domain names for the Dashboard, API, and OpenVPN services.
- **Site Manager Email:** Email address of the site manager. This email address will serve as the default sender address for all email communications from OpenWISP.

- **Let's Encrypt Email:** Email address for Let's Encrypt to use for certificate generation. If you leave this blank, a self-signed certificate will be generated.

Important

The Docker OpenWISP installation responds only to the fully qualified domain names (FQDN) defined in the configuration. If you are deploying locally (for testing), you need to update the `/etc/hosts` file on your machine to resolve the configured domains to localhost.

For example, the following command will update the `/etc/hosts` file to resolve the domains used in the default configurations:

```
echo "127.0.0.1 dashboard.openwisp.org api.openwisp.org openvpn.openwisp.org" | \
    sudo tee -a /etc/hosts
```

Run the following commands to download the auto-install script and execute it:

```
curl https://raw.githubusercontent.com/openwisp/docker-openwisp/master/deploy/auto-install.s
sudo bash auto-install.sh
```

The auto-install script maintains a log, which is useful for debugging or checking the real-time output of the script. You can view the log by running the following command:

```
tail -n 50 -f /opt/openwisp/autoinstall.log
```

The auto-install script can be used to upgrade installations that were originally deployed using this script. You can upgrade your installation by using the following command

```
sudo bash auto-install.sh --upgrade
```

Note

- If you're having any installation issues with the `latest` version, you can try auto-installation with the `edge` version, which ships the development version of OpenWISP.
- Still facing errors while installation? Please read the FAQ.

Using Docker Compose

This setup is suitable for single-server setup requirements. It is quicker and requires less prior knowledge about OpenWISP & networking.

1. Install requirements:

   ```
   sudo apt -y update
   sudo apt -y install git docker.io make
   # Please ensure docker is installed properly and the following
   # command show system information. In most machines, you'll need to
   # add your user to the `docker` group and re-login to the shell.
   docker info
   ```

2. Setup repository:

   ```
   git clone https://github.com/openwisp/docker-openwisp.git
   cd docker-openwisp
   ```

3. Configure:

   Please refer to the Settings and Advanced Customization pages to configure any aspect of your OpenWISP instance.

Make sure to change the values for essential and security variables.

4. Deploy:

   Use the `make start` command to pull images and start the containers.

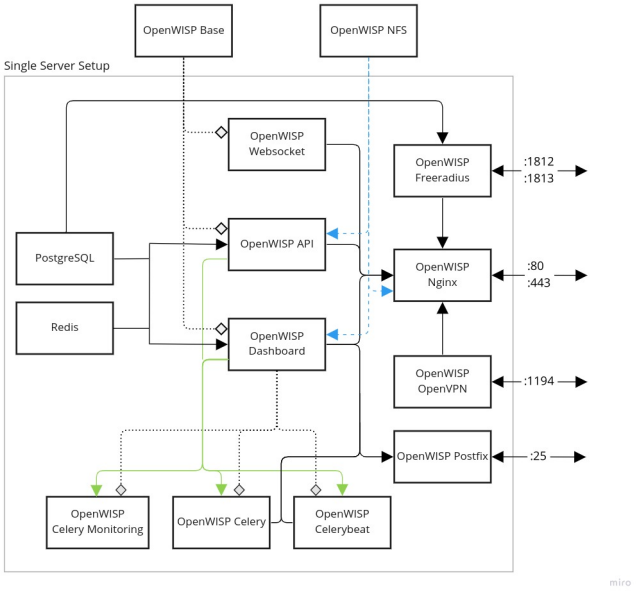
> **Note**
>
> If you want to shutdown services for maintenance or any other purposes, please use `make stop`.

If you are facing errors during the installation process, read the FAQ for known issues.

Architecture

A typical OpenWISP installation is made of multiple components (e.g. application servers, background workers, web servers, database, messaging queue, VPN server, etc.) that have different scaling requirements.

The aim of Docker OpenWISP is to allow deploying OpenWISP in cloud based environments which allow potentially infinite horizontal scaling. That is the reason for which there are different docker images shipped in this repository.



Architecture

- **openwisp-dashboard**: Your OpenWISP device administration dashboard.
- **openwisp-api**: HTTP API from various OpenWISP modules which can be scaled simply by having multiple API containers as per requirement.
- **openwisp-websocket**: Dedicated container for handling websocket requests, e.g. for updating location of mobile network devices.
- **openwisp-celery**: Runs all the background tasks for OpenWISP, e.g. updating configurations of your device.
- **openwisp-celery-monitoring**: Runs background tasks that perform active monitoring checks, e.g. ping checks and configuration checks. It also executes task for writing monitoring data to the timeseries DB.
- **openwisp-celerybeat**: Runs periodic background tasks. e.g. revoking all the expired certificates.
- **openwisp-nginx**: Internet facing container that facilitates all the HTTP and Websocket communication between the outside world and the service containers.
- **openwisp-freeradius**: Freeradius container for OpenWISP.
- **openwisp-openvpn**: OpenVPN container for out-of-the-box management VPN.

- **openwisp-postfix**: Mail server for sending mails to MTA.
- **openwisp-nfs**: NFS server that allows shared storage between different machines. It does not run in single server machines but provided for K8s setup.
- **openwisp-base**: It is the base image which does not run on your server, but openwisp-api & openwisp-dashboard use it as a base.
- **Redis**: data caching service (required for actions like login).
- **PostgreSQL**: SQL database container for OpenWISP.

Settings

The OpenWISP Docker images are designed for customization. You can easily modify environment variables to tailor the containers to your needs.

- **Docker Compose:** Simply change the values in the `.env` file.

Below are listed the available configuration options divided by section:

Additionally, you can search for the following prefixes:

- `OPENWISP_`: OpenWISP application settings.
- `DB_`: PostgreSQL Database settings.
- `INFLUXDB_`: InfluxDB settings.
- `DJANGO_`: Django settings.
- `EMAIL_`: Email settings (see also `POSTFIX_`).
- `POSTFIX_`: Postfix settings (see also `EMAIL_`).
- `NGINX_`: Nginx web server settings.
- `UWSGI_`: uWSGI application server settings.
- `DASHBOARD_`: Settings specific to the OpenWISP dashboard.
- `API_`: Settings specific to the OpenWISP API.
- `X509_`: Configurations related to x509 CA and certificates.
- `VPN_`: Default VPN and VPN template configurations.
- `CRON_`: Periodic task configurations.
- `EXPORT_`: NFS server configurations.

Essential

You will need to adapt these values to get the docker images working properly on your system.

`DASHBOARD_DOMAIN`

- **Explanation:** Domain on which you want to access OpenWISP dashboard.
- **Valid Values:** Any valid domain.
- **Default:** `dashboard.example.com`.

`API_DOMAIN`

- **Explanation:** Domain on which you want to access OpenWISP APIs.
- **Valid Values:** Any valid domain.
- **Default:** `api.example.com`.

`VPN_DOMAIN`

- **Explanation:** Valid domain / IP address to reach the OpenVPN server.
- **Valid Values:** Any valid domain or IP address.
- **Default:** `openvpn.example.com`.

`TZ`

- **Explanation:** Sets the timezone for the OpenWISP containers.
- **Valid Values:** Find list of timezone database here.
- **Default:** `UTC`.

`CERT_ADMIN_EMAIL`

- **Explanation:** Required by certbot. Email used for registration and recovery contact.
- **Valid Values:** A comma separated list of valid email addresses.
- **Default:** `example@example.com`.

`SSL_CERT_MODE`

- **Explanation:** Flag to enable or disable HTTPs. If it is set to `Yes`, letsencrypt certificates are automatically fetched with the help of certbot and a cronjob to ensure they stay updated is added. If it is set to `SelfSigned`, self-signed certificates are used and cronjob for the certificates is set. If set to `No`, site is accessible via HTTP, if set if `EXTERNAL`, it tells HTTPs is used but managed by external tool like loadbalancer / provider. Setting this option as `No` is not recommended and might break some features, only do it when you know what you are doing.
- **Valid Values:** `External`, `Yes`, `SelfSigned`, `No`.
- **Default:** `Yes`.

Security

Tune these options to strengthen the security of your instance.

`DJANGO_SECRET_KEY`

- **Explanation:** A random unique string that must be kept secret for security reasons. You can generate it with the command: `python build.py get-secret-key` at the root of the repository to get a key or make a random key yourself.
- **Valid Values:** STRING.
- **Default:** `default_secret_key`

`DJANGO_ALLOWED_HOSTS`

- **Explanation:** Used to validate a request's HTTP Host header. The default value `*` allows all domains. For security, it is recommended to specify only trusted domains, such as `.mydomain.com`. If left blank, it defaults to your dashboard's root domain.
- **Valid Values:** Refer to the Django documentation for ALLOWED_HOSTS.
- **Default:** Root domain extracted from DASHBOARD_DOMAIN.
- **Example:** `.openwisp.org,.example.org,www.example.com`.

`OPENWISP_RADIUS_FREERADIUS_ALLOWED_HOSTS`

- **Explanation:** Default IP address or subnet of your freeradius instance.
- **Valid Values:** A comma separated string of valid IP address or IP Networks.
- **Default:** `172.18.0.0/16`.
- **Example:** `127.0.0.1,192.0.2.20,172.18.0.0/16`.

OpenWISP

Settings for the OpenWISP application and the underlying Django web framework.

Note

Any OpenWISP Configuration of type `string`. `int`, `bool` or `json` is supported and can be used as per the documentation in the module.

If you need to change a Django setting that has a more complex datatype, please refer to Supplying Custom Django Settings.

`EMAIL_HOST`

- **Explanation:** Host to be used when connecting to the STMP. `localhost` or empty string are not allowed.
- **Valid Values:** A valid hostname or IP address.
- **Example:** `smtp.gmail.com`.
- **Default:** `postfix`.

`EMAIL_DJANGO_DEFAULT`

- **Explanation:** It is the email address to use for various automated correspondence from the site manager(s).
- **Valid Values:** Any valid email address.
- **Default:** `example@example.com`.

`EMAIL_HOST_PORT`

- **Explanation:** Port to use for the SMTP server defined in EMAIL_HOST.
- **Valid Values:** INTEGER.
- **Default:** `25`.

`EMAIL_HOST_USER`

- **Explanation:** Username to use for the SMTP server defined in EMAIL_HOST. If empty, Django won't attempt authentication.
- **Valid Values:** STRING.
- **Default:** `""` (empty string).
- **Example:** `example@example.com`

`EMAIL_HOST_PASSWORD`

- **Explanation:** Password to use for the SMTP server defined in EMAIL_HOST.. If empty, Django won't attempt authentication.
- **Valid Values:** STRING.
- **Default:** `""` (empty string)

`EMAIL_HOST_TLS`

- **Explanation:** Whether to use a TLS (secure) connection when talking to the SMTP server. This is used for explicit TLS connections, generally on port 587.
- **Valid Values:** `True`, `False`.
- **Default:** `False`.

`EMAIL_TIMEOUT`

- **Explanation:** Specifies a timeout in seconds used by Django for blocking operations like the connection attempt.
- **Valid Values:** INTEGER.
- **Default:** `10`.

`EMAIL_BACKEND`

- **Explanation:** Email will be sent using this backend.
- **Valid Values:** Refer to the "Email backends" section on the Django documentation.

- **Default:** `djcelery_email.backends.CeleryEmailBackend`.

```
DJANGO_X509_DEFAULT_CERT_VALIDITY
```

- **Explanation:** Validity of your x509 cert in days.
- **Valid Values:** INTEGER.
- **Default:** `1825`

```
DJANGO_X509_DEFAULT_CA_VALIDITY
```

- **Explanation:** Validity of your x509 CA in days.
- **Valid Values:** INTEGER.
- **Default:** `3650`.

```
DJANGO_CORS_HOSTS
```

- **Explanation:** Hosts for which CORS. is whitelisted.
- **Valid Values:** Comma separated list of CORS domains.
- **Default:** `http://localhost`
- **Example:** `https://www.openwisp.org,openwisp.example.org`

```
DJANGO_LANGUAGE_CODE
```

- **Explanation:** Language for your OpenWISP application.
- **Valid Values:** Refer to the related Django documentation section.
- **Default:** `en-gb`.

```
DJANGO_SENTRY_DSN
```

- **Explanation:** Sentry DSN.
- **Valid Values:** Your DSN value provided by sentry.
- **Example:** `https://example@sentry.io/example`.
- **Default:** `""` (empty string).

```
DJANGO_LEAFET_CENTER_X_AXIS
```

- **Explanation:** X-axis coordinate of the leaflet default center property. Refer to the django-leaflet docs for more information.
- **Valid Values:** FLOAT.
- **Example:** `26.357896`.
- **Default:** `0`.

```
DJANGO_LEAFET_CENTER_Y_AXIS
```

- **Explanation:** Y-axis coordinate of the leaflet default center property. Refer to the django-leaflet docs for more information.
- **Valid Values:** FLOAT.
- **Example:** `127.783809`.
- **Default:** `0`.

```
DJANGO_LEAFET_ZOOM
```

- **Explanation:** Default zoom for leaflet. Refer to the django-leaflet docs for more information.
- **Valid Values:** INT (1-16).
- **Default:** `1`.

```
DJANGO_WEBSOCKET_HOST
```

- **Explanation:** Host on which Daphne should listen for websocket connections.
- **Valid Values:** Any valid domain or IP Address.
- **Default:** `0.0.0.0`.

```
OPENWISP_GEOCODING_CHECK
```

- **Explanation:** Used to check if geocoding is working as expected or not.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

```
USE_OPENWISP_CELERY_TASK_ROUTES_DEFAULTS
```

- **Explanation:** Whether the default celery task routes should be used by celery. Turn this off if you're defining custom task routing rules.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

```
OPENWISP_CELERY_COMMAND_FLAGS
```

- **Explanation:** Additional flags passed to the command that starts the celery worker for the `default` queue. It can be used to configure different attributes of the celery worker (e.g. auto-scaling, concurrency, etc.). Refer to the celery worker documentation for more information on configurable properties.
- **Valid Values:** STRING.
- **Default:** `--concurrency=1`.

```
USE_OPENWISP_CELERY_NETWORK
```

- **Explanation:** Whether the dedicated worker for the celery "network" queue is enabled. Must be turned on unless there's another server running a worker for this queue.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

`OPENWISP_CELERY_NETWORK_COMMAND_FLAGS`

- **Explanation:** Additional flags passed to the command that starts the celery worker for the `network` queue. It can be used to configure different attributes of the celery worker (e.g. auto-scaling, concurrency, etc.). Refer to the celery worker documentation for more information on configurable properties.
- **Valid Values:** STRING.
- **Default:** `--concurrency=1`

`USE_OPENWISP_CELERY_FIRMWARE`

- **Explanation:** Whether the dedicated worker for the celery `firmware_upgrader` queue is enabled. Must be turned on unless there's another server running a worker for this queue.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

`OPENWISP_CELERY_FIRMWARE_COMMAND_FLAGS`

- **Explanation:** Additional flags passed to the command that starts the celery worker for the `firmware_upgrader` queue. It can be used to configure different attributes of the celery worker (e.g. auto-scaling, concurrency, etc.). Refer to the celery worker documentation for more information on configurable properties.
- **Valid Values:** STRING
- **Default:** `--concurrency=1`

`USE_OPENWISP_CELERY_MONITORING`

- **Explanation:** Whether the dedicated worker for the celery `monitoring` queue is enabled. Must be turned on unless there's another server running a worker for this queue.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

`OPENWISP_CELERY_MONITORING_COMMAND_FLAGS`

- **Explanation:** Additional flags passed to the command that starts the celery worker for the `monitoring` queue. It can be used to configure different attributes of the celery worker (e.g. auto-scaling, concurrency, etc.). Refer to the celery worker documentation for more information on configurable properties.
- **Valid Values:** STRING.
- **Default:** `--concurrency=1`.

`OPENWISP_CELERY_MONITORING_CHECKS_COMMAND_FLAGS`

- **Explanation:** Additional flags passed to the command that starts the celery worker for the `monitoring_checks` queue. It can be used to configure different attributes of the celery worker (e.g. auto-scaling, concurrency, etc.). Refer to the celery worker documentation for more information on configurable properties.
- **Valid Values:** STRING.
- **Default:** `--concurrency=1`.

`OPENWISP_CUSTOM_OPENWRT_IMAGES`

- **Explanation:** JSON representation of the related Firmware Upgrader setting.
- **Valid Values:** JSON
- **Default:** `None`
- **Example:** `[{"name": "Name1","label": "Label1","boards": ["TestA", "TestB"]}, {"name": "Name2","label": "Label2","boards": ["TestC", "TestD"]}]`

`METRIC_COLLECTION`

- **Explanation:** Whether Collection of Usage Metrics is enabled or not.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

`CRON_DELETE_OLD_RADACCT`

- **Explanation:** (Value in days) Deletes RADIUS accounting sessions older than given number of days.
- **Valid Values:** INTEGER.
- **Default:** `365`.

`CRON_DELETE_OLD_POSTAUTH`

- **Explanation:** (Value in days) Deletes RADIUS *post-auth* logs older than given number of days.
- **Valid Values:** INTEGER.
- **Default:** `365`.

`CRON_CLEANUP_STALE_RADACCT`

- **Explanation:** (Value in days) Closes stale RADIUS sessions that have remained open for the number of specified days.
- **Valid Values:** INTEGER.
- **Default:** `365`.

`CRON_DELETE_OLD_RADIUSBATCH_USERS`

- **Explanation:** (Value in days) Deactivates expired user accounts which were created temporarily and have an expiration date set.
- **Valid Values:** INTEGER.
- **Default:** `365`.

`DEBUG_MODE`

- **Explanation:** Enable Django Debugging. Refer to the related Django documentation section for details.
- **Valid Values:** `True`, `False`.
- **Default:** `False`.

DJANGO_LOG_LEVEL

- **Explanation:** Logging level for Django. Refer to the related Django documentation section for details.
- **Valid Values:** STRING.
- **Default:** `ERROR`.

Enabled OpenWISP Modules

These options allow to disable the optional OpenWISP modules.

USE_OPENWISP_TOPOLOGY

- **Explanation:** Whether the Network Topology module is enabled or not.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

USE_OPENWISP_RADIUS

- **Explanation:** Whether the RADIUS module is enabled or not.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

USE_OPENWISP_FIRMWARE

- **Explanation:** Whether the Firmware Upgrader module is enabled or not.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

USE_OPENWISP_MONITORING

- **Explanation:** Whether the Monitoring module is enabled or not.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

PostgreSQL Database

DB_NAME

- **Explanation:** The name of the database to use.
- **Valid Values:** STRING.
- **Default:** `openwisp_db`.

DB_USER

- **Explanation:** The username to use when connecting to the database.

- **Valid Values:** STRING.
- **Default:** `admin`.

`DB_PASS`

- **Explanation:** The password to use when connecting to the database.
- **Valid Values:** STRING.
- **Default:** `admin`.

`DB_HOST`

- **Explanation:** Host to be used when connecting to the database. `localhost` or empty string are not allowed.
- **Valid Values:** A hostname or an IP address.
- **Default:** `postgres`.

`DB_PORT`

- **Explanation:** The port to use when connecting to the database.
- **Valid Values:** INTEGER.
- **Default:** `5432`.

`DB_SSLMODE`

- **Explanation:** Postgresql SSLMode option.
- **Valid Values:** Consult the related PostgreSQL documentation.
- **Default:** `disable`.

`DB_SSLCERT`

- **Explanation:** Path inside container to a valid client certificate.
- **Valid Values:** STRING.
- **Default:** `None`.

`DB_SSLKEY`

- **Explanation:** Path inside container to valid client private key.
- **Valid Values:** STRING.
- **Default:** `None`.

`DB_SSLROOTCERT`

- **Explanation:** Path inside container to a valid server certificate for the database.
- **Valid Values:** STRING.
- **Default:** `None`.

`DB_OPTIONS`

- **Explanation:** Additional database options to connect to the database. These options must be supported by your DB_HOST.
- **Valid Values:** JSON.
- **Default:** `{}`.

`DB_ENGINE`

- **Explanation:** Django spatial database backend to use.
- **Valid Values:** Refer to Spatial Backends on the Django documentation.
- **Default:** `django.contrib.gis.db.backends.postgis`

InfluxDB

InfluxDB is the default time series database used by the Monitoring module.

`INFLUXDB_USER`

- **Explanation:** Username of InfluxDB user.
- **Valid Values:** STRING.
- **Default:** `admin`.

`INFLUXDB_PASS`

- **Explanation:** Password for InfluxDB user.
- **Valid Values:** STRING.
- **Default:** `admin`.

`INFLUXDB_NAME`

- **Explanation:** Name of InfluxDB database.
- **Valid Values:** STRING.
- **Default:** `openwisp`.

`INFLUXDB_HOST`

- **Explanation:** Host to be used when connecting to influxDB. Values as `localhost` or empty string are not allowed.
- **Valid Values:** any valid hostname or IP address.
- **Default:** `influxdb`.

`INFLUXDB_PORT`

- **Explanation:** Port on which InfluxDB is listening to.

- **Valid Values:** INTEGER.
- **Default:** `8086`.

`INFLUXDB_DEFAULT_RETENTION_POLICY`

- **Explanation:** The default retention policy that applies to the time series data.
- **Valid Values:** STRING.
- **Default:** `26280h0m0s` (3 years).

Postfix

> **Note**
>
> Keep in mind that Postfix is optional. You can avoid running the Postfix container if you already have an external SMTP server available.

`POSTFIX_ALLOWED_SENDER_DOMAINS`

- **Explanation:** Due to in-built spam protection in Postfix you will need to specify sender domains.
- **Valid Values:** Any valid domain name.
- **Default:** `example.org`.

`POSTFIX_MYHOSTNAME`

- **Explanation:** You may configure a specific hostname that the SMTP server will use to identify itself.
- **Valid Values:** STRING.
- **Default:** `example.org`.

`POSTFIX_DESTINATION`

- **Explanation:** Destinations of the postfix service.
- **Valid Values:** Any valid domain name.
- **Default:** `$mydomain, $myhostname`.

`POSTFIX_MESSAGE_SIZE_LIMIT`

- **Explanation:** By default, this limit is set to 0 (zero), which means unlimited. Why would you want to set this? Well, this is especially useful in relation with `RELAYHOST` setting.
- **Valid Values:** INTEGER.
- **Default:** `0`
- **Example:** `26214400`

`POSTFIX_MYNETWORKS`

- **Explanation:** Postfix is exposed only in `mynetworks` to prevent any issues with this postfix being inadvertently exposed on the internet.
- **Valid Values:** space separated IP Networks.
- **Default:** `127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128`.

`POSTFIX_RELAYHOST_TLS_LEVEL`

- **Explanation:** Define relay host TLS connection level.
- **Valid Values:** See list.
- **Default:** `may`.

`POSTFIX_RELAYHOST`

- **Explanation:** Host that relays your mails.
- **Valid Values:** any valid IP address or domain name.
- **Default:** `null`.
- **Example:** `[smtp.gmail.com]:587`.

`POSTFIX_RELAYHOST_USERNAME`

- **Explanation:** Username for the relay server.
- **Valid Values:** STRING.
- **Default:** `null`.
- **Example:** `example@example.com`.

`POSTFIX_RELAYHOST_PASSWORD`

- **Explanation:** Login password for the relay server.
- **Valid Values:** STRING.
- **Default:** `null`.
- **Example:** `example`.

`POSTFIX_DEBUG_MYNETWORKS`

- **Explanation:** Set debug_peer_list for given list of networks.
- **Valid Values:** STRING.
- **Default:** `null`.
- **Example:** `127.0.0.0/8`.

uWSGI

`UWSGI_PROCESSES`

- **Explanation:** Number of uWSGI process to spawn.
- **Valid Values:** INTEGER.
- **Default:** 2.

`UWSGI_THREADS`

- **Explanation:** Number of threads each uWSGI process will have.
- **Valid Values:** INTEGER.
- **Default:** 2.

`UWSGI_LISTEN`

- **Explanation:** Value of the listen queue of uWSGI.
- **Valid Values:** INTEGER.
- **Default:** `100`.

Nginx

`NGINX_HTTP2`

- **Explanation:** Used by nginx to enable http2. Refer to the related Nginx documentation section for details.
- **Valid Values:** `http2` or empty string.
- **Default:** `http2`.

`NGINX_CLIENT_BODY_SIZE`

- **Explanation:** Client body size. Refer to the related Nginx documentation section for details.
- **Valid Values:** INTEGER.
- **Default:** `30`.

`NGINX_IP6_STRING`

- **Explanation:** Nginx listen on IPv6 for SSL connection. You can either enter a valid nginx statement or leave this value empty.
- **Valid Values:** `listen [::]:443 ssl http2;` or empty string.
- **Default:** `""` (empty string).

`NGINX_IP6_80_STRING`

- **Explanation:** Nginx listen on IPv6 connection. You can either enter a valid nginx statement or leave this value empty.

- **Valid Values:** `listen [::]:80;` or empty string.
- **Default:** `""` (empty string).

`NGINX_ADMIN_ALLOW_NETWORK`

- **Explanation:** IP address allowed to access OpenWISP services.
- **Valid Values:** `all`, IP network.
- **Example:** `12.213.43.54/16`.
- **Default:** `all`.

`NGINX_SERVER_NAME_HASH_BUCKET`

- **Explanation:** Define the Nginx domain hash bucket size. Values should be only in powers of 2.
- **Valid Values:** INTEGER.
- **Default:** 32.

`NGINX_SSL_CONFIG`

- **Explanation:** Additional nginx configurations. You can add any valid server block element here. As an example `index` option is configured. You may add options to this string or leave this variable blank. This variable is only applicable when `SSL_CERT_MODE` is `Yes` or `SelfSigned`.
- **Example:** `index index.html index.htm;`.
- **Default:** `""` (empty string).

`NGINX_80_CONFIG`

- **Explanation:** Additional nginx configurations. You can add any valid server block element here. As an example `index` option is configured. You may add options to this string or leave this variable blank. This variable is only applicable when `SSL_CERT_MODE` is `False`.
- **Example:** `index index.html index.htm;`.
- **Default:** `""` (empty string).

`NGINX_GZIP_SWITCH`

- **Explanation:** Turn on/off Nginx GZIP.
- **Valid Values:** `on`, `off`.
- **Default:** on.

`NGINX_GZIP_LEVEL`

- **Explanation:** Sets a gzip compression level of a response. Acceptable values are in the range from 1 to 9.
- **Valid Values:** `INTEGER`.
- **Default:** 6.

```
NGINX_GZIP_PROXIED
```

- **Explanation:** Enables or disables gzipping of responses for proxied requests depending on the request and response.
- **Valid Values:** `off`, `expired`, `no-cache`, `no-store | private`, `no_last_modified`, `no_etag`, `auth`, `any`.
- **Default:** `any`.

```
NGINX_GZIP_MIN_LENGTH
```

- **Explanation:** Sets the minimum length of a response that will be gzipped. The length is determined only from the "Content-Length" response header field.
- **Valid Values:** INTEGER.
- **Default:** `1000`.

```
NGINX_GZIP_TYPES
```

- **Explanation:** Enables gzipping of responses for the specified MIME types in addition to "text/html". The special value "*" matches any MIME type. Responses with the "text/html" type are always compressed.
- **Valid Values:** MIME type
- **Example:** `text/plain image/svg+xml application/json application/javascript text/xml text/css application/xml application/x-font-ttf font/opentype`.
- **Default:** `\*`.

```
NGINX_HTTPS_ALLOWED_IPS
```

- **Explanation:** Allow these IP addresses to access the website over http when `SSL_CERT_MODE` is set to `Yes` .
- **Valid Values:** `all`, any valid IP address.
- **Example:** `12.213.43.54/16`.
- **Default:** `all`.

```
NGINX_HTTP_ALLOW
```

- **Explanation:** Allow http access with https access. Valid only when `SSL_CERT_MODE` is set to `Yes` or `SelfSigned`.
- **Valid Values:** `True`, `False`.
- **Default:** `True`.

```
NGINX_CUSTOM_FILE
```

- **Explanation:** If you have a custom configuration file mounted, set this to `True`.
- **Valid Values:** `True`, `False`.
- **Default:** `False`.

```
NINGX_REAL_REMOTE_ADDR
```

- **Explanation:** The nginx header to get the value of the real IP address of Access points. Example if a reverse proxy is used in your cluster (Example if you are using an Ingress), then the real IP of the AP is most likely the `$http_x_forwarded_for`. If `$http_x_forwarded_for` returns a list, you can use `$real_ip` for getting first element of the list.
- **Valid Values:** `$remote_addr`, `$http_x_forwarded_for`, `$realip_remote_addr`, `$real_ip`.
- **Default:** `$real_ip`.

OpenVPN

`VPN_NAME`

- **Explanation:** Name of the VPN Server that will be visible on the OpenWISP dashboard.
- **Valid Values:** STRING.
- **Default:** `default`.

`VPN_CLIENT_NAME`

- **Explanation:** Name of the VPN client template that will be visible on the OpenWISP dashboard.
- **Valid Values:** STRING.
- **Default:** `default-management-vpn`.

Topology

`TOPOLOGY_UPDATE_INTERVAL`

- **Explanation:** Interval in minutes to upload the topology data to the OpenWISP,
- **Valid Values:** INTEGER.
- **Default:** 3.

X509 Certificates

`X509_NAME_CA`

- **Explanation:** Name of the default certificate authority visible on the OpenWISP dashboard.
- **Valid Values:** STRING.
- **Default:** `default`.

`X509_NAME_CERT`

- **Explanation:** Name of the default certificate visible on the OpenWISP dashboard.
- **Valid Values:** STRING.
- **Default:** `default`.

`X509_COUNTRY_CODE`

- **Explanation:** ISO code of the country of issuance of the certificate.

- **Valid Values:** Country code, see list here.
- **Default:** `IN`.

```
X509_STATE
```

- **Explanation:** Name of the state / province of issuance of the certificate.
- **Valid Values:** STRING.
- **Default:** `Delhi`.

```
X509_CITY
```

- **Explanation:** Name of the city of issuance of the certificate.
- **Valid Values:** STRING.
- **Default:** `New Delhi`.

```
X509_ORGANIZATION_NAME
```

- **Explanation:** Name of the organization issuing the certificate.
- **Valid Values:** STRING.
- **Default:** `OpenWISP`.

```
X509_ORGANIZATION_UNIT_NAME
```

- **Explanation:** Name of the unit of the organization issuing the certificate.
- **Valid Values:** STRING.
- **Default:** `OpenWISP`.

```
X509_EMAIL
```

- **Explanation:** Organization email address that'll be available to view in the certificate.
- **Valid Values:** STRING.
- **Default:** `certificate@example.com`.

```
X509_COMMON_NAME
```

- **Explanation:** Common name for the CA and certificate.
- **Valid Values:** STRING.
- **Default:** `OpenWISP`.

Misc Services

```
REDIS_HOST
```

- **Explanation:** Host to establish redis connection.
- **Valid Values:** A valid hostname or IP address.

- **Default:** `redis`.

```
REDIS_PORT
```

- **Explanation:** Port to establish redis connection.
- **Valid Values:** INTEGER.
- **Default:** `6379`.

```
REDIS_PASS
```

- **Explanation:** Redis password, optional.
- **Valid Values:** STRING.
- **Default:** `None`.

```
DASHBOARD_APP_SERVICE
```

- **Explanation:** Host to establish OpenWISP dashboard connection.
- **Valid Values:** Any hostname or IP address.
- **Default:** `dashboard`.

```
API_APP_SERVICE
```

- **Explanation:** Host to establish OpenWISP api connection.
- **Valid Values:** Any hostname or IP address.
- **Default:** `api`.

```
DASHBOARD_APP_PORT
```

- **Explanation:** The port on which nginx tries to get the OpenWISP dashboard container. Don't Change unless you know what you are doing.
- **Valid Values:** INTEGER.
- **Default:** `8000`.

```
API_APP_PORT
```

- **Explanation:** The port on which nginx tries to get the OpenWISP api container. Don't Change unless you know what you are doing.
- **Valid Values:** INTEGER.
- **Default:** `8001`.

```
WEBSOCKET_APP_PORT
```

- **Explanation:** The port on which nginx tries to get the OpenWISP websocket container. Don't Change unless you know what you are doing.
- **Valid Values:** INTEGER.

- **Default:** `8002`.

`DASHBOARD_INTERNAL`

- **Explanation:** Internal dashboard domain to reach dashboard from other containers.
- **Valid Values:** STRING.
- **Default:** `dashboard.internal`.

`API_INTERNAL`

- **Explanation:** Internal api domain to reach api from other containers.
- **Valid Values:** STRING.
- **Default:** `api.internal`.

NFS Server

`EXPORT_DIR`

- **Explanation:** Directory to be exported by the NFS server. Don't change this unless you know what you are doing.
- **Valid Values:** STRING.
- **Default:** `/exports`.

`EXPORT_OPTS`

- **Explanation:** NFS export options for the directory in `EXPORT_DIR` variable.
- **Valid Values:** STRING.
- **Default:** `10.0.0.0/8(rw,fsid=0,insecure,no_root_squash,no_subtree_check,sync)`.

Advanced Customization

This page describes advanced customization options for the OpenWISP Docker images.

The table of contents below provides a quick overview of the specific areas that can be customized.

Creating the `customization` Directory

The following commands will create the directory structure required for adding customizations. Execute these commands in the same location as the `docker-compose.yml` file.

```
mkdir -p customization/configuration/django
touch customization/configuration/django/__init__.py
touch customization/configuration/django/custom_django_settings.py
mkdir -p customization/theme
```

You can also refer to the directory structure of Docker OpenWISP repository for an example.

Supplying Custom Django Settings

The `customization/configuration/django` directory created in the previous section is mounted at `/opt/openwisp/openwisp/configuration` in the `dashboard`, `api`, `celery`, `celery_monitoring` and `celerybeat` containers.

You can specify additional Django settings (e.g. SMTP configuration) in the `customization/configuration/django/custom_django_settings.py` file. OpenWISP will include these settings during the startup phase.

You can also put additional files in `customization/configuration/django` that need to be mounted at `/opt/openwisp/openwisp/configuration` in the containers.

Supplying Custom CSS and JavaScript Files

If you want to use your custom styles, add custom JavaScript you can follow the following guide.

1. Read about the option OPENWISP_ADMIN_THEME_LINKS. Please make ensure the value you have enter is a valid JSON and add the desired JSON in `.env` file. example:

```
# OPENWISP_ADMIN_THEME_LINKS = [
#     {
#         "type": "text/css",
#         "href": "/static/custom/css/custom-theme.css",
#         "rel": "stylesheet",
#         "media": "all",
#     },
#     {
#         "type": "image/x-icon",
#         "href": "/static/custom/bootload.png",
#         "rel": "icon",
#     },
#     {
#         "type": "image/svg+xml",
#         "href": "/static/ui/openwisp/images/openwisp-logo-small.svg",
#         "rel": "icons",
#     },
# ]
# JSON string of the above configuration:
OPENWISP_ADMIN_THEME_LINKS='[{"type": "text/css", "href": "/static/custom/css/custom-theme.c
```

2. Create your custom CSS / Javascript file in `customization/theme` directory created in the above section. E.g. `customization/theme/static/custom/css/custom-theme.css`.
3. Start the nginx containers.

Note

1. You can edit the styles / JavaScript files now without restarting the container, as long as file is in the correct place, it will be picked.

2. You can create a `maintenance.html` file inside the `customize` directory to have a custom maintenance page for scheduled downtime.

Supplying Custom uWSGI configuration

By default, you can only configure "processes", "threads" and "listen" settings of uWSGI using environment variables. If you want to configure more uWSGI settings, you can supply your uWSGI configuration by following these steps:

1. Create the uWSGI configuration file in the `customization/configuration` directory. For the sake of this example, let's assume the filename is `custom_uwsgi.ini`.
2. In `dashboard` and `api` services of `docker-compose.yml`, add volumes as following

```
services:
  dashboard:
    ... # other configuration
    volumes:
      ... # other volumes
      - ${PWD}/customization/configuration/custom_uwsgi.ini:/opt/openwisp/uwsgi.ini:ro
  api:
    ... # other configuration
    volumes:
      ... # other volumes
      - ${PWD}/customization/configuration/custom_uwsgi.ini:/opt/openwisp/uwsgi.ini:ro
```

Supplying Custom Nginx Configurations

Docker

1. Create nginx your configuration file.
2. Set `NGINX_CUSTOM_FILE` to `True` in `.env` file.
3. Mount your file in `docker-compose.yml` as following:

```
nginx:
  ...
  volumes:
      ...
      PATH/TO/YOUR/FILE:/etc/nginx/nginx.conf
  ...
```

Supplying Custom Freeradius Configurations

Note: `/etc/raddb/clients.conf`, `/etc/raddb/radiusd.conf`, `/etc/raddb/sites-enabled/default`, `/etc/raddb/mods-enabled/`, `/etc/raddb/mods-available/` are the default files you may want to overwrite and you can find all of default files in `build/openwisp_freeradius/raddb`. The following are examples for including custom `radiusd.conf` and `sites-enabled/default` files.

Docker

1. Create file configuration files that you want to edit / add to your container.
2. Mount your file in `docker-compose.yml` as following:

```
nginx:
  ...
```

```
    volumes:
        ...
        PATH/TO/YOUR/RADIUSD:/etc/raddb/radiusd.conf
        PATH/TO/YOUR/DEFAULT:/etc/raddb/sites-enabled/default
    ...
```

Supplying Custom Python Source Code

You can build the images and supply custom python source code by creating a file named `.build.env` in the root of the repository, then set the variables inside `.build.env` file in `<variable>=<value>` format. Multiple variable should be separated in newline.

These are the variables that can be changed:

- `OPENWISP_MONITORING_SOURCE`
- `OPENWISP_FIRMWARE_SOURCE`
- `OPENWISP_CONTROLLER_SOURCE`
- `OPENWISP_NOTIFICATION_SOURCE`
- `OPENWISP_TOPOLOGY_SOURCE`
- `OPENWISP_RADIUS_SOURCE`
- `OPENWISP_IPAM_SOURCE`
- `OPENWISP_USERS_SOURCE`
- `OPENWISP_UTILS_SOURCE`
- `DJANGO_X509_SOURCE`
- `DJANGO_SOURCE`

For example, if you want to supply your own Django and OpenWISP Controller source, your `.build.env` should be written like this:

```
DJANGO_SOURCE=https://github.com/<username>/Django/tarball/master
OPENWISP_CONTROLLER_SOURCE=https://github.com/<username>/openwisp-controller/tarball/master
```

Disabling Services

- `openwisp-dashboard`: You cannot disable the openwisp-dashboard. It is the heart of OpenWISP and performs core functionalities.
- `openwisp-api`: You cannot disable the openwisp-api. It is required for interacting with your devices.
- `openwisp-websocket`: Removing this container will cause the system to not able to update real-time location for mobile devices.

If you want to disable a service, you can simply remove the container for that service, however, there are additional steps for some images:

- `openwisp-network-topology`: Set the `USE_OPENWISP_TOPOLOGY` variable to `False`.
- `openwisp-firmware-upgrader` : Set the `USE_OPENWISP_FIRMWARE` variable to `False`.
- `openwisp-monitoring` : Set the `USE_OPENWISP_MONITORING` variable to `False`.
- `openwisp-radius` : Set the `USE_OPENWISP_RADIUS` variable to `False`.
- `openwisp-postgres`: If you are using a separate database instance,
    - Ensure your database instance is reachable by the following OpenWISP containers: `openvpn`, `freeradius`, `celerybeat`, `celery`, `celery_monitoring`, `websocket`, `api`, `dashboard`.
    - Ensure your database server supports GeoDjango. (Install PostGIS for PostgreSQL)

- Change the PostgreSQL Database Setting to point to your instances, if you are using SSL, remember to set `DB_SSLMODE`, `DB_SSLKEY`, `DB_SSLCERT`, `DB_SSLROOTCERT`.
- If you are using SSL, remember to mount volume containing the certificates and key in all the containers which contact the database server and make sure that the private key permission is `600` and owned by `root:root`.
- In your database, create database with name `<DB_NAME>`.

- `openwisp-postfix`:
  - Ensure your SMTP instance reachable by the OpenWISP containers.
  - Change the email configuration variables to point to your instances.

Docker OpenWISP FAQs

1. Setup fails, it couldn't find the images on DockerHub?

Answer: The setup requires following ports and destinations to be unblocked, if you are using a firewall or any external control to block traffic, please whitelist:

	User Id	Protocol	DstPort	Destination	Process
1	0	tcp,udp	443,53	gitlab.com	`/usr/bin/dockerd`
2	0	tcp,udp	443,53	registry.gitlab.com	`/usr/bin/dockerd`
3	0	tcp,udp	443,53	storage.googleapis.com	`/usr/bin/dockerd`
4	0	udp	53	registry.gitlab.com	`/usr/bin/docker`
5	0	tcp,udp	443,53	github.com	`/usr/lib/git-core/git-remote-http`
6	0	tcp	443,80	172.18.0.0/16	`/usr/bin/docker-proxy`
7	0	udp	1812, 1813	172.18.0.0/16	`/usr/bin/docker-proxy`
8	0	tcp	25	172.18.0.0/16	`/usr/bin/docker-proxy`

2. Makefile failed without any information, what's wrong?

Answer: You are using an old version of a requirement, please consider upgrading:

```
$ git --version
git version 2.25.1
$ docker --version
Docker version 27.0.2, build 912c1dd
$ docker compose version
Docker Compose version v2.28.1
$ make --version
GNU Make 4.2.1
$ bash --version
GNU bash, version 5.0.3(1)-release (x86_64-pc-linux-gnu)
$ uname -v # kernel-version
#1 SMP Debian 4.19.181-1 (2021-03-19)
```

3. Can I run the containers as the `root` or `docker`

No, please do not run the Docker containers as these users.

Ensure you use a less privileged user and tools like `sudo` or `su` to escalate privileges during the installation phase.

Developer Docs

Note

This page is for developers who want to customize or extend OpenWISP Controller, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- Docker OpenWISP User Docs

Important

The Docker OpenWISP installation responds only to the fully qualified domain names (FQDN) defined in the configuration. If you are deploying locally (for testing), you need to update the `/etc/hosts` file on your machine to resolve the configured domains to localhost.

For example, the following command will update the `/etc/hosts` file to resolve the domains used in the default configurations:

```
echo "127.0.0.1 dashboard.openwisp.org api.openwisp.org openvpn.openwisp.org" | \
    sudo tee -a /etc/hosts
```

Building and Running Images

1. Install Docker.
2. In the root directory of the repository, run `make develop`. Once the containers are ready, you can test them by accessing the domain names of the modules.

Important

- The default username and password are `admin`.
- The default domains are `dashboard.openwisp.org` and `api.openwisp.org`.

- You will need to repeat step 2 each time you make changes and want to rebuild the images.
- If you want to perform actions such as cleaning everything produced by `docker-openwisp`, please refer to the makefile options.

Running Tests

You can run tests using either `geckodriver` (Firefox) or `chromedriver` (Chromium).

Chromium is preferred as it also checks for console log errors.

Using Chromedriver

Install WebDriver for Chromium for your browser version from https://chromedriver.chromium.org/home and extract `chromedriver` to one of directories from your `$PATH` (example: `~/.local/bin/`).

Using Geckodriver

Install Geckodriver for Firefox for your browser version from https://github.com/mozilla/geckodriver/releases and extract `geckodriver` to one of directories from your `$PATH` (example: `~/.local/bin/`).

Finish Setup and Run Tests

1. Install test requirements:

   ```
   python3 -m pip install -r requirements-test.txt
   ```

2. (Optional) Modify configuration options in `tests/config.json`:

   ```
   driver: Name of the driver to use for tests, "chromium" or "firefox"
   logs: Print container logs if an error occurs
   logs_file: Location of the log file for saving logs generated during tests
   headless: Run Selenium Chrome driver in headless mode
   load_init_data: Flag for running tests/data.py, only needs to be done once after databas
   app_url: URL to reach the admin dashboard
   username: Username for logging into the admin dashboard
   password: Password for logging into the admin dashboard
   services_max_retries: Maximum number of retries to check if services are running
   services_delay_retries: Delay time (in seconds) for each retry when checking if services
   ```

3. Run tests with:

   ```
   make runtests
   ```

4. To run a single test suite, use the following command:

   ```
   python3 tests/runtests.py <TestSuite>.<TestCase>
   ```

Run Quality Assurance Checks

We use shfmt to format shell scripts and hadolint to lint Dockerfiles.

To format all files, run:

```
./qa-format
```

To run quality assurance checks, use the `run-qa-checks` script:

```
# Run QA checks before committing code
./run-qa-checks
```

Makefile Options

Most commonly used:

- `make start [USER=docker-username] [TAG=image-tag]`: Start OpenWISP containers on your server.
- `make pull [USER=docker-username] [TAG=image-tag]`: Pull images from the registry.
- `make stop`: Stop OpenWISP containers on your server.
- `make develop`: Bundle all the commands required to build the images and run containers.
- `make runtests`: Start containers and run test cases to ensure all services are working. It stops containers after the test suite passes.
- `make clean`: Aggressively purge all containers, images, volumes, and networks related to `docker-openwisp`.

Other options:

- `make publish [USER=docker-username] [TAG=image-tag]`: Build, test, and publish images.
- `make python-build`: Generate a random Django secret and set it in the `.env` file.
- `make nfs-build`: Build the OpenWISP NFS server image.
- `make base-build`: Build the OpenWISP base image. The base image is used in other OpenWISP images.
- `make compose-build`: (default) Build OpenWISP images for development.
- `make develop-runtests`: Similar to `runtests`, but it doesn't stop the containers after running the tests, which may be desired for debugging and analyzing failing container logs.
- `make develop-pythontests`: Similar to `develop-runtests`, but it requires containers to be already running.

Modules

Users

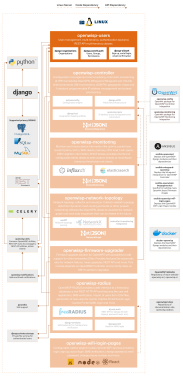
Seealso

Source code: github.com/openwisp/openwisp-users.

The OpenWISP Users module leverages the capabilities of the Django Framework and its rich ecosystem to provide OpenWISP with features for managing user accounts, permission groups, supporting different authentication schemes, implementing multi-tenancy for allowing multiple organizations to be managed by different users within a single OpenWISP instance and more.

For a full introduction please refer to Users: Structure & Features.

The following diagram illustrates the role of the Users module within the OpenWISP architecture.



OpenWISP Architecture: highlighted users module

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

Users: Structure & Features

The OpenWISP Users module leverages the capabilities of the Django Framework and its rich ecosystem to provide OpenWISP with features for managing user accounts, permission groups, supporting different authentication schemes, and implementing multi-tenancy. This allows multiple organizations to be managed by different users within a single OpenWISP instance, among other functionalities.

User Management

- Create, read, update, and delete user accounts.
- Support for custom user fields through extensible models (see Extending OpenWISP Users for more information).
- Export user data through a management command (from the Linux shell).

Multi-tenancy

- Create multiple organizations (also commonly referred to as tenants).
- Users can be associated with one or multiple organizations as members, managers, or owners.
- Each organization can access only their data.
- Shared data: some objects can be shared among multiple organizations.

See Basic Concepts for more information.

Permissions and Roles

- Granular permission control for users and organizations.
- Default roles for administrators, managers, and regular users.
- Customizable permission sets for specific needs.

See Basic Concepts for more information.

API Integration

- RESTful API endpoints for user and organization management.
- Secure API access with token-based authentication.

See REST API for more information.

Admin Interface

- User-friendly Django admin interface.
- Customizable admin views for user and organization management (see Extending OpenWISP Users for more information).

Extensible Authentication

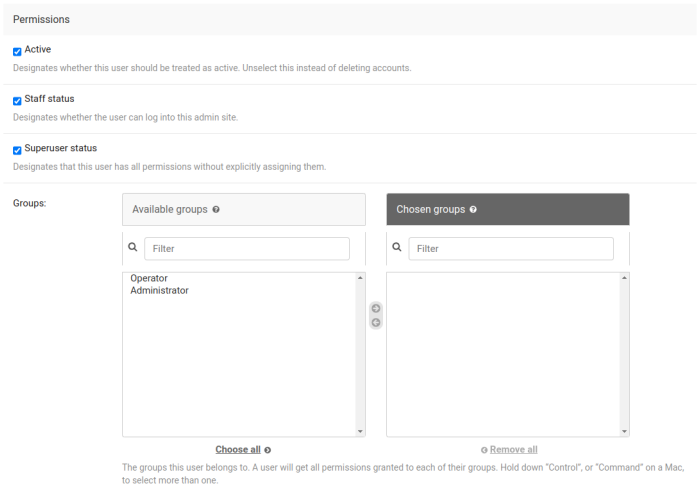
With some additional work, it is possible to leverage the rich ecosystem of Django third party apps to implement the following:

- Possibility to log in in the admin interface via authentication schemes like OAuth, SAML, MS Azure Authentication, etc.
- Support multi-factor authentication (MFA).

On a similar note, the OpenWISP RADIUS module ships logic that allows end-users to log into WiFi services using OAuth (e.g.: social login provided by Google, Facebook) or SAML (e.g.: EIDAS, SPID).

Basic Concepts

Superusers



A superuser, also known as a "super administrator," is a special type of admin user account with full access to all aspects of an OpenWISP instance.

The **Superuser status** flag in the user details page indicates whether a user is a superuser or not. Only superusers are allowed to edit this flag.

Superusers have all permissions enabled by default and can create, manage and delete any organization available in the system.

However, it's essential to use superuser accounts sparingly due to their elevated privileges.

To grant access to specific features and organizations within your OpenWISP system, consider creating staff users without the "superuser status" flag enabled. Assign them to one of the available permission groups, as explained in the following sections. These users will have limited administrative capabilities, managing only the objects permitted by their assigned permissions and associated organization.

Staff Users

Users with the **Staff status** flag enabled, as shown in the screenshot above, have access to the OpenWISP Admin interface. This access allows them to manage various aspects of the OpenWISP instance according to their assigned permissions and organizational role.

Users with this flag disabled will still be able to interact with OpenWISP, but in a more limited way. They can use non-administrative user interfaces or specific REST API HTTP endpoints designed for end-users.

Note

An example of an end-user is someone who signs up for a public WiFi hotspot service via the WiFi Login Pages module. This optional OpenWISP module is commonly used in public WiFi hotspot deployments.

Permissions

The permission system used by OpenWISP is based on the Django Permission System.

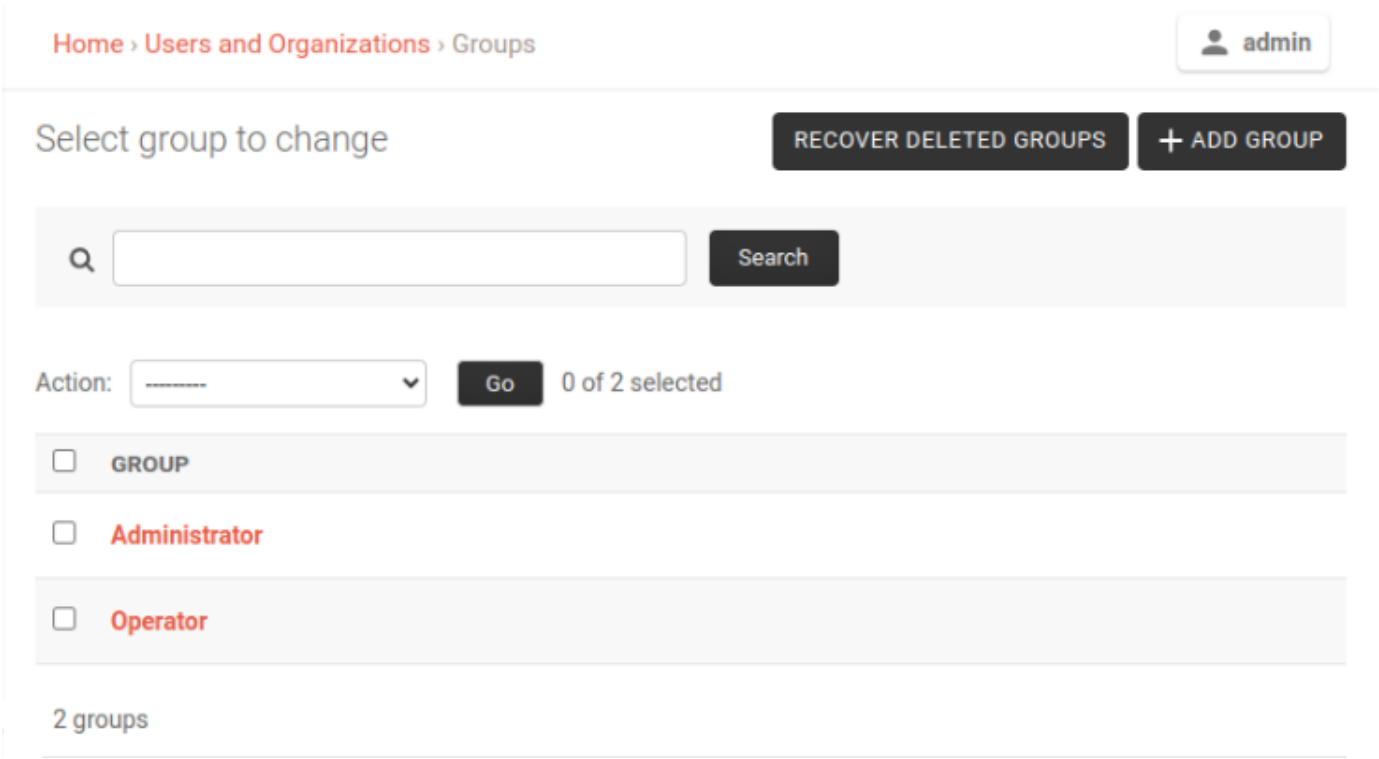
In short, a permission indicates whether a user has the authority to perform the following operations:

- **View**: Access the details of a specific class of objects, e.g., view the details of users.
- **Add**: Create a new object of a specific class, e.g., add a new user.
- **Change**: Edit the details of a specific class, e.g., modify existing user details.

- **Delete**: Remove an object of a specific class, e.g., delete users.

> **Note**
>
> For more detailed technical information, please refer to the Django Documentation.

Default Permission Groups



A permission group is a collection of permissions that can be assigned to multiple users.

It is then possible to change the permissions on the group to reflect the changes on all the users who are part of the permission group.

This allows to avoid having to assign permissions to individual users, which is hard to maintain and leads to inconsistent permission configuration over time.

OpenWISP provides a few permission groups which are explained below.

Administrator

This permission group is designed for users who need to manage most aspects of an organization without having superuser access.

Operator

This permission group is designed for users who need to be able to perform a limited amount of operations like provisioning new devices and perform regular network maintenance operations but are not allowed to create new users or change the permissions settings of other users.

Use this for users who have very specific and limited responsibilities in the network.

Organizations & Multi-Tenancy

The concept of multi-tenancy in OpenWISP is implemented through "organizations".

An organization in OpenWISP represents a distinct entity or tenant within the system. Each organization has its own set of users, configurations, data, and administrative controls, allowing for isolation and management of network resources.

Key Features of Organizations:

- **Isolation & Privacy**: Organizations provide a logical separation of resources, ensuring that data and configurations are segregated between different entities or tenants. Each tenant can only see and interact with the data of their organizations and Shared Objects defined by super administrators.
- **User Management**: Each organization can have its own set of users with specific roles and permissions tailored to their responsibilities within that organization.
- **Administrative Controls**: Super administrators can define, oversee, and manage Shared Objects, permission policies, and any other processes relating to organizations to ensure consistency across the entire system.

By leveraging organizations, OpenWISP provides a robust framework for implementing multi-tenancy, allowing for the efficient management of network resources across diverse entities or tenants within a single instance of the platform.

Note

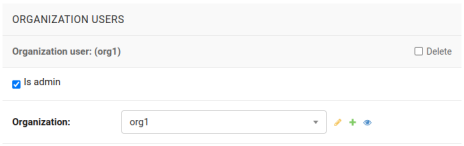
Multi-Tenancy and Organizations are implemented in OpenWISP with the django-organizations third-party app.

Organization Membership and Roles

A user can be associated to one or multiple organizations and have different roles in each.

Here's a summary of the default organization roles.

Organization Manager



Any user with the "Is admin" flag enabled for a specific organization (as shown in the screenshot above) is considered by the system a manager of that organization. Organization managers have the authority to view and interact with the data belonging to that organization according to their set of permissions (as defined in Permission Groups).

To modify this flag, navigate to the "ORGANIZATION USERS" section on the "Change user" page.

Organization Members (End-Users)

Any user with the "Is admin" flag disabled for a specific organization (as shown in the screenshot above) is considered by the system a regular end-user of that organization.

These users are consumers of a service provided by the organization. They will not be able to see or interact with any object of that organization via the administrative interface, even if they are flagged as Staff users.

They can only consume REST API endpoints or other non administrative user interface pages.

A real-world example of this is the User API endpoints of OpenWISP RADIUS, which allow users to sign up to an organization, verify their phone number by receiving a verification code via SMS, see their RADIUS sessions, etc. All those endpoints are tied to an organization because different organizations can have very different configurations. Users are allowed to consume those endpoints only if they're members.

Organization Owners

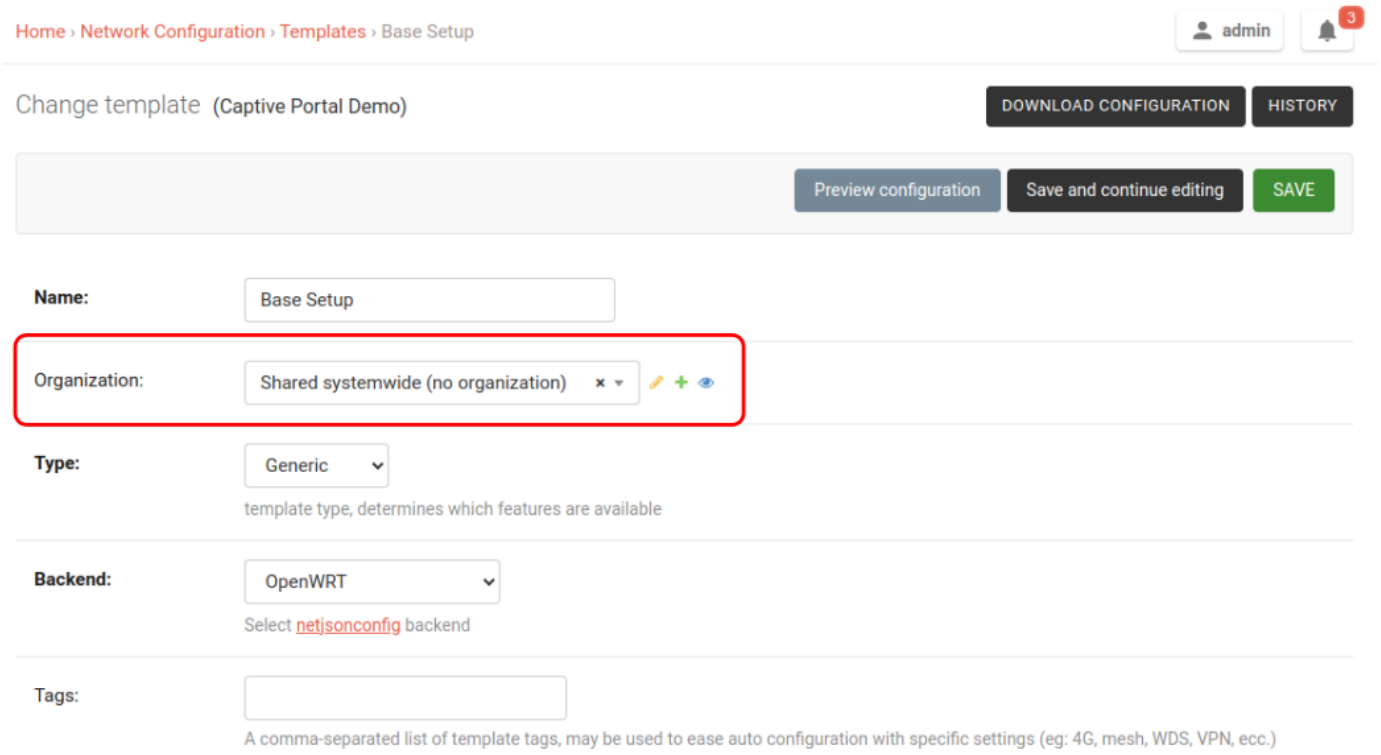
An organization owner is a user designated as the owner of a particular organization. This owner cannot be deleted or edited by other administrators; only superusers have permission to perform these actions.

By default, the first manager of an organization is designated as the owner of that organization.

Only superusers and organization owners are allowed to change the owner of an organization. Organization owners can be changed from the "Change organization" page by navigating to the "ORGANIZATION OWNER" section.

If the `OrganizationUser` instance related to the owner of an organization is deleted or flagged as `is_admin=False`, the admin interface will return an error informing users that the operation is not allowed. The owner should be changed before attempting to perform such actions.

Shared Objects



A shared object is a resource that can be used by multiple organizations or tenants within the system.

Shared objects do not belong to any specific organization. In the user interface, the organization field is empty, and it displays *"Shared systemwide (no organization)"* as shown in the screenshot above. These objects are defined and managed by super administrators and can include configurations, policies, or other data that need to be consistent across all organizations.

By sharing common resources, global uniformity and consistency can be enforced across the entire system.

Note

Only a specific subset of object classes can be shared. You can determine if an object can be shared by attempting to create a new object for that class while logged in as a superuser. If the organization field shows the option *"Shared systemwide (no organization)"*, it means the object can be shared.

Examples of shared objects include:

- Shared Configuration Templates
- Shared VPN servers
- Shared Subnets

Management Commands

`export_users`

This command exports user data to a CSV file, including related data such as organizations.

Arguments:

- `--exclude-fields`: Optional, comma-separated list of fields to exclude from the export.
- `--filename`: Optional, filename for the exported CSV, defaults to "openwisp_exported_users.csv".

Example usage:

```
./manage.py export_users --exclude-fields birth_date,location --filename users.csv
```

For advanced customizations (e.g., adding fields for export), you can use the OPENWISP_USERS_EXPORT_USERS_COMMAND_CONFIG setting.

Settings

> **Note**
>
> If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

`OPENWISP_ORGANIZATION_USER_ADMIN`

type:	`boolean`
default:	`True`

Indicates whether the admin section for managing `OrganizationUser` items is enabled or not.

`OPENWISP_ORGANIZATION_OWNER_ADMIN`

type:	`boolean`
default:	`True`

Indicates whether the admin section for managing `OrganizationOwner` items is enabled or not.

Refer to Organization Owners for more information.

`OPENWISP_USERS_AUTH_API`

type:	`boolean`

default:	`True`

Indicates whether the REST API is enabled or not.

`OPENWISP_USERS_AUTH_THROTTLE_RATE`

type:	`str`
default:	`100/day`

Indicates the rate throttling for the Obtain Authentication Token API endpoint.

Please note that the current rate throttler is very basic and will also count valid requests for rate limiting. For more information, check Django-rest-framework throttling guide.

`OPENWISP_USERS_AUTH_BACKEND_AUTO_PREFIXES`

type:	`tuple`
default:	`tuple()`

A tuple or list of international prefixes which will be automatically tested by the authentication backend of OpenWISP Users when parsing phone numbers.

Each prefix will be prepended to the username string automatically and parsed with the `phonenumbers` library to find out if the result is a valid number of not.

This allows users to log in by using only the national phone number, without having to specify the international prefix.

`OPENWISP_USERS_EXPORT_USERS_COMMAND_CONFIG`

type:	`dict`
default:	`{ "fields": [ "id", "username", "email", "password", "first_name", "last_name", "is_staff", "is_active", "date_joined", "phone_number", "birth_date", "location", "notes", "language", "organizations", ], "select_related": [], }`

This setting can be used to configure the exported fields for the export_users command.

The `select_related` property can be used to optimize the database query.

`OPENWISP_USERS_USER_PASSWORD_EXPIRATION`

type:	`integer`
default:	`0`

Number of days after which a user's password will expire. In other words, it determines when users will be prompted to change their passwords.

If set to `0`, this feature is disabled, and users are not required to change their passwords.

`OPENWISP_USERS_STAFF_USER_PASSWORD_EXPIRATION`

type:	`integer`
default:	`0`

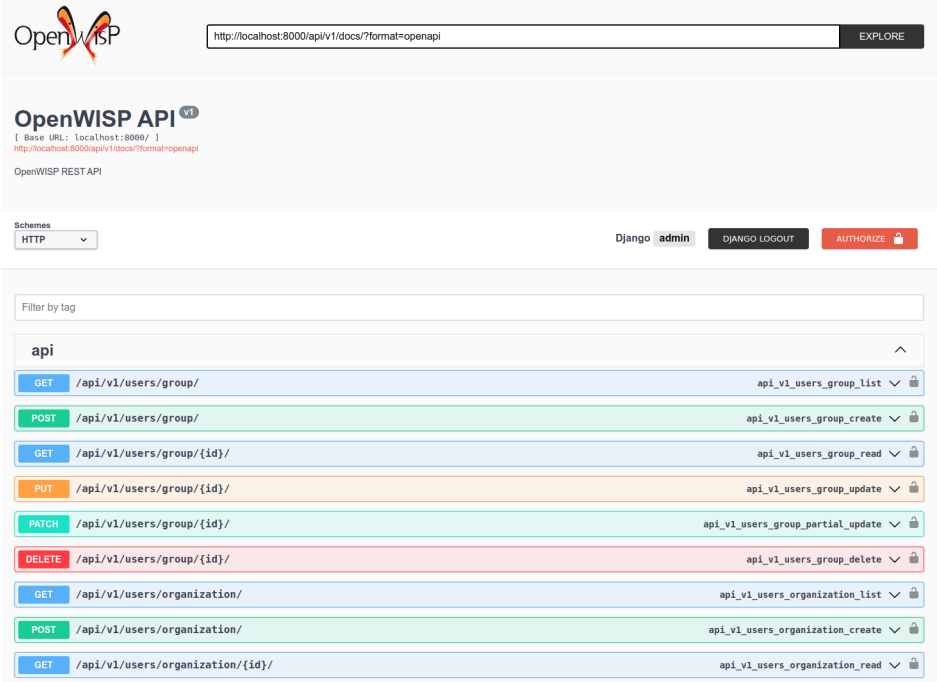
Similar to OPENWISP_USERS_USER_PASSWORD_EXPIRATION, but for **staff users**.

REST API

Note

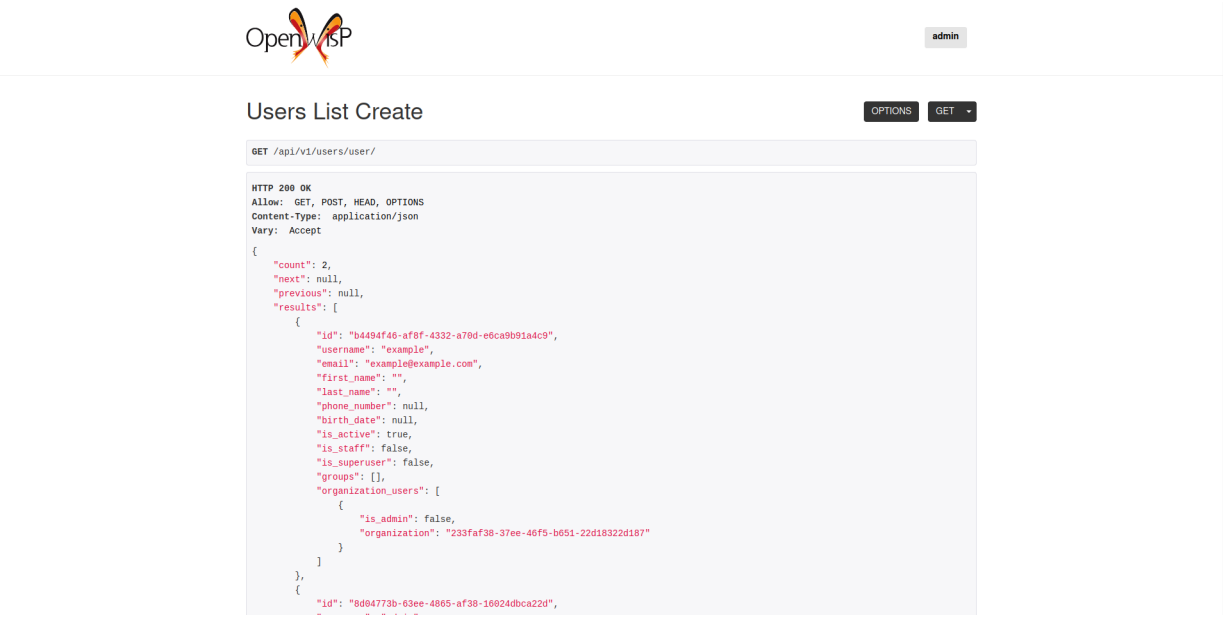
The REST API is enabled by default but can be disabled by setting OPENWISP_USERS_AUTH_API to `False`.

Live Documentation



General live API documentation, following the OpenAPI specification, is available at `/api/v1/docs/`.

Browsable Web Interface



Additionally, opening any of the endpoints listed below directly in the browser will show the browsable API interface of Django-REST-Framework, which makes it even easier to find out the details of each endpoint.

Obtain Authentication Token

`/api/v1/users/token/`

This endpoint only accepts the `POST` method and is used to retrieve the Bearer token that is required to make API requests to other endpoints.

Example usage:

```
curl -i -X POST http://localhost:8000/api/v1/users/token/ -d "username=openwisp" -d "passwor

HTTP/1.1 200 OK
Date: Wed, 05 Jun 2024 16:31:33 GMT
Server: WSGIServer/0.2 CPython/3.8.10
Content-Type: application/json
Vary: Accept
Allow: POST, OPTIONS
X-Frame-Options: DENY
Content-Length: 52
X-Content-Type-Options: nosniff
Referrer-Policy: same-origin
Cross-Origin-Opener-Policy: same-origin

{"token": "7a2e1d3d008253c123c61d56741003db5a194256"}
```

Authenticating with the User Token

The authentication class `openwisp_users.api.authentication.BearerAuthentication` is used across the different OpenWISP modules for authentication.

To use it, first of all get the user token as described above in Obtain Authentication Token, then send the token in the `Authorization` header:

```
# Get the bearer token
TOKEN=$(curl -X POST http://localhost:8000/api/v1/users/token/ -d "username=openwisp" -d "pa

# Get user list, send bearer token in authorization header
curl http://localhost:8000/api/v1/users/user/ -H "Authorization: Bearer $TOKEN"
```

List of Endpoints

Since the detailed explanation is contained in the Live Documentation and in the Browsable Web Interface of each endpoint, here we'll provide just a list of the available endpoints, for further information please open the URL of the endpoint in your browser.

Change User password

```
PUT /api/v1/users/user/{id}/password/
```

List Groups

```
GET /api/v1/users/group/
```

Create New Group

```
POST /api/v1/users/group/
```

Get Group Detail

```
GET /api/v1/users/group/{id}/
```

Change Group Detail

```
PUT /api/v1/users/group/{id}/
```

Patch Group Detail

```
PATCH /api/v1/users/group/{id}/
```

Delete Group

```
DELETE /api/v1/users/group/{id}/
```

List Email Addresses

```
GET /api/v1/users/user/{id}/email/
```

Add Email Address

```
POST/api/v1/users/user/{id}/email/
```

Get Email Address

```
GET /api/v1/users/user/{id}/email/{id}/
```

Change Email Address

```
PUT /api/v1/users/user/{id}/email/{id}/
```

Patch Email Address

```
PATCH /api/v1/users/user/{id}/email/{id}/
```

Make/Unmake Email Address Primary

```
PATCH /api/v1/users/user/{id}/email/{id}/
```

Mark/Unmark Email Address as Verified

```
PATCH /api/v1/users/user/{id}/email/{id}/
```

Remove Email Address

```
DELETE /api/v1/users/user/{id}/email/{id}/
```

List Organizations

```
GET /api/v1/users/organization/
```

Create new Organization

```
POST /api/v1/users/organization/
```

Get Organization Detail

```
GET /api/v1/users/organization/{id}/
```

Change Organization Detail

```
PUT /api/v1/users/organization/{id}/
```

Patch Organization Detail

```
PATCH /api/v1/users/organization/{id}/
```

Delete Organization

```
DELETE /api/v1/users/organization/{id}/
```

List Users

```
GET /api/v1/users/user/
```

Create User

```
POST /api/v1/users/user/
```

> **Note**
>
> Passing `true` to the optional `is_verified` field allows creating users with their email address flagged as verified. This will also skip sending the verification link to their email address.

Get User Detail

```
GET /api/v1/users/user/{id}/
```

Change User Detail

```
PUT /api/v1/users/user/{id}/
```

Patch User Detail

```
PATCH /api/v1/users/user/{id}/
```

Delete User

```
DELETE /api/v1/users/user/{id}/
```

Developer Docs

> **Note**
>
> This page is for developers who want to customize or extend OpenWISP Users, whether for bug fixes, new features, or contributions.
>
> For user guides and general information, please see:
>
> - General OpenWISP Quickstart
> - OpenWISP Users Usage Docs

Developer Installation Instructions

> **Note**
>
> This page is for developers who want to customize or extend OpenWISP Users, whether for bug fixes, new features, or contributions.
>
> For user guides and general information, please see:
>
> - General OpenWISP Quickstart
> - OpenWISP Users Usage Docs

Installing for Development

Install sqlite:

```
sudo apt-get install sqlite3 libsqlite3-dev openssl libssl-dev
```

Install your forked repo:

```
git clone git://github.com/<your_fork>/openwisp-users
cd openwisp-users/
pip install -e .[rest]
```

Install test requirements:

```
pip install -r requirements-test.txt
```

Start Redis

```
docker-compose up -d
```

Create database:

```
cd tests/
./manage.py migrate
./manage.py createsuperuser
```

Run celery and celery-beat with the following commands (separate terminal windows are needed):

```
cd tests/
celery -A openwisp2 worker -l info
celery -A openwisp2 beat -l info
```

Launch development server:

```
./manage.py runserver
```

You can access the admin interface at `http://127.0.0.1:8000/admin/`.

Run tests with:

```
# --parallel and --keepdb are optional but help to speed up the operation
./runtests.py --parallel --keepdb
```

Alternative Sources

Pypi

To install the latest Pypi:

```
pip install openwisp-users
```

Github

To install the latest development version tarball via HTTPs:

```
pip install https://github.com/openwisp/openwisp-users/tarball/master
```

Alternatively you can use the git protocol:

```
pip install -e git+git://github.com/openwisp/openwisp-users#egg=openwisp_users
```

Admin Utilities

This section outlines the admin utilities provided by the OpenWISP Users module.

`MultitenantAdminMixin`

Full python path: `openwisp_users.multitenancy.MultitenantAdminMixin`.

Adding this mixin to a `ModelAdmin` class makes it multitenant-capable, allowing users to see only items of the organizations they manage or own.

This class has two important attributes:

- `multitenant_shared_relations`: If the model has relations (e.g., `ForeignKey`, `OneToOne`) to other multitenant models with an `organization` field, list those model attributes here as a list of strings. See how it is used in OpenWISP Controller for a real-world example.
- `multitenant_parent`: If the admin model relies on a parent model with the `organization` field, specify the field pointing to the parent here. See how it is used in OpenWISP Firmware Upgrader for a real-world example.

`MultitenantOrgFilter`

Full python path: `openwisp_users.multitenancy.MultitenantOrgFilter`.

This auto complete admin filter displays only organizations the current user can manage. Below is an example of adding the auto complete organization filter in `BookAdmin`:

```
from django.contrib import admin
from openwisp_users.multitenancy import MultitenantOrgFilter


class BookAdmin(admin.ModelAdmin):
    list_filter = [
        MultitenantOrgFilter,
    ]
    # other attributes
```

`MultitenantRelatedOrgFilter`

Full python path: `openwisp_users.multitenancy.MultitenantRelatedOrgFilter`.

This filter is similar to `MultitenantOrgFilter` but displays only objects related to organizations the current user can manage. Use this for creating filters for related multitenant models.

Consider the following example from IpAddressAdmin in openwisp-ipam. `IpAddressAdmin` allows filtering *IpAddress objects by ``Subnet`* belonging to organizations managed by the user.

```
from django.contrib import admin
from openwisp_users.multitenancy import MultitenantRelatedOrgFilter
from swapper import load_model

Subnet = load_model("openwisp_ipam", "Subnet")


class SubnetFilter(MultitenantRelatedOrgFilter):
    field_name = "subnet"
    parameter_name = "subnet_id"
    title = _("subnet")


@admin.register(IpAddress)
class IpAddressAdmin(
    VersionAdmin,
    MultitenantAdminMixin,
    TimeReadonlyAdminMixin,
    ModelAdmin,
):
    list_filter = [SubnetFilter]
    # other options
```

Django REST Framework Utilities

Note

This page is for developers who want to customize or extend OpenWISP Users, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Users Usage Docs

This page details the Django REST Framework classes and utilities provided in the OpenWISP Users module. These tools support various REST API features such as authentication, permission enforcement, multi-tenancy, and filtering.

These utilities ensure consistency and reusability across the OpenWISP modules.

Authentication

`openwisp_users.api.authentication.BearerAuthentication`

`BearerAuthentication` is the primary authentication class used in OpenWISP's REST APIs. It is based on TokenAuthentication from Django REST Framework.

For detailed usage instructions, please refer to the *authenticating with the user token :ref:`authenticating_rest_api* section.

`openwisp_users.api.authentication.SesameAuthentication`

`SesameAuthentication` allows authentication using tokens generated by django-sesame.

This method is primarily used for password-less authentication, such as magic login links sent via email or SMS.

To use this authentication class, you must configure `django-sesame`.

For more details, please see the django-sesame documentation.

Permission Classes

The custom Django REST Framework permission classes `IsOrganizationMember`, `IsOrganizationManager`, and `IsOrganizationOwner` ensure that the requesting user belongs to the same organization as the requested object and has the appropriate role: member, manager, or owner, respectively.

Usage example:

```
from openwisp_users.api.permissions import IsOrganizationManager
from rest_framework import generics


class MyApiView(generics.APIView):
    permission_classes = (IsOrganizationManager,)
```

`organization_field`

type:	`string`
default:	`organization`

`organization_field` specifies where to find the organization of the current object. In most cases, this default value does not need to be changed. However, it may need to be adjusted if the `organization` is defined only on a parent object.

For example, in openwisp-firmware-upgrader, the `organization` is defined on `Category`, and `Build` has a relation to `Category`. Therefore, the organization of `Build` instances is inferred from the `Category` organization.

To implement the permission class correctly in such cases, you would use:

```
from openwisp_users.api.permissions import IsOrganizationManager
from rest_framework import generics


class MyApiView(generics.APIView):
    permission_classes = (IsOrganizationManager,)
    organization_field = "category__organization"
```

This setup translates to accessing `obj.category.organization`. Ensure your view's querysets use select_related to avoid generating too many queries.

`DjangoModelPermissions`

The default `DjangoModelPermissions` class does not check for the `view` permission on objects for `GET` requests. The extended `DjangoModelPermissions` class addresses this issue. It checks for the availability of either the `view` or `change` permissions to allow `GET` requests on any object.

Usage example:

```
from openwisp_users.api.permissions import DjangoModelPermissions
from rest_framework.generics import ListCreateAPIView


class TemplateListCreateView(ListCreateAPIView):
    serializer_class = TemplateSerializer
    permission_classes = (DjangoModelPermissions,)
    queryset = Template.objects.all()
```

Note: `DjangoModelPermissions` allows users who are either organization managers or owners to view shared objects in read-only mode.

Standard users will not be able to view or list shared objects.

`ProtectedAPIMixin`

Full python path: `openwisp_users.api.mixins.ProtectedAPIMixin`.

This mixin provides a set of authentication and permission classes that are commonly used across various OpenWISP modules API views.

Usage example:

```
# Used in openwisp-ipam
from openwisp_users.api.mixins import (
    ProtectedAPIMixin as BaseProtectedAPIMixin,
)


class ProtectedAPIMixin(BaseProtectedAPIMixin):
    throttle_scope = "ipam"


class SubnetView(ProtectedAPIMixin, RetrieveUpdateDestroyAPIView):
    serializer_class = SubnetSerializer
    queryset = Subnet.objects.all()
```

Mixins for Multi-Tenancy

Filtering Items by Organization

The custom Django REST Framework mixins `FilterByOrganizationMembership`, `FilterByOrganizationManaged` and `FilterByOrganizationOwned` can be used in the API views to ensure that the current user is able to see only the data related to their organization when accessing the API view.

These classes work by filtering the queryset so that only items related to organizations the user is member, manager or owner of, respectively.

These mixins ship the Django REST Framework's IsAuthenticated permission class by default because the organization filtering works only on authenticated users. Always remember to include this class when overriding `permission_classes` in a view.

Usage example:

```
from openwisp_users.api.mixins import FilterByOrganizationManaged
from rest_framework import generics


class UsersListView(FilterByOrganizationManaged, generics.ListAPIView):
    """
    UsersListView will show only users from organizations managed
    by current user in the list.
    """

    pass


class ExampleListView(FilterByOrganizationManaged, generics.ListAPIView):
    """
    Example showing how to extend ``permission_classes``.
    """

    permission_classes = FilterByOrganizationManaged.permission_classes + [
        # additional permission classes here
    ]
```

Checking Parent Objects

Sometimes, the API view needs to check the existence and the `organization` field of a parent object.

In such cases, `FilterByParentMembership`, `FilterByParentManaged` and `FilterByParentOwned` can be used.

For example, given a hypothetical URL `/api/v1/device/{device_id}/config/`, the view must check that `{device_id}` exists and that the user has access to it, here's how to do it:

```
import swapper
from rest_framework import generics
from openwisp_users.api.mixins import FilterByParentManaged

Device = swapper.load_model("config", "Device")
Config = swapper.load_model("config", "Config")

# URL is:
# /api/v1/device/{device_id}/config/


class ConfigListView(FilterByParentManaged, generics.DetailAPIView):
    model = Config

    def get_parent_queryset(self):
        qs = Device.objects.filter(pk=self.kwargs["device_id"])
        return qs
```

Multi-tenant Serializers for the Browsable Web UI

Django REST Framework provides a browsable API which can be used to create HTTP requests right from the browser.

The relationship fields in this interface show all the relationships, without filtering by the organization the user has access to, which breaks multi-tenancy.

The `FilterSerializerByOrgMembership`, `FilterSerializerByOrgManaged` and `FilterSerializerByOrgOwned` can be used to solve this issue.

These serializers do not allow non-superusers to create shared objects.

Usage example:

```
from openwisp_users.api.mixins import FilterSerializerByOrgOwned
from rest_framework.serializers import ModelSerializer
from .models import Device


class DeviceSerializer(FilterSerializerByOrgOwned, ModelSerializer):
    class Meta:
        model = Device
        fields = "__all__"
```

The `include_shared` boolean attribute can be used to include shared objects in the accepted values of the multi-tenant serializers.

Shared objects have the `organization` field set to `None` and can be used by any organization. A common use case is shared templates in OpenWISP Controller.

Usage example:

```
from openwisp_users.api.mixins import FilterSerializerByOrgOwned
from rest_framework.serializers import ModelSerializer
from .models import Book


class BookSerializer(FilterSerializerByOrgOwned, ModelSerializer):
    include_shared = True

    class Meta:
```

```
        model = Book
        fields = "__all__"
```

To filter items based on the `organization` of their parent object, `organization_field` attribute can be defined in the view function which is inheriting any of the mixin classes.

Usage example: organization_field.

Multi-tenant Filtering Capabilities for the Browsable Web UI

Integration of Django filters with Django REST Framework is provided through a DRF-specific `FilterSet` and a `filter backend`.

The relationship fields of `django-filters` show all the available results, without filtering by the organization the user has access to, which breaks multi-tenancy.

The `FilterDjangoByOrgMembership`, `FilterDjangoByOrgManaged` and `FilterDjangoByOrgOwned` can be used to solve this issue.

Usage example:

```
from django_filters import rest_framework as filters
from openwisp_users.api.mixins import FilterDjangoByOrgManaged
from ..models import FloorPlan


class FloorPlanOrganizationFilter(FilterDjangoByOrgManaged):
    organization_slug = filters.CharFilter(
        field_name="organization__slug"
    )

    class Meta:
        model = FloorPlan
        fields = ["organization", "organization_slug"]


class FloorPlanListCreateView(
    ProtectedAPIMixin, generics.ListCreateAPIView
):
    serializer_class = FloorPlanSerializer
    queryset = FloorPlan.objects.select_related().order_by("-created")
    pagination_class = ListViewPagination
    filter_backends = [filters.DjangoFilterBackend]
    filterset_class = FloorPlanOrganizationFilter
```

You can also use the organization filter classes such as `OrganizationManagedFilter` from `openwisp_users.api.filters` which includes `organization` and `organization_slug` filter fields by default.

Usage example:

```
from django_filters import rest_framework as filters
from openwisp_users.api.filters import OrganizationManagedFilter
from ..models import FloorPlan


class FloorPlanFilter(OrganizationManagedFilter):
    class Meta(OrganizationManagedFilter.Meta):
        model = FloorPlan


class FloorPlanListCreateView(
    ProtectedAPIMixin, generics.ListCreateAPIView
```

```
):
    serializer_class = FloorPlanSerializer
    queryset = FloorPlan.objects.select_related().order_by("-created")
    pagination_class = ListViewPagination
    filter_backends = [filters.DjangoFilterBackend]
    filterset_class = FloorPlanFilter
```

Miscellaneous Utilities

Note

This page is for developers who want to customize or extend OpenWISP Users, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Users Usage Docs

This section covers miscellaneous utilities provided by the OpenWISP Users module.

Organization Membership Helpers

The `User` model offers methods to efficiently check whether the user is a member, manager, or owner of an organization.

Use these methods to distinguish between different user roles across organizations and minimize database queries.

```
import swapper

User = swapper.load_model("openwisp_users", "User")
Organization = swapper.load_model("openwisp_users", "Organization")

user = User.objects.first()
org = Organization.objects.first()
user.is_member(org)
user.is_manager(org)
user.is_owner(org)

# Also valid (avoids query to retrieve Organization instance)
device = Device.objects.first()
user.is_member(device.organization_id)
```

```
user.is_manager(device.organization_id)
user.is_owner(device.organization_id)
```

`is_member(org)`

Returns `True` if the user is a member of the specified `Organization` instance. Alternatively, you can pass a `UUID` or `str` representing the organization's primary key, which allows you to avoid an additional database query to fetch the organization instance.

Use this check to grant access to end-users who need to consume services offered by organizations they're members of, such as authenticating to public WiFi services.

`is_manager(org)`

Returns `True` if the user is a member of the specified `Organization` instance and has the `OrganizationUser.is_admin` field set to `True`. Alternatively, you can pass a `UUID` or `str` representing the organization's primary key, which allows you to avoid an additional database query to fetch the organization instance.

Use this check to grant access to managers of organizations, who need to perform administrative tasks such as creating, editing, or deleting objects of their organization, or accessing sensitive information like firmware images.

`is_owner(org)`

Returns `True` if the user is a member of the specified `Organization` instance and is the owner of the organization, checked against the presence of an `OrganizationOwner` instance for the user. Alternatively, you can pass a `UUID` or `str` representing the organization's primary key, which allows you to avoid an additional database query to fetch the organization instance.

Use this check to prevent managers from taking control of organizations without the original owner's consent.

`organizations_dict`

The methods described above utilize the `organizations_dict` property method, which builds a dictionary containing the primary keys of organizations the user is a member of, along with information about whether the user is a manager (`is_admin`) or owner (`is_owner`).

This data structure is cached automatically to prevent multiple database queries across multiple requests.

The cache is automatically invalidated on the following events:

- An `OrganizationUser` is added, changed, or deleted.
- An `OrganizationOwner` is added, changed, or deleted.
- The `is_active` field of an `Organization` changes.

Usage example:

```
>>> user.organizations_dict
... {'20135c30-d486-4d68-993f-322b8acb51c4': {'is_admin': True, 'is_owner': False}}
>>> user.organizations_dict.keys()
... dict_keys(['20135c30-d486-4d68-993f-322b8acb51c4'])
```

`organizations_managed`

Returns a list of primary keys of organizations the user can manage.

Usage example:

```
>>> user.organizations_managed
... ['20135c30-d486-4d68-993f-322b8acb51c4']
```

`organizations_owned`

Returns a list of primary keys of organizations the user owns.

Usage example:

```
>>> user.organizations_owned
... ['20135c30-d486-4d68-993f-322b8acb51c4']
```

`UsersAuthenticationBackend`

Full python path: `openwisp_users.backends.UsersAuthenticationBackend`.

This authentication backend enables users to authenticate using their email or phone number, as well as their username. Email authentication takes precedence over the username, while phone number authentication takes precedence if the identifier passed as argument is a valid phone number.

Phone numbers are parsed using the phonenumbers library, ensuring recognition even if users include characters like spaces, dots, or dashes.

The OPENWISP_USERS_AUTH_BACKEND_AUTO_PREFIXES setting allows specifying a list of international prefixes that can be automatically prepended to the username string, enabling users to log in without typing the international prefix.

Additionally, the backend supports phone numbers with a leading zero, ensuring successful authentication even with the leading zero included.

You can also use the backend programmatically:

```
from openwisp_users.backends import UsersAuthenticationBackend

backend = UsersAuthenticationBackend()
backend.authenticate(request, identifier, password)
```

`PasswordExpirationMiddleware`

Full python path: `openwisp_users.middleware.PasswordExpirationMiddleware`.

When the password expiration feature is enabled (see OPENWISP_USERS_USER_PASSWORD_EXPIRATION and OPENWISP_USERS_STAFF_USER_PASSWORD_EXPIRATION), this middleware restricts users to the *password change view* until they change their password.

Ensure this middleware follows `AuthenticationMiddleware` and `MessageMiddleware`:

```
# in your_project/settings.py
MIDDLEWARE = [
    # Other middlewares
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "openwisp_users.middleware.PasswordExpirationMiddleware",
]
```

`PasswordReuseValidator`

Full python path: `openwisp_users.password_validation.PasswordReuseValidator`.

On password change views, this validator ensures users cannot reuse their current password as the new password.

Add the validator to the `AUTH_PASSWORD_VALIDATORS` Django setting:

```
# in your-project/settings.py
AUTH_PASSWORD_VALIDATORS = [
    # Other password validators
    {
        "NAME": "openwisp_users.password_validation.PasswordReuseValidator",
```

```
    },
]
```

Extending OpenWISP Users

Note

This page is for developers who want to customize or extend OpenWISP Users, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Users Usage Docs

One of the core values of the OpenWISP project is Software Reusability, which ensures long-term sustainability. For this reason, *OpenWISP Users* provides a set of base classes that can be imported, extended, and reused to create derivative apps.

This is extremely beneficial if you want to add additional fields to the User model, such as requesting a Social Security Number during registration.

To implement your custom version of *OpenWISP Users*, follow the steps described in this section.

If you have any doubts, refer to the code in the test project and the sample app. These resources will serve as your source of truth: replicate and adapt that code to get a basic derivative of *OpenWISP Users* working.

Important

If you plan on using a customized version of this module, we suggest to start with it since the beginning, because migrating your data from the default module to your extended version may be time consuming.

1. Initialize Your Custom Module

The first thing you need to do is create a new Django app which will contain your custom version of *OpenWISP Users*.

A Django app is nothing more than a Python package (a directory of Python scripts). In the following examples, we'll call this Django app `myusers`, but you can name it however you like:

```
django-admin startapp myusers
```

Keep in mind that the command mentioned above must be called from a directory that is available in your PYTHON_PATH so that you can then import the result into your project.

Now you need to add `myusers` to `INSTALLED_APPS` in your `settings.py`, ensuring also that `openwisp_users` has been removed:

```
INSTALLED_APPS = [
    # ... other apps ...
    # 'openwisp_users'  <-- comment out or delete this line
    "myusers"
]
```

For more information about how to work with Django projects and Django apps, please refer to the Django documentation.

2. Install OpenWISP Users

Install (and add to the requirements of your project) openwisp-users:

```
pip install openwisp-users
```

3. Add `EXTENDED_APPS`

Add the following to your `settings.py`:

```
EXTENDED_APPS = ("openwisp_users",)
```

4. Add `openwisp_utils.staticfiles.DependencyFinder`

Add `openwisp_utils.staticfiles.DependencyFinder` to `STATICFILES_FINDERS` in your `settings.py`:

```
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
    "openwisp_utils.staticfiles.DependencyFinder",
]
```

5. Add `openwisp_utils.loaders.DependencyLoader`

Add `openwisp_utils.loaders.DependencyLoader` to `TEMPLATES` before `django.template.loaders.app_directories.Loader` in your `settings.py`:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "OPTIONS": {
            "loaders": [
                "django.template.loaders.filesystem.Loader",
                "openwisp_utils.loaders.DependencyLoader",
                "django.template.loaders.app_directories.Loader",
            ],
            "context_processors": [
```

```
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    }
]
```

6. Inherit the AppConfig Class

Please refer to the following files in the sample app of the test project:

- openwisp_users/__init__.py
- openwisp_users/apps.py

You have to replicate and adapt that code in your project.

For more information regarding the concept of `AppConfig` please refer to the "Applications" section in the django documentation.

7. Create Your Custom Models

For the purpose of showing an example, we added a simple `social_security_number` field in the User model to the models of the sample app in the test project.

You can add fields in a similar way in your `models.py` file.

For doubts regarding how to use, extend, or develop models please refer to the "Models" section in the django documentation.

8. Add Swapper Configurations

Once you have created the models, add the following to your `settings.py`:

```
# Setting models for swapper module
AUTH_USER_MODEL = "myusers.User"
OPENWISP_USERS_GROUP_MODEL = "myusers.Group"
OPENWISP_USERS_ORGANIZATION_MODEL = "myusers.Organization"
OPENWISP_USERS_ORGANIZATIONUSER_MODEL = "myusers.OrganizationUser"
OPENWISP_USERS_ORGANIZATIONOWNER_MODEL = "myusers.OrganizationOwner"
# The following model is not used in OpenWISP yet
# but users are free to implement it in their projects if needed
# for more information refer to the django-organizations docs:
# https://django-organizations.readthedocs.io/
OPENWISP_USERS_ORGANIZATIONINVITATION_MODEL = (
    "myusers.OrganizationInvitation"
)
```

Substitute `myusers` with the name you chose in step 1.

9. Create Database Migrations

Create database migrations:

```
./manage.py makemigrations
```

Now, manually create a file `0004_default_groups.py` in the migrations directory just created by the `makemigrations` command and copy the contents of the sample_users/migrations/0004_default_groups.py.

Then, run the migrations:

```
./manage.py migrate
```

> **Note**
>
> The `0004_default_groups` is required because other OpenWISP modules depend on it. If it's not created as documented here, the migrations of other OpenWISP modules will fail.

10. Create the admin

Refer to the admin.py file of the sample app.

To introduce changes to the admin, you can do it in two main ways which are described below.

For more information regarding how the Django admin works, or how it can be customized, please refer to "The Django admin site" section in the Django documentation.

1. Monkey Patching

If the changes you need to add are relatively small, you can resort to monkey patching.

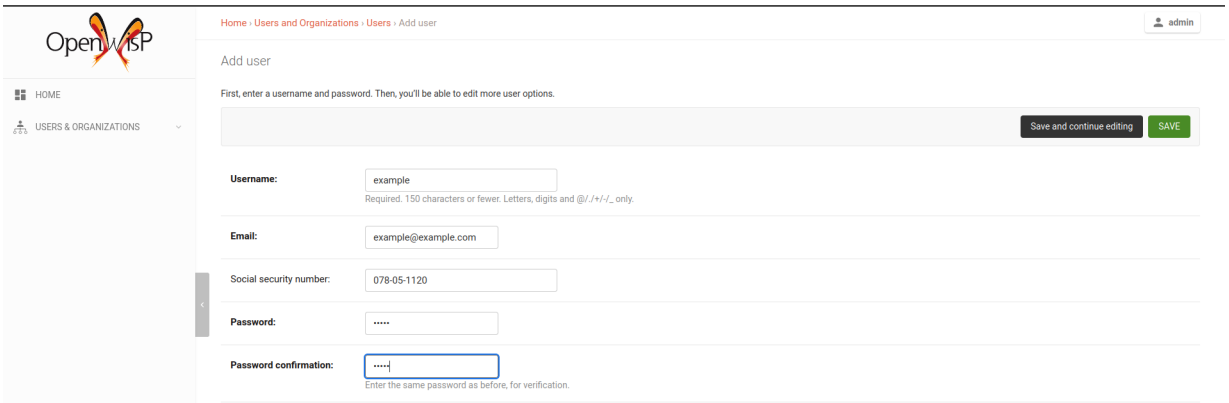
For example:

```
from openwisp_users.admin import (
    UserAdmin,
    GroupAdmin,
    OrganizationAdmin,
    OrganizationOwnerAdmin,
    BaseOrganizationUserAdmin,
)

# OrganizationAdmin.field += ['example_field'] <-- Monkey patching changes example
```

For your convenience in adding fields in User forms, we provide the following functions:

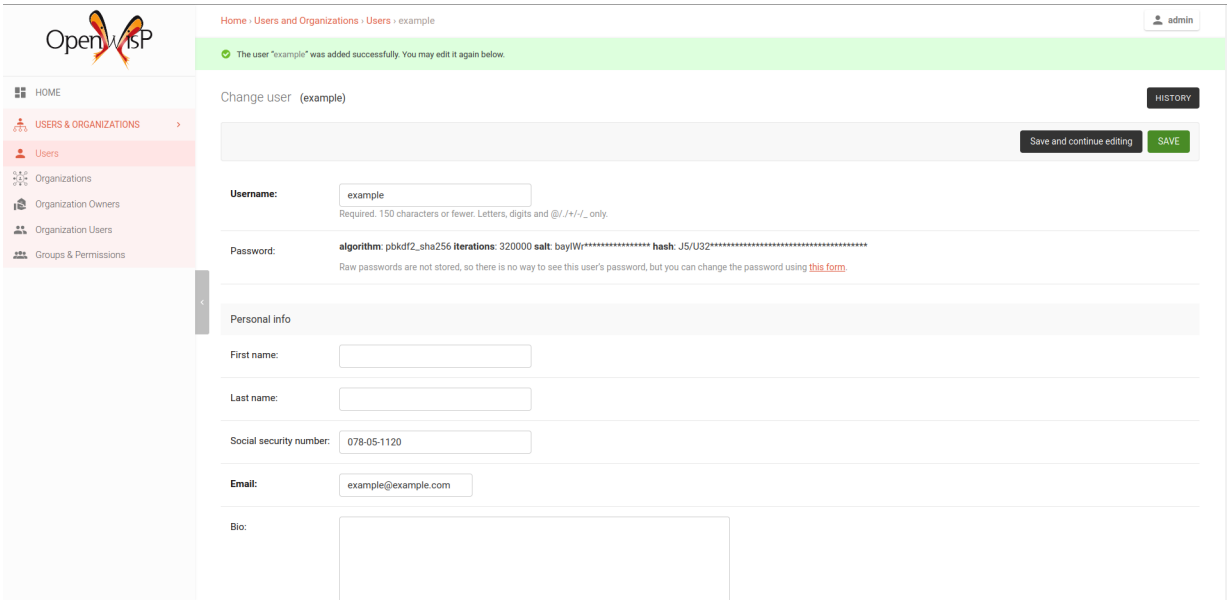
`usermodel_add_form`

When monkey patching the `UserAdmin` class to add fields in the "Add user" form, you can use this function. In the example, Social Security Number is added in the add form:



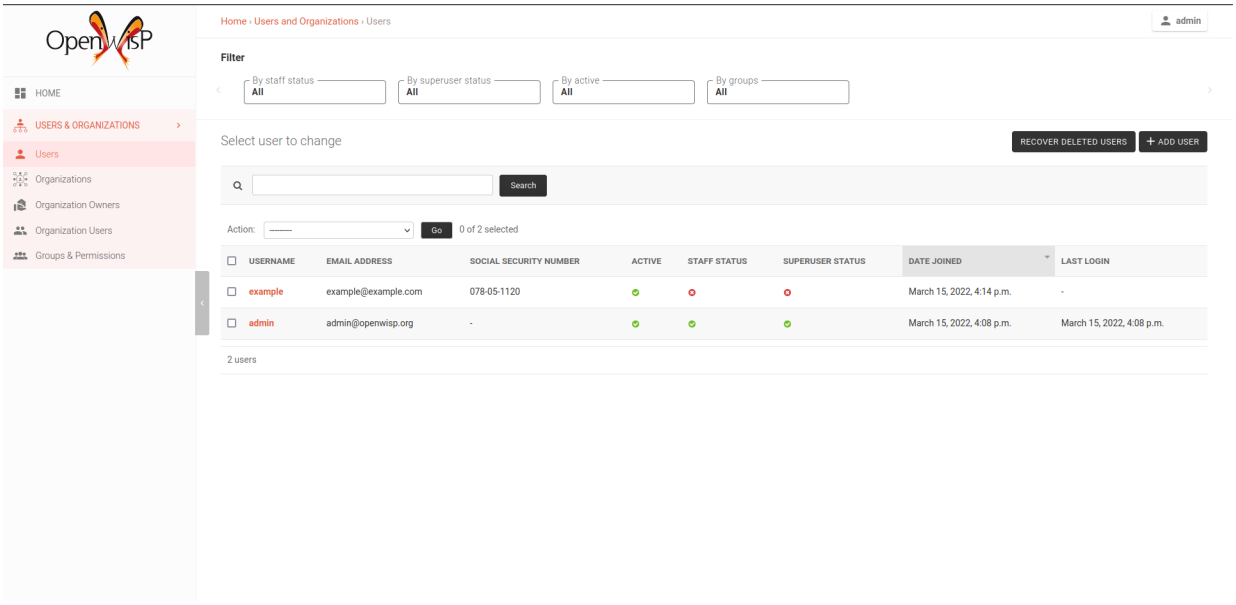
`usermodel_change_form`

When monkey patching the `UserAdmin` class to add fields in the "Change user" form to change/modify the user form's profile section, you can use this function. In the example, Social Security Number is added in the change form:



`usermodel_list_and_search`

When monkey patching the `UserAdmin` class, you can use this function to make a field searchable and add it to the user display list view. In the example, Social Security Number is added in the changelist view:



2. Inheriting Admin Classes

If you need to introduce significant changes and/or you don't want to resort to monkey patching, you can proceed as follows:

```
from django.contrib import admin
from openwisp_users.admin import (
    UserAdmin as BaseUserAdmin,
    GroupAdmin as BaseGroupAdmin,
    OrganizationAdmin as BaseOrganizationAdmin,
    OrganizationOwnerAdmin as BaseOrganizationOwnerAdmin,
    OrganizationUserAdmin as BaseOrganizationUserAdmin,
)
from swapper import load_model
from django.contrib.auth import get_user_model
```

```
Group = load_model("openwisp_users", "Group")
Organization = load_model("openwisp_users", "Organization")
OrganizationOwner = load_model("openwisp_users", "OrganizationOwner")
OrganizationUser = load_model("openwisp_users", "OrganizationUser")
User = get_user_model()

admin.site.unregister(Group)
admin.site.unregister(Organization)
admin.site.unregister(OrganizationOwner)
admin.site.unregister(OrganizationUser)
admin.site.unregister(User)


@admin.register(Group)
class GroupAdmin(BaseGroupAdmin):
    pass


@admin.register(Organization)
class OrganizationAdmin(BaseOrganizationAdmin):
    pass


@admin.register(OrganizationOwner)
class OrganizationOwnerAdmin(BaseOrganizationOwnerAdmin):
    pass


@admin.register(OrganizationUser)
class OrganizationUserAdmin(BaseOrganizationUserAdmin):
    pass


@admin.register(User)
class UserAdmin(BaseUserAdmin):
    pass
```

11. Create Root URL Configuration

Please refer to the urls.py file in the sample project.

For more information about URL configuration in Django, please refer to the "URL dispatcher" section in the Django documentation.

12. Import the Automated Tests

When developing a custom application based on this module, it's a good idea to import and run the base tests too, so that you can be sure the changes you're introducing are not breaking some of the existing features of *OpenWISP Users*.

In case you need to add breaking changes, you can overwrite the tests defined in the base classes to test your own behavior.

See the tests of the sample app to find out how to do this.

You can then run tests with:

```
# the --parallel flag is optional
./manage.py test --parallel myusers
```

Substitute `myusers` with the name you chose in step 1.

Other Base Classes that can be Inherited and Extended

The following steps are not required and are intended for more advanced customization.

Extending the API Views

The API view classes can be extended into other Django applications as well. Note that it is not required for extending *OpenWISP Users* to your app and this change is required only if you plan to make changes to the API views.

Create a view file as done in API views.py.

Remember to use these views in root URL configurations in point 11.

For more information about Django views, please refer to the views section in the Django documentation.

Other useful resources:

- REST API
- Settings

Controller

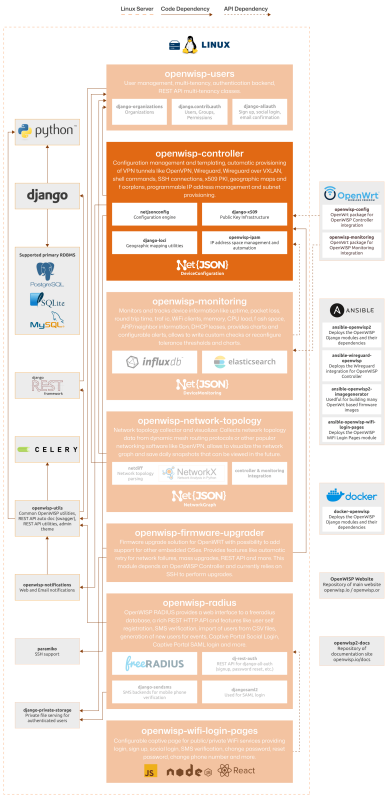
Seealso

Source code: github.com/openwisp/openwisp-controller.

OpenWISP Controller is responsible of of managing the core resources of the network and allows automating several aspects like adoption, provisioning, VPN tunnel configuration, generation of X509 certificates, subnet and IP address allocation and more.

For a full introduction please refer to Controller: Structure & Features.

The following diagram illustrates the role of the Controller module within the OpenWISP architecture.



OpenWISP Architecture: highlighted controller module

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

Controller: Structure & Features

OpenWISP Controller is a Python package which ships five Django apps.

Config App

The config app is the core of the controller module and implements all the following features:

- **Configuration management for embedded devices supporting:**
    - OpenWrt
    - OpenWISP Firmware
    - additional firmware can be added by specifying custom configuration backends
- **Configuration editor** based on JSON-Schema editor
- **Advanced edit mode**: edit NetJSON *DeviceConfiguration* objects for maximum flexibility
- Configuration Templates: reduce repetition to the minimum, configure default and required templates
- Configuration Variables: reference variables in the configuration and templates
- Device Groups: define different set of default configuration and metadata in device groups
- Template Tags: define different sets of default templates (e.g.: mesh, WDS, 4G)
- **HTTP resources**: allow devices to automatically check for and download configuration updates
- **VPN management**: automatically provision VPN tunnel configurations, including cryptographic keys and IP addresses, e.g.: OpenVPN, WireGuard
- Import/Export Device Data

It exposes various REST API endpoints.

PKI App

The PKI app is based on django-x509, allowing you to create, import, and view x509 CAs and certificates directly from the administration dashboard.

It exposes various REST API endpoints.

Connection App

This app enables OpenWISP Controller to use different protocols to reach network devices. Currently, the default connection protocols are SSH and SNMP, but the protocol mechanism is extensible, allowing for implementation of additional protocols if needed.

It exposes various REST API endpoints.

SSH

The SSH connector allows the controller to initialize connections to the devices in order to perform push operations, e.g.:

- Sending configuration updates.
- Executing shell commands.
- Perform firmware upgrades via the additional firmware upgrade module.

The default connection protocol implemented is SSH, but other protocol mechanism is extensible and custom protocols can be implemented as well.

Access via SSH key is recommended, the SSH key algorithms supported are:

- RSA
- Ed25519

SNMP

The SNMP connector is useful to collect monitoring information and it's used in OpenWISP Monitoring for performing checks to collect monitoring information. Read more on how to use it.

Geo App

The geographic app is based on django-loci and allows to define the geographic coordinates of the devices, as well as their indoor coordinates on floor plan images.

It exposes various REST API endpoints.

Subnet Division App

> **Note**
>
> This app is optional, if you don't need it you can avoid adding it to `settings.INSTALLED_APPS`.

This app allows to automatically provision subnets and IP addresses which will be available as system defined configuration variables that can be used in Configuration Templates.

The purpose of this app is to allow users to automatically provision and configure specific subnets and IP addresses to the devices without the need of manual intervention.

Refer to Automating Subnet and IP Address Provisioning for more information.

Device Configuration Status

The device's configuration status (*Device.config.status*) indicates the current state of the configuration as managed by OpenWISP. The possible statuses and their meanings are explained below.

`modified`

The device configuration has been updated in OpenWISP, but these changes have not yet been applied to the device. The device is pending an update.

`applied`

The device has successfully applied the configuration changes made in OpenWISP. The current configuration on the device matches the latest changes.

`error`

An issue occurred while applying the configuration to the device, causing the device to revert to its previous working configuration.

`deactivating`

The device is in the process of being deactivated. The configuration is scheduled to be removed from the device.

`deactivated`

The device has been deactivated. The configuration applied through OpenWISP has been removed, and any other operation to manage the device will be prevented or rejected.

Configuration Templates

What is a Template?

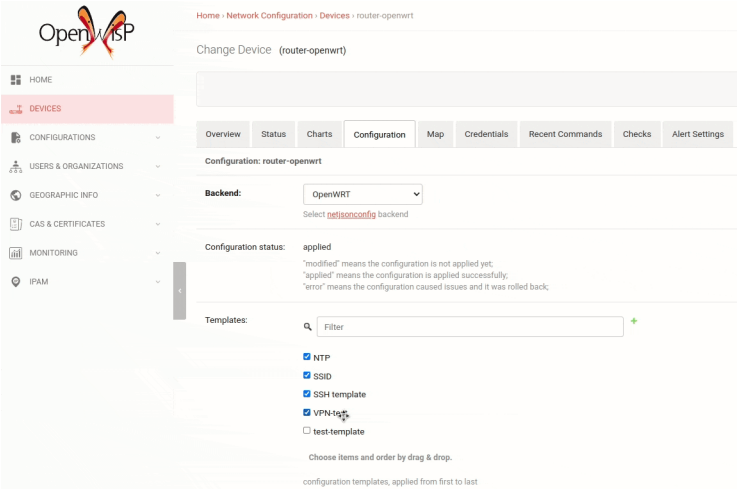
Templates are designed to store configuration that can be reused by some or all the devices in the system.

Updating the configuration stored in a template allows to update the configuration of all the devices that have that template assigned.

This means that configuration can be defined only once for multiple devices, and if the need to update a specific piece of configuration arises, it can be easily achieved by updating the template.

Template Ordering and Override

A device can use multiple templates, **the order in which templates are assigned to each device matters**: templates assigned last can override templates assigned earlier, the order can be changed by drag and dropping the template in the device configuration page as in the animated screenshot below.



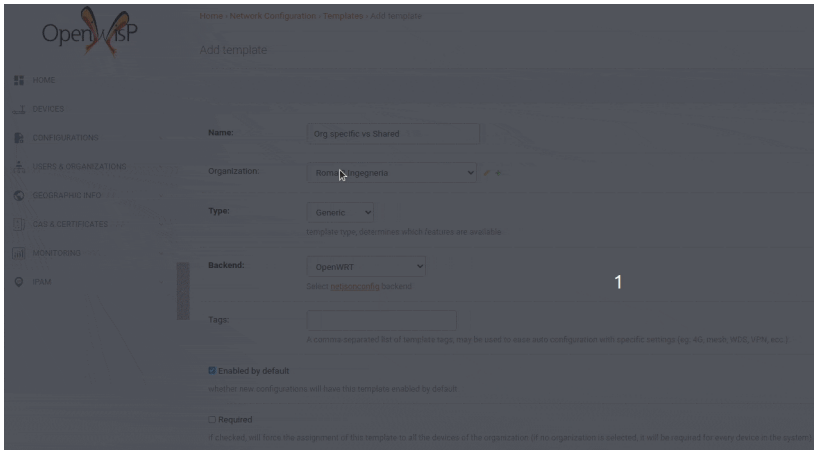
The device configuration can also override what is defined in templates.

Overriding means redefining a specific configuration section in a way that overwrites the template.

Overriding involves some form of duplication of information, which is not great, it should be used as a last resort. The recommended way to define parts of the configuration that are specific for each device is to use Configuration variables.

Shared Templates vs Organization Specific

Templates can be *organization specific* or *shared* (no organization specified).



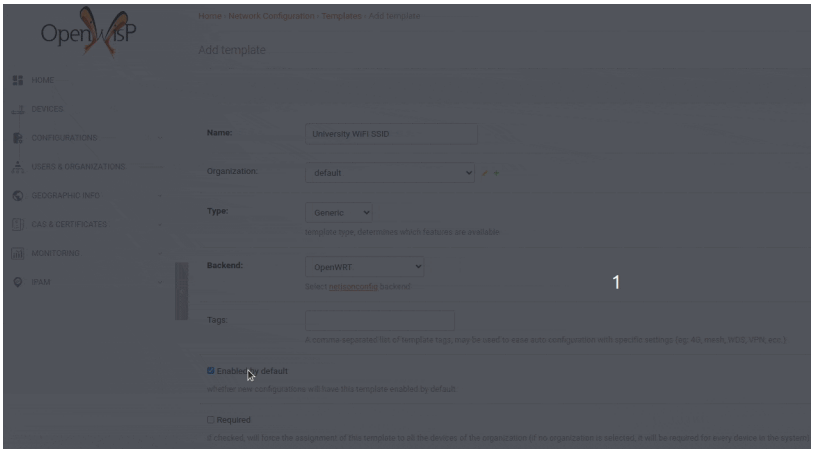
Organization specific templates will be available and usable only within the same organization which they are assigned to.

If no organization is specified when creating a template, a shared template will be created, **shared templates are available to any organization in the system**.

Here are a few typical use cases of shared templates:

- Management VPN
- Authorized SSH keys belonging to network administrators
- Crontab with generic periodic management operations

Default Templates



When templates are flagged as **"Enabled by default"**, they will be automatically assigned to new devices.

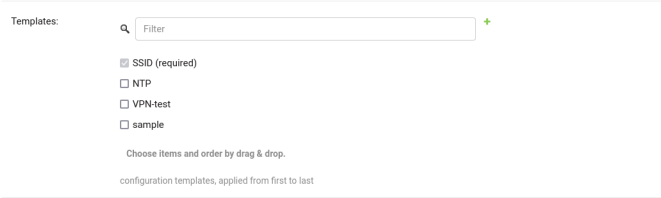
This is a very powerful feature: **once default templates are correctly configured to implement the use case you need, you will only have to register a device into OpenWISP for it to auto-configure itself**.

Moreover, you can change the default templates any time you need, which is the reason this feature has replaced the practice of storing default configuration in firmware images (which would need to be recompiled and redistributed): with default templates, the default firmware image only needs to contain the bare minimum configuration to connect to OpenWISP, once the device connects to OpenWISP it will download and apply the default templates without the need of manual intervention from the network operators.

An organization specific template flagged as default will be automatically assigned to any new device which will be created in the same organization.

A shared default template instead will be automatically assigned to all the new devices which will be created in the system, regardless of organization.

Required Templates



Required templates are similar to Default Templates but cannot be unassigned from a device configuration, they can only be overridden.

They will be always assigned earlier than default templates, so they can be overridden if needed.

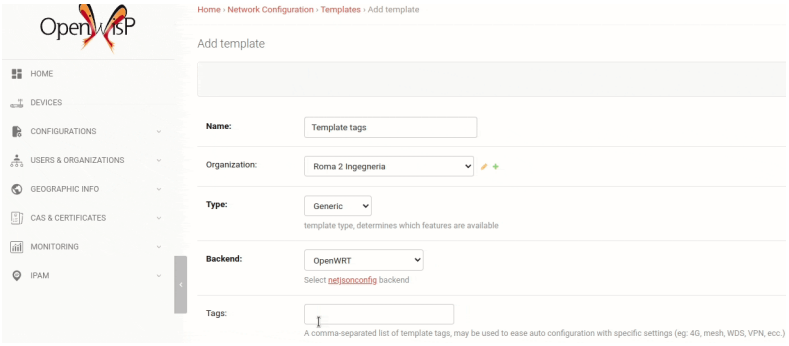
In the example above, the "SSID" template is flagged as "(required)" and its checkbox is always checked and disabled.

Device Group Templates

Default Templates are an incredibly useful tool, but they're limited: **only one set of default templates can be created** per each organization.

With Group Templates it is possible to specify a set of default templates for each device group.

Template Tags



In some cases, you may have multiple set of default settings to use, let's explain this with a practical example: you may have 2 different device types in your network:

- Mesh routers: they connect to one another, forming a wireless mesh network
- Dumb access points: they connect to the mesh routers on the LAN port and offer internet access which is routed via the mesh network by the routers

In this example case, the default configuration to use in each device type can greatly differ.

In such a setup, default templates would only contain configuration which is common to both device types, while configuration which is specific for each type would be stored in specific templates which are then tagged with specific keywords:

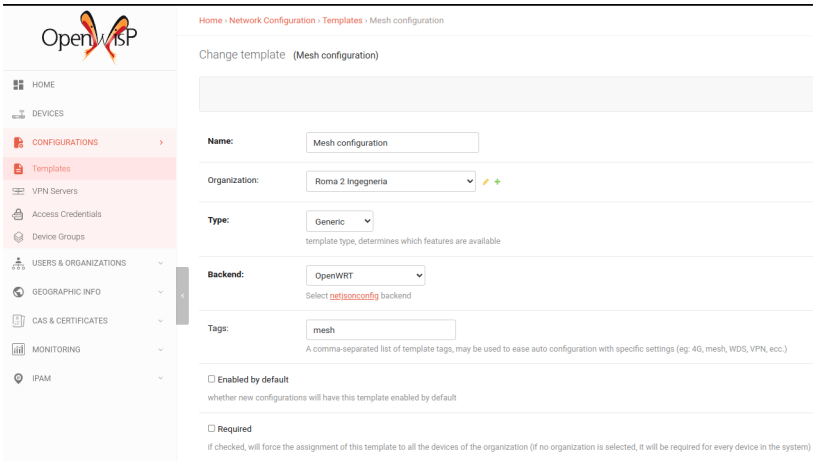
- `mesh`: tag to use for mesh configuration
- `dumb-ap`: tag to use for dumb AP configuration

The openwisp-config configuration of each device type must specify the correct tag before each device registers in the system.

Here's the sample `/etc/config/openwisp` configuration for mesh devices:

```
config controller 'http'
    option url 'https://openwisp2.mynetwork.com'
    option shared_secret 'mySharedSecret123'
    option tags 'mesh'
```

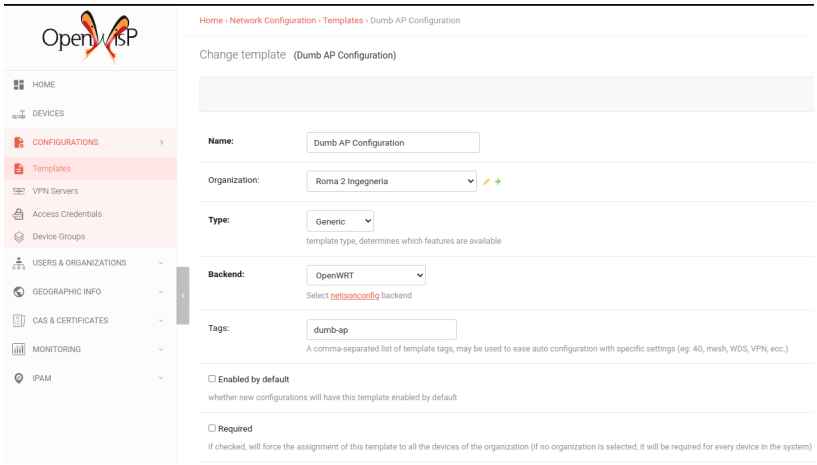
Once devices with the above configuration will register into the system, any template tagged as `mesh` (as in the screenshot below) will be assigned to them.



The sample `/etc/config/openwisp` configuration for dumb access points is the following:

```
config controller 'http'
    option url 'https://openwisp2.mynetwork.com'
    option shared_secret 'mySharedSecret123'
    option tags 'dumb-ap'
```

Once devices with the above configuration will register into the system, any template tagged as `dumb-ap` (as in the screenshot below) will be assigned to them.



Implementation Details of Templates

Templates are implemented under the hood by the OpenWISP configuration engine: netjsonconfig.

For more advanced technical information about templates, consult the netjsonconfig documentation: Basic Concepts, Template.

Configuration Variables

Sometimes the configuration is not exactly equal on all the devices, some parameters are unique to each device or need to be changed by the user.

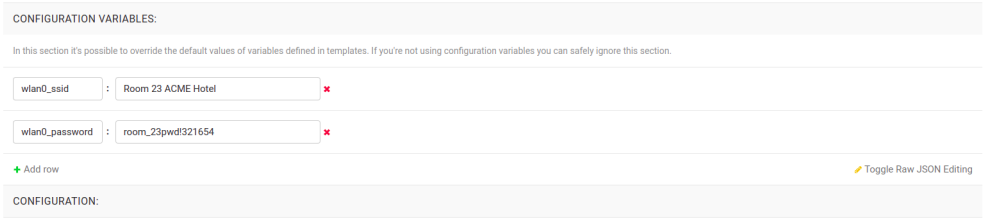
In these cases it is possible to use configuration variables in conjunction with templates, this feature is also known as *configuration context*, think of it like a dictionary which is passed to the function which renders the configuration, so that it can fill variables according to the passed context.

Different Types of Variables

The different ways in which variables are defined are described below in the order (high to low) of their precedence.

1. User Defined Device Variables

In the device configuration section you can find a section named "Configuration variables" where it is possible to define the configuration variables and their values, as shown in the example below:



2. Predefined Device Variables

Each device gets the following attributes passed as configuration variables:

- `id`
- `key`
- `name`
- `mac_address`

3. Group Variables

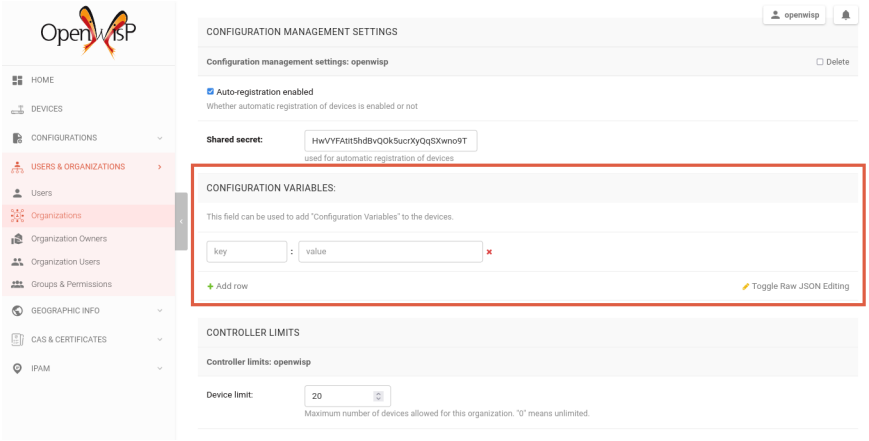
Variables can also be defined in Device Groups.

Refer to Group Configuration Variables for more information.

4. Organization Variables

Variables can also be defined at the organization level.

You can set the *organization variables* from the organization change page `/admin/openwisp_users/organization/<organization-id>/change/`, under the **Configuration Management Settings**.



5. Global Variables

Variables can also be defined globally using the OPENWISP_CONTROLLER_CONTEXT setting, see also How to Edit Django Settings.

6. Template Default Values

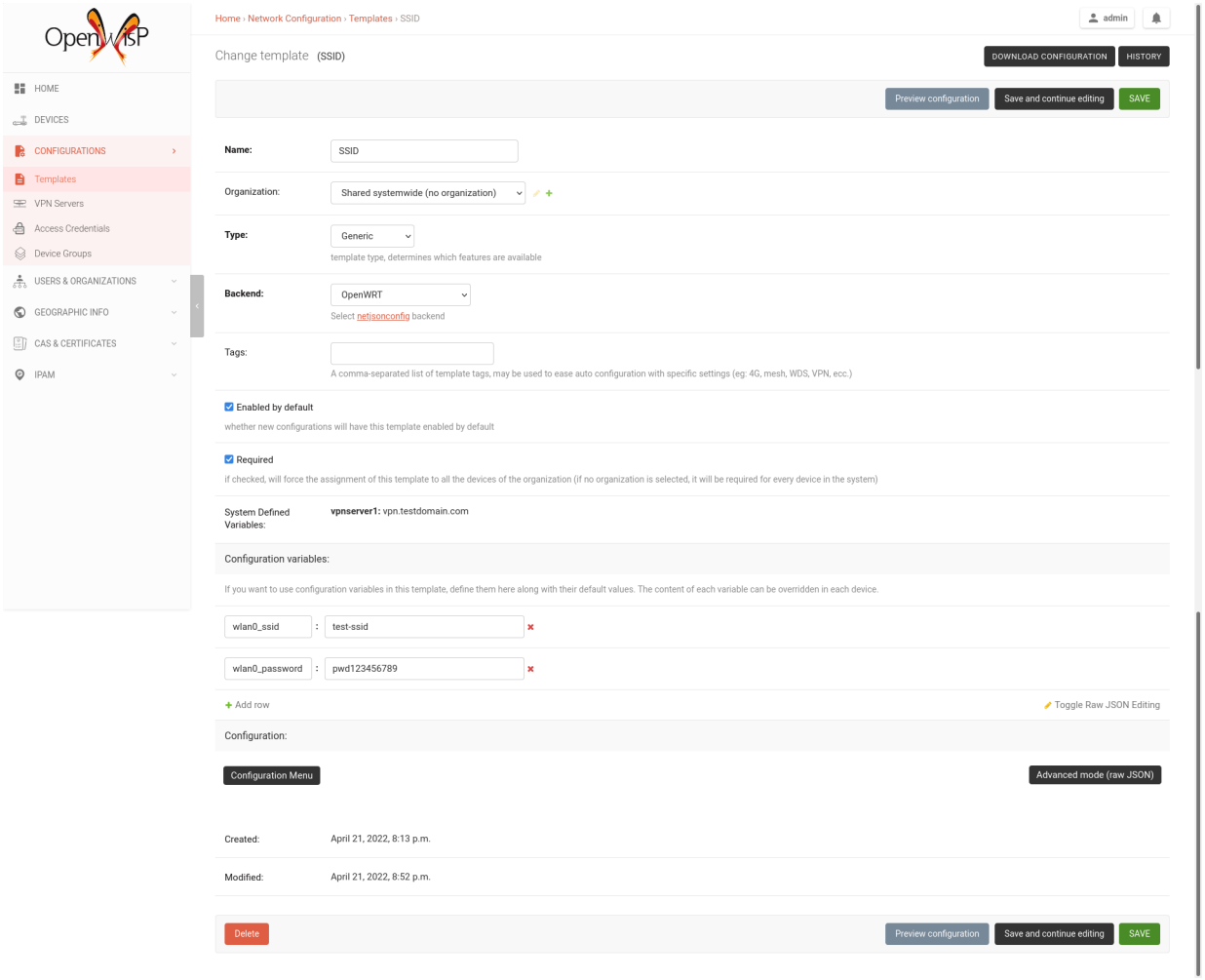
It's possible to specify the default values of variables defined in a template.

This allows to achieve 2 goals:

1. pass schema validation without errors (otherwise it would not be possible to save the template in the first place)
2. provide good default values that are valid in most cases but can be overridden in the device if needed

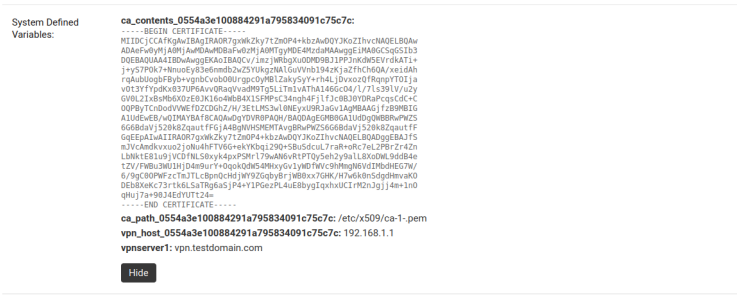
These default values will be overridden by the User defined device variables.

The default values of variables can be manipulated from the section "configuration variables" in the edit template page:



7. System Defined Variables

Predefined device variables, global variables and other variables that are automatically managed by the system (e.g.: when using templates of type VPN-client) are displayed in the admin UI as *System Defined Variables* in read-only mode.



Example Usage of Variables

Here's a typical use case, the WiFi SSID and WiFi password. You don't want to define this for every device, but you may want to allow operators to easily change the SSID or WiFi password for a specific device without having to re-define the whole wifi interface to avoid duplicating information.

This would be the template:

```
{
    "interfaces": [
        {
            "type": "wireless",
```

```
                "name": "wlan0",
                "wireless": {
                    "mode": "access_point",
                    "radio": "radio0",
                    "ssid": "{{wlan0_ssid}}",
                    "encryption": {
                        "protocol": "wpa2_personal",
                        "key": "{{wlan0_password}}",
                        "cipher": "auto"
                    }
                }
            }
        ]
}
```

These would be the default values in the template:

```
{
    "wlan0_ssid": "SnakeOil PublicWiFi",
    "wlan0_password": "Snakeoil_pwd!321654"
}
```

The default values can then be overridden at device level if needed, e.g.:

```
{
    "wlan0_ssid": "Room 23 ACME Hotel",
    "wlan0_password": "room_23pwd!321654"
}
```

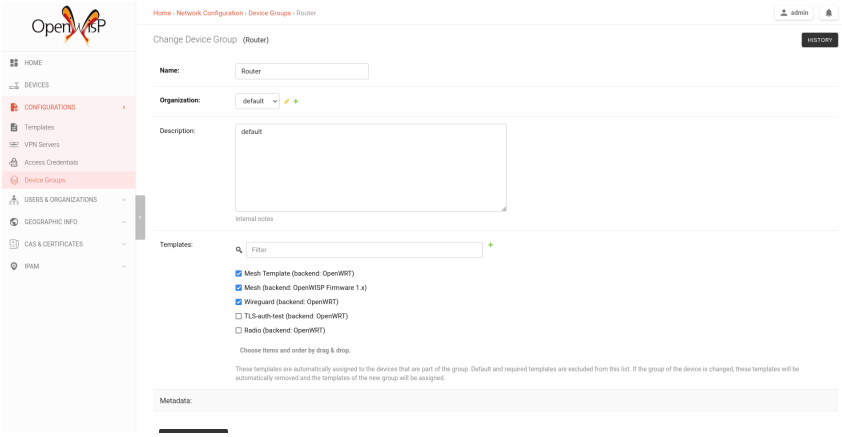
Implementation Details of Variables

Variables are implemented under the hood by the OpenWISP configuration engine: netjsonconfig.

For more advanced technical information about variables, consult the netjsonconfig documentation: Basic Concepts, Context (configuration variables).

Device Groups

Device groups allow to group similar devices together, the groups usually share not only a common characteristic but also some kind of organizational need: they need to have specific configuration templates, variables and/or associated metadata which differs from the rest of the network.



Group Templates

Groups allow to define templates which are automatically assigned to devices belonging to the group. When using this feature, keep in mind the following important points:

- Templates of any configuration backend can be selected, when a device is assigned to a group, only the templates which matches the device configuration backend are applied to the device.
- The system will not force group templates onto devices, this means that users can remove the applied group templates from a specific device if needed.
- If a device group is changed, the system will automatically remove the group templates of the old group and apply the new templates of the new group (this operation is implemented by leveraging the group_templates_changed signal).
- If the group templates are changed, the devices which belong to the group will be automatically updated to reflect the changes (this operation is executed in a background task).
- In case the configuration backend of a device is changed, the system will handle this automatically too and update the group templates accordingly (this operation is implemented by leveraging the config_backend_changed signal).
- If a device does not have a configuration defined yet, but it is assigned to a group which has templates defined, the system will automatically create a configuration for it using the default backend specified in the OPENWISP_CONTROLLER_DEFAULT_BACKEND setting.

Note: the list of templates shown in the edit group page do not contain templates flagged as "default" or "required" to avoid redundancy because those templates are automatically assigned by the system to new devices.

This feature works also when editing group templates or the group assigned to a device via the REST API.

Group Configuration Variables

Groups allow to define configuration variables which are automatically added to the device's context in the **System Defined Variables**. Check the Configuration Variables section to learn more about precedence of different configuration variables.

This feature also works when editing group templates or the group assigned to a device via the REST API.

Group Metadata

Groups allow to store additional information regarding a group in the structured metadata field (which can be accessed via the REST API).

The metadata field allows custom structure and validation to standardize information across all groups using the OPENWISP_CONTROLLER_DEVICE_GROUP_SCHEMA setting.

Variables vs Metadata

Group configuration variables and *Group metadata* serves different purposes.

The group configuration variables should be used when the device configuration is required to be changed for particular group of devices.

Group metadata should be used to store additional data for the device group, this data can be fetched and/or tweaked via the REST API if needed. Group metadata is not designed to be used for configuration purposes.

Configuring Push Operations

Introduction

Important

If you have installed OpenWISP with one of the Official installers you can skip the following steps, which are handled automatically during the first installation.

The Ansible role automatically creates a default template to update `authorized_keys` on networking devices using the default access credentials.

Follow the procedure described below to enable secure SSH access from OpenWISP to your devices, this is required to enable push operations (whenever the configuration is changed, OpenWISP will trigger the update in the background) and/or firmware upgrades (via the additional module openwisp-firmware-upgrader).

1. Generate SSH Key

First of all, we need to generate the SSH key which will be used by OpenWISP to access the devices, to do so, you can use the following command:

```
ssh-keygen -f ./sshkey -t ed25519 -C "openwisp" -N ""
```

This will create two files in the current directory, one called `sshkey` (the private key) and one called `sshkey.pub` (the public key).

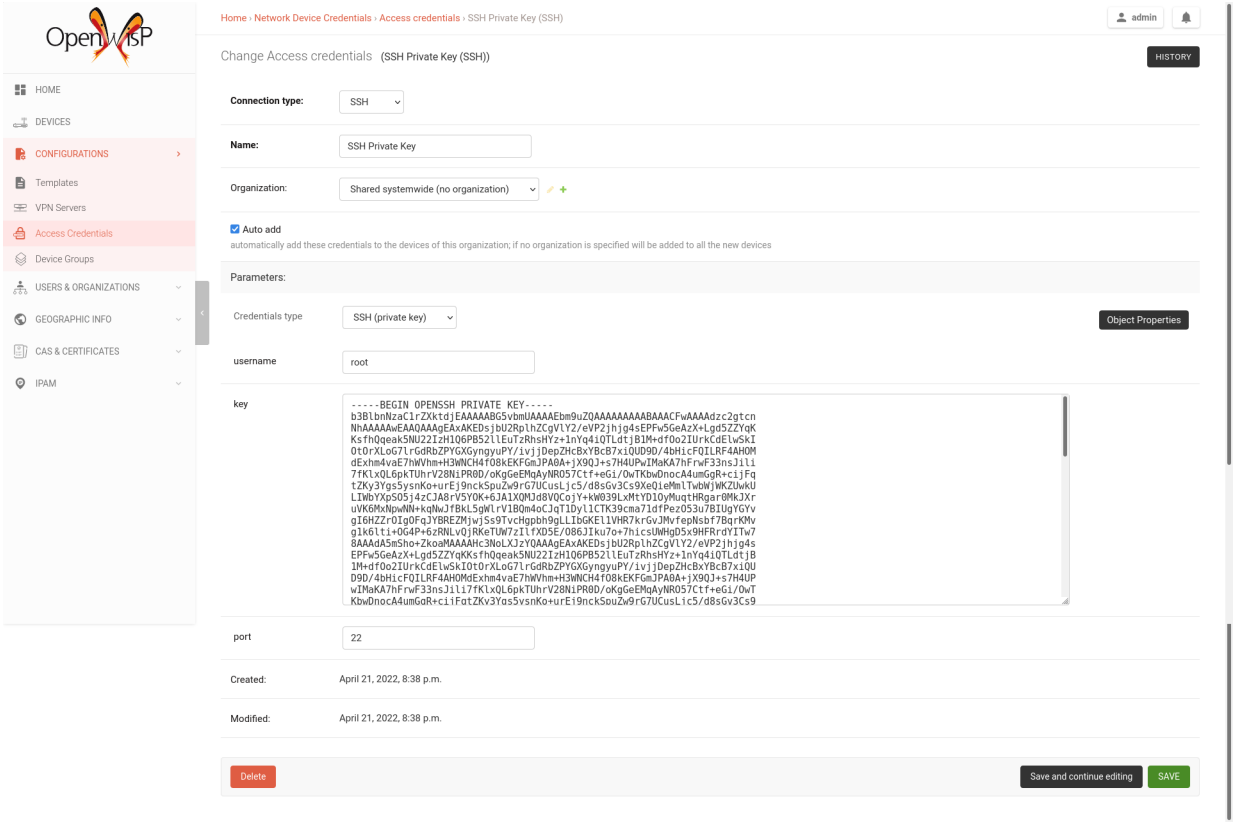
Store the content of these files in a secure location.

Note

Support for **ED25519** was added in OpenWrt 21.02 (requires Dropbear > 2020.79). If you are managing devices with OpenWrt < 21, then you will need to use RSA keys:

```
ssh-keygen -f ./sshkey -t rsa -b 4096 -C "openwisp"
```

2. Save SSH Private Key in "Access Credentials"



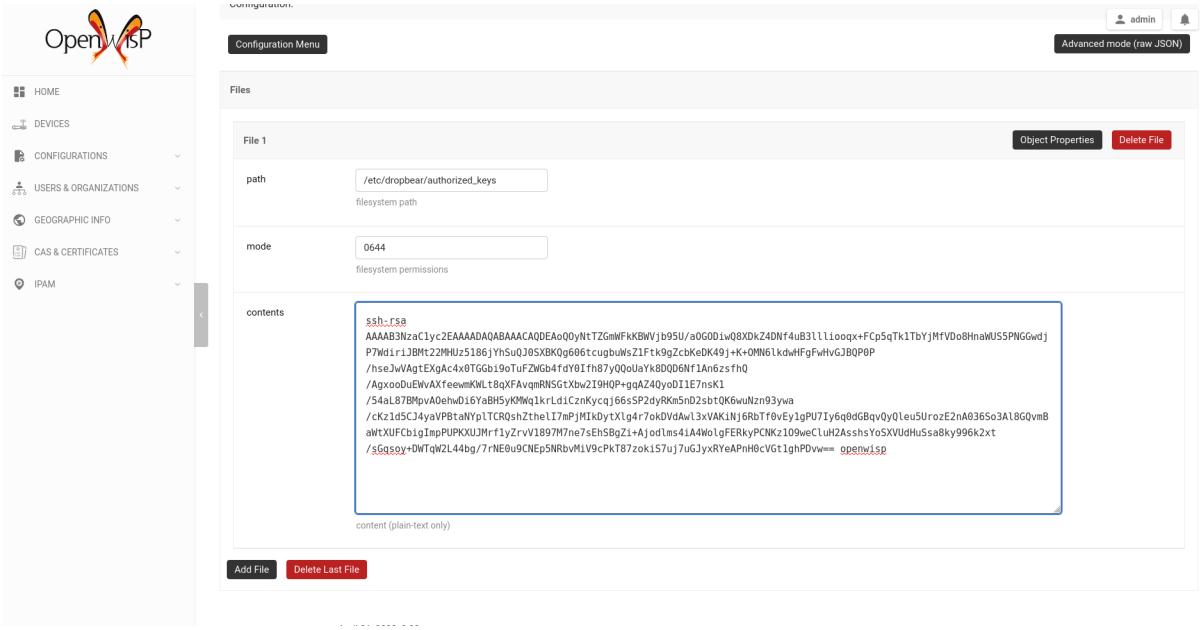
From the first page of OpenWISP click on "CONFIGURATIONS" in the left navigation menu, then "Access credentials", then click on the **"ADD ACCESS CREDENTIALS"** button in the upper right corner (alternatively, go to the following URL path: `/admin/connection/credentials/add/`).

Select SSH as `type`, enable the **Auto add** checkbox, then at the field "Credentials type" select "SSH (private key)", now type "root" in the `username` field, while in the `key` field you have to paste the contents of the private key just created.

Now hit save.

The credentials just created will be automatically enabled for all the devices in the system (both existing devices and devices which will be added in the future).

3. Add the Public Key to Your Devices



Now we need to instruct your devices to allow OpenWISP accessing via SSH, in order to do this we need to add the contents of the public key file created in step 1 (`sshkey.pub`) in the file `/etc/dropbear/authorized_keys` on the devices, the recommended way to do this is to create a configuration template in OpenWISP: from the first page of OpenWISP, click on "CONFIGURATIONS" in the left navigation menu, then and click on the **"ADD TEMPLATE"** button in the upper right corner (alternatively, go to the following URL: `/admin/config/template/add/`).

Check **enabled by default**, then scroll down the configuration section, click on "Configuration Menu", scroll down, click on "Files" then close the menu by clicking again on "Configuration Menu". Now type `/etc/dropbear/authorized_keys` in the `path` field of the file, then paste the contents of `sshkey.pub` in `contents`.

Now hit save.

There's a catch: you will need to assign the template to any existing device.

4. Test It

Once you have performed the 3 steps above, you can test it as follows:

1. Ensure there's at least one device turned on and connected to OpenWISP, ensure this device has the "SSH Authorized Keys" assigned to it.
2. Ensure the celery worker of OpenWISP Controller is running (e.g.: `ps aux | grep celery`)
3. SSH into the device and wait (maximum 2 minutes) until `/etc/dropbear/authorized_keys` appears as specified in the template.
4. While connected via SSH to the device run the following command in the console: `logread -f`, now try changing the device name in OpenWISP
5. Shortly after you change the name in OpenWISP, you should see some output in the SSH console indicating another SSH access and the configuration update being performed.

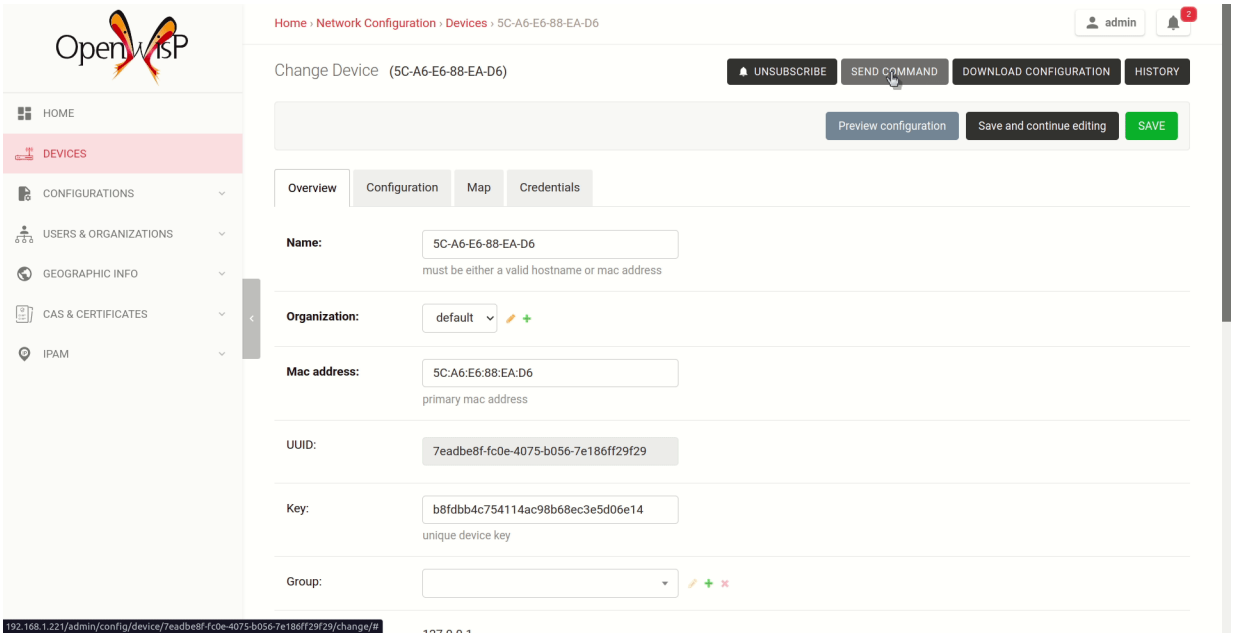
Sending Commands to Devices

Default Commands

By default, there are three options in the **Send Command** dropdown:

1. Reboot
2. Change Password
3. Custom Command

While the first two options are self-explanatory, the **custom command** option allows you to execute any command on the device as shown in the example below.



Important

In order for this feature to work, a device needs to have at least one valid **Access Credential** (see How to configure push updates).

The **Send Command** button will be hidden until the device has at least one **Access Credential**.

If you need to allow your users to quickly send specific commands that are used often in your network regardless of your users' knowledge of Linux shell commands, you can add new commands by following instructions in the Defining New Options in the Commands Menu section below.

> **Note**
>
> If you're an advanced user and want to learn how to register commands programmatically, refer to the Registering / Unregistering Commands section.

Defining New Options in the Commands Menu

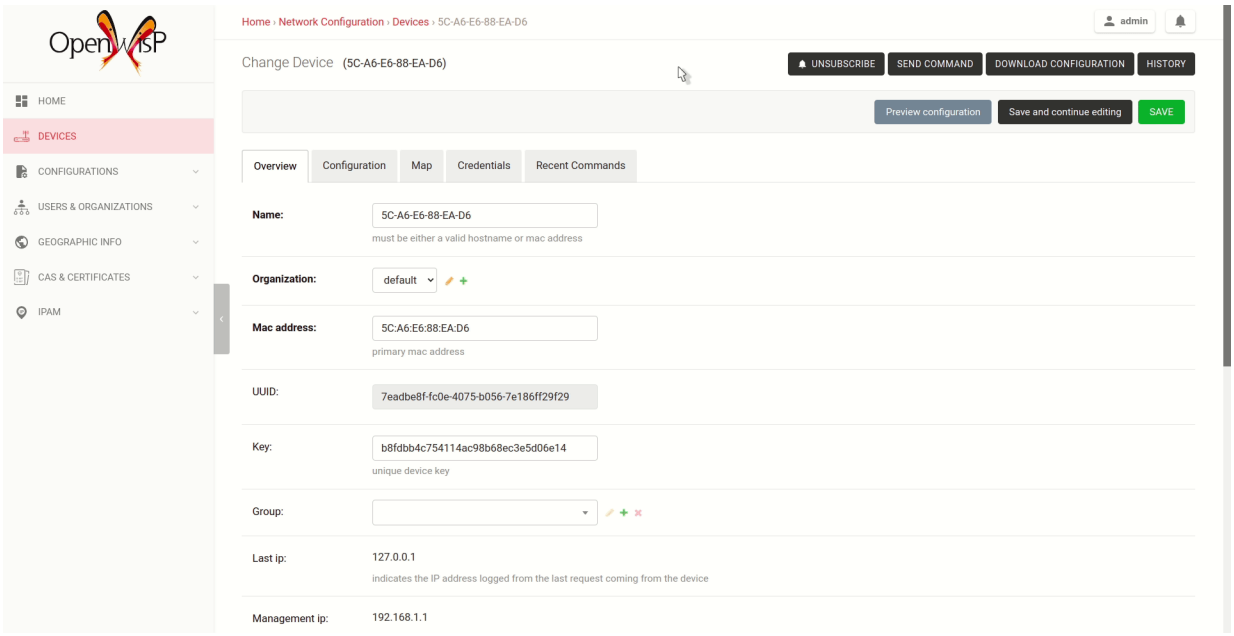
Let's explore to define new custom commands to help users perform additional management actions without having to be Linux/Unix experts.

We can do so by using the `OPENWISP_CONTROLLER_USER_COMMANDS` django setting.

The following example defines a simple command that can `ping` an input `destination_address` through a network interface, `interface_name`.

```
# In yourproject/settings.py


def ping_command_callable(destination_address, interface_name=None):
    command = f"ping -c 4 {destination_address}"
    if interface_name:
        command += f" -I {interface_name}"
    return command


OPENWISP_CONTROLLER_USER_COMMANDS = [
    (
        "ping",
        {
            "label": "Ping",
            "schema": {
                "title": "Ping",
                "type": "object",
                "required": ["destination_address"],
                "properties": {
                    "destination_address": {
                        "type": "string",
                        "title": "Destination Address",
                    },
                    "interface_name": {
                        "type": "string",
                        "title": "Interface Name",
                    },
                },
                "message": "Destination Address cannot be empty",
                "additionalProperties": False,
            },
            "callable": ping_command_callable,
        },
    )
]
```

The above code will add the *Ping* command in the user interface as show in the GIF below:



The `OPENWISP_CONTROLLER_USER_COMMANDS` setting takes a `list` of `tuple` each containing two elements. The first element of the tuple should contain an identifier for the command and the second element should contain a `dict` defining configuration of the command.

Command Configuration

The `dict` defining configuration for command should contain following keys:

1. `label`

A `str` defining label for the command used internally by Django.

2. `schema`

A `dict` defining JSONSchema for inputs of command. You can specify the inputs for your command, add rules for performing validation and make inputs required or optional.

Here is a detailed explanation of the schema used in above example:

```
{
    # Name of the command displayed in *Send Command* widget
    "title": "Ping",
    # Use type *object* if the command needs to accept inputs
    # Use type *null* if the command does not accepts any input
    "type": "object",
    # Specify list of inputs that are required
    "required": ["destination_address"],
    # Define the inputs for the commands along with their properties
    "properties": {
        "destination_address": {
            # type of the input value
            "type": "string",
            # label used for displaying this input field
            "title": "Destination Address",
        },
        "interface_name": {
            "type": "string",
            "title": "Interface Name",
        },
```

```
        },
        # Error message to be shown if validation fails
        "message": "Destination Address cannot be empty",
        # Whether specifying addtionaly inputs is allowed from the input form
        "additionalProperties": False,
    }
```

This example uses only handful of properties available in JSONSchema. You can experiment with other properties of JSONSchema for schema of your command.

3. `callable`

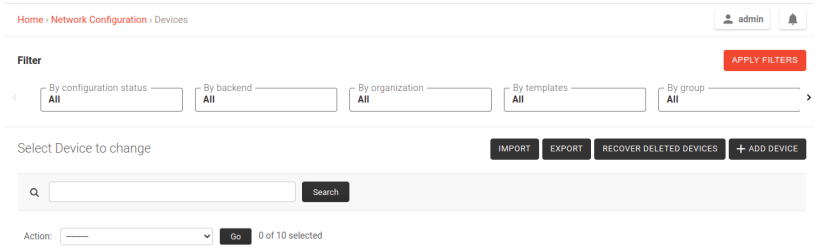
A `callable` or `str` defining dotted path to a callable. It should return the command (`str`) to be executed on the device. Inputs of the command are passed as arguments to this callable.

The example above includes a callable(`ping_command_callable`) for `ping` command.

How to register or unregister commands

Refer to Registering / Unregistering Commands in the developer documentation.

Import/Export Device Data

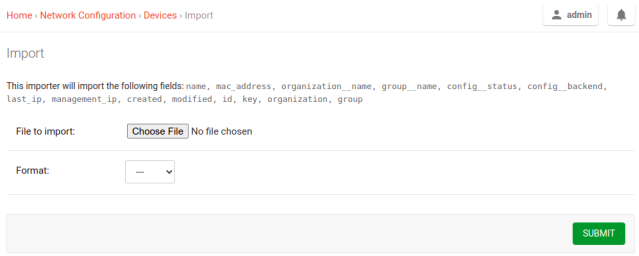


The device list page offers two buttons to export and import device data in different formats.

Importing

For importing devices into the system, only the required fields are needed, for example, the following CSV file will import a device named `TestImport` with mac address `00:11:22:09:44:55` in the organization with UUID `3cb5e18c-0312-48ab-8dbd-038b8415bd6f`:

```
organization,name,mac_address
3cb5e18c-0312-48ab-8dbd-038b8415bd6f,TestImport,00:11:22:09:44:55
```



Exporting

The export feature respects any filters selected in the device list.

Organization Limits

You can restrict the number of devices managed by each organization.

To set these limits:

1. Navigate to **USERS & ORGANIZATIONS** on the left-hand navigation menu.
2. Go to **Organizations**.
3. Click on the specific organization you want to limit.
4. In the **CONTROLLER LIMIT** section, set the desired limit.

Refer to the screenshot below for guidance:

Automating WireGuard Tunnels

Important

This guide assumes your OpenWrt firmware has the `wireguard-tools` package and its dependencies installed. If these packages are not present, you will need to install them.

This guide will help you to set up the automatic provisioning of WireGuard tunnels for your devices.

Note

This guide creates the VPN server and VPN client templates as **Shared systemwide (no organization)** objects. This allows any device of any organization to use the automation.

If needed, you can use any organization as long as the VPN server, the VPN client template, and devices have the same organization.

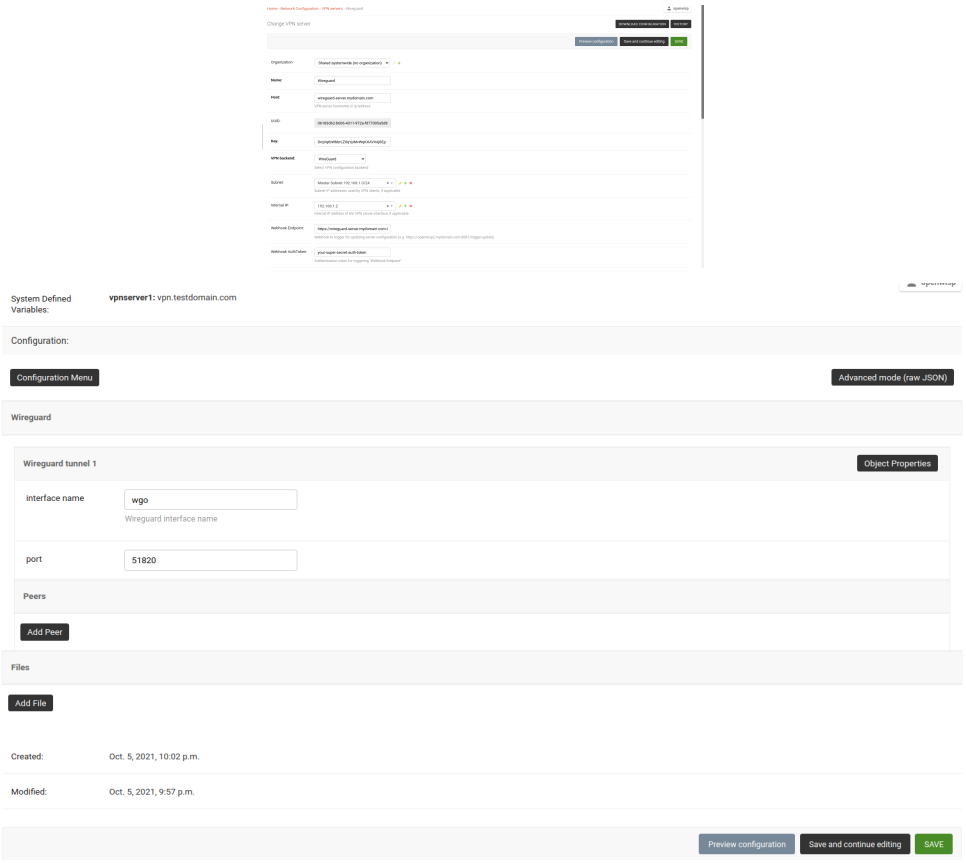
1. Create VPN Server Configuration for WireGuard

1. Visit `/admin/config/vpn/add/` to add a new VPN server.
2. Set the **Name** of this VPN server as `WireGuard` and the **Host** as `wireguard-server.mydomain.com` (update this to point to your WireGuard VPN server).
3. Select `WireGuard` from the dropdown as the **VPN Backend**.
4. When using WireGuard, OpenWISP takes care of managing IP addresses, assigning an IP address to each VPN peer. Create a new subnet or select an existing one from the dropdown menu. You can also assign an **Internal IP** to the WireGuard Server or leave it empty for OpenWISP to configure. This IP address will be used by the WireGuard interface on the server.

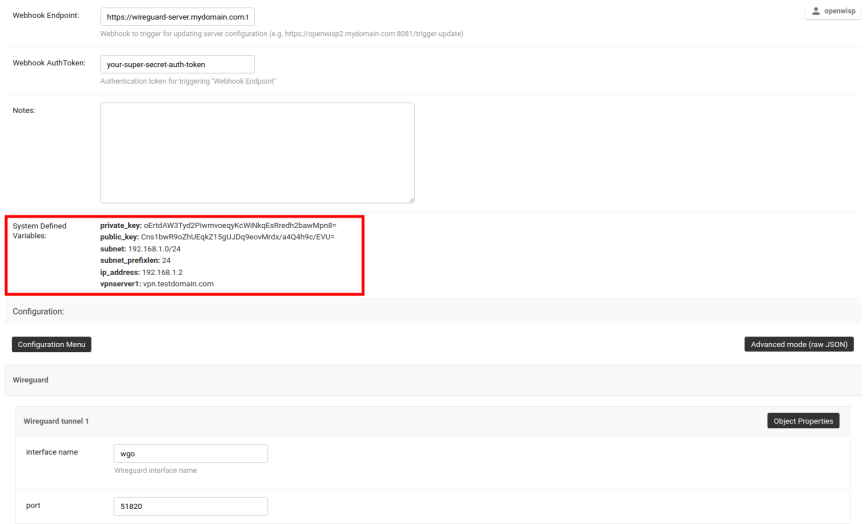
5. Set the **Webhook Endpoint** as `https://wireguard-server.mydomain.com:8081/trigger-update` for this example. Update this according to your VPN upgrader endpoint. Set **Webhook AuthToken** to any strong passphrase; this will be used to ensure that configuration upgrades are requested from trusted sources.

> **Note**
>
> If you are setting up a WireGuard VPN server, substitute `wireguard-server.mydomain.com` with the hostname of your VPN server and follow the steps in the next section.

6. Under the configuration section, set the name of the WireGuard tunnel 1 interface. In this example, we have used `wg0`.



7. After clicking on **Save and continue editing**, you will see that OpenWISP has automatically created public and private keys for the WireGuard server in **System Defined Variables**, along with internal IP address information.



2. Deploy WireGuard VPN Server

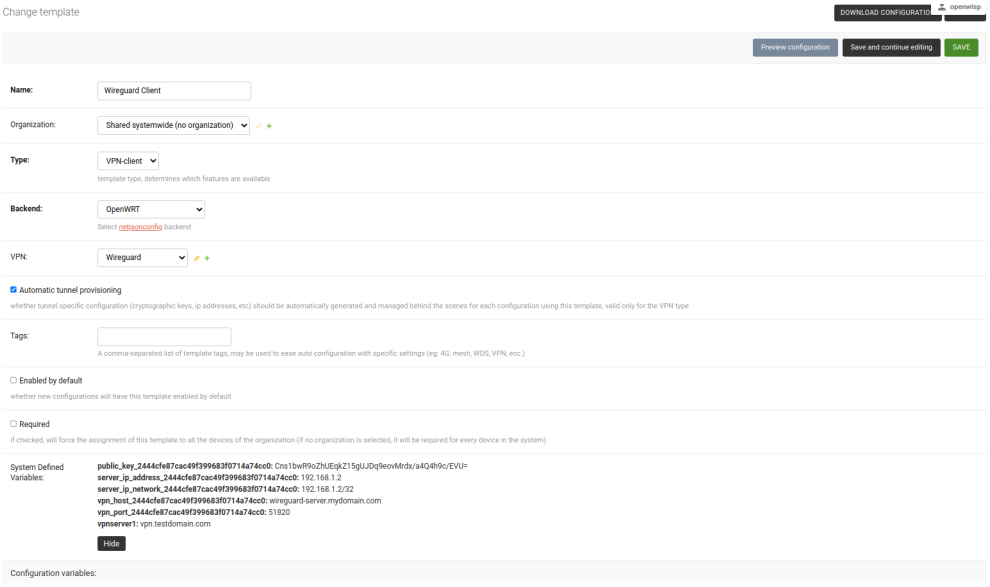
If you haven't already set up WireGuard on your VPN server, this would be a good time to do so.

We recommend using the ansible-wireguard-openwisp role for installing WireGuard, as it also installs scripts that allow OpenWISP to manage the WireGuard VPN server.

Ensure that the VPN server attributes used in your playbook match the VPN server configuration in OpenWISP.

3. Create VPN Client Template for WireGuard VPN Server

1. Visit `/admin/config/template/add/` to add a new template.
2. Set `WireGuard Client` as **Name** (you can set whatever you want) and select `VPN-client` as **type** from the dropdown list.
3. The **Backend** field refers to the backend of the device this template can be applied to. For this example, we will leave it to `OpenWrt`.
4. Select the correct VPN server from the dropdown for the **VPN** field. Here it is `WireGuard`.
5. Ensure that **Automatic tunnel provisioning** is checked. This will make OpenWISP to automatically generate public and private keys and provision IP address for each WireGuard VPN client.
6. After clicking on **Save and continue editing** button, you will see details of *WireGuard* VPN server in **System Defined Variables**. The template configuration will be automatically generated which you can tweak accordingly. We will use the automatically generated VPN client configuration for this example.



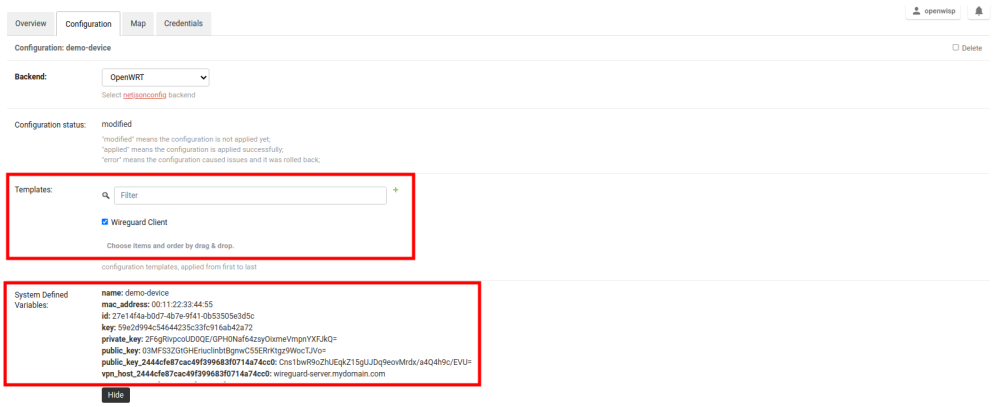
4. Apply WireGuard VPN Template to Devices

Note

This step assumes that you already have a device registered on OpenWISP. Register or create a device before proceeding.

1. Open the **Configuration** tab of the concerned device.
2. Select the *WireGuard Client* template.

3. Upon clicking on **Save and continue editing** button, you will see some entries in **System Defined Variables**. It will contain internal IP address, private and public key for the WireGuard client on the device along with details of WireGuard VPN server.



Voila! You have successfully configured OpenWISP to manage WireGuard tunnels for your devices.

Seealso

You may also want to explore other automated VPN tunnel provisioning options:

- Wireguard over VXLAN
- Zerotier
- OpenVPN

Automating VXLAN over WireGuard Tunnels

Important

This guide assumes your OpenWrt firmware has the `vxlan` and `wireguard-tools` packages installed. If these packages are not present, you will need to install them.

By following these steps, you will be able to setup layer 2 VXLAN tunnels encapsulated in WireGuard tunnels which work on layer 3.

Note

This guide creates the VPN server and VPN client templates as **Shared systemwide (no organization)** objects. This allows any device of any organization to use the automation.

If needed, you can use any organization as long as the VPN server, the VPN client template, and devices have the same organization.

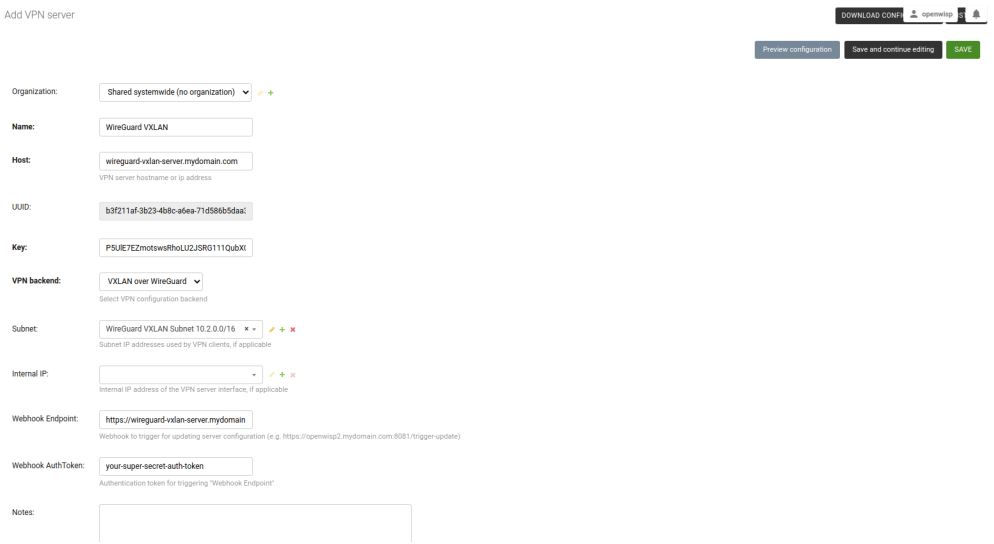
1. Create VPN Server Configuration for VXLAN Over WireGuard

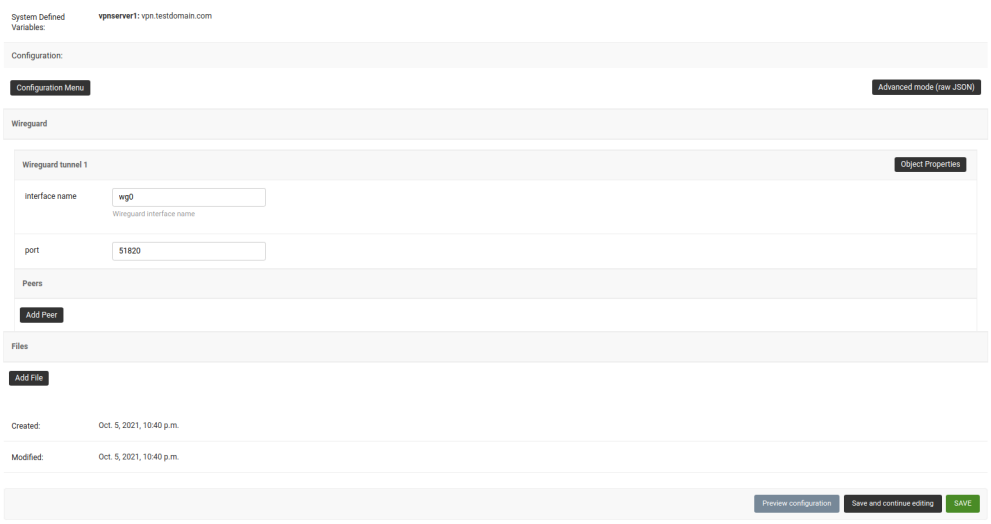
1. Visit `/admin/config/vpn/add/` to add a new VPN server.
2. We will set **Name** of this VPN server `Wireguard VXLAN` and **Host** as `wireguard-vxlan-server.mydomain.com` (update this to point to your WireGuard VXLAN VPN server).
3. Select `VXLAN over WireGuard` from the dropdown as **VPN Backend**.
4. When using VXLAN over WireGuard, OpenWISP takes care of managing IP addresses (assigning an IP address to each VPN peer). You can create a new subnet or select an existing one from the dropdown menu. You can also assign an **Internal IP** to the WireGuard Server or leave it empty for OpenWISP to configure. This IP address will be used by the WireGuard interface on server.
5. We have set the **Webhook Endpoint** as `https://wireguard-vxlan-server.mydomain.com:8081/trigger-update` for this example. You will need to update this according to you VPN upgrader endpoint. Set **Webhook AuthToken** to any strong passphrase, this will be used to ensure that configuration upgrades are requested from trusted sources.

Note

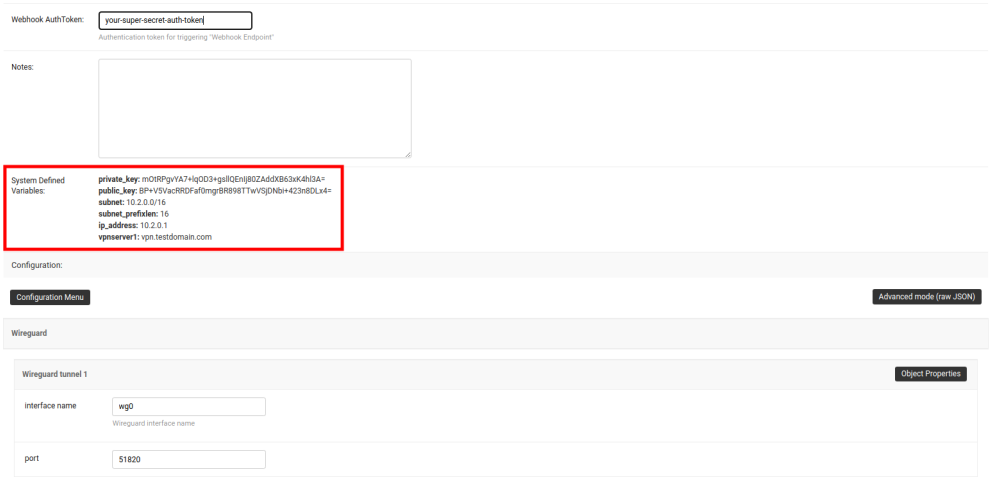
If you are following this tutorial for also setting up WireGuard VPN server, just substitute `wireguard-server.mydomain.com` with hostname of your VPN server and follow the steps in next section.

6. Under the configuration section, set the name of WireGuard tunnel 1 interface. We have used `wg0` in this example.





7. After clicking on **Save and continue editing**, you will see that OpenWISP has automatically created public and private key for WireGuard server in **System Defined Variables** along with internal IP address information.



2. Deploy Wireguard VXLAN VPN Server

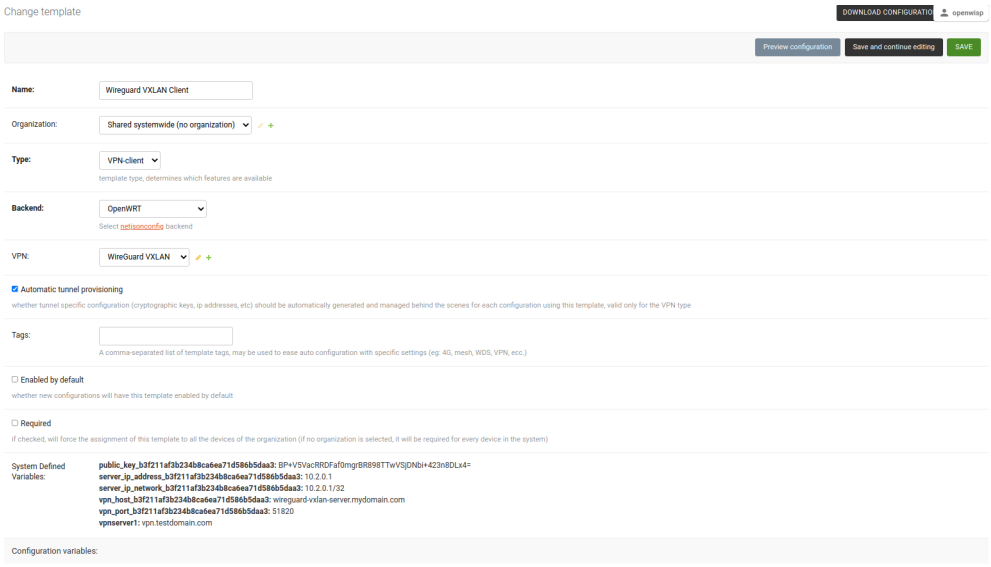
If you haven't already set up WireGuard on your VPN server, this is a good time to do so. We recommend using the ansible-wireguard-openwisp role for installing WireGuard since it also installs scripts that allow OpenWISP to manage the WireGuard VPN server along with VXLAN tunnels.

Pay attention to the VPN server attributes used in your playbook. It should be the same as the VPN server configuration in OpenWISP.

3. Create VPN Client Template for WireGuard VXLAN VPN Server

1. Visit `/admin/config/template/add/` to add a new template.
2. Set `Wireguard VXLAN Client` as **Name** (you can set whatever you want) and select `VPN-client` as **type** from the dropdown list.
3. The **Backend** field refers to the backend of the device this template can be applied to. For this example, we will leave it as `OpenWrt`.
4. Select the correct VPN server from the dropdown for the **VPN** field. Here it is `Wireguard VXLAN`.
5. Ensure that **Automatic tunnel provisioning** is checked. This will make OpenWISP automatically generate public and private keys and provision IP addresses for each WireGuard VPN client along with the VXLAN Network Identifier (VNI).

6. After clicking on **Save and continue editing** button, you will see details of the *Wireguard VXLAN* VPN server in **System Defined Variables**. The template configuration will be automatically generated which you can tweak accordingly. We will use the automatically generated VPN client configuration for this example.

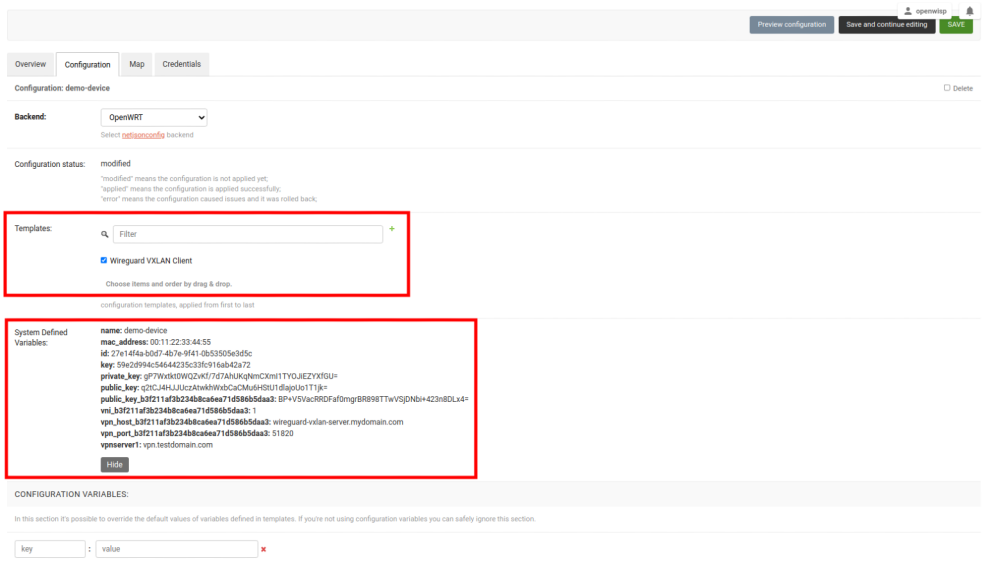


4. Apply Wireguard VXLAN VPN Template to Devices

Note

This step assumes that you already have a device registered on OpenWISP. Register or create a device before proceeding.

1. Open the **Configuration** tab of the concerned device.
2. Select the *WireGuard VXLAN Client* template.
3. Upon clicking on **Save and continue editing** button, you will see some entries in **System Defined Variables**. It will contain internal IP address, private and public key for the WireGuard client on the device and details of WireGuard VPN server along with VXLAN Network Identifier(VNI) of this device.



Voila! You have successfully configured OpenWISP to manage VXLAN over WireGuard tunnels for your devices.

Seealso

You may also want to explore other automated VPN tunnel provisioning options:

- Wireguard
- Zerotier
- OpenVPN

Automating ZeroTier Tunnels

Important

This guide assumes your OpenWrt firmware has the `zerotier` package installed. If this package is not present, you will need to install it.

Follow the procedure described below to set up ZeroTier tunnels on your devices.

Note

This guide creates the VPN server and VPN client templates as **Shared systemwide (no organization)** objects. This allows any device of any organization to use the automation.

If needed, you can use any organization as long as the VPN server, the VPN client template, and devices have the same organization.

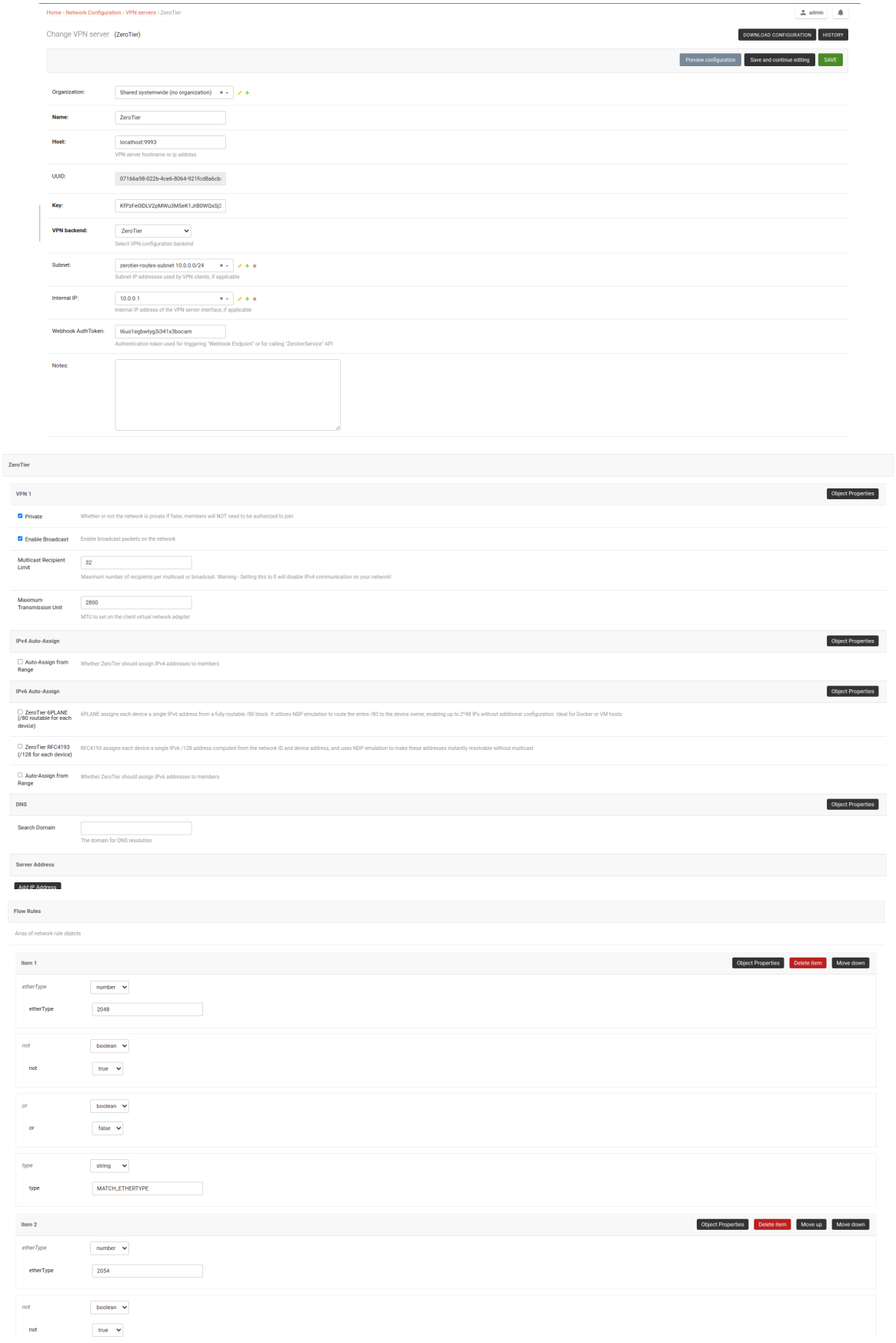
1. Configure Self-Hosted ZeroTier Network Controller

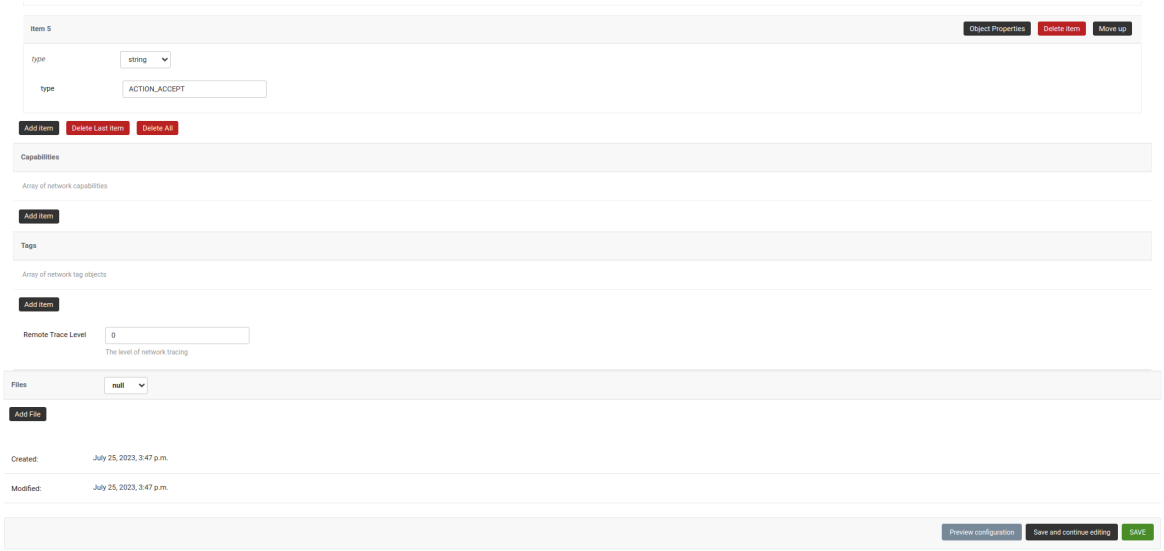
If you haven't already set up a self-hosted ZeroTier network controller on your server, now is a good time to do so. You can start by simply installing ZeroTier on your server from the official website.

2. Create VPN Server Configuration for ZeroTier

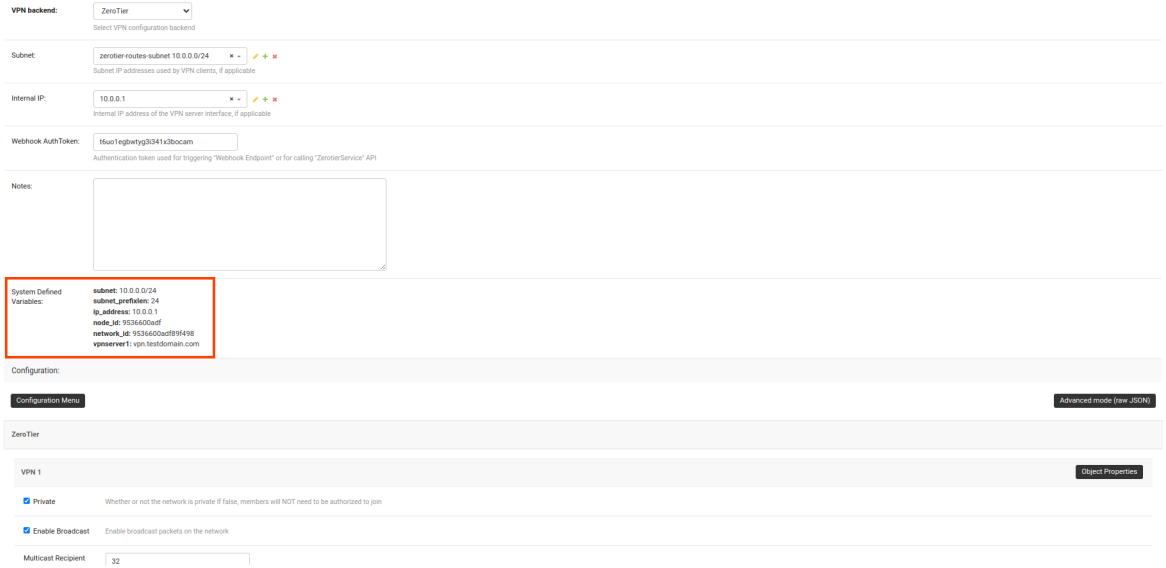
1. Visit `/admin/config/vpn/add/` to add a new VPN server.
2. We will set **Name** of this VPN server `ZeroTier` and **Host** as `my-zerotier-server.mydomain.com:9993` (update this to point to your ZeroTier VPN server).
3. Select `ZeroTier` from the dropdown as **VPN Backend**.
4. When using ZeroTier, OpenWISP takes care of managing IP addresses (assigning an IP address to each VPN client (ZeroTier network members)). You can create a new subnet or select an existing one from the dropdown menu. You can also assign an **Internal IP** to the ZeroTier controller or leave it empty for OpenWISP to configure. This IP address will be used to assign it to the ZeroTier controller running on the server.
5. Set the **Webhook AuthToken**, this will be the ZeroTier authorization token which you can obtain by running the following command on the ZeroTier controller:

```
sudo cat /var/lib/zerotier-one/authtoken.secret
```





6. After clicking on **Save and continue editing**, OpenWISP automatically detects the node address of the ZeroTier controller and creates a ZeroTier network. The **network_id** of this network can be viewed in the **System Defined Variables** section, where it also provides internal IP address information.

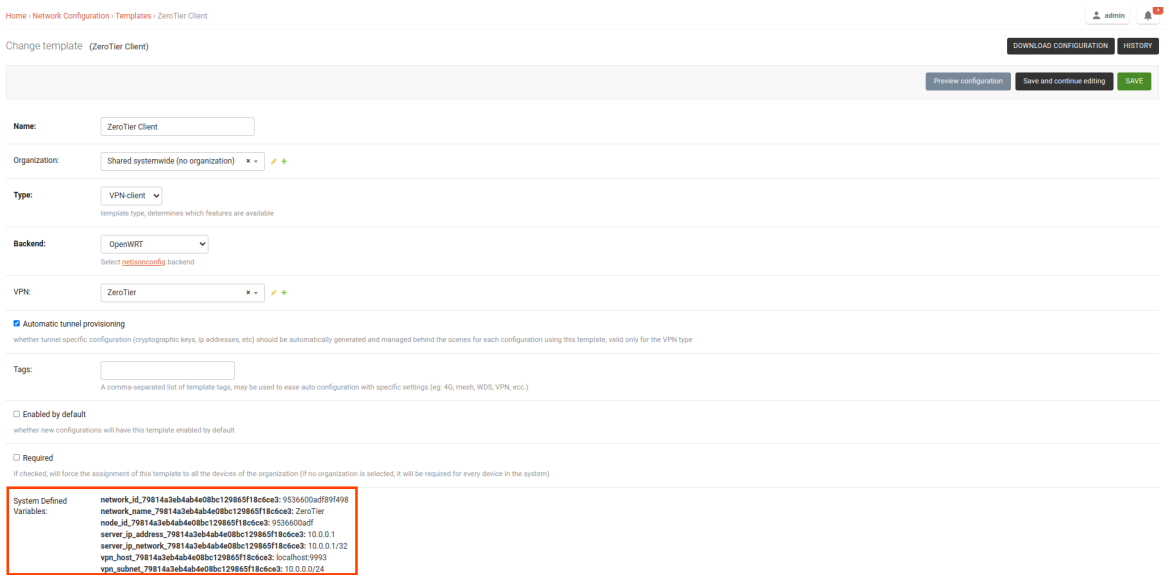


3. Create VPN Client Template for ZeroTier VPN Server

1. Visit `/admin/config/template/add/` to add a new template.
2. Set `ZeroTier Client` as **Name** (you can set whatever you want) and select `VPN-client` as **type** from the dropdown list.
3. The **Backend** field refers to the backend of the device this template can be applied to. For this example, we will leave it to `OpenWrt`.
4. Select the correct VPN server from the dropdown for the **VPN** field. Here it is `ZeroTier`.
5. Ensure that the **Automatic tunnel provisioning** option is checked. This will enable OpenWISP to automatically provision an IP address and ZeroTier identity secrets (used for assigning member IDs) for each ZeroTier VPN client.
6. After clicking on **Save and continue editing** button, you will see details of *ZeroTier* VPN server in **System Defined Variables**. The template configuration will be automatically generated which you can tweak accordingly. We will use the automatically generated VPN client configuration for this example.

Note

OpenWISP uses zerotier-idtool to manage **ZeroTier identity secrets**. Please make sure that you have ZeroTier package installed on the server.

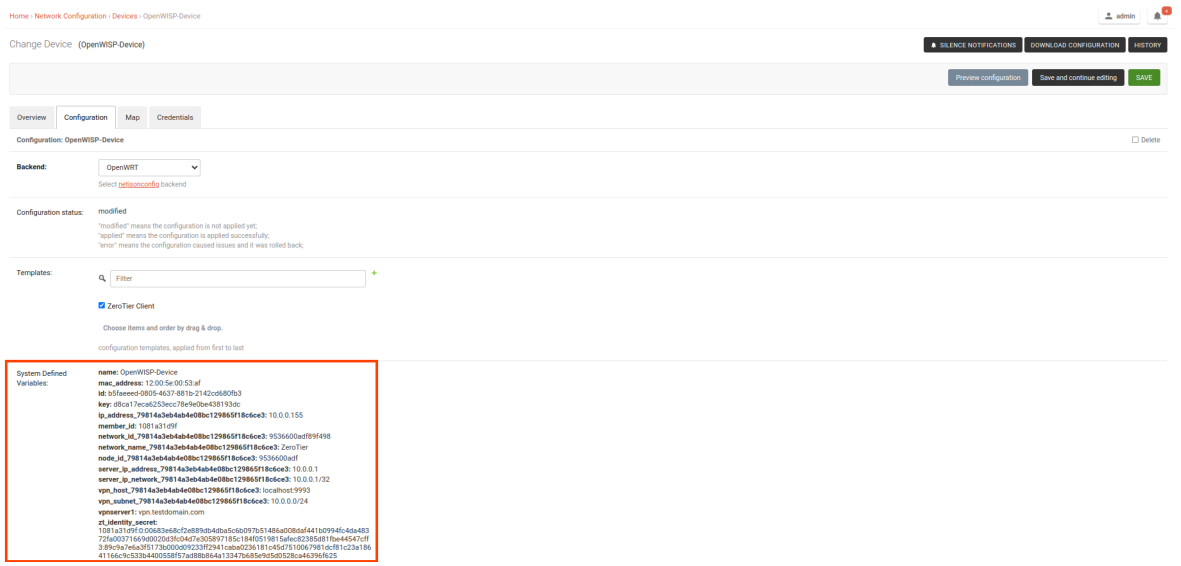


4. Apply ZeroTier VPN Template to Devices

Note

This step assumes that you already have a device registered on OpenWISP. Register or create a device before proceeding.

1. Open the **Configuration** tab of the concerned device.
2. Select the *ZeroTier Client* template.
3. Upon clicking the **Save and Continue Editing** button, you will see entries in the **System Defined Variables** section. These entries will include **zerotier_member_id**, **identity_secret**, and the internal **IP address** of the ZeroTier client (network member) on the device, along with details of the VPN server.



4. Once the configuration is successfully applied to the device, you will notice a new ZeroTier interface that is up and running. This interface will have the name `owzt89f498` (where `owzt` is followed by the last six hexadecimal characters of the ZeroTier **network ID**).

Congratulations! You've successfully configured OpenWISP to manage ZeroTier tunnels on your devices.

Seealso

You may also want to explore other automated VPN tunnel provisioning options:

- Wireguard
- Wireguard over VXLAN
- OpenVPN

Automating OpenVPN Tunnels

Important

This guide assumes your OpenWrt firmware has the `openvpn-mbedtls` package (or equivalent versions like `openvpn-wolfssl` or `openvpn-openssl`) installed. If this package is not present, you will need to install it.

In this guide, we will explore how to set up the automatic provisioning and management of **OpenVPN tunnels**.

Table of Contents:

Setting up the OpenVPN Server

The first step is to install the OpenVPN server. In this tutorial, to perform this step we will use Ansible.

If you already have experience installing an OpenVPN server, feel free to use any method you prefer.

Important

If you have already set up your OpenVPN server or prefer to install the OpenVPN server using a different method, you can skip forward to Import the CA and the Server Certificate in OpenWISP.

For simplicity, **the OpenVPN server must be installed on the same server where OpenWISP is also installed**.

While it is possible to install the OpenVPN server on a different server, it requires additional steps not covered in this tutorial.

1. Install Ansible and Required Ansible Roles

Install Ansible **on your local machine** (please ensure that you do not install it on the server).

To **install Ansible**, we suggest following the official Ansible installation guide.

After installing Ansible, you need to install Git (example for Linux Debian/Ubuntu systems):

```
sudo apt-get install git
```

After installing both Ansible and Git, install the required roles:

```
ansible-galaxy install git+https://github.com/Stouts/Stouts.openvpn,3.0.0 nkakouros.easyrsa
```

2. Create Inventory File and Playbook YAML

Create an Ansible inventory file named `inventory` **on your local machine** (not on the server) with the following contents:

```
[openvpn]
your_server_domain_or_ip
```

For example, if your server IP is `192.168.56.2`:

```
[openvpn]
192.168.56.2
```

In the same directory where you created the `inventory` file, create a file named `playbook.yml` with the following content:

```
- hosts: openvpn
  vars:
    # EasyRSA
    easyrsa_generate_dh: true
    easyrsa_servers:
      - name: server
    easyrsa_clients: []
    easyrsa_pki_dir: /etc/easyrsa/pki

    # OpenVPN
    openvpn_keydir: "{{ easyrsa_pki_dir }}"
    openvpn_clients: []
    openvpn_use_pam: false
  roles:
    - role: nkakouros.easyrsa
    - role: Stouts.openvpn
```

Hint

You can further customize the configuration using the role variables. Read more about other options in EasyRSA and OpenVPN.

3. Run the Playbook

Run the Ansible playbook:

```
ansible-playbook -i inventory playbook.yml -b -k -K --become-method=su
```

Import the CA and the Server Certificate in OpenWISP

Important

If you chose an alternative installation method for OpenVPN and you did not create the CA and certificate yet, you can create the certificates from scratch via the OpenWISP web interface instead of importing them.

Follow the instructions below and instead of selecting Import Existing as Operation Type, select Create new.

You also won't need to copy any file from the server as OpenWISP generates the x509 certificates automatically.

To import the CA and Server Certificate into OpenWISP, you need to access your server via `ssh` or any other method that suits you.

Change your directory to `/etc/easyrsa/pki/`.

Note

If you incur in the following error: `-bash: cd: /etc/easyrsa/pki: Permission denied`, you may need to log in as the root user.

Import the CA

In your OpenWISP dashboard, go to `/admin/pki/ca/add/`.

In Operation Type, choose Import Existing.

Get your CA certificate from the `ca.crt` file and the private key from the `private/ca.key` file, then enter them in the respective fields.

Import the Server Certificate

In your OpenWISP dashboard, go to `/admin/pki/cert/add/`.

In Operation Type, choose Import Existing and in **CA**, choose the CA you just created.

Get your server certificate from the `issued/server.crt` file and the server private key from the `private/server.key` file, then enter them in the respective fields.

Create the VPN Server in OpenWISP

In the OpenWISP dashboard, go to `/admin/config/vpn/add/`.

In the Host field, enter your server IP address. In the Certification Authority and X509 Certificate fields, select the CA and certificate you created in the previous step.

Under Configuration, click on Configuration Menu, then change Server (Bridged) to Server (Routed).

Setting up a Bridged Server is similar to setting up a Routed Server but is not covered in this tutorial.

Adjust the rest of the VPN configuration to match the settings in `/etc/openvpn/server.conf`.

Tip

You can verify if your VPN configuration matches the `server.conf` file by using the Preview Configuration button at the top right corner of the page.

Create the VPN-Client Template in OpenWISP

In your OpenWISP dashboard, go to `/admin/config/template/add/`.

Set the Type to VPN-client.

Once the VPN field appears, select the VPN you created in the previous step.

Ensure the Automatic tunnel provisioning flag remains enabled.

If this template is for your management VPN or the default VPN option, we recommend checking the Enabled by default flag. For more information about this flag, refer to Default Templates.

Now, save the template.

After saving the template, you can tweak the VPN Client configuration, which is automatically generated to be compatible with the server configuration.

Finally you can add the new template to your devices.

Tip

If you need to troubleshoot any issue, increase the verbosity of the OpenVPN logging, both on the server and the clients, and check both logs (on the server and on the client).

Seealso

You may also want to explore other automated VPN tunnel provisioning options:

- Wireguard
- Wireguard over VXLAN
- Zerotier

Automating Subnet and IP Address Provisioning

This guide helps you automate provisioning subnets and IP addresses for your network devices.

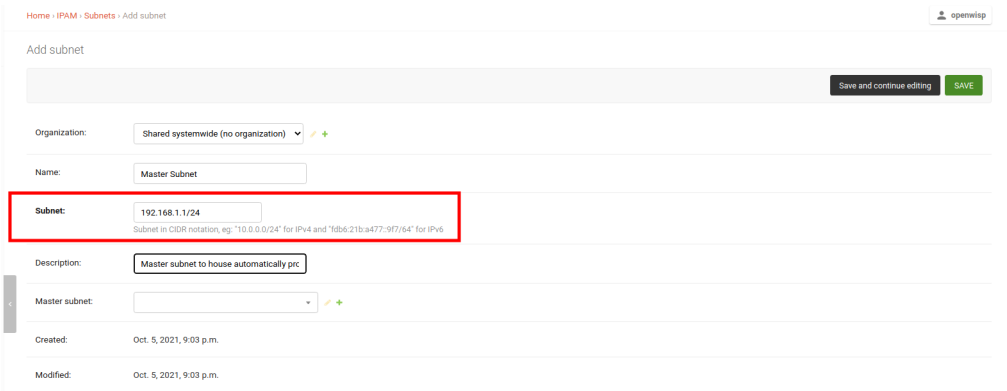
1. Create a Subnet and a Subnet Division Rule

Create a master subnet.

This is the parent subnet from which automatically generated subnets will be provisioned.

Note

Choose a subnet size appropriate for the needs of your network.



On the same page, add a **subnet division rule**. This rule defines the criteria for automatically provisioning subnets under the master subnet.

The type of subnet division rule determines when subnets and IP addresses are assigned to devices.

The currently supported rule types are described below.

> **Note**
>
> For information on how to write your own subnet division rule types, please refer to: Custom Subnet Division Rule Types.

Device Subnet Division Rule

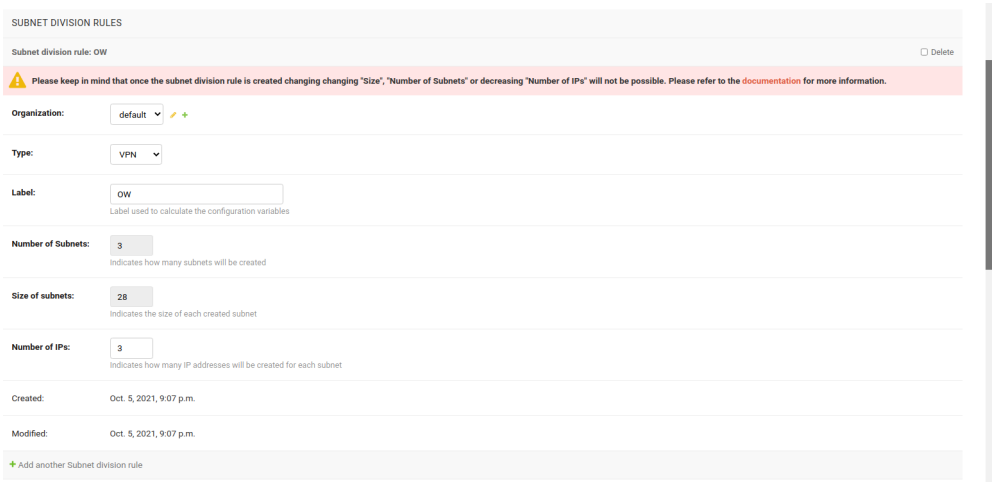
This rule triggers when a device configuration (`config.Config` model) is created for the organization specified in the rule.

> **Note**
>
> If a device object is created without any related configuration object, it will not trigger this rule.

Creating a new *"Device"* rule will also automatically provision subnets and IP addresses for existing devices within the organization.

VPN Subnet Division Rule

This rule triggers when a template flagged as *VPN-client* is assigned to a device configuration, but only if the VPN server associated with the VPN-client template uses the same subnet to which the subnet division rule is assigned to.



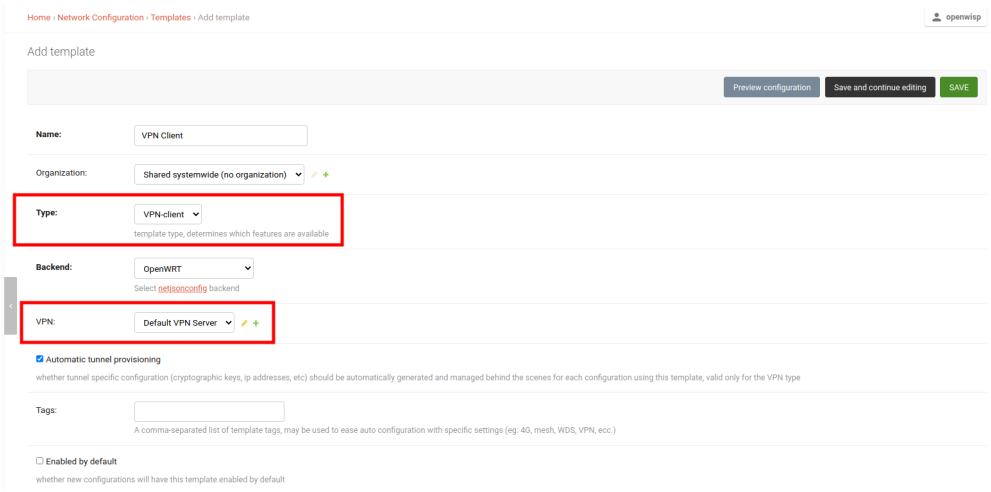
In this example, **VPN subnet division rule** is used.

2. Create a VPN Server

Now create a VPN Server and choose the previously created **master subnet** as the subnet for this VPN Server.

3. Create a VPN Client Template

Create a template, setting the **Type** field to **VPN Client** and **VPN** field to use the previously created VPN Server.

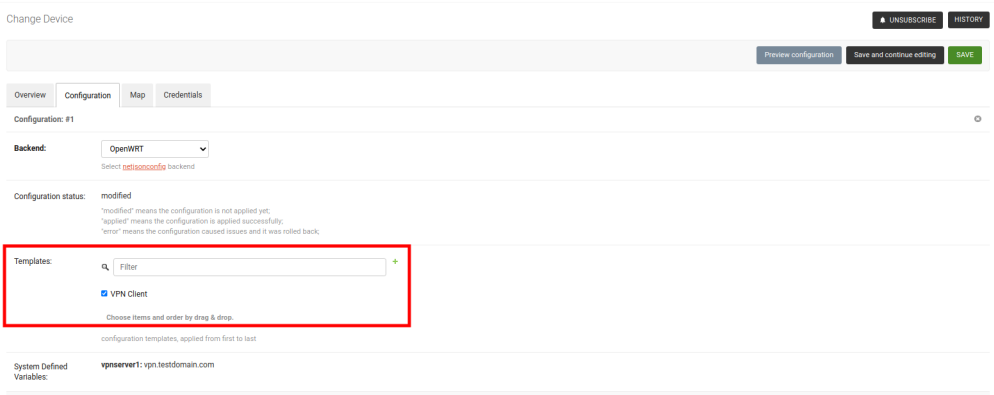


Note

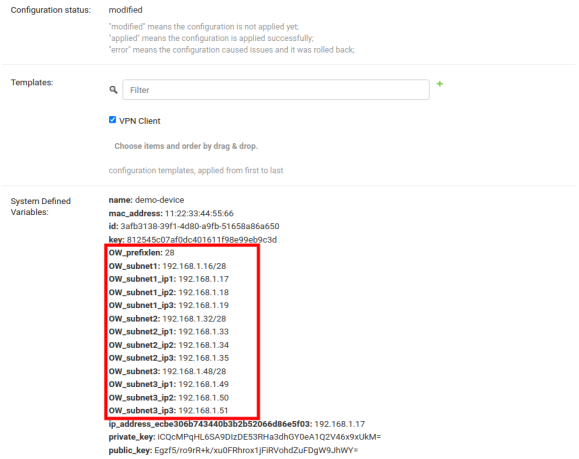
You can also check the **Enable by default** field if you want to automatically apply this template to devices that will register in future.

4. Apply VPN Client Template to Devices

With everything in place, you can now apply the VPN Client Template to devices.



After saving the device, you should see all provisioned Subnets and IPs for this device under System Defined Variables.



You can now use these Configuration Variables in the configuration of devices of your network.

Important notes for using Subnet Division

- In the example provided, the Subnet, VPN Server, and VPN Client Template were associated with the **default** organization. You can also utilize **Systemwide Shared** Subnet, VPN Server, or VPN Client Template; however, remember that the Subnet Division Rule will always be linked to an organization. It will only be triggered when a VPN Client Template is applied to a Device with the same organization as the Subnet Division Rule.
- Configuration variables can be used for provisioned subnets and IPs in the Template. Each variable will resolve differently for different devices. For example, `OW_subnet1_ip1` will resolve to `10.0.0.1` for one device and `10.0.0.55` for another. Every device receives its own set of subnets and IPs. Ensure to provide default fallback values in the *default values* template field (mainly used for validation).
- The Subnet Division Rule automatically creates a reserved subnet, which can be utilized to provision any IP addresses that need to be created manually. The remaining address space of the master subnet must not be interfered with, or the automation implemented in this module will not function.
- The example provided used the VPN subnet division rule. Similarly, the device subnet division rule can be employed, requiring only the creation of a subnet and a subnet division rule.

Limitations of Subnet Division Rules

In the current implementation, it is not possible to change *Size*, *Number of Subnets* and *Number of IPs* fields of an existing subnet division rule due to following reasons:

Size

Allowing to change size of provisioned subnets of an existing subnet division rule will require rebuilding of Subnets and IP addresses which has possibility of breaking existing configurations.

Number of Subnets

Allowing to decrease number of subnets of an existing subnet division rule can create patches of unused subnets dispersed everywhere in the master subnet. Allowing to increase number of subnets will break the continuous allocation of subnets for every device. It can also break configuration of devices.

Number of IPs

Decreasing the number of IPs in an existing subnet division rule is not allowed as it can lead to deletion of IP addresses, potentially breaking configurations of existing devices.

Increasing the number of IPs is allowed.

If you need to modify any of these fields (**Size**, **Number of Subnets**, or **Number of IPs**), we recommend to proceed as follows:

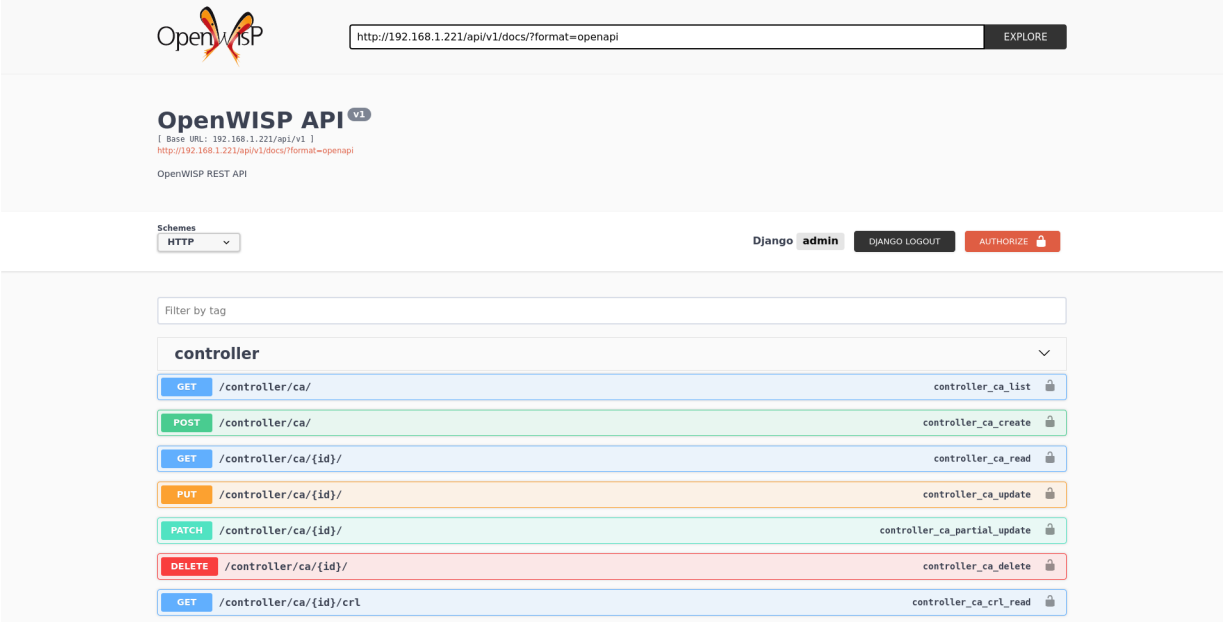
1. Delete the existing rule.
2. Create a new rule.

The automation will provision new subnets and addresses according to the new parameters to any existing devices that are eligible to the subnet division rule.

However, be aware that existing devices **will probably be reassigned different subnets and IP addresses** than the ones used previously.

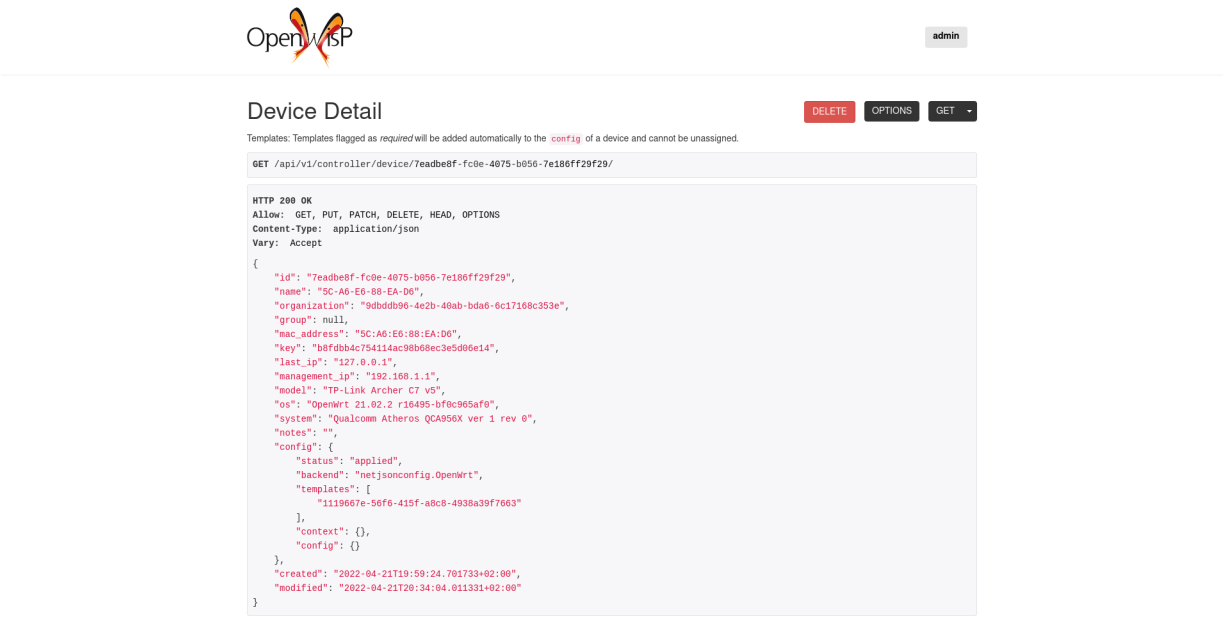
REST API Reference

Live Documentation



A general live API documentation (following the OpenAPI specification) at `/api/v1/docs/`.

Browsable Web Interface



Additionally, opening any of the endpoints listed below directly in the browser will show the browsable API interface of Django-REST-Framework, which makes it even easier to find out the details of each endpoint.

Authentication

See Authenticating with the User Token.

When browsing the API via the Live Documentation or the Browsable Web Interface, you can also use the session authentication by logging in the django admin.

Pagination

All *list* endpoints support the `page_size` parameter that allows paginating the results in conjunction with the `page` parameter.

```
GET /api/v1/controller/template/?page_size=10
GET /api/v1/controller/template/?page_size=10&page=2
```

List of Endpoints

Since the detailed explanation is contained in the Live Documentation and in the Browsable Web Interface of each point, here we'll provide just a list of the available endpoints, for further information please open the URL of the endpoint in your browser.

List Devices

```
GET /api/v1/controller/device/
```

Available filters

You can filter a list of devices based on their configuration status using the `status` (e.g modified, applied, or error).

```
GET /api/v1/controller/device/?config__status={status}
```

You can filter a list of devices based on their configuration backend using the `backend` (e.g `netjsonconfig.OpenWrt` or `netjsonconfig.OpenWisp`).

```
GET /api/v1/controller/device/?config__backend={backend}
```

You can filter a list of devices based on their organization using the `organization_id` or `organization_slug`.

```
GET /api/v1/controller/device/?organization={organization_id}

GET /api/v1/controller/device/?organization_slug={organization_slug}
```

You can filter a list of devices based on their configuration templates using the `template_id`.

```
GET /api/v1/controller/device/?config__templates={template_id}
```

You can filter a list of devices based on their device group using the `group_id`.

```
GET /api/v1/controller/device/?group={group_id}
```

You can filter a list of devices that have a device location object using the `with_geo` (e.g. true or false).

```
GET /api/v1/controller/device/?with_geo={with_geo}
```

You can filter a list of devices based on their creation time using the `creation_time`.

```
# Created exact
GET /api/v1/controller/device/?created={creation_time}

# Created greater than or equal to
GET /api/v1/controller/device/?created__gte={creation_time}

# Created is less than
GET /api/v1/controller/device/?created__lt={creation_time}
```

Create Device

```
POST /api/v1/controller/device/
```

Get Device Detail

```
GET /api/v1/controller/device/{id}/
```

Download Device Configuration

```
GET /api/v1/controller/device/{id}/configuration/
```

The above endpoint triggers the download of a `tar.gz` file containing the generated configuration for that specific device.

Change Details of Device

```
PUT /api/v1/controller/device/{id}/
```

Patch Details of Device

```
PATCH /api/v1/controller/device/{id}/
```

Note

To assign, unassign, and change the order of the assigned templates add, remove, and change the order of the `{id}` of the templates under the `config` field in the JSON response respectively. Moreover, you can also select and unselect templates in the HTML Form of the Browsable API.

The required template(s) from the organization(s) of the device will added automatically to the `config` and cannot be removed.

Example usage: For assigning template(s) add the/their {id} to the config of a device,

```
curl -X PATCH \
    http://127.0.0.1:8000/api/v1/controller/device/76b7d9cc-4ffd-4a43-b1b0-8f8befd1a7c0/ \
    -H 'authorization: Bearer dc8d497838d4914c9db9aad9b6ec66f6c36ff46b' \
    -H 'content-type: application/json' \
    -d '{
            "config": {
                "templates": ["4791fa4c-2cef-4f42-8bb4-c86018d71bd3"]
            }
        }'
```

Example usage: For removing assigned templates, simply remove the/their {id} from the config of a device,

```
curl -X PATCH \
    http://127.0.0.1:8000/api/v1/controller/device/76b7d9cc-4ffd-4a43-b1b0-8f8befd1a7c0/ \
    -H 'authorization: Bearer dc8d497838d4914c9db9aad9b6ec66f6c36ff46b' \
    -H 'content-type: application/json' \
    -d '{
            "config": {
                "templates": []
            }
        }'
```

Example usage: For reordering the templates simply change their order from the config of a device,

```
curl -X PATCH \
    http://127.0.0.1:8000/api/v1/controller/device/76b7d9cc-4ffd-4a43-b1b0-8f8befd1a7c0/ \
    -H 'authorization: Bearer dc8d497838d4914c9db9aad9b6ec66f6c36ff46b' \
    -H 'cache-control: no-cache' \
    -H 'content-type: application/json' \
    -H 'postman-token: b3f6a1cc-ff13-5eba-e460-8f394e485801' \
    -d '{
            "config": {
                "templates": [
                    "c5bbc697-170e-44bc-8eb7-b944b55ee88f",
                    "4791fa4c-2cef-4f42-8bb4-c86018d71bd3"
                ]
            }
        }'
```

Delete Device

> **Note**
>
> A device must be deactivated before it can be deleted. Otherwise, an `HTTP 403 Forbidden` response will be returned.

```
DELETE /api/v1/controller/device/{id}/
```

Deactivate Device

```
POST /api/v1/controller/device/{id}/deactivate/
```

Activate Device

```
POST /api/v1/controller/device/{id}/activate/
```

List Device Connections

```
GET /api/v1/controller/device/{id}/connection/
```

Create Device Connection

```
POST /api/v1/controller/device/{id}/connection/
```

Get Device Connection Detail

```
GET /api/v1/controller/device/{id}/connection/{id}/
```

Change Device Connection Detail

```
PUT /api/v1/controller/device/{id}/connection/{id}/
```

Patch Device Connection Detail

```
PATCH /api/v1/controller/device/{id}/connection/{id}/
```

Delete Device Connection

```
DELETE /api/v1/controller/device/{id}/connection/{id}/
```

List Credentials

```
GET /api/v1/connection/credential/
```

Create Credential

```
POST /api/v1/connection/credential/
```

Get Credential Detail

```
GET /api/v1/connection/credential/{id}/
```

Change Credential Detail

```
PUT /api/v1/connection/credential/{id}/
```

Patch Credential Detail

```
PATCH /api/v1/connection/credential/{id}/
```

Delete Credential

```
DELETE /api/v1/connection/credential/{id}/
```

List Commands of a Device

```
GET /api/v1/controller/device/{id}/command/
```

Execute a Command a Device

```
POST /api/v1/controller/device/{id}/command/
```

Get Command Details

```
GET /api/v1/controller/device/{device_id}/command/{command_id}/
```

List Device Groups

```
GET /api/v1/controller/group/
```

Available filters

You can filter a list of device groups based on their organization using the `organization_id` or `organization_slug`.

```
GET /api/v1/controller/group/?organization={organization_id}
```

```
GET /api/v1/controller/group/?organization_slug={organization_slug}
```

You can filter a list of device groups that have a device object using the `empty` (e.g. true or false).

```
GET /api/v1/controller/group/?empty={empty}
```

Create Device Group

```
POST /api/v1/controller/group/
```

Get Device Group Detail

```
GET /api/v1/controller/group/{id}/
```

Change Device Group Detail

```
PUT /api/v1/controller/group/{id}/
```

This endpoint allows to change the Group Templates too.

Get Device Group from Certificate Common Name

```
GET /api/v1/controller/cert/{common_name}/group/
```

This endpoint can be used to retrieve group information and metadata by the common name of a certificate used in a VPN client tunnel, this endpoint is used in layer 2 tunneling solutions for firewall/captive portals.

It is also possible to filter device group by providing organization slug of certificate's organization as show in the example below:

```
GET /api/v1/controller/cert/{common_name}/group/?org={org1_slug},{org2_slug}
```

Get Device Location

```
GET /api/v1/controller/device/{id}/location/
```

Create Device Location

```
PUT /api/v1/controller/device/{id}/location/
```

You can create `DeviceLocation` object by using primary keys of existing `Location` and `FloorPlan` objects as shown in the example below.

```
{
    "location": "f0cb5762-3711-4791-95b6-c2f6656249fa",
```

```
    "floorplan": "dfeb6724-aab4-4533-aeab-f7feb6648acd",
    "indoor": "-36,264"
}
```

Note

The `indoor` field represents the coordinates of the point placed on the image from the top left corner. E.g. if you placed the pointer on the top left corner of the floor plan image, its indoor coordinates will be `0,0`.

```
curl -X PUT \
    http://127.0.0.1:8000/api/v1/controller/device/8a85cc23-bad5-4c7e-b9f4-ffe298defb5c/loca
    -H 'authorization: Bearer dc8d497838d4914c9db9aad9b6ec66f6c36ff46b' \
    -H 'content-type: application/json' \
    -d '{
        "location": "f0cb5762-3711-4791-95b6-c2f6656249fa",
        "floorplan": "dfeb6724-aab4-4533-aeab-f7feb6648acd",
        "indoor": "-36,264"
        }'
```

You can also create related `Location` and `FloorPlan` objects for the device directly from this endpoint.

The following example demonstrates creating related location object in a single request.

```
{
    "location": {
        "name": "Via del Corso",
        "address": "Via del Corso, Roma, Italia",
        "geometry": {
            "type": "Point",
            "coordinates": [12.512124, 41.898903]
        },
        "type": "outdoor",
    }
}
```

```
curl -X PUT \
    http://127.0.0.1:8000/api/v1/controller/device/8a85cc23-bad5-4c7e-b9f4-ffe298defb5c/loca
    -H 'authorization: Bearer dc8d497838d4914c9db9aad9b6ec66f6c36ff46b' \
    -H 'content-type: application/json' \
    -d '{
            "location": {
                "name": "Via del Corso",
                "address": "Via del Corso, Roma, Italia",
                "geometry": {
                    "type": "Point",
                    "coordinates": [12.512124, 41.898903]
                },
                "type": "outdoor"
            }
        }'
```

Note

You can also specify the `geometry` in **Well-known text (WKT)** format, like following:

```
{
    "location": {
        "name": "Via del Corso",
        "address": "Via del Corso, Roma, Italia",
        "geometry": "POINT (12.512124 41.898903)",
        "type": "outdoor",
    }
}
```

Similarly, you can create `Floorplan` object with the same request. But, note that a `FloorPlan` can be added to `DeviceLocation` only if the related `Location` object defines an indoor location. The example below demonstrates creating both `Location` and `FloorPlan` objects.

```
{
    "location.name": "Via del Corso",
    "location.address": "Via del Corso, Roma, Italia",
    "location.geometry.type": "Point",
    "location.geometry.coordinates": [12.512124, 41.898903],
    "location.type": "outdoor",
    "floorplan.floor": 1,
    "floorplan.image": "floorplan.png"
}
```

```
curl -X PUT \
    http://127.0.0.1:8000/api/v1/controller/device/8a85cc23-bad5-4c7e-b9f4-ffe298defb5c/loca
    -H 'authorization: Bearer dc8d497838d4914c9db9aad9b6ec66f6c36ff46b' \
    -H 'content-type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW' \
    -F 'location.name=Via del Corso' \
    -F 'location.address=Via del Corso, Roma, Italia' \
    -F location.geometry.type=Point \
    -F 'location.geometry.coordinates=[12.512124, 41.898903]' \
    -F location.type=indoor \
    -F floorplan.floor=1 \
    -F 'floorplan.image=@floorplan.png'
```

Note

The example above uses `multipart content-type` for uploading floor plan image.

You can also use an existing `Location` object and create a new floor plan for that location using this endpoint.

```
{
    "location": "f0cb5762-3711-4791-95b6-c2f6656249fa",
    "floorplan.floor": 1,
    "floorplan.image": "floorplan.png"
}
```

```
curl -X PUT \
    http://127.0.0.1:8000/api/v1/controller/device/8a85cc23-bad5-4c7e-b9f4-ffe298defb5c/loca
    -H 'authorization: Bearer dc8d497838d4914c9db9aad9b6ec66f6c36ff46b' \
    -H 'content-type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW' \
    -F location=f0cb5762-3711-4791-95b6-c2f6656249fa \
    -F floorplan.floor=1 \
    -F 'floorplan.image=@floorplan.png'
```

Change Details of Device Location

```
PUT /api/v1/controller/device/{id}/location/
```

> **Note**
>
> This endpoint can be used to update related `Location` and `Floorplan` objects. Refer to the examples in the "Create device location" section for information on payload format.

Delete Device Location

```
DELETE /api/v1/controller/device/{id}/location/
```

Get Device Coordinates

```
GET /api/v1/controller/device/{id}/coordinates/
```

> **Note**
>
> This endpoint is intended to be used by devices.

This endpoint skips multi-tenancy and permission checks if the device `key` is passed as `query_param` because the system assumes that the device is updating it's position.

```
curl -X GET \
    'http://127.0.0.1:8000/api/v1/controller/device/8a85cc23-bad5-4c7e-b9f4-ffe298defb5c/coo
```

Update Device Coordinates

```
PUT /api/v1/controller/device/{id}/coordinates/
```

> **Note**
>
> This endpoint is intended to be used by devices.

This endpoint skips multi-tenancy and permission checks if the device `key` is passed as `query_param` because the system assumes that the device is updating it's position.

```
{
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [12.512124, 41.898903]
    },
}
```

```
curl -X PUT \
    'http://127.0.0.1:8000/api/v1/controller/device/8a85cc23-bad5-4c7e-b9f4-ffe298defb5c/coo
    -H 'content-type: application/json' \
    -d '{
            "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [12.512124, 41.898903]
            },
        }'
```

List Locations

```
GET /api/v1/controller/location/
```

Available filters

You can filter using `organization_id` or `organization_slug` to get list locations that belongs to an organization.

```
GET /api/v1/controller/location/?organization={organization_id}

GET /api/v1/controller/location/?organization_slug={organization_slug}
```

Create Location

```
POST /api/v1/controller/location/
```

If you are creating an `indoor` location, you can use this endpoint to create floor plan for the location.

The following example demonstrates creating floor plan along with location in a single request.

```
{
    "name": "Via del Corso",
    "address": "Via del Corso, Roma, Italia",
    "geometry.type": "Point",
    "geometry.location": [12.512124, 41.898903],
    "type": "indoor",
    "is_mobile": "false",
    "floorplan.floor": "1",
    "floorplan.image": "floorplan.png",
    "organization": "1f6c5666-1011-4f1d-bce9-fc6fcb4f3a05"
}

curl -X POST \
    http://127.0.0.1:8000/api/v1/controller/location/ \
    -H 'authorization: Bearer dc8d497838d4914c9db9aad9b6ec66f6c36ff46b' \
    -H 'content-type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW' \
    -F 'name=Via del Corso' \
    -F 'address=Via del Corso, Roma, Italia' \
    -F geometry.type=Point \
    -F 'geometry.coordinates=[12.512124, 41.898903]' \
    -F type=indoor \
    -F is_mobile=false \
    -F floorplan.floor=1 \
    -F 'floorplan.image=@floorplan.png' \
    -F organization=1f6c5666-1011-4f1d-bce9-fc6fcb4f3a05
```

Note

You can also specify the `geometry` in **Well-known text (WKT)** format, like following:

```
{
    "name": "Via del Corso",
    "address": "Via del Corso, Roma, Italia",
    "geometry": "POINT (12.512124 41.898903)",
    "type": "indoor",
    "is_mobile": "false",
    "floorplan.floor": "1",
    "floorplan.image": "floorplan.png",
    "organization": "1f6c5666-1011-4f1d-bce9-fc6fcb4f3a05"
}
```

Get Location Details

```
GET /api/v1/controller/location/{pk}/
```

Change Location Details

```
PUT /api/v1/controller/location/{pk}/
```

Note

Only the first floor plan data present can be edited or changed. Setting the `type` of location to outdoor will remove all the floor plans associated with it.

Refer to the examples in the "Create device location" section for information on payload format.

Delete Location

```
DELETE /api/v1/controller/location/{pk}/
```

List Devices in a Location

```
GET /api/v1/controller/location/{id}/device/
```

List Locations with Devices Deployed (in GeoJSON Format)

Note

this endpoint will only list locations that have been assigned to a device.

```
GET /api/v1/controller/location/geojson/
```

Available filters

You can filter using `organization_id` or `organization_slug` to get list location of devices from that organization.

```
GET /api/v1/controller/location/geojson/?organization_id={organization_id}

GET /api/v1/controller/location/geojson/?organization_slug={organization_slug}
```

Floor Plan List

```
GET /api/v1/controller/floorplan/
```

Available filters

You can filter using `organization_id` or `organization_slug` to get list floor plans that belongs to an organization.

```
GET /api/v1/controller/floorplan/?organization={organization_id}

GET /api/v1/controller/floorplan/?organization_slug={organization_slug}
```

Create Floor Plan

```
POST /api/v1/controller/floorplan/
```

Get Floor Plan Details

```
GET /api/v1/controller/floorplan/{pk}/
```

Change Floor Plan Details

```
PUT /api/v1/controller/floorplan/{pk}/
```

Delete Floor Plan

```
DELETE /api/v1/controller/floorplan/{pk}/
```

List Templates

```
GET /api/v1/controller/template/
```

Available filters

You can filter a list of templates based on their organization using the `organization_id` or `organization_slug`.

```
GET /api/v1/controller/template/?organization={organization_id}

GET /api/v1/controller/template/?organization_slug={organization_slug}
```

You can filter a list of templates based on their backend using the `backend` (e.g `netjsonconfig.OpenWrt` or `netjsonconfig.OpenWisp`).

```
GET /api/v1/controller/template/?backend={backend}
```

You can filter a list of templates based on their type using the `type` (e.g. vpn or generic).

```
GET /api/v1/controller/template/?type={type}
```

You can filter a list of templates that are enabled by default or not using the `default` (e.g. true or false).

```
GET /api/v1/controller/template/?default={default}
```

You can filter a list of templates that are required or not using the `required` (e.g. true or false).

```
GET /api/v1/controller/template/?required={required}
```

You can filter a list of templates based on their creation time using the `creation_time`.

```
# Created exact

GET /api/v1/controller/template/?created={creation_time}

# Created greater than or equal to

GET /api/v1/controller/template/?created__gte={creation_time}

# Created is less than

GET /api/v1/controller/template/?created__lt={creation_time}
```

Create Template

```
POST /api/v1/controller/template/
```

Get Template Detail

```
GET /api/v1/controller/template/{id}/
```

Download Template Configuration

```
GET /api/v1/controller/template/{id}/configuration/
```

The above endpoint triggers the download of a `tar.gz` file containing the generated configuration for that specific template.

Change Details of Template

```
PUT /api/v1/controller/template/{id}/
```

Patch Details of Template

```
PATCH /api/v1/controller/template/{id}/
```

Delete Template

```
DELETE /api/v1/controller/template/{id}/
```

List VPNs

```
GET /api/v1/controller/vpn/
```

Available filters

You can filter a list of vpns based on their backend using the `backend` (e.g `openwisp_controller.vpn_backends.OpenVpn` or `openwisp_controller.vpn_backends.Wireguard`).

```
GET /api/v1/controller/vpn/?backend={backend}
```

You can filter a list of vpns based on their subnet using the `subnet_id`.

```
GET /api/v1/controller/vpn/?subnet={subnet_id}
```

You can filter a list of vpns based on their organization using the `organization_id` or `organization_slug`.

```
GET /api/v1/controller/vpn/?organization={organization_id}

GET /api/v1/controller/vpn/?organization_slug={organization_slug}
```

Create VPN

```
POST /api/v1/controller/vpn/
```

Get VPN detail

```
GET /api/v1/controller/vpn/{id}/
```

Download VPN Configuration

```
GET /api/v1/controller/vpn/{id}/configuration/
```

The above endpoint triggers the download of a `tar.gz` file containing the generated configuration for that specific VPN.

Change Details of VPN

```
PUT /api/v1/controller/vpn/{id}/
```

Patch Details of VPN

```
PATCH /api/v1/controller/vpn/{id}/
```

Delete VPN

```
DELETE /api/v1/controller/vpn/{id}/
```

List CA

```
GET /api/v1/controller/ca/
```

Create New CA

```
POST /api/v1/controller/ca/
```

Import Existing CA

```
POST /api/v1/controller/ca/
```

> **Note**
>
> To import an existing CA, only `name`, `certificate` and `private_key` fields have to be filled in the `HTML` form or included in the `JSON` format.

Get CA Detail

```
GET /api/v1/controller/ca/{id}/
```

Change Details of CA

```
PUT /api/v1/controller/ca/{id}/
```

Patch Details of CA

```
PATCH /api/v1/controller/ca/{id}/
```

Download CA(crl)

```
GET /api/v1/controller/ca/{id}/crl/
```

The above endpoint triggers the download of `{id}.crl` file containing up to date CRL of that specific CA.

Delete CA

```
DELETE /api/v1/controller/ca/{id}/
```

Renew CA

```
POST /api/v1/controller/ca/{id}/renew/
```

List Cert

```
GET /api/v1/controller/cert/
```

Create New Cert

```
POST /api/v1/controller/cert/
```

Import Existing Cert

```
POST /api/v1/controller/cert/
```

> **Note**
>
> To import an existing Cert, only `name`, `ca`, `certificate` and `private_key` fields have to be filled in the `HTML` form or included in the `JSON` format.

Get Cert Detail

```
GET /api/v1/controller/cert/{id}/
```

Change Details of Cert

```
PUT /api/v1/controller/cert/{id}/
```

Patch Details of Cert

```
PATCH /api/v1/controller/cert/{id}/
```

Delete Cert

```
DELETE /api/v1/controller/cert/{id}/
```

Renew Cert

```
POST /api/v1/controller/cert/{id}/renew/
```

Revoke Cert

```
POST /api/v1/controller/cert/{id}/revoke/
```

Settings

> **Note**
>
> If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

`OPENWISP_SSH_AUTH_TIMEOUT`

type:	`int`
default:	`2`
unit:	`seconds`

Configure timeout to wait for an authentication response when establishing a SSH connection.

`OPENWISP_SSH_BANNER_TIMEOUT`

type:	`int`
default:	`60`
unit:	`seconds`

Configure timeout to wait for the banner to be presented when establishing a SSH connection.

`OPENWISP_SSH_COMMAND_TIMEOUT`

type:	`int`
default:	`30`
unit:	`seconds`

Configure timeout on blocking read/write operations when executing a command in a SSH connection.

`OPENWISP_SSH_CONNECTION_TIMEOUT`

type:	`int`
default:	`5`
unit:	`seconds`

Configure timeout for the TCP connect when establishing a SSH connection.

`OPENWISP_CONNECTORS`

type:	`tuple`
default:	

```
(
    ("openwisp_controller.connection.connectors.ssh.Ssh", "SSH"),
    (
        "openwisp_controller.connection.connectors.openwrt.snmp.OpenWRTSnmp",
        "OpenWRT SNMP",
    ),
    (
        "openwisp_controller.connection.connectors.airos.snmp.AirOsSnmp",
        "Ubiquiti AirOS SNMP",
    ),
)
```

Available connector classes. Connectors are python classes that specify ways in which OpenWISP can connect to devices in order to launch commands.

OPENWISP_UPDATE_STRATEGIES

type:	`tuple`
default:	`(("openwisp_controller.connection.connectors.openwrt.ssh.OpenWrt", "OpenWRT SSH"), ("openwisp_controller.connection.connectors.openwrt.ssh.OpenWisp1", "OpenWISP 1.x SSH"),)`

Available update strategies. An update strategy is a subclass of a connector class which defines an `update_config` method which is in charge of updating the configuration of the device.

This operation is launched in a background worker when the configuration of a device is changed.

It's possible to write custom update strategies and add them to this setting to make them available in OpenWISP.

OPENWISP_CONFIG_UPDATE_MAPPING

type:	`dict`
default:	`{"netjsonconfig.OpenWrt": OPENWISP_UPDATE_STRATEGIES[0][0],}`

A dictionary that maps configuration backends to update strategies in order to automatically determine the update strategy of a device connection if the update strategy field is left blank by the user.

OPENWISP_CONTROLLER_BACKENDS

type:	`tuple`
default:	`(("netjsonconfig.OpenWrt", "OpenWRT"), ("netjsonconfig.OpenWisp", "OpenWISP"),)`

Available configuration backends. For more information, see netjsonconfig backends.

OPENWISP_CONTROLLER_VPN_BACKENDS

type:	`tuple`

default:	(("openwisp_controller.vpn_backends.OpenVpn", "OpenVPN"), ("openwisp_controller.vpn_backends.Wireguard", "WireGuard"), ("openwisp_controller.vpn_backends.VxlanWireguard", "VXLAN over WireGuard",), ("openwisp_controller.vpn_backends.ZeroTier", "ZeroTier"),)

Available VPN backends for VPN Server objects. For more information, see netjsonconfig VPN backends.

A VPN backend must follow some basic rules in order to be compatible with *openwisp-controller*:

- it MUST allow at minimum and at maximum one VPN instance
- the main *NetJSON* property MUST match the lowercase version of the class name, e.g.: when using the `OpenVpn` backend, the system will look into `config['openvpn']`
- it SHOULD focus on the server capabilities of the VPN software being used

OPENWISP_CONTROLLER_DEFAULT_BACKEND

type:	`str`
default:	`OPENWISP_CONTROLLER_BACKENDS[0][0]`

The preferred backend that will be used as initial value when adding new `Config` or `Template` objects in the admin.

This setting defaults to the raw value of the first item in the `OPENWISP_CONTROLLER_BACKENDS` setting, which is `netjsonconfig.OpenWrt`.

Setting it to `None` will force the user to choose explicitly.

OPENWISP_CONTROLLER_DEFAULT_VPN_BACKEND

type:	`str`
default:	`OPENWISP_CONTROLLER_VPN_BACKENDS[0][0]`

The preferred backend that will be used as initial value when adding new `Vpn` objects in the admin.

This setting defaults to the raw value of the first item in the `OPENWISP_CONTROLLER_VPN_BACKENDS` setting, which is `openwisp_controller.vpn_backends.OpenVpn`.

Setting it to `None` will force the user to choose explicitly.

OPENWISP_CONTROLLER_REGISTRATION_ENABLED

type:	`bool`
default:	`True`

Whether devices can automatically register through the controller or not.

This feature is enabled by default.

Auto-registration must be supported on the devices in order to work, see openwisp-config automatic registration for more information.

OPENWISP_CONTROLLER_CONSISTENT_REGISTRATION

type:	bool
default:	True

Whether devices that are already registered are recognized when reflashed or reset, hence keeping the existing configuration without creating a new one.

This feature is enabled by default.

Auto-registration must be enabled also on the devices in order to work, see openwisp-config consistent key generation for more information.

OPENWISP_CONTROLLER_REGISTRATION_SELF_CREATION

type:	bool
default:	True

Whether devices that are not already present in the system are allowed to register or not.

Turn this off if you still want to use auto-registration to avoid having to manually set the device UUID and key in its configuration file but also want to avoid indiscriminate registration of new devices without explicit permission.

OPENWISP_CONTROLLER_CONTEXT

type:	dict
default:	{}

Additional context that is passed to the default context of each device object.

`OPENWISP_CONTROLLER_CONTEXT` can be used to define system-wide configuration variables.

For more information regarding how to use configuration variables in OpenWISP, refer to Configuration Variables.

For technical information about how variables are handled in the lower levels of OpenWISP, see netjsonconfig context: configuration variables.

OPENWISP_CONTROLLER_DEFAULT_AUTO_CERT

type:	bool
default:	True

The default value of the `auto_cert` field for new `Template` objects.

The `auto_cert` field is valid only for templates which have `type` set to `VPN` and indicates whether configuration regarding the VPN tunnel is provisioned automatically to each device using the template, e.g.:

- when using OpenVPN, new x509 certificates will be generated automatically using the same CA assigned to the related VPN object
- when using WireGuard, new pair of private and public keys (using Curve25519) will be generated, as well as an IP address of the subnet assigned to the related VPN object
- when using VXLAN tunnels over Wireguard, in addition to the configuration generated for Wireguard, a new VID will be generated automatically for each device if the configuration option "auto VNI" is turned on in the VPN object

All these auto generated configuration options will be available as template variables.

The objects that are automatically created will also be removed when they are not needed anymore (e.g.: when the VPN template is removed from a configuration object).

OPENWISP_CONTROLLER_CERT_PATH

type:	`str`
default:	`/etc/x509`

The file system path where x509 certificate will be installed when downloaded on routers when `auto_cert` is being used (enabled by default).

OPENWISP_CONTROLLER_COMMON_NAME_FORMAT

type:	`str`
default:	`{mac_address}-{name}`

Defines the format of the `common_name` attribute of VPN client certificates that are automatically created when using VPN templates which have `auto_cert` set to `True`. A unique slug generated using shortuuid is appended to the common name to introduce uniqueness. Therefore, resulting common names will have `{OPENWISP_CONTROLLER_COMMON_NAME_FORMAT}-{unique-slug}` format.

Note

If the `name` and `mac address` of the device are equal, the `name` of the device will be omitted from the common name to avoid redundancy.

OPENWISP_CONTROLLER_MANAGEMENT_IP_DEVICE_LIST

type:	`bool`
default:	`True`

In the device list page, the column `IP` will show the `management_ip` if available, defaulting to `last_ip` otherwise.

If this setting is set to `False` the `management_ip` won't be shown in the device list page even if present, it will be shown only in the device detail page.

You may set this to `False` if for some reason the majority of your user doesn't care about the management ip address.

OPENWISP_CONTROLLER_CONFIG_BACKEND_FIELD_SHOWN

type:	`bool`
default:	`True`

This setting toggles the `backend` fields in add/edit pages in Device and Template configuration, as well as the `backend` field/filter in Device list and Template list.

If this setting is set to `False` these items will be removed from the UI.

Note

This setting affects only the configuration backend and NOT the VPN backend.

`OPENWISP_CONTROLLER_DEVICE_NAME_UNIQUE`

type:	`bool`
default:	`True`

This setting conditionally enforces unique Device names in an Organization. The query to enforce this is case-insensitive.

Note: For this constraint to be optional, it is enforced on an application level and not on database.

`OPENWISP_CONTROLLER_HARDWARE_ID_ENABLED`

type:	`bool`
default:	`False`

The field `hardware_id` can be used to store a unique hardware id, for example a serial number.

If this setting is set to `True` then this field will be shown first in the device list page and in the add/edit device page.

This feature is disabled by default.

`OPENWISP_CONTROLLER_HARDWARE_ID_OPTIONS`

type:	`dict`
default:	`{ "blank": not OPENWISP_CONTROLLER_HARDWARE_ID_ENABLED, "null": True, "max_length": 32, "unique": True, "verbose_name": _("Serial number"), "help_text": _("Serial number of this device"), }`

Options for the model field `hardware_id`.

- `blank`: whether the field is allowed to be blank
- `null`: whether an empty value will be stored as `NULL` in the database
- `max_length`: maximum length of the field
- `unique`: whether the value of the field must be unique
- `verbose_name`: text for the human readable label of the field
- `help_text`: help text to be displayed with the field

`OPENWISP_CONTROLLER_HARDWARE_ID_AS_NAME`

type:	`bool`

default:	`True`

When the hardware ID feature is enabled, devices will be referenced with their hardware ID instead of their name.

If you still want to reference devices by their name, set this to `False`.

`OPENWISP_CONTROLLER_DEVICE_VERBOSE_NAME`

type:	`tuple`
default:	`('Device', 'Devices')`

Defines the `verbose_name` attribute of the `Device` model, which is displayed in the admin site. The first and second element of the tuple represent the singular and plural forms.

For example, if we want to change the verbose name to "Hotspot", we could write:

```
OPENWISP_CONTROLLER_DEVICE_VERBOSE_NAME = ("Hotspot", "Hotspots")
```

`OPENWISP_CONTROLLER_HIDE_AUTOMATICALLY_GENERATED_SUBNETS_AND_IPS`

type:	`bool`
default:	`False`

Setting this to `True` will hide subnets and IP addresses generated by subnet division rules from being displayed in the list of Subnets and IP addresses in the admin dashboard.

`OPENWISP_CONTROLLER_SUBNET_DIVISION_TYPES`

type:	`tuple`

default:	`(`
	`  (`
	`  "openwisp_controller.subnet_division.rule_types.device.DeviceSubnetDivisio`
	`  "Device",`
	`  ),`
	`  (`
	`  "openwisp_controller.subnet_division.rule_types.vpn.VpnSubnetDivisionRule`
	`  "VPN",`
	`  ),`
	`)`

Available types for Subject Division Rule objects.

For more information on how to write your own types, please refer to: Custom Subnet Division Rule Types.

`OPENWISP_CONTROLLER_API`

type:	`bool`
default:	`True`

Indicates whether the API for Openwisp Controller is enabled or not. To disable the API by default add `OPENWISP_CONTROLLER_API = False` in your project `settings.py` file.

`OPENWISP_CONTROLLER_API_HOST`

type:	`str`
default:	`None`

Allows to specify backend URL for API requests, if the frontend is hosted separately.

`OPENWISP_CONTROLLER_USER_COMMANDS`

type:	`list`
default:	`[]`

Allows to specify a `list` of tuples for adding commands as described in the section: Defining New Options in the Commands Menu.

`OPENWISP_CONTROLLER_ORGANIZATION_ENABLED_COMMANDS`

type:	`dict`
default:	`{ # By default all commands are allowed "__all__": "*", }`

This setting controls the command types that are enabled on the system By default, all command types are enabled to all the organizations, but it's possible to disable a specific command for a specific organization as shown in the following example:

```
OPENWISP_CONTROLLER_ORGANIZATION_ENABLED_COMMANDS = {
    "__all__": "*",
    # Organization UUID: # Tuple of enabled commands
    "7448a190-6e65-42bf-b8ea-bb6603e593a5": ("reboot", "change_password"),
}
```

In the example above, the organization with UUID `7448a190-6e65-42bf-b8ea-bb6603e593a5` will allow to send only commands of type `reboot` and `change_password`, while all the other organizations will have all command types enabled.

`OPENWISP_CONTROLLER_DEVICE_GROUP_SCHEMA`

type:	`dict`
default:	`{'type': 'object', 'properties': {}}`

Allows specifying JSONSchema used for validating the meta-data of Device Groups.

`OPENWISP_CONTROLLER_SHARED_MANAGEMENT_IP_ADDRESS_SPACE`

type:	`bool`
default:	`True`

By default, the system assumes that the address space of the management tunnel is shared among all the organizations using the system, that is, the system assumes there's only one management VPN, tunnel or other networking technology to reach the devices it controls.

When set to `True`, any device belonging to any organization will never have the same `management_ip` as another device, the latest device declaring the management IP will take the IP and any other device who declared the same IP in the past will have the field reset to empty state to avoid potential conflicts.

Set this to `False` if every organization has its dedicated management tunnel with a dedicated address space that is reachable by the OpenWISP server.

`OPENWISP_CONTROLLER_MANAGEMENT_IP_ONLY`

type:	`bool`
default:	`True`

By default, only the management IP will be used to establish connection with the devices.

If the devices are connecting to your OpenWISP instance using a shared layer2 network, hence the OpenWSP server can reach the devices using the `last_ip` field, you can set this to `False`.

`OPENWISP_CONTROLLER_DSA_OS_MAPPING`

type:	`dict`
default:	`{}`

OpenWISP Controller can figure out whether it should use the new OpenWrt syntax for DSA interfaces (Distributed Switch Architecture) introduced in OpenWrt 21 by reading the `os` field of the `Device` object. However, if the firmware you are using has a custom firmware identifier, the system will not be able to figure out whether it should use the new syntax and it will default to OPENWISP_CONTROLLER_DSA_DEFAULT_FALLBACK.

If you want to make sure the system can parse your custom firmware identifier properly, you can follow the example below.

For the sake of the example, the OS identifier `MyCustomFirmware 2.0` corresponds to `OpenWrt 19.07`, while `MyCustomFirmware 2.1` corresponds to `OpenWrt 21.02`. Configuring this setting as indicated below will allow OpenWISP to supply the right syntax automatically.

Example:

```
OPENWISP_CONTROLLER_DSA_OS_MAPPING = {
    "netjsonconfig.OpenWrt": {
        # OpenWrt >=21.02 configuration syntax will be used for
        # these OS identifiers.
        ">=21.02": [r"MyCustomFirmware 2.1(.*)"],
        # OpenWrt <=21.02 configuration syntax will be used for
        # these OS identifiers.
        "<21.02": [r"MyCustomFirmware 2.0(.*)"],
    }
}
```

Note

The OS identifier should be a regular expression as shown in above example.

`OPENWISP_CONTROLLER_DSA_DEFAULT_FALLBACK`

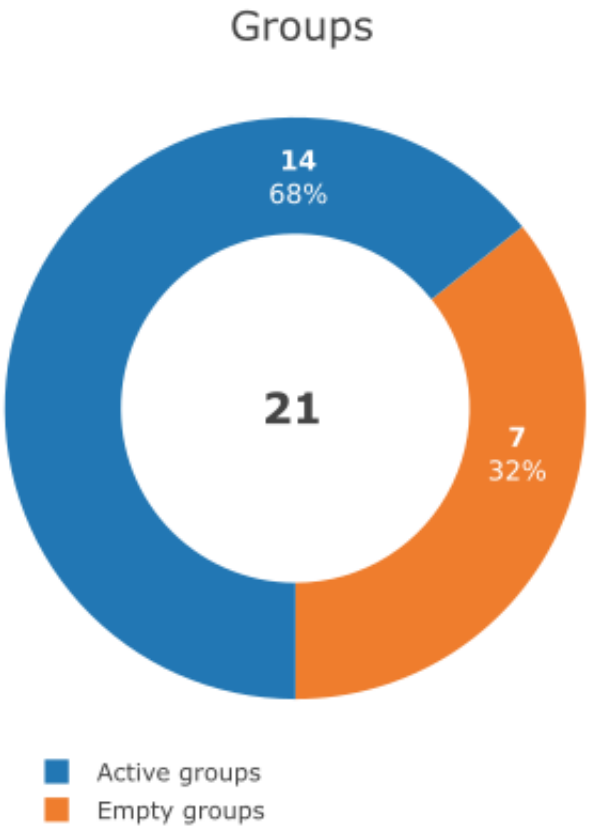
type:	`bool`
default:	`True`

The value of this setting decides whether to use DSA syntax (OpenWrt >=21 configuration syntax) if openwisp-controller fails to make that decision automatically.

OPENWISP_CONTROLLER_GROUP_PIE_CHART

type:	`bool`
default:	`False`

Allows to show a pie chart like the one in the screenshot.



Active groups are groups which have at least one device in them, while empty groups do not have any device assigned.

OPENWISP_CONTROLLER_API_TASK_RETRY_OPTIONS

type:	`dict`
default:	see below

```
# default value of OPENWISP_CONTROLLER_API_TASK_RETRY_OPTIONS:

dict(
    max_retries=5,  # total number of retries
    retry_backoff=True,  # exponential backoff
    retry_backoff_max=600,  # 10 minutes
    retry_jitter=True,  # randomness into exponential backoff
)
```

This setting is utilized by background API tasks executed by ZeroTier VPN servers and ZeroTier VPN clients to handle recoverable HTTP status codes such as 429, 500, 502, 503, and 504.

These tasks are retried with a maximum of 5 attempts with an exponential backoff and jitter, with a maximum delay of 10 minutes.

This feature ensures that ZeroTier Service API calls are resilient to recoverable failures, improving the reliability of the system.

For more information on these settings, you can refer to the the celery documentation regarding automatic retries for known errors.

Developer Docs

Note

This page is for developers who want to customize or extend OpenWISP Controller, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Controller User Docs

Developer Installation Instructions

Note

This page is for developers who want to customize or extend OpenWISP Controller, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Controller User Docs

Dependencies

- Python >= 3.8
- OpenSSL

Installing for Development

Install the system dependencies:

```
sudo apt update
sudo apt install -y sqlite3 libsqlite3-dev openssl libssl-dev
sudo apt install -y gdal-bin libproj-dev libgeos-dev libspatialite-dev libsqlite3-mod-spatia
sudo apt install -y chromium-browser
```

Fork and clone the forked repository:

```
git clone git://github.com/<your_fork>/openwisp-controller
```

Navigate into the cloned repository:

```
cd openwisp-controller/
```

Launch Redis and PostgreSQL:

```
docker-compose up -d redis postgres
```

Setup and activate a virtual-environment (we'll be using virtualenv):

```
python -m virtualenv env
source env/bin/activate
```

Make sure that your base python packages are up to date before moving to the next step:

```
pip install -U pip wheel setuptools
```

Install development dependencies:

```
pip install -e .
pip install -r requirements-test.txt
sudo npm install -g jshint stylelint
```

Install WebDriver for Chromium for your browser version from https://chromedriver.chromium.org/home and Extract `chromedriver` to one of directories from your `$PATH` (example: `~/.local/bin/`).

Create database:

```
cd tests/
./manage.py migrate
./manage.py createsuperuser
```

Launch celery worker (for background jobs):

```
celery -A openwisp2 worker -l info
```

Launch development server:

```
./manage.py runserver 0.0.0.0:8000
```

You can access the admin interface at `http://127.0.0.1:8000/admin/`.

Run tests with:

```
./runtests.py --parallel
```

Some tests, such as the Selenium UI tests, require a PostgreSQL database to run. If you don't have a PostgreSQL database running on your system, you can use the Docker Compose configuration provided in this repository. Once set up, you can run these specific tests as follows:

```
# Run database tests against PostgreSQL backend
POSTGRESQL=1 ./runtests.py --parallel

# Run only specific selenium tests classes
cd tests/
DJANGO_SETTINGS_MODULE=openwisp2.postgresql_settings ./manage.py test openwisp_controller.co
```

Run quality assurance tests with:

```
./run-qa-checks
```

Alternative Sources

Pypi

To install the latest Pypi:

```
pip install openwisp-controller
```

Github

To install the latest development version tarball via HTTPs:

```
pip install https://github.com/openwisp/openwisp-controller/tarball/master
```

Alternatively you can use the git protocol:

```
pip install -e git+git://github.com/openwisp/openwisp-controller#egg=openwisp_controller
```

Install and Run on Docker

> **Warning**
>
> This Docker image is for development purposes only.
>
> For the official OpenWISP Docker images, see: docker-openwisp.

Build from the Dockerfile:

```
docker-compose build
```

Run the docker container:

```
docker-compose up
```

Troubleshooting Steps for Common Installation Issues

You may encounter some issues while installing GeoDjango.

Unable to Load SpatiaLite library Extension?

If you are incurring in the following exception:

```
django.core.exceptions.ImproperlyConfigured: Unable to load the SpatiaLite library extension
```

You need to specify `SPATIALITE_LIBRARY_PATH` in your `settings.py` as explained in django documentation regarding how to install and configure spatialte.

Having Issues with Other Geospatial Libraries?

Please refer troubleshooting issues related to geospatial libraries.

Important

If you want to add OpenWISP Controller to an existing Django project, then you can refer to the test project in the openwisp-controller repository.

Code Utilities

Note

This page is for developers who want to customize or extend OpenWISP Controller, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Controller User Docs

Registering / Unregistering Commands

OpenWISP Controller allows to register new command options or unregister existing command options through two utility functions:

- `openwisp_controller.connection.commands.register_command`
- `openwisp_controller.connection.commands.unregister_command`

You can use these functions to register new custom commands or unregister existing commands from your code.

Note

These functions are to be used as an alternative to the OPENWISP_CONTROLLER_USER_COMMANDS setting when extending openwisp-controller or when developing custom applications based on OpenWISP Controller.

`register_command`

Parameter	Description
`command_name`	A `str` defining identifier for the command.
`command_config`	A `dict` like the one shown in Command Configuration: schema.

Note: It will raise `ImproperlyConfigured` exception if a command is already registered with the same name.

`unregister_command`

Parameter	Description
`command_name`	A `str` defining name of the command.

Note: It will raise `ImproperlyConfigured` exception if such command does not exists.

Controller Notifications

The notification types registered and used by OpenWISP Controller are listed in the following table.

Notification Type	Use
`config_error`	Fires when the status of a device configuration changes to `error`.
`device_registered`	Fires when a new device registers itself.

Registering Notification Types

You can define your own notification types using `register_notification_type` function from OpenWISP Notifications.

For more information, see the relevant documentation section about registering notification types in the Notifications module.

Once a new notification type is registered, you can use the "notify" signal provided by the Notifications module to send notifications with this new type.

Signals

> **Note**
>
> If you're not familiar with signals, please refer to the Django Signals documentation.

`config_modified`

Path: `openwisp_controller.config.signals.config_modified`

Arguments:

- `instance`: instance of `Config` which got its `config` modified
- `previous_status`: indicates the status of the config object before the signal was emitted
- `action`: action which emitted the signal, can be any of the list below: - `config_changed`: the configuration of the config object was changed - `related_template_changed`: the configuration of a related template was changed - `m2m_templates_changed`: the assigned templates were changed (either templates were added, removed or their order was changed)

This signal is emitted every time the configuration of a device is modified.

It does not matter if `Config.status` is already modified, this signal will be emitted anyway because it signals that the device configuration has changed.

This signal is used to trigger the update of the configuration on devices, when the push feature is enabled (requires Device credentials).

The signal is also emitted when one of the templates used by the device is modified or if the templates assigned to the device are changed.

Special cases in which `config_modified` is not emitted

This signal is not emitted when the device is created for the first time.

It is also not emitted when templates assigned to a config object are cleared (`post_clear` m2m signal), this is necessary because sortedm2m, the package we use to implement ordered templates, uses the clear action to reorder templates (m2m relationships are first cleared and then added back), therefore we ignore `post_clear` to avoid emitting signals twice (one for the clear action and one for the add action). Please keep this in mind if you plan on using the clear method of the m2m manager.

`config_status_changed`

Path: `openwisp_controller.config.signals.config_status_changed`

Arguments:

- `instance`: instance of `Config` which got its `status` changed

This signal is emitted only when the configuration status of a device has changed.

The signal is emitted also when the m2m template relationships of a config object are changed, but only on `post_add` or `post_remove` actions, `post_clear` is ignored for the same reason explained in the previous section.

`config_deactivating`

Path: `openwisp_controller.config.signals.config_deactivating`

Arguments:

- `instance`: instance of the object being deactivated
- `previous_status`: previous status of the object before deactivation

This signal is emitted when a configuration status of device is set to `deactivating`.

`config_deactivated`

Path: `openwisp_controller.config.signals.config_deactivated`

Arguments:

- `instance`: instance of the object being deactivated
- `previous_status`: previous status of the object before deactivation

This signal is emitted when a configuration status of device is set to `deactivated`.

`device_deactivated`

Path: `openwisp_controller.config.signals.device_deactivated`

Arguments:

- `instance`: instance of the device being deactivated

This signal is emitted when a device is flagged for deactivation.

`device_activated`

Path: `openwisp_controller.config.signals.device_activated`

Arguments:

- `instance`: instance of the device being activated

This signal is emitted when a device is flagged for activation (after deactivation).

`config_backend_changed`

Path: `openwisp_controller.config.signals.config_backend_changed` **Arguments**:

- `instance`: instance of `Config` which got its `backend` changed
- `old_backend`: the old backend of the config object
- `backend`: the new backend of the config object

It is not emitted when the device or config is created.

`checksum_requested`

Path: `openwisp_controller.config.signals.checksum_requested`

Arguments:

- `instance`: instance of `Device` for which its configuration checksum has been requested
- `request`: the HTTP request object

This signal is emitted when a device requests a checksum via the controller views.

The signal is emitted just before a successful response is returned, it is not sent if the response was not successful.

`config_download_requested`

Path: `openwisp_controller.config.signals.config_download_requested`

Arguments:

- `instance`: instance of `Device` for which its configuration has been requested for download
- `request`: the HTTP request object

This signal is emitted when a device requests to download its configuration via the controller views.

The signal is emitted just before a successful response is returned, it is not sent if the response was not successful.

`is_working_changed`

Path: `openwisp_controller.connection.signals.is_working_changed`

Arguments:

- `instance`: instance of `DeviceConnection`
- `is_working`: value of `DeviceConnection.is_working`
- `old_is_working`: previous value of `DeviceConnection.is_working`, either `None` (for new connections), `True` or `False`
- `failure_reason`: error message explaining reason for failure in establishing connection
- `old_failure_reason`: previous value of `DeviceConnection.failure_reason`

This signal is emitted every time `DeviceConnection.is_working` changes.

It is not triggered when the device is created for the first time.

`management_ip_changed`

Path: `openwisp_controller.config.signals.management_ip_changed`

Arguments:

- `instance`: instance of `Device`
- `management_ip`: value of `Device.management_ip`
- `old_management_ip`: previous value of `Device.management_ip`

This signal is emitted every time `Device.management_ip` changes.

It is not triggered when the device is created for the first time.

`device_registered`

Path: `openwisp_controller.config.signals.device_registered`

Arguments:

- `instance`: instance of `Device` which got registered.
- `is_new`: boolean, will be `True` when the device is new, `False` when the device already exists (e.g.: a device which gets a factory reset will register again)

This signal is emitted when a device registers automatically through the controller HTTP API.

`device_name_changed`

Path: `openwisp_controller.config.signals.device_name_changed`

Arguments:

- `instance`: instance of `Device`.

The signal is emitted when the device name changes.

It is not emitted when the device is created.

device_group_changed

Path: `openwisp_controller.config.signals.device_group_changed`

Arguments:

- `instance`: instance of `Device`.
- `group_id`: primary key of `DeviceGroup` of `Device`
- `old_group_id`: primary key of previous `DeviceGroup` of `Device`

The signal is emitted when the device group changes.

It is not emitted when the device is created.

group_templates_changed

Path: `openwisp_controller.config.signals.group_templates_changed`

Arguments:

- `instance`: instance of `DeviceGroup`.
- `templates`: list of `Template` objects assigned to `DeviceGroup`
- `old_templates`: list of `Template` objects assigned earlier to `DeviceGroup`

The signal is emitted when the device group templates changes.

It is not emitted when the device is created.

subnet_provisioned

Path: `openwisp_controller.subnet_division.signals.subnet_provisioned`

Arguments:

- `instance`: instance of `VpnClient`.
- `provisioned`: dictionary of `Subnet` and `IpAddress` provisioned, `None` if nothing is provisioned

The signal is emitted when subnets and IP addresses have been provisioned for a `VpnClient` for a VPN server with a subnet with subnet division rule.

vpn_server_modified

Path: `openwisp_controller.config.signals.vpn_server_modified`

Arguments:

- `instance`: instance of `Vpn`.

The signal is emitted when the VPN server is modified.

vpn_peers_changed

Path: `openwisp_controller.config.signals.vpn_peers_changed`

Arguments:

- `instance`: instance of `Vpn`.

The signal is emitted when the peers of VPN server gets changed.

It is only emitted for `Vpn` object with **WireGuard** or **VXLAN over WireGuard** backend.

Extending OpenWISP Controller

Note

This page is for developers who want to customize or extend OpenWISP Controller, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Controller User Docs

One of the core values of the OpenWISP project is Software Reusability, for this reason *OpenWISP Controller* provides a set of base classes which can be imported, extended and reused to create derivative apps.

In order to implement your custom version of *OpenWISP Controller*, you need to perform the steps described in this section.

When in doubt, the code in the test project will serve you as source of truth: just replicate and adapt that code to get a basic derivative of *OpenWISP Controller* working.

If you want to add new users fields, please follow the tutorial to extend the openwisp-users module. As an example, we have extended *openwisp-users* to *sample_users* app and added a field `social_security_number` in the sample_users/models.py.

Important

If you plan on using a customized version of this module, we suggest to start with it since the beginning, because migrating your data from the default module to your extended version may be time consuming.

1. Initialize Your Project & Custom Apps

Firstly, to get started you need to create a django project:

```
django-admin startproject mycontroller
```

Now, you need to do is to create some new django apps which will contain your custom version of *OpenWISP Controller*.

A django project is a collection of django apps. There are 4 django apps in the openwisp_controller project, namely *config*, *pki*, *connection* & *geo*. You'll need to create 4 apps in your project for each app in *openwisp-controller*.

A django app is nothing more than a python package (a directory of python scripts), in the following examples we'll call these django app `sample_config`, `sample_pki`, `sample_connection`, `sample_geo` & `sample_subnet_division`. but you can name it how you want:

```
django-admin startapp sample_config
django-admin startapp sample_pki
django-admin startapp sample_connection
django-admin startapp sample_geo
django-admin startapp sample_subnet_division
```

Keep in mind that the command mentioned above must be called from a directory which is available in your PYTHON_PATH so that you can then import the result into your project.

For more information about how to work with django projects and django apps, please refer to the django documentation.

2. Install `openwisp-controller`

Install (and add to the requirement of your project) openwisp-controller:

```
pip install openwisp-controller
```

3. Add Your Apps to `INSTALLED_APPS`

Now you need to add `mycontroller.sample_config`, `mycontroller.sample_pki`, `mycontroller.sample_connection`, `mycontroller.sample_geo` & `mycontroller.sample_subnet_division` to `INSTALLED_APPS` in your `settings.py`, ensuring also that `openwisp_controller.config`, `openwisp_controller.geo`, `openwisp_controller.pki`, `openwisp_controller.connnection` & `openwisp_controller.subnet_division` have been removed:

```
# Remember: Order in INSTALLED_APPS is important.
INSTALLED_APPS = [
    # other django installed apps
    "openwisp_utils.admin_theme",
    "admin_auto_filters",
    # all-auth
    "django.contrib.sites",
    "allauth",
    "allauth.account",
    "allauth.socialaccount",
    # openwisp2 module
    # 'openwisp_controller.config', <-- comment out or delete this line
    # 'openwisp_controller.pki', <-- comment out or delete this line
    # 'openwisp_controller.geo', <-- comment out or delete this line
```

```
    # 'openwisp_controller.connection', <-- comment out or delete this line
    # 'openwisp_controller.subnet_division', <-- comment out or delete this line
    "mycontroller.sample_config",
    "mycontroller.sample_pki",
    "mycontroller.sample_geo",
    "mycontroller.sample_connection",
    "mycontroller.sample_subnet_division",
    "openwisp_users",
    # admin
    "django.contrib.admin",
    # other dependencies
    "sortedm2m",
    "reversion",
    "leaflet",
    # rest framework
    "rest_framework",
    "rest_framework_gis",
    # channels
    "channels",
    # django-import-export
    "import_export",
]
```

Substitute `mycontroller`, `sample_config`, `sample_pki`, `sample_connection`, `sample_geo` & `sample_subnet_division` with the name you chose in step 1.

4. Add `EXTENDED_APPS`

Add the following to your `settings.py`:

```
EXTENDED_APPS = (
    "django_x509",
    "django_loci",
    "openwisp_controller.config",
    "openwisp_controller.pki",
    "openwisp_controller.geo",
    "openwisp_controller.connection",
    "openwisp_controller.subnet_division",
)
```

5. Add `openwisp_utils.staticfiles.DependencyFinder`

Add `openwisp_utils.staticfiles.DependencyFinder` to `STATICFILES_FINDERS` in your `settings.py`:

```
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
    "openwisp_utils.staticfiles.DependencyFinder",
]
```

6. Add `openwisp_utils.loaders.DependencyLoader`

Add `openwisp_utils.loaders.DependencyLoader` to `TEMPLATES` in your `settings.py`, but ensure it comes before `django.template.loaders.app_directories.Loader`:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "OPTIONS": {
            "loaders": [
```

```
                "django.template.loaders.filesystem.Loader",
                "openwisp_utils.loaders.DependencyLoader",
                "django.template.loaders.app_directories.Loader",
            ],
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
                "openwisp_utils.admin_theme.context_processor.menu_items",
                "openwisp_notifications.context_processors.notification_api_settings",
            ],
        },
    }
]
```

7. Initial Database Setup

Ensure you are using one of the available geodjango backends, e.g.:

```
DATABASES = {
    "default": {
        "ENGINE": "openwisp_utils.db.backends.spatialite",
        "NAME": "openwisp-controller.db",
    }
}
```

For more information about GeoDjango, please refer to the geodjango documentation.

8. Django Channels Setup

Create `asgi.py` in your project folder and add following lines in it:

```
from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from django.core.asgi import get_asgi_application

from openwisp_controller.routing import get_routes

# You can also add your routes like this
from my_app.routing import my_routes

application = ProtocolTypeRouter(
    {
        "http": get_asgi_application(),
        "websocket": AllowedHostsOriginValidator(
            AuthMiddlewareStack(URLRouter(get_routes() + my_routes))
        ),
    }
)
```

9. Other Settings

Add the following settings to `settings.py`:

```
FORM_RENDERER = "django.forms.renderers.TemplatesSetting"

ASGI_APPLICATION = "my_project.asgi.application"
CHANNEL_LAYERS = {
```

```
    "default": {"BACKEND": "channels.layers.InMemoryChannelLayer"},
}
```

For more information about FORM_RENDERER setting, please refer to the FORM_RENDERER documentation. For more information about ASGI_APPLICATION setting, please refer to the ASGI_APPLICATION documentation. For more information about CHANNEL_LAYERS setting, please refer to the CHANNEL_LAYERS documentation.

10. Inherit the AppConfig Class

Please refer to the following files in the sample app of the test project:

- `sample_config`:
    - sample_config/__init__.py.
    - sample_config/apps.py.
- `sample_geo`:
    - sample_geo/__init__.py.
    - sample_geo/apps.py.
- `sample_pki`:
    - sample_pki/__init__.py.
    - sample_pki/apps.py.
- `sample_connection`:
    - sample_connection/__init__.py.
    - sample_connection/apps.py.
- `sample_subnet_division`:
    - sample_subnet_division/__init__.py.
    - sample_subnet_division/apps.py.

You have to replicate and adapt that code in your project.

For more information regarding the concept of `AppConfig` please refer to the "Applications" section in the django documentation.

11. Create Your Custom Models

For the purpose of showing an example, we added a simple "details" field to the models of the sample app in the test project.

- sample_config models
- sample_geo models
- sample_pki models
- sample_connection models
- sample_subnet_division

You can add fields in a similar way in your `models.py` file.

Note

If you have any doubt regarding how to use, extend or develop models please refer to the "Models" section in the django documentation.

12. Add Swapper Configurations

Once you have created the models, add the following to your `settings.py`:

```
# Setting models for swapper module
CONFIG_DEVICE_MODEL = "sample_config.Device"
CONFIG_DEVICEGROUP_MODEL = "sample_config.DeviceGroup"
CONFIG_CONFIG_MODEL = "sample_config.Config"
CONFIG_TEMPLATETAG_MODEL = "sample_config.TemplateTag"
CONFIG_TAGGEDTEMPLATE_MODEL = "sample_config.TaggedTemplate"
CONFIG_TEMPLATE_MODEL = "sample_config.Template"
CONFIG_VPN_MODEL = "sample_config.Vpn"
CONFIG_VPNCLIENT_MODEL = "sample_config.VpnClient"
CONFIG_ORGANIZATIONCONFIGSETTINGS_MODEL = (
    "sample_config.OrganizationConfigSettings"
)
CONFIG_ORGANIZATIONLIMITS_MODEL = "sample_config.OrganizationLimits"
DJANGO_X509_CA_MODEL = "sample_pki.Ca"
DJANGO_X509_CERT_MODEL = "sample_pki.Cert"
GEO_LOCATION_MODEL = "sample_geo.Location"
GEO_FLOORPLAN_MODEL = "sample_geo.FloorPlan"
GEO_DEVICELOCATION_MODEL = "sample_geo.DeviceLocation"
CONNECTION_CREDENTIALS_MODEL = "sample_connection.Credentials"
CONNECTION_DEVICECONNECTION_MODEL = "sample_connection.DeviceConnection"
CONNECTION_COMMAND_MODEL = "sample_connection.Command"
SUBNET_DIVISION_SUBNETDIVISIONRULE_MODEL = (
    "sample_subnet_division.SubnetDivisionRule"
)
SUBNET_DIVISION_SUBNETDIVISIONINDEX_MODEL = (
    "sample_subnet_division.SubnetDivisionIndex"
)
```

Substitute `sample_config`, `sample_pki`, `sample_connection`, `sample_geo` & `sample_subnet_division` with the name you chose in step 1.

13. Create Database Migrations

Create database migrations:

```
./manage.py makemigrations
```

Now, to use the default `administrator` and `operator` user groups like the used in the openwisp_controller module, you'll manually need to make a migrations file which would look like:

- sample_config/migrations/0002_default_groups_permissions.py
- sample_geo/migrations/0002_default_group_permissions.py
- sample_pki/migrations/0002_default_group_permissions.py
- sample_connection/migrations/0002_default_group_permissions.py
- sample_subnet_division/migrations/0002_default_group_permissions.py

Create database migrations:

```
./manage.py migrate
```

For more information, refer to the "Migrations" section in the django documentation.

14. Create the Admin

Refer to the `admin.py` file of the sample app.

- sample_config admin.py.

- sample_geo admin.py.
- sample_pki admin.py.
- sample_connection admin.py.
- sample_subnet_division admin.py.

To introduce changes to the admin, you can do it in two main ways which are described below.

> **Note**
>
> For more information regarding how the django admin works, or how it can be customized, please refer to "The django admin site" section in the django documentation.

14.1. Monkey Patching

If the changes you need to add are relatively small, you can resort to monkey patching.

For example:

sample_config

```
from openwisp_controller.config.admin import (
    DeviceAdmin,
    DeviceGroupAdmin,
    TemplateAdmin,
    VpnAdmin,
)

DeviceAdmin.fields += ["example"]  # <-- monkey patching example
```

sample_connection

```
from openwisp_controller.connection.admin import CredentialsAdmin

CredentialsAdmin.fields += ["example"]  # <-- monkey patching example
```

sample_geo

```
from openwisp_controller.geo.admin import FloorPlanAdmin, LocationAdmin

FloorPlanAdmin.fields += ["example"]  # <-- monkey patching example
```

sample_pki

```
from openwisp_controller.pki.admin import CaAdmin, CertAdmin

CaAdmin.fields += ["example"]  # <-- monkey patching example
```

```
sample_subnet_division
```

```
from openwisp_controller.subnet_division.admin import (
    SubnetDivisionRuleInlineAdmin,
)

SubnetDivisionRuleInlineAdmin.fields += [
    "example"
]  # <-- monkey patching example
```

14.2. Inheriting admin classes

If you need to introduce significant changes and/or you don't want to resort to monkey patching, you can proceed as follows:

sample_config

```
from django.contrib import admin
from openwisp_controller.config.admin import (
    DeviceAdmin as BaseDeviceAdmin,
    TemplateAdmin as BaseTemplateAdmin,
    VpnAdmin as BaseVpnAdmin,
    DeviceGroupAdmin as BaseDeviceGroupAdmin,
)
from swapper import load_model

Vpn = load_model("openwisp_controller", "Vpn")
Device = load_model("openwisp_controller", "Device")
DeviceGroup = load_model("openwisp_controller", "DeviceGroup")
Template = load_model("openwisp_controller", "Template")

admin.site.unregister(Vpn)
admin.site.unregister(Device)
admin.site.unregister(DeviceGroup)
admin.site.unregister(Template)


@admin.register(Vpn)
class VpnAdmin(BaseVpnAdmin):
    # add your changes here
    pass


@admin.register(Device)
class DeviceAdmin(BaseDeviceAdmin):
    # add your changes here
    pass


@admin.register(DeviceGroup)
class DeviceGroupAdmin(BaseDeviceGroupAdmin):
    # add your changes here
    pass


@admin.register(Template)
class TemplateAdmin(BaseTemplateAdmin):
    # add your changes here
    pass
```

sample_connection

```
from openwisp_controller.connection.admin import (
    CredentialsAdmin as BaseCredentialsAdmin,
)
from django.contrib import admin
from swapper import load_model

Credentials = load_model("openwisp_controller", "Credentials")

admin.site.unregister(Credentials)


@admin.register(Device)
class CredentialsAdmin(BaseCredentialsAdmin):
    pass
    # add your changes here
```

sample_geo

```
from openwisp_controller.geo.admin import (
    FloorPlanAdmin as BaseFloorPlanAdmin,
    LocationAdmin as BaseLocationAdmin,
)
from django.contrib import admin
from swapper import load_model

Location = load_model("openwisp_controller", "Location")
FloorPlan = load_model("openwisp_controller", "FloorPlan")

admin.site.unregister(FloorPlan)
admin.site.unregister(Location)


@admin.register(FloorPlan)
class FloorPlanAdmin(BaseFloorPlanAdmin):
    pass
    # add your changes here


@admin.register(Location)
class LocationAdmin(BaseLocationAdmin):
    pass
    # add your changes here
```

```
sample_pki
```

```
from openwisp_controller.geo.admin import (
    CaAdmin as BaseCaAdmin,
    CertAdmin as BaseCertAdmin,
)
from django.contrib import admin
from swapper import load_model

Ca = load_model("openwisp_controller", "Ca")
Cert = load_model("openwisp_controller", "Cert")

admin.site.unregister(Ca)
admin.site.unregister(Cert)


@admin.register(Ca)
class CaAdmin(BaseCaAdmin):
    pass
    # add your changes here


@admin.register(Cert)
class CertAdmin(BaseCertAdmin):
    pass
    # add your changes here
```

sample_subnet_division

```
from openwisp_controller.subnet_division.admin import (
    SubnetAdmin as BaseSubnetAdmin,
    IpAddressAdmin as BaseIpAddressAdmin,
    SubnetDivisionRuleInlineAdmin as BaseSubnetDivisionRuleInlineAdmin,
)
from django.contrib import admin
from swapper import load_model

Subnet = load_model("openwisp_ipam", "Subnet")
IpAddress = load_model("openwisp_ipam", "IpAddress")
SubnetDivisionRule = load_model("subnet_division", "SubnetDivisionRule")

admin.site.unregister(Subnet)
admin.site.unregister(IpAddress)
admin.site.unregister(SubnetDivisionRule)


@admin.register(Subnet)
class SubnetAdmin(BaseSubnetAdmin):
    pass
    # add your changes here


@admin.register(IpAddress)
class IpAddressAdmin(BaseIpAddressAdmin):
    pass
    # add your changes here


@admin.register(SubnetDivisionRule)
class SubnetDivisionRuleInlineAdmin(BaseSubnetDivisionRuleInlineAdmin):
    pass
    # add your changes here
```

15. Create Root URL Configuration

```
from django.contrib import admin
from openwisp_controller.config.utils import get_controller_urls
from openwisp_controller.geo.utils import get_geo_urls

# from .sample_config import views as config_views
# from .sample_geo import views as geo_views

urlpatterns = [
    # ... other urls in your project ...
    # Use only when changing controller API views (discussed below)
    # url(r'^controller/', include((get_controller_urls(config_views), 'controller'), namesp
    # Use only when changing geo API views (discussed below)
    # url(r'^geo/', include((get_geo_urls(geo_views), 'geo'), namespace='geo')),
    # openwisp-controller urls
    url(
        r"",
        include(
            ("openwisp_controller.config.urls", "config"),
            namespace="config",
        ),
    ),
    url(r"", include("openwisp_controller.urls")),
]
```

For more information about URL configuration in django, please refer to the "URL dispatcher" section in the django documentation.

16. Import the Automated Tests

When developing a custom application based on this module, it's a good idea to import and run the base tests too, so that you can be sure the changes you're introducing are not breaking some of the existing features of *OpenWISP Controller*.

In case you need to add breaking changes, you can overwrite the tests defined in the base classes to test your own behavior.

See the tests in sample_app to find out how to do this.

- project common tests.py
- sample_config tests.py
- sample_geo tests.py
- sample_geo pytest.py
- sample_pki tests.py
- sample_connection tests.py
- sample_subnet_division tests.py

For running the tests, you need to copy fixtures as well:

- Change *sample_config* to your config app's name in sample_config fixtures and paste it in the `sample_config/fixtures/` directory.

You can then run tests with:

```
# the --parallel flag is optional
./manage.py test --parallel mycontroller
```

Substitute `mycontroller` with the name you chose in step 1.

For more information about automated tests in django, please refer to "Testing in Django".

Other Base Classes that Can Be Inherited and Extended

The following steps are not required and are intended for more advanced customization.

1. Extending the Controller API Views

Extending the sample_config/views.py is required only when you want to make changes in the controller API, Remember to change `config_views` location in `urls.py` in point 11 for extending views.

For more information about django views, please refer to the views section in the django documentation.

2. Extending the Geo API Views

Extending the sample_geo/views.py is required only when you want to make changes in the geo API, Remember to change `geo_views` location in `urls.py` in point 11 for extending views.

For more information about django views, please refer to the views section in the django documentation.

Custom Subnet Division Rule Types

It is possible to create your own subnet division rule types. The rule type determines when subnets and IPs will be provisioned and when they will be destroyed.

You can create your custom rule types by extending `openwisp_controller.subnet_division.rule_types.base.BaseSubnetDivisionRuleType`.

Below is an example to create a subnet division rule type that will provision subnets and IPs when a new device is created and will delete them upon deletion for that device.

```
# In mycontroller/sample_subnet_division/rules_types/custom.py

from django.db.models.signals import post_delete, post_save
from swapper import load_model

from openwisp_controller.subnet_division.rule_types.base import (
    BaseSubnetDivisionRuleType,
)

Device = load_model("config", "Device")


class CustomRuleType(BaseSubnetDivisionRuleType):
    # The signal on which provisioning should be triggered
    provision_signal = post_save
    # The sender of the provision_signal
    provision_sender = Device
    # Dispatch UID for connecting provision_signal to provision_receiver
    provision_dispatch_uid = "some_unique_identifier_string"

    # The signal on which deletion should be triggered
    destroyer_signal = post_delete
    # The sender of the destroyer_signal
    destroyer_sender = Device
    # Dispatch UID for connecting destroyer_signal to destroyer_receiver
    destroyer_dispatch_uid = "another_unique_identifier_string"

    # Attribute path to organization_id
    # Example 1: If organization_id is direct attribute of provision_signal
    #            sender instance, then
    #   organization_id_path = 'organization_id'
    # Example 2: If organization_id is indirect attribute of provision signal
```

```
        #            sender instance, then
        #   organization_id_path = 'some_attribute.another_intermediate.organization_id'
        organization_id_path = "organization_id"

        # Similar to organization_id_path but for the required subnet attribute
        subnet_path = "subnet"

        # An intermediate method through which you can specify conditions for provisions
        @classmethod
        def should_create_subnets_ips(cls, instance, **kwargs):
            # Using "post_save" provision_signal, the rule should be only
            # triggered when a new object is created.
            return kwargs["created"]

        # You can define logic to trigger provisioning for existing objects
        # using following classmethod. By default, BaseSubnetDivisionRuleType
        # performs no operation for existing objects.
        @classmethod
        def provision_for_existing_objects(cls, rule_obj):
            for device in Device.objects.filter(
                organization=rule_obj.organization
            ):
                cls.provision_receiver(device, created=True)
```

After creating a class for your custom rule type, you will need to set OPENWISP_CONTROLLER_SUBNET_DIVISION_TYPES setting as follows:

```
OPENWISP_CONTROLLER_SUBNET_DIVISION_TYPES = (
    (
        "openwisp_controller.subnet_division.rule_types.vpn.VpnSubnetDivisionRuleType",
        "VPN",
    ),
    (
        "openwisp_controller.subnet_division.rule_types.device.DeviceSubnetDivisionRuleType",
        "Device",
    ),
    (
        "mycontroller.sample_subnet_division.rules_types.custom.CustomRuleType",
        "Custom Rule",
    ),
)
```

More Utilities to Extend OpenWISP Controller

See Code Utilities.

Other useful resources:

- REST API Reference
- Settings

Monitoring

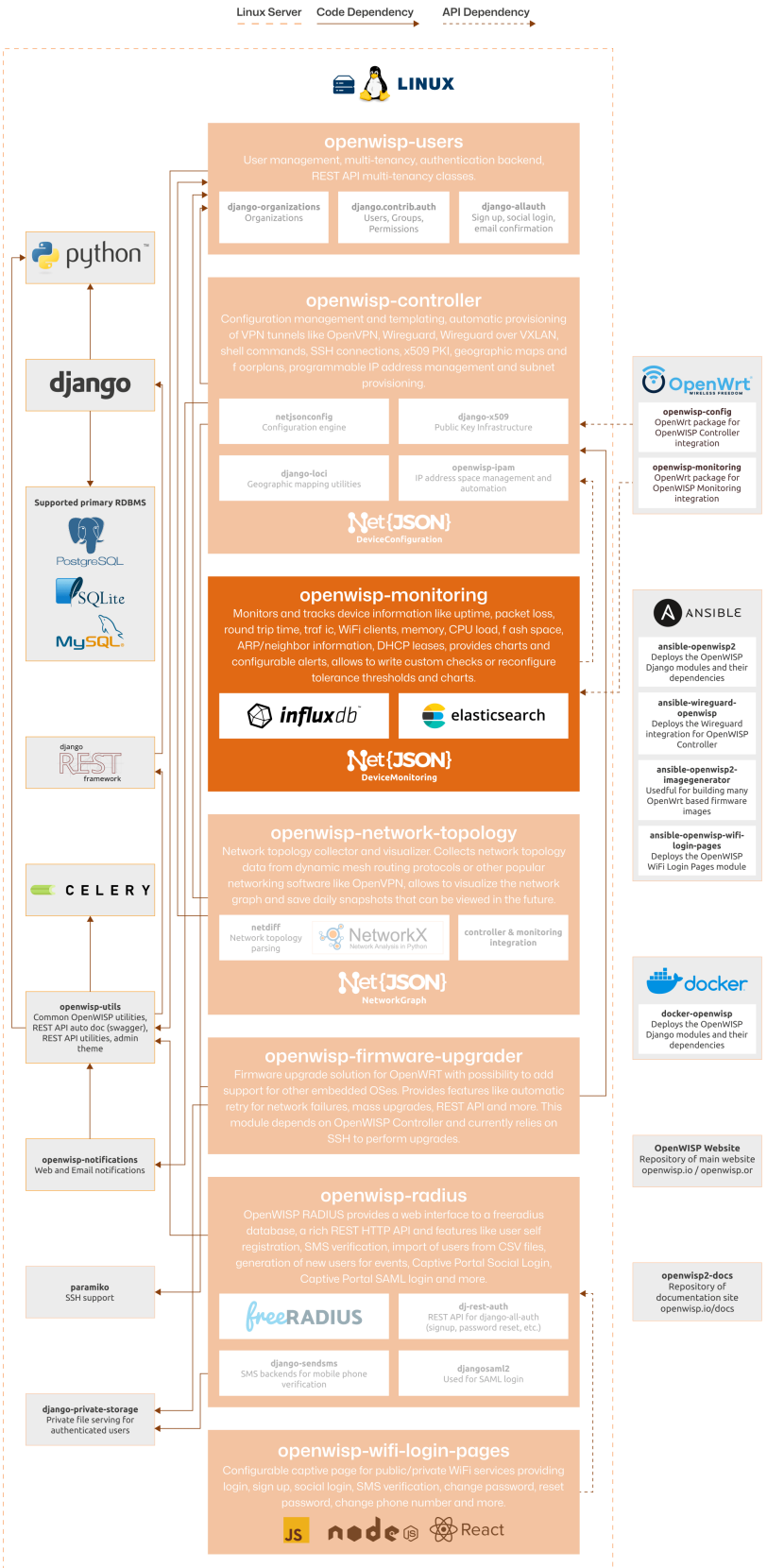
Seealso

Source code: github.com/openwisp/openwisp-monitoring.

The OpenWISP Monitoring module leverages the capabilities of Python and the Django Framework to provide OpenWISP with robust network monitoring features. Designed to be extensible, programmable, scalable, and user-friendly, this module automates monitoring checks, alerts, and metric collection, ensuring efficient and comprehensive network management.

For a comprehensive overview of its features, please refer to the Monitoring: Features page.

The following diagram illustrates the role of the Monitoring module within the OpenWISP architecture.



OpenWISP Architecture: highlighted monitoring module

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

Monitoring: Features

OpenWISP provides the following monitoring capabilities:

- An overview of the status of the network is shown in the admin dashboard, a chart shows the percentages of devices which are online, offline or having issues; there are also two timeseries charts which show the total unique WiFI clients and the traffic flowing to the network, a geographic map is also available for those who use the geographic features of OpenWISP
- Collection of monitoring information in a timeseries database (currently only **InfluxDB** is supported)
- Allows to browse alerts easily from the user interface with one click
- Collects and displays device status information like uptime, RAM status, CPU load averages, Interface properties and addresses, WiFi interface status and associated clients, Neighbors information, DHCP Leases, Disk/Flash status
- Monitoring charts for ping success rate, packet loss, round trip time (latency), associated wifi clients, interface traffic, RAM usage, CPU load, flash/disk usage, mobile signal (LTE/UMTS/GSM signal strength, signal quality, access technology in use), bandwidth, transferred data, restransmits, jitter, datagram, datagram loss
- Maintains a record of WiFi sessions with clients' MAC address and vendor, session start and stop time and connected device along with other information
- Charts can be viewed at resolutions of the last 1 day, 3 days, 7 days, 30 days, and 365 days
- Configurable alerts
- CSV Export of monitoring data
- Possibility to configure additional Metrics and Charts
- Extensible active check system: it's possible to write additional checks that are run periodically using python classes
- Extensible metrics and charts: it's possible to define new metrics and new charts
- API to retrieve the chart metrics and status information of each device based on NetJSON DeviceMonitoring
- Iperf3 check that provides network performance measurements such as maximum achievable bandwidth, jitter, datagram loss etc of the openwrt device using iperf3 utility

Quick Start Guide

Install Monitoring Packages on the Device

First of all, Install the OpenWrt Monitoring Agent on your device.

The agent is responsible for collecting some of the monitoring metrics from the device and sending these to the server. It's required to collect interface traffic, WiFi clients, CPU load, memory usage, storage usage, cellular signal strength, etc.

Make Sure OpenWISP can Reach your Devices

Please make sure that OpenWISP can reach your devices.

Device Health Status

The possible values for the health status field (`DeviceMonitoring.status`) are explained below.

`UNKNOWN`

Whenever a new device is created it will have `UNKNOWN` as it's default Heath Status.

It implies that the system doesn't know whether the device is reachable yet.

`OK`

Everything is working normally.

`PROBLEM`

One of the metrics has a value which is not in the expected range (the threshold value set in the alert settings has been crossed).

Example: CPU usage should be less than 90% but current value is at 95%.

`CRITICAL`

One of the metrics defined in `OPENWISP_MONITORING_CRITICAL_DEVICE_METRICS` has a value which is not in the expected range (the threshold value set in the alert settings has been crossed).

Example: ping is by default a critical metric which is expected to be always 1 (reachable).

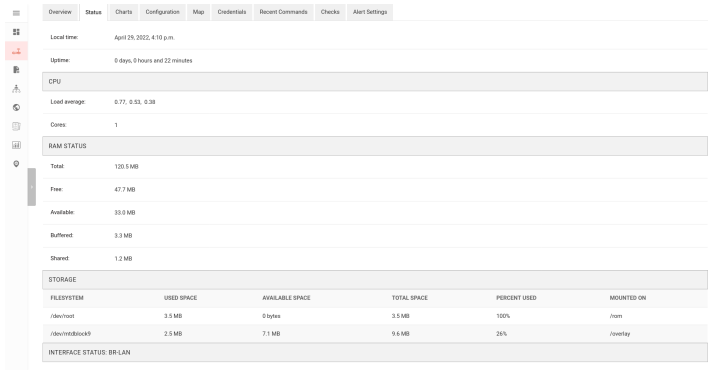
`DEACTIVATED`

The device is deactivated. All active and passive checks are disabled.

Metrics

Device Status

This metric stores the status of the device for viewing purposes.



INTERFACE STATUS: BR-LAN

ADDRESS / MASK	PROTOCOL
192.168.254.1 / 24	static
192.168.1.39 / 24	dhcp

INTERFACE STATUS: WLAN1

Type:	Wireless
Mode:	access point
SSID:	publicwifi-demo
Channel:	11
Frequency:	2.462 GHz
Transmission Power:	20 dBm
Signal:	-60 dBm
Noise:	-95 dBm
Associated clients:	2

ASSOCIATED CLIENT MAC ADDRESS	VENDOR	HT	VHT	WMM	WDS	WPS
b0:e1:7e:30:16:44	HUAWEI TECHNOLOGIES CO.,LTD	✓	✗	✓	✗	✗
20:f4:78:19:3b:38	Xiaomi Communications Co Ltd	✓	✗	✓	✗	✗

NEIGHBORS

IP ADDRESS	MAC ADDRESS	VENDOR	INTERFACE	STATE
192.168.1.1	34:57:60:cb:7f:48	MitraStar Technology Corp.	br-lan	REACHABLE
192.168.1.42	b4:69:21:d2:a6:d1	Intel Corporate	br-lan	REACHABLE
fe80::b2e1:7eff:fe30:1644	b0:e1:7e:30:16:44	HUAWEI TECHNOLOGIES CO.,LTD	br-fvgwifi	STALE
fe80::22f4:78ff:fe19:3b38	20:f4:78:19:3b:38	Xiaomi Communications Co Ltd	br-fvgwifi	STALE

DHCP LEASES

IP ADDRESS	MAC ADDRESS	VENDOR	CLIENT NAME	CLIENT ID	EXPIRY
10.0.0.240	b8:ae:ed:73:ee:51	Elitegroup Computer Systems Co.,Ltd.	DESKTOP-J7NT068	01:b8:ae:ed:73:ee:51	25 Jul 2020, 1:02 a.m.
10.0.0.207	00:04:f2:5d:0e:f3	Polycom	Polycom_0004f25d0ef3	*	24 Jul 2020, 10:33 p.m.

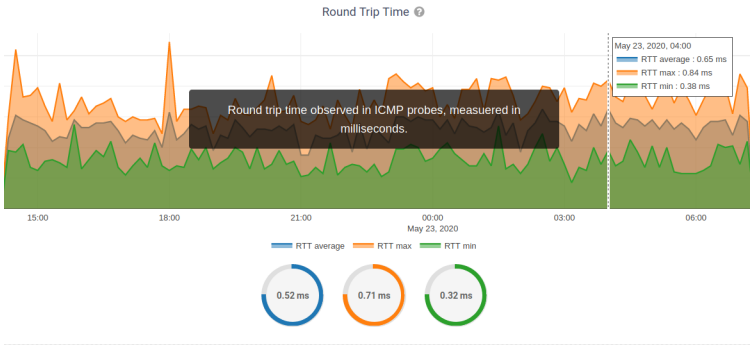
Ping

measurement:	`ping`
types:	`int` (reachable and loss), `float` (rtt)
fields:	`reachable`, `loss`, `rtt_min`, `rtt_max`, `rtt_avg`
configuration:	`ping`
charts:	`uptime` (Ping Success Rate), `packet_loss`, `rtt`

Ping Success Rate:

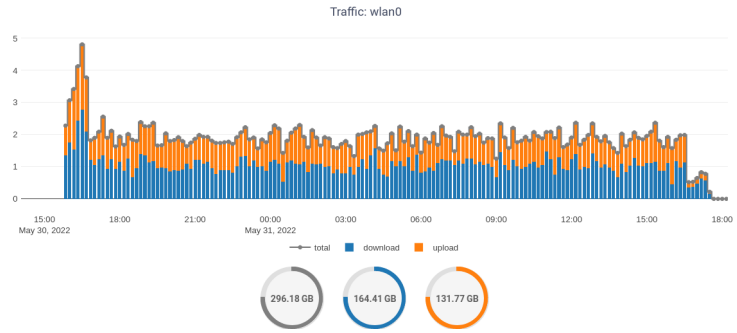
Packet loss:

Round Trip Time:



Traffic

measurement:	`traffic`
type:	`int`
fields:	`rx_bytes`, `tx_bytes`
tags:	`{ "organization_id": "<organization-id-of-the-related-device>", "ifname": "<interface-name>", # optional "location_id": "<location-id-of-the-related-device-if-present>", "floorplan_id": "<floorplan-id-of-the-related-device-if-present>", }`
configuration:	`traffic`
charts:	`traffic`

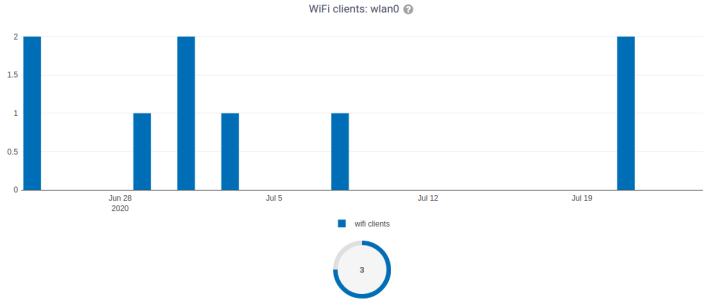


WiFi Clients

measurement:	`wifi_clients`
type:	`int`
fields:	`clients`

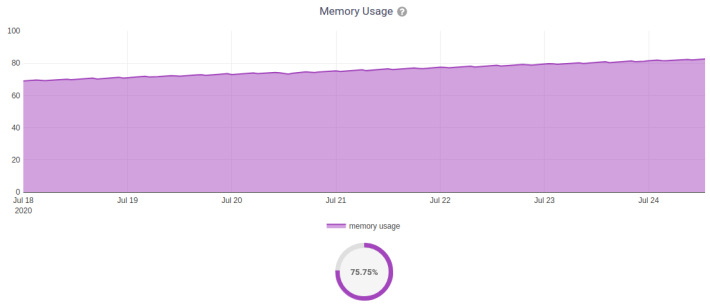
tags:	<pre>{
	    "organization_id": "<organization-id-of-the-related-device>",
	    "ifname": "<interface-name>",
	    # optional
	    "location_id": "<location-id-of-the-related-device-if-present>",
	    "floorplan_id": "<floorplan-id-of-the-related-device-if-present>",
	}</pre>

configuration:	`clients`
charts:	`wifi_clients`



Memory Usage

measurement:	`<memory>`
type:	`float`
fields:	`percent_used`, `free_memory`, `total_memory`, `buffered_memory`, `shared_memory`, `cached_memory`, `available_memory`
configuration:	`memory`
charts:	`memory`

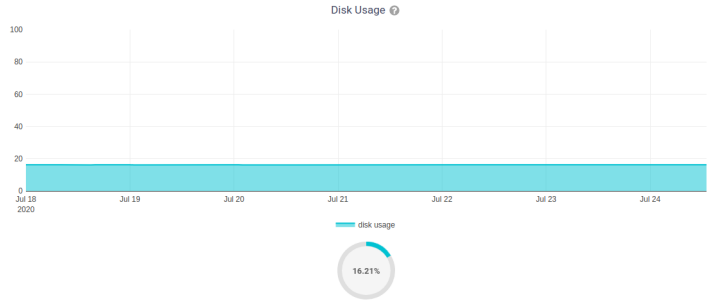


CPU Load

measurement:	`load`
type:	`float`
fields:	`cpu_usage`, `load_1`, `load_5`, `load_15`
configuration:	`load`
charts:	`load`

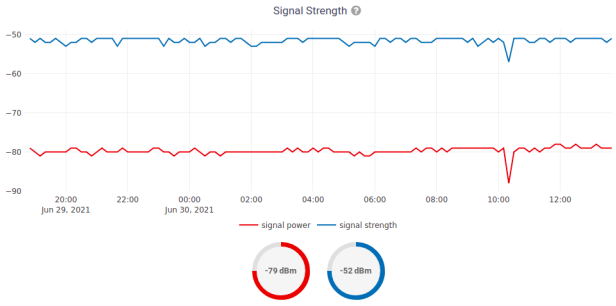
Disk Usage

measurement:	`disk`
type:	`float`
fields:	`used_disk`
configuration:	`disk`
charts:	`disk`



Mobile Signal Strength

measurement:	`signal_strength`
type:	`float`
fields:	`signal_strength`, `signal_power`
configuration:	`signal_strength`
charts:	`signal_strength`

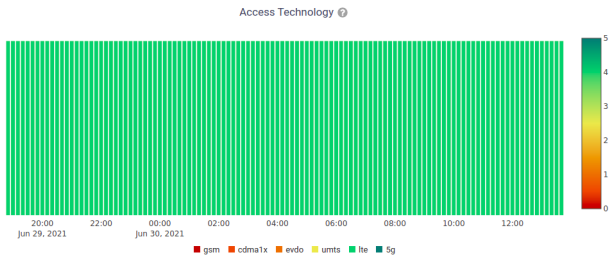


Mobile Signal Quality

measurement:	`signal_quality`
type:	`float`
fields:	`signal_quality`, `signal_quality`
configuration:	`signal_quality`
charts:	`signal_quality`

Mobile Access Technology in Use

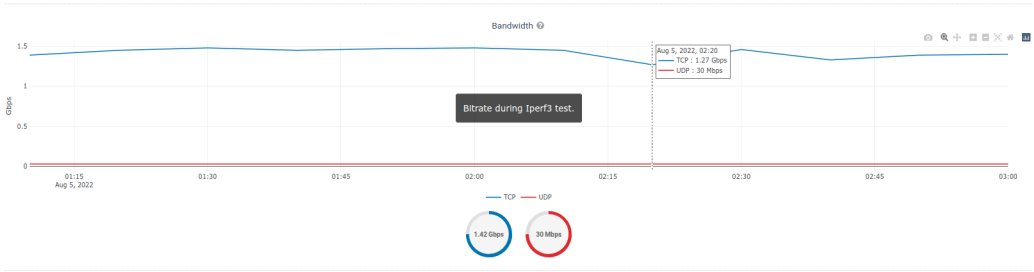
measurement:	`access_tech`
type:	`int`
fields:	`access_tech`
configuration:	`access_tech`
charts:	`access_tech`



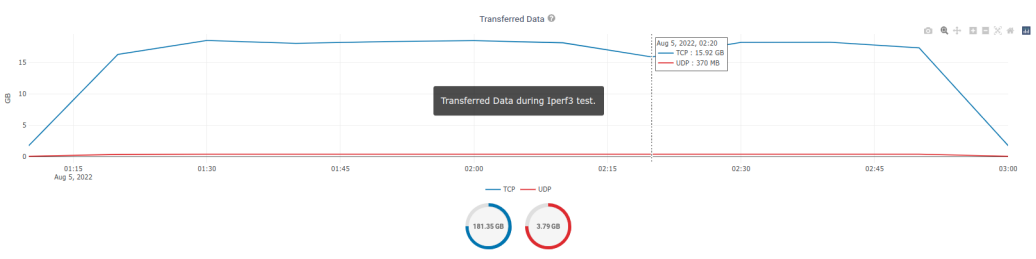
Iperf3

measurement:	`iperf3`
types:	`int` (iperf3_result, sent_bytes_tcp, received_bytes_tcp, retransmits, sent_bytes_udp, total_packets, lost_packets), `float` (sent_bps_tcp, received_bps_tcp, sent_bps_udp, jitter, lost_percent)
fields:	`iperf3_result`, `sent_bps_tcp`, `received_bps_tcp`, `sent_bytes_tcp`, `received_bytes_tcp`, `retransmits`, `sent_bps_udp`, `sent_bytes_udp`, `jitter`, `total_packets`, `lost_packets`, `lost_percent`
configuration:	`iperf3`
charts:	`bandwidth`, `transfer`, `retransmits`, `jitter`, `datagram`, `datagram_loss`

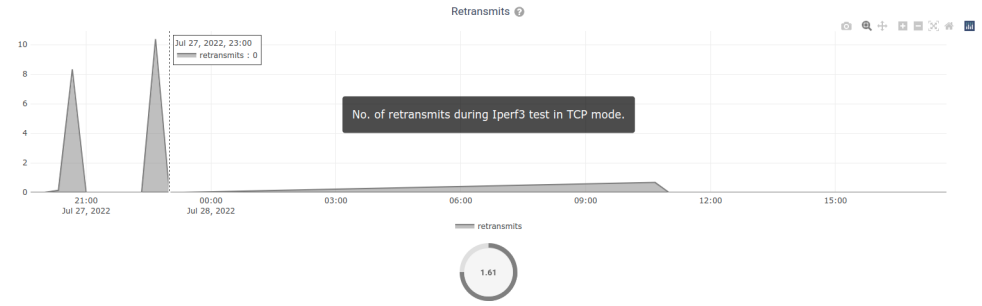
Bandwidth:



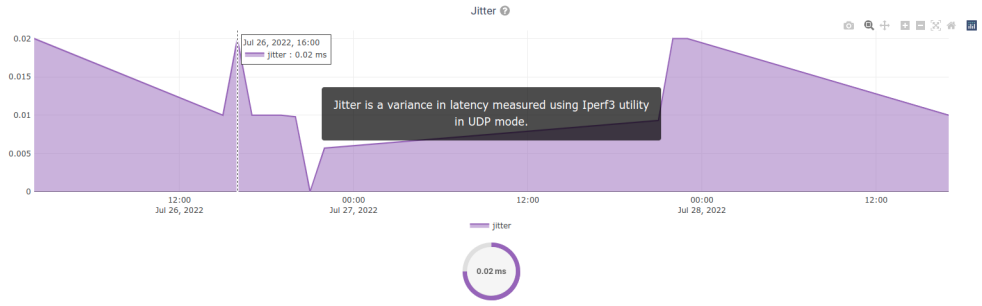
Transferred Data:



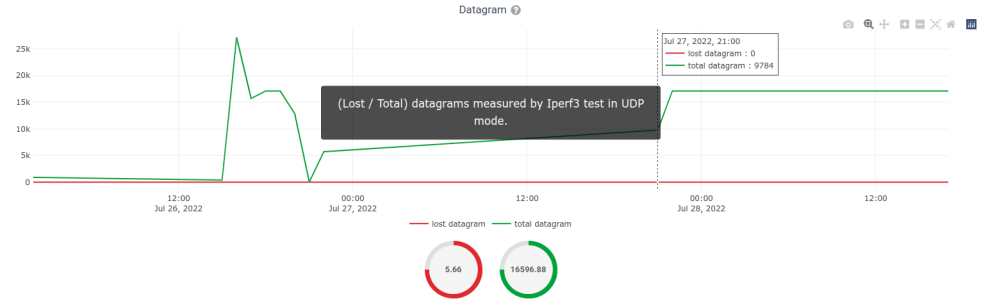
Retransmits:



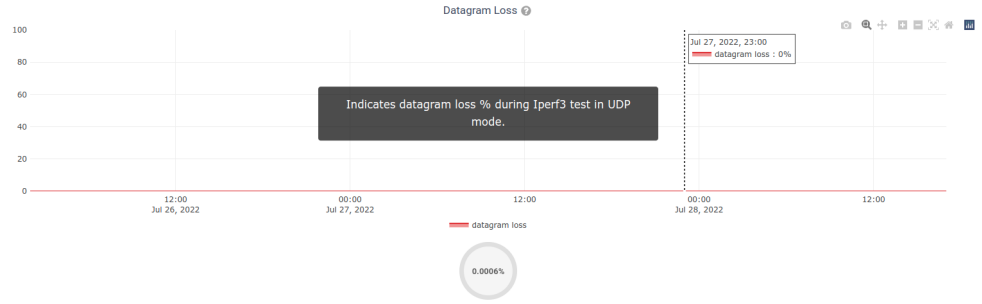
Jitter:



Datagram:



Datagram loss:



For more info on how to configure and use Iperf3, please refer to Configuring Iperf3 Check.

Note

Iperf3 charts uses `connect_points=True` in default chart configuration that joins it's individual chart data points.

Passive vs Active Metric Collection

The the different device metric collected by OpenWISP Monitoring can be divided in two categories:

1. **metrics collected actively by OpenWISP**: these metrics are collected by the celery workers running on the OpenWISP server, which continuously sends network requests to the devices and store the results;
2. **metrics collected passively by OpenWISP**: these metrics are sent by the OpenWrt Monitoring Agent installed on the network devices and are collected by OpenWISP via its REST API.

The Checks section of the documentation lists the currently implemented **active checks**.

Checks

Ping

This check returns information on Ping Success Rate and RTT (Round trip time). It creates charts like Ping Success Rate, Packet Loss and RTT. These metrics are collected using the `fping` Linux program. You may choose to disable auto creation of this check by setting OPENWISP_MONITORING_AUTO_PING to `False`.

You can change the default values used for ping checks using OPENWISP_MONITORING_PING_CHECK_CONFIG setting.

Configuration Applied

This check ensures that the openwisp-config agent is running and applying configuration changes in a timely manner. You may choose to disable auto creation of this check by using the setting OPENWISP_MONITORING_AUTO_DEVICE_CONFIG_CHECK.

This check runs periodically, but it is also triggered whenever the configuration status of a device changes, this ensures the check reacts quickly to events happening in the network and informs the user promptly if there's anything that is not working as intended.

Iperf3

This check provides network performance measurements such as maximum achievable bandwidth, jitter, datagram loss etc of the device using iperf3 utility.

This check is **disabled by default**. You can enable auto creation of this check by setting the OPENWISP_MONITORING_AUTO_IPERF3 to `True`.

You can also add the iperf3 check directly from the device page.

It also supports tuning of various parameters. You can change the parameters used for iperf3 checks (e.g. timing, port, username, password, `rsa_publc_key`, etc.) using the OPENWISP_MONITORING_IPERF3_CHECK_CONFIG setting.

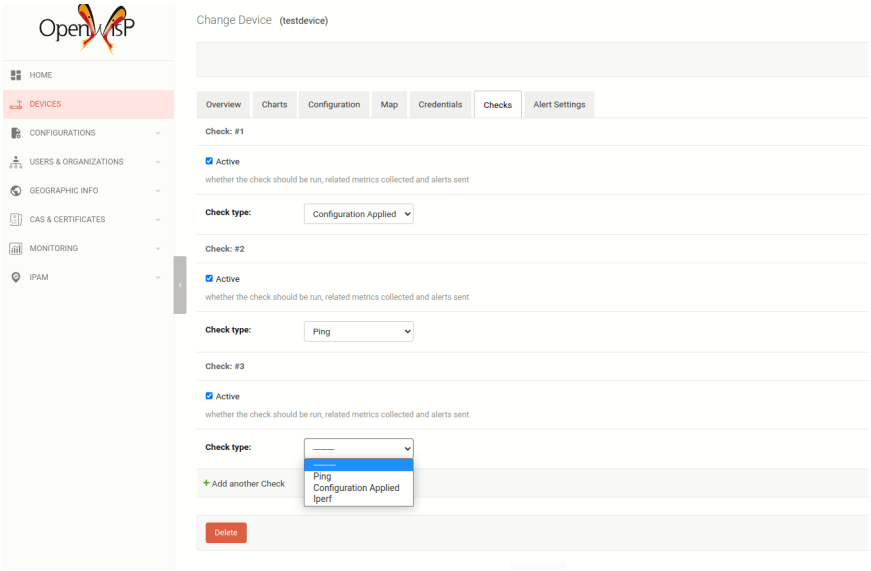
Note

When setting OPENWISP_MONITORING_AUTO_IPERF3 to `True`, you may need to update the metric configuration to enable alerts for the iperf3 check.

Managing Device Checks & Alert Settings

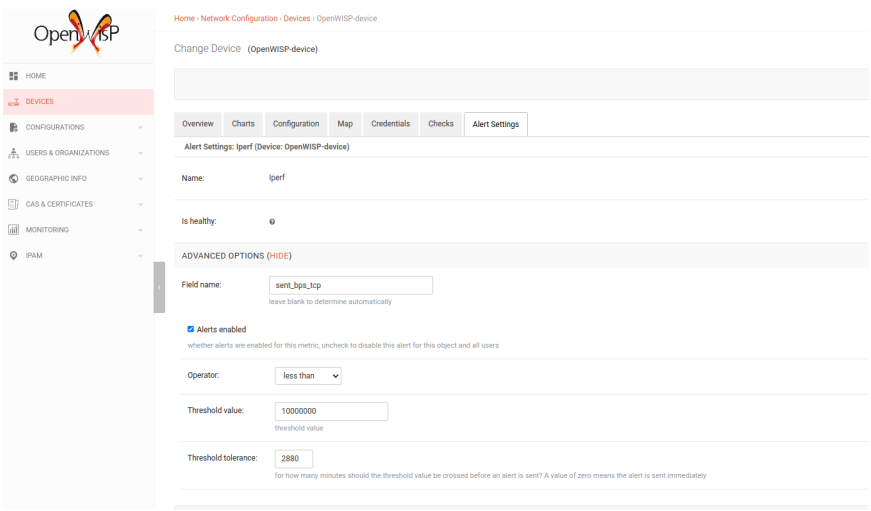
We can add checks and define alert settings directly from the **device page**.

To add a check, you just need to select an available **check type** as shown below:



The following example shows how to use the OPENWISP_MONITORING_METRICS setting to reconfigure the system for iperf3 check to send an alert if the measured **TCP bandwidth** has been less than **10Mbit/s** for more than **2 days**.

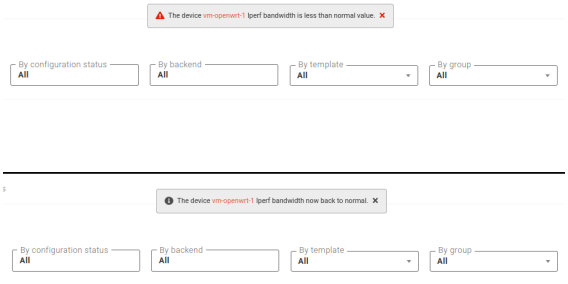
1. By default, Iperf3 checks come with default alert settings, but it is easy to customize alert settings through the device page as shown below:



2. Now, add the following notification configuration to send an alert for **TCP bandwidth**:

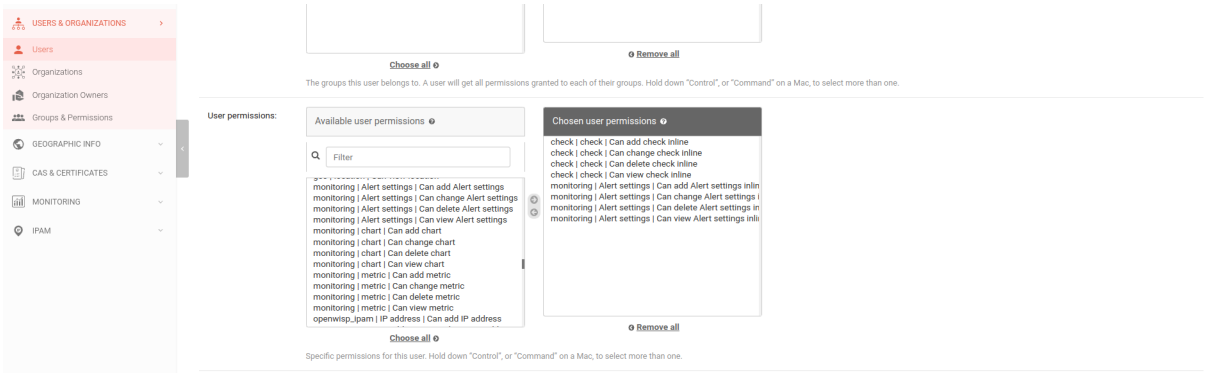
```
# Main project settings.py
from django.utils.translation import gettext_lazy as _

OPENWISP_MONITORING_METRICS = {
    "iperf3": {
        "notification": {
            "problem": {
                "verbose_name": "Iperf3 PROBLEM",
                "verb": _("Iperf3 bandwidth is less than normal value"),
                "level": "warning",
                "email_subject": _(
                    "[{site.name}] PROBLEM: {notification.target} {notification.verb}"
```

```
                ),
                "message": _(
                    "The device [{notification.target}]({notification.target_link}) "
                    "{notification.verb}."
                ),
            },
            "recovery": {
                "verbose_name": "Iperf3 RECOVERY",
                "verb": _("Iperf3 bandwidth now back to normal"),
                "level": "info",
                "email_subject": _(
                    "[{site.name}] RECOVERY: {notification.target} {notification.verb}"
                ),
                "message": _(
                    "The device [{notification.target}]({notification.target_link}) "
                    "{notification.verb}."
                ),
            },
        },
    },
}
```



Note

To access the features described above, the user must have permissions for `Check` and `AlertSetting` *inlines*, these permissions are included by default in the "Administrator" and "Operator" groups and are shown in the screenshot below.



Configuring Iperf3 Check

1. Make Sure Iperf3 is Installed on the Device

Register your device to OpenWISP and make sure the iperf3 openwrt package is installed on the device, e.g.:

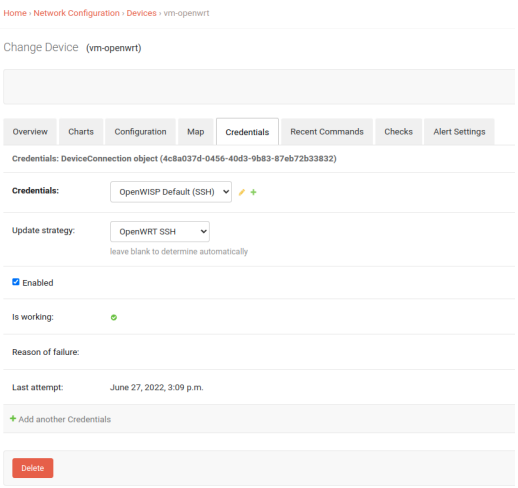
```
opkg install iperf3  # if using without authentication
opkg install iperf3-ssl  # if using with authentication (read below for more info)
```

2. Ensure SSH Access from OpenWISP is Enabled on your Devices

Follow the steps in "Configuring Push Operations" section of the documentation to allow SSH access to you device from OpenWISP.

Important

Make sure device connection is enabled & working with right update strategy i.e. `OpenWrt SSH`.



3. Set Up and Configure Iperf3 Server Settings

Note

If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

After having deployed your Iperf3 servers, you need to configure the iperf3 settings on the Django side of OpenWISP, see the test project settings for reference.

The host can be specified by hostname, IPv4 literal, or IPv6 literal. Example:

```
OPENWISP_MONITORING_IPERF3_CHECK_CONFIG = {
    # 'org_pk' : {'host' : [], 'client_options' : {}}
    "a9734710-db30-46b0-a2fc-01f01046fe4f": {
        # Some public iperf3 servers
        # https://iperf.fr/iperf-servers.php#public-servers
        "host": ["iperf3.openwisp.io", "2001:db8::1", "192.168.5.2"],
        "client_options": {
            "port": 5209,
            "udp": {"bitrate": "30M"},
            "tcp": {"bitrate": "0"},
        },
    },
    # another org
    "b9734710-db30-46b0-a2fc-01f01046fe4f": {
        # available iperf3 servers
        "host": ["iperf3.openwisp2.io", "192.168.5.3"],
        "client_options": {
            "port": 5207,
            "udp": {"bitrate": "50M"},
            "tcp": {"bitrate": "20M"},
        },
    },
}
```

Note

If an organization has more than one iperf3 server configured, then it enables the iperf3 checks to run concurrently on different devices. If all of the available servers are busy, then it will add the check back in the queue.

The celery-beat configuration for the iperf3 check needs to be added too:

```
from celery.schedules import crontab

# Celery TIME_ZONE should be equal to Django TIME_ZONE
# In order to schedule run_iperf3_checks on the correct time intervals
CELERY_TIMEZONE = TIME_ZONE
CELERY_BEAT_SCHEDULE.update(
    {
        # Other celery beat configurations
        # Celery beat configuration for iperf3 check
        "run_iperf3_checks": {
            "task": "openwisp_monitoring.check.tasks.run_checks",
            # https://docs.celeryq.dev/en/latest/userguide/periodic-tasks.html#crontab-sched
            # Executes check every 5 mins from 00:00 AM to 6:00 AM (night)
            "schedule": crontab(minute="*/5", hour="0-6"),
            # Iperf3 check path
            "args": (["openwisp_monitoring.check.classes.Iperf3"],),
            "relative": True,
        }
    }
)
```

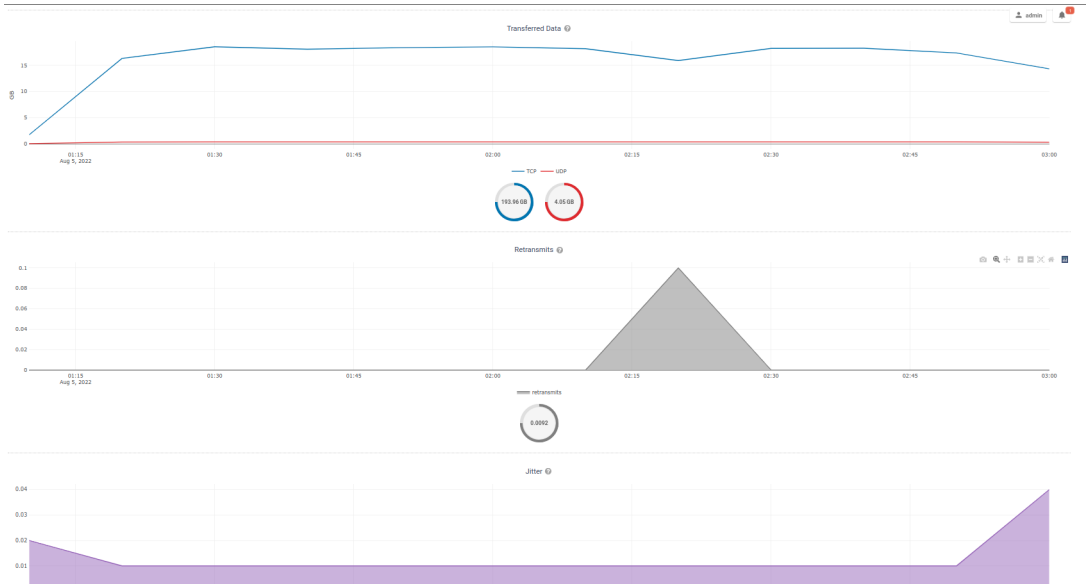
Once the changes are saved, you will need to restart all the processes.

> **Note**
>
> We recommended to configure this check to run in non peak traffic times to not interfere with standard traffic.

4. Run the Check

This should happen automatically if you have celery-beat correctly configured and running in the background. For testing purposes, you can run this check manually using the run_checks command.

After that, you should see the iperf3 network measurements charts.



Iperf3 Check Parameters

Currently, iperf3 check supports the following parameters:

Parameter	Type	Default Value
`host`	`list`	`[]`
`username`	`str`	`''`
`password`	`str`	`''`
`rsa_public_key`	`str`	`''`
`client_options`	`dict`	Refer the Iperf3 Client Options table below for available parameters

Iperf3 Client Options

Parameters	Type	Default Value
`port`	`int`	`5201`
`time`	`int`	`10`
`bytes`	`str`	`''`
`blockcount`	`str`	`''`

window	str	0
parallel	int	1
reverse	bool	False
bidirectional	bool	False
connect_timeout	int	1000
tcp	dict	Refer the Iperf3 Client's TCP Options table below for available parameters
udp	dict	Refer the Iperf3 Client's UDP Options table below for available parameters

Iperf3 Client's TCP Options

Parameters	Type	Default Value
bitrate	str	0
length	str	128K

Iperf3 Client's UDP Options

Parameters	Type	Default Value
bitrate	str	30M
length	str	0

To learn how to use these parameters, please see the iperf3 check configuration example.

Visit the official documentation to learn more about the iperf3 parameters.

Iperf3 Authentication

By default iperf3 check runs without any kind of **authentication**, in this section we will explain how to configure **RSA authentication** between the **client** and the **server** to restrict connections to authenticated clients.

Server Side

1. Generate RSA Key Pair

```
openssl genrsa -des3 -out private.pem 2048
openssl rsa -in private.pem -outform PEM -pubout -out public_key.pem
openssl rsa -in private.pem -out private_key.pem -outform PEM
```

After running the commands mentioned above, the public key will be stored in `public_key.pem` which will be used in **rsa_public_key** parameter in OPENWISP_MONITORING_IPERF3_CHECK_CONFIG and the private key will be contained in the file `private_key.pem` which will be used with **--rsa-private-key-path** command option when starting the iperf3 server.

2. Create User Credentials

```
USER=iperfuser PASSWD=iperfpass
echo -n "{$USER}$PASSWD" | sha256sum | awk '{ print $1 }'
----
ee17a7f98cc87a6424fb52682396b2b6c058e9ab70e946188faa0714905771d7 #This is the hash of "iperf
```

Add the above hash with username in `credentials.csv`

```
# file format: username,sha256
iperfuser,ee17a7f98cc87a6424fb52682396b2b6c058e9ab70e946188faa0714905771d7
```

3. Now Start the Iperf3 Server with Authentication Options

```
iperf3 -s --rsa-private-key-path ./private_key.pem --authorized-users-path ./credentials.csv
```

Client Side (OpenWrt Device)

1. Install iperf3-ssl

Install the iperf3-ssl openwrt package instead of the normal iperf3 openwrt package because the latter comes without support for authentication.

You may also check your installed **iperf3 openwrt package** features:

```
root@vm-openwrt:- iperf3 -v
iperf 3.7 (cJSON 1.5.2)
Linux vm-openwrt 4.14.171 #0 SMP Thu Feb 27 21:05:12 2020 x86_64
Optional features available: CPU affinity setting, IPv6 flow label, TCP congestion algorithm
sendfile / zerocopy, socket pacing, authentication # contains 'authentication'
```

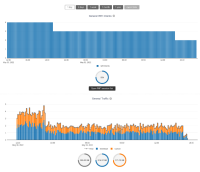
2. Configure Iperf3 Check Authentication Parameters

Now, add the following iperf3 authentication parameters to OPENWISP_MONITORING_IPERF3_CHECK_CONFIG in the Django settings:

```
OPENWISP_MONITORING_IPERF3_CHECK_CONFIG = {
    "a9734710-db30-46b0-a2fc-01f01046fe4f": {
        "host": [
            "iperf1.openwisp.io",
            "iperf2.openwisp.io",
            "192.168.5.2",
        ],
        # All three parameters (username, password, rsa_publc_key)
        # are required for iperf3 authentication
        "username": "iperfuser",
        "password": "iperfpass",
        # Add RSA public key without any headers
        # ie. -----BEGIN PUBLIC KEY-----, -----BEGIN END KEY-----
        "rsa_public_key": (
            """
            MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwuEm+iYrfSWJOupy6X3N
            dxZvUCxvmoL3uoGAs0O0Y32unUQrwcTIxudy38JSuCccD+k2Rf8S4WuZSiTxaoea
            6Du99YQGVZeY67uJ21SWFqWU+w6ONUj3TrNNWoICN7BXGLE2BbSBz9YaXefE3aqw
            GhEjQz364Itwm425vHn2MntSp0weWb4hUCjQUyyooRXPrFUGBOuY+VvAvMyAG4Uk
            msapnWnBSxXt7Tbb++A5XbOMdM2mwNYDEtkD5ksC/x3EVBrI9FvENsH9+u/8J9Mf
            2oPl4MnlCMY86MQypkeUn7eVWfDnseNky7TyC0/IgCXve/iaydCCFdkjyo1MTAA4
            BQIDAQAB
            """
```

```
        ),
        "client_options": {
            "port": 5209,
            "udp": {"bitrate": "20M"},
            "tcp": {"bitrate": "0"},
        },
    }
}
```

Dashboard Monitoring Charts



OpenWISP Monitoring adds two timeseries charts to the admin dashboard:

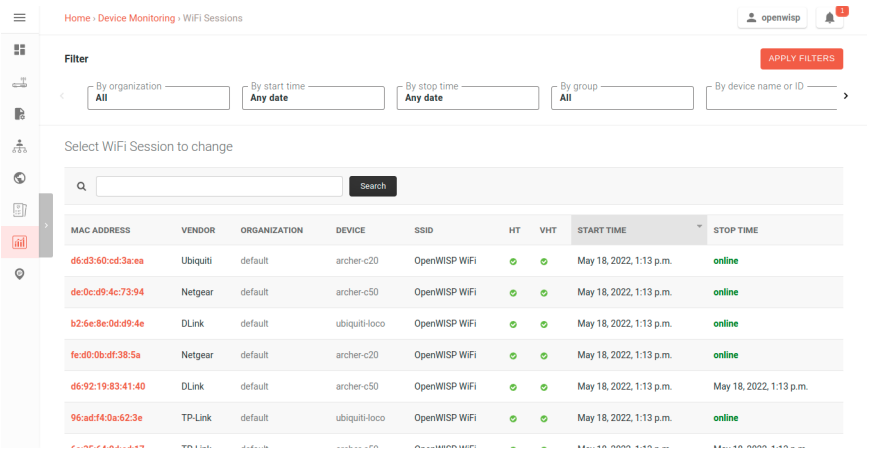
- **General WiFi clients Chart**: Shows the number of connected clients to the WiFi interfaces of devices in the network.
- **General traffic Chart**: Shows the amount of traffic flowing in the network.

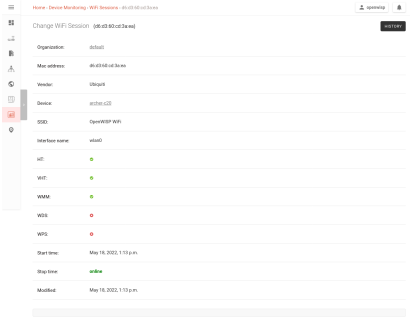
You can configure the interfaces included in the **General traffic chart** using the OPENWISP_MONITORING_DASHBOARD_TRAFFIC_CHART setting.

Monitoring WiFi Sessions

OpenWISP Monitoring maintains a record of WiFi sessions created by clients joined to a radio of managed devices. The WiFi sessions are created asynchronously from the monitoring data received from the device.

You can filter both currently open sessions and past sessions by their *start* or *stop* time or *organization* or *group* of the device clients are connected to or even directly by a *device* name or ID.





You can disable this feature by configuring OPENWISP_MONITORING_WIFI_SESSIONS_ENABLED setting.

You can also view open WiFi sessions of a device directly from the device's change admin under the "WiFi Sessions" tab.

Scheduled Deletion of WiFi Sessions

Important

If you have deployed OpenWISP using ansible-openwisp2 or docker-openwisp, then this feature has been already configured for you. Refer to the documentation of your deployment method to know the default value. This section is only for reference for users who wish to customize OpenWISP, or who have deployed OpenWISP in a different way.

OpenWISP Monitoring provides a celery task to automatically delete WiFi sessions older than a preconfigured number of days.

The celery task takes only one argument, i.e. number of days. You can provide any number of days in *args* key while configuring `CELERY_BEAT_SCHEDULE` setting.

Note

If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

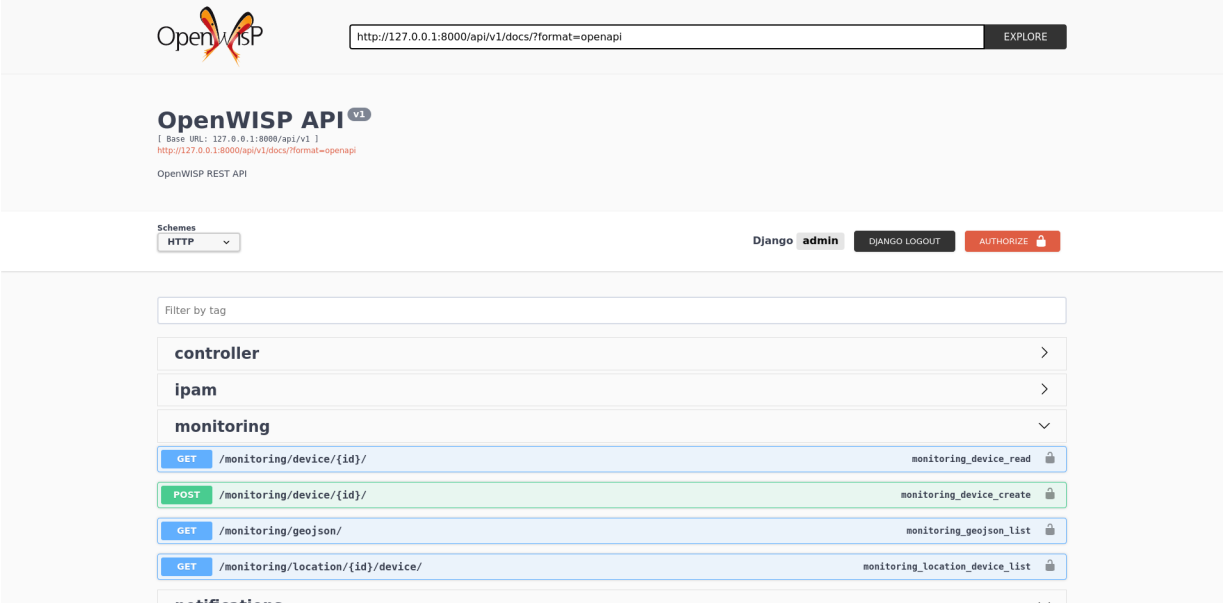
E.g., if you want WiFi Sessions older than 30 days to get deleted automatically, then configure `CELERY_BEAT_SCHEDULE` as follows:

```
from datetime import timedelta

CELERY_BEAT_SCHEDULE.update(
    {
        "delete_wifi_clients_and_sessions": {
            "task": "openwisp_monitoring.monitoring.tasks.delete_wifi_clients_and_sessions",
            "schedule": timedelta(days=1),
            "args": (30,),  # Here we have defined 30 days
        },
    }
)
```

Please refer to "Periodic Tasks" section of Celery's documentation to learn more.

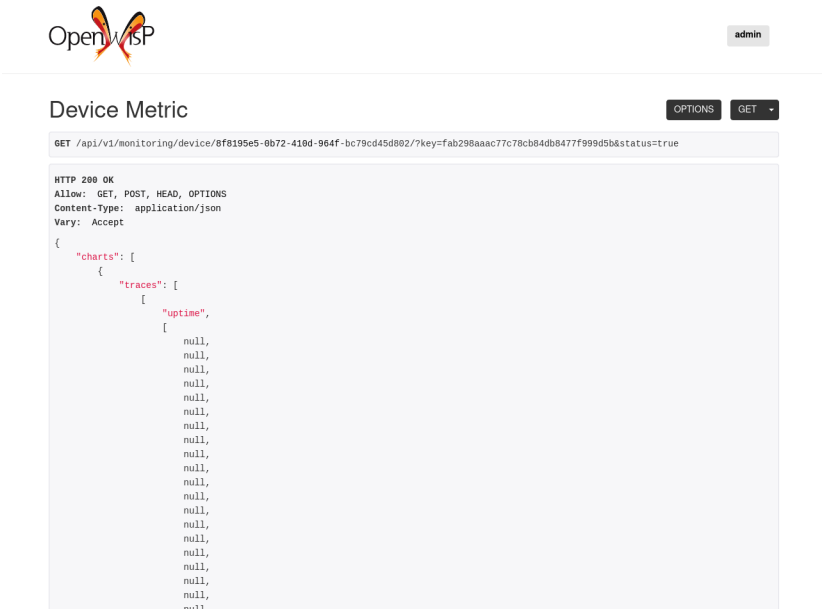
REST API Reference

Live Documentation



A general live API documentation (following the OpenAPI specification) at `/api/v1/docs/`.

Browsable Web Interface



```
"data": {
    "dns_search": [
        "www.tendawifi.com"
    ],
    "type": "DeviceMonitoring",
    "general": {
        "local_time": 1651246948,
        "uptime": 6858,
        "hostname": "5C-A6-E6-88-EA-D6"
    },
    "dhcp_leases": [
        {
            "mac": "24:ee:9a:a4:5d:49",
            "client_id": "01:24:ee:9a:a4:5d:49",
            "client_name": "sd",
            "ip": "192.168.1.221",
            "expiry": 1651283575,
            "vendor": "Intel Corporate"
        }
    ],
    "interfaces": [
        {
            "mac": "5c:a6:e6:88:ea:d7",
            "type": "vlan",
            "up": true,
            "txqueuelen": 1000,
            "name": "eth0.2",
            "multicast": true,
            "speed": "1000F",
            "addresses": [
                {
                    "proto": "dhcp",
                    "address": "192.168.0.110",
                    "family": "ipv4",
                    "mask": 24,
                    "gateway": "192.168.0.1"
                },
                {
                    "family": "ipv6",
                    "mask": 64,
                    "proto": "static",
                    "address": "fe80::5ea6:e6ff:fe88:ead7"
                }
            ],
```


Additionally, opening any of the endpoints listed below directly in the browser will show the browsable API interface of Django-REST-Framework, which makes it even easier to find out the details of each endpoint.

List of Endpoints

Since the detailed explanation is contained in the Live Documentation and in the Browsable Web Interface of each point, here we'll provide just a list of the available endpoints, for further information please open the URL of the endpoint in your browser.

Retrieve General Monitoring Charts

```
GET /api/v1/monitoring/dashboard/
```

This API endpoint is used to show dashboard monitoring charts. It supports multi-tenancy and allows filtering monitoring data by `organization_slug`, `location_id` and `floorplan_id` e.g.:

```
GET /api/v1/monitoring/dashboard/?organization_slug=<org1-slug>,<org2-slug>&location_id=<loc
```

- When retrieving chart data, the `time` parameter allows to specify the time frame, e.g.:
  - `1d`: returns data of the last day
  - `3d`: returns data of the last 3 days
  - `7d`: returns data of the last 7 days
  - `30d`: returns data of the last 30 days
  - `365d`: returns data of the last 365 days
- In alternative to `time` it is possible to request chart data for a custom date range by using the `start` and `end` parameters, e.g.:

```
GET /api/v1/monitoring/dashboard/?start={start_datetime}&end={end_datetime}
```

Note

The `start` and `end` parameters should be in the format `YYYY-MM-DD H:M:S`, otherwise 400 Bad Response will be returned.

Retrieve Device Charts and Device Status Data

```
GET /api/v1/monitoring/device/{pk}/?key={key}&status=true&time={timeframe}
```

The format used for Device Status is inspired by NetJSON DeviceMonitoring.

Note

- If the request is made without `?status=true` the response will contain only charts data and will not include any device status information (current load average, ARP table, DCHP leases, etc.).
- When retrieving chart data, the `time` parameter allows to specify the time frame, e.g.:
    - `1d`: returns data of the last day
    - `3d`: returns data of the last 3 days
    - `7d`: returns data of the last 7 days
    - `30d`: returns data of the last 30 days
    - `365d`: returns data of the last 365 days

- In alternative to `time` it is possible to request chart data for a custom date range by using the `start` and `end` parameters, e.g.:
- The response contains device information, monitoring status (health status), a list of metrics with their respective statuses, chart data and device status information (only if `?status=true`).
- This endpoint can be accessed with session authentication, token authentication, or alternatively with the device key passed as query string parameter as shown below (`?key={key}`); note: this method is meant to be used by the devices.

```
GET /api/v1/monitoring/device/{pk}/?key={key}&status=true&start={start_datetime}&end={end_da
```

Note

The `start` and `end` parameters must be in the format `YYYY-MM-DD H:M:S`, otherwise 400 Bad Response will be returned.

List Device Monitoring Information

```
GET /api/v1/monitoring/device/
```

Note

- The response contains device information and monitoring status (health status), but it does not include the information and health status of the specific metrics, this information can be retrieved in the detail endpoint of each device.
- This endpoint can be accessed with session authentication and token authentication.

Available filters

Data can be filtered by health status (e.g. *critical*, *ok*, *problem*, and *unknown*) to obtain the list of devices in the corresponding status, for example, to retrieve the list of devices which are in critical conditions (e.g.: unreachable), the following will work:

```
GET /api/v1/monitoring/device/?monitoring__status=critical
```

To filter a list of device monitoring data based on their organization, you can use the `organization_id`.

```
GET /api/v1/monitoring/device/?organization={organization_id}
```

To filter a list of device monitoring data based on their organization slug, you can use the `organization_slug`.

```
GET /api/v1/monitoring/device/?organization_slug={organization_slug}
```

Collect Device Metrics and Status

```
POST /api/v1/monitoring/device/{pk}/?key={key}&time={datetime}
```

If data is latest then an additional parameter current can also be passed. For e.g.:

```
POST /api/v1/monitoring/device/{pk}/?key={key}&time={datetime}&current=true
```

The format used for Device Status is inspired by NetJSON DeviceMonitoring.

Note

The device data will be saved in the timeseries database using the date time specified `time`, this should be in the format `%d-%m-%Y_%H:%M:%S.%f`, otherwise 400 Bad Response will be returned.

If the request is made without passing the `time` argument, the server local time will be used.

The `time` parameter was added to support resilient collection and sending of data by the OpenWISP Monitoring Agent, this feature allows sending data collected while the device is offline.

List Nearby Devices

```
GET /api/v1/monitoring/device/{pk}/nearby-devices/
```

Returns list of nearby devices along with respective distance (in metres) and monitoring status.

Available filters

The list of nearby devices provides the following filters:

- `organization` (Organization ID of the device)
- `organization__slug` (Organization slug of the device)
- `monitoring__status` (Monitoring status (`unknown`, `ok`, `problem`, or `critical`))
- `model` (Pipe | separated list of device models)
- `distance__lte` (Distance in metres)

Here's a few examples:

```
GET /api/v1/monitoring/device/{pk}/nearby-devices/?organization={organization_id}
GET /api/v1/monitoring/device/{pk}/nearby-devices/?organization__slug={organization_slug}
GET /api/v1/monitoring/device/{pk}/nearby-devices/?monitoring__status={monitoring_status}
GET /api/v1/monitoring/device/{pk}/nearby-devices/?model={model1,model2}
GET /api/v1/monitoring/device/{pk}/nearby-devices/?distance__lte={distance}
```

List WiFi Session

```
GET /api/v1/monitoring/wifi-session/
```

Available filters

The list of wifi session provides the following filters:

- `device__organization` (Organization ID of the device)
- `device` (Device ID)
- `device__group` (Device group ID)
- `start_time` (Start time of the wifi session)
- `stop_time` (Stop time of the wifi session)

Here's a few examples:

```
GET /api/v1/monitoring/wifi-session/?device__organization={organization_id}
GET /api/v1/monitoring/wifi-session/?device={device_id}
GET /api/v1/monitoring/wifi-session/?device__group={group_id}
GET /api/v1/monitoring/wifi-session/?start_time={stop_time}
GET /api/v1/monitoring/wifi-session/?stop_time={stop_time}
```

> **Note**
>
> Both `start_time` and `stop_time` support greater than or equal to, as well as less than or equal to, filter lookups.

For example:

```
GET /api/v1/monitoring/wifi-session/?start_time__gt={start_time}
GET /api/v1/monitoring/wifi-session/?start_time__gte={start_time}
GET /api/v1/monitoring/wifi-session/?stop_time__lt={stop_time}
GET /api/v1/monitoring/wifi-session/?stop_time__lte={stop_time}
```

Get WiFi Session

```
GET /api/v1/monitoring/wifi-session/{id}/
```

Pagination

WiFi session endpoint support the `page_size` parameter that allows paginating the results in conjunction with the page parameter.

```
GET /api/v1/monitoring/wifi-session/?page_size=10
GET /api/v1/monitoring/wifi-session/?page_size=10&page=1
```

Settings

Note

If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

TIMESERIES_DATABASE

type:	str
default:	see below

```
TIMESERIES_DATABASE = {
    "BACKEND": "openwisp_monitoring.db.backends.influxdb",
    "USER": "openwisp",
    "PASSWORD": "openwisp",
    "NAME": "openwisp2",
    "HOST": "localhost",
    "PORT": "8086",
    "OPTIONS": {
        "udp_writes": False,
        "udp_port": 8089,
    },
}
```

The following table describes all keys available in TIMESERIES_DATABASE setting:

Key	Description
BACKEND	The timeseries database backend to use. You can select one of the backends located in openwisp_monitoring.db.backends
USER	User for logging into the timeseries database
PASSWORD	Password of the timeseries database user
NAME	Name of the timeseries database
HOST	IP address/hostname of machine where the timeseries database is running
PORT	Port for connecting to the timeseries database
OPTIONS	These settings depends on the timeseries backend. Refer the Timeseries Database Options table below for available options

Timeseries Database Options

udp_writes	Whether to use UDP for writing data to the timeseries database
udp_port	Timeseries database port for writing data using UDP

Important

UDP packets can have a maximum size of 64KB. When using UDP for writing timeseries data, if the size of the data exceeds 64KB, TCP mode will be used instead.

Note

If you want to use the `openwisp_monitoring.db.backends.influxdb` backend with UDP writes enabled, then you need to enable two different ports for UDP (each for different retention policy) in your InfluxDB configuration. The UDP configuration section of your InfluxDB should look similar to the following:

```
# For writing data with the "default" retention policy
[[udp]]
enabled = true
bind-address = "127.0.0.1:8089"
database = "openwisp2"

# For writing data with the "short" retention policy
[[udp]]
enabled = true
bind-address = "127.0.0.1:8090"
database = "openwisp2"
retention-policy = 'short'
```

If you are using ansible-openwisp2 for deploying OpenWISP, you can set the `influxdb_udp_mode` ansible variable to `true` in your playbook, this will make the ansible role automatically configure the InfluxDB UDP listeners. You can refer to the ansible-ow-influxdb's (a dependency of ansible-openwisp2) documentation to learn more.

`OPENWISP_MONITORING_DEFAULT_RETENTION_POLICY`

type:	`str`
default:	`26280h0m0s` (3 years)

The default retention policy that applies to the timeseries data.

`OPENWISP_MONITORING_SHORT_RETENTION_POLICY`

type:	`str`
default:	`24h0m0s`

The default retention policy used to store raw device data.

This data is only used to assess the recent status of devices, keeping it for a long time would not add much benefit and would cost a lot more in terms of disk space.

`OPENWISP_MONITORING_AUTO_PING`

type:	`bool`
default:	`True`

Whether ping checks are created automatically for devices.

`OPENWISP_MONITORING_PING_CHECK_CONFIG`

type:	`dict`

default:	`{}`

This setting allows to override the default ping check configuration defined in `openwisp_monitoring.check.classes.ping.DEFAULT_PING_CHECK_CONFIG`.

For example, if you want to change only the **timeout** of `ping` you can use:

```
OPENWISP_MONITORING_PING_CHECK_CONFIG = {
    "timeout": {
        "default": 1000,
    },
}
```

If you are overriding the default value for any parameter beyond the maximum or minimum value defined in `openwisp_monitoring.check.classes.ping.DEFAULT_PING_CHECK_CONFIG`, you will also need to override the `maximum` or `minimum` fields as following:

```
OPENWISP_MONITORING_PING_CHECK_CONFIG = {
    "timeout": {
        "default": 2000,
        "minimum": 1500,
        "maximum": 2500,
    },
}
```

Note

Above `maximum` and `minimum` values are only used for validating custom parameters of a `Check` object.

`OPENWISP_MONITORING_AUTO_DEVICE_CONFIG_CHECK`

type:	`bool`
default:	`True`

This setting allows you to choose whether config_applied checks should be created automatically for newly registered devices. It's enabled by default.

`OPENWISP_MONITORING_CONFIG_CHECK_INTERVAL`

type:	`int`
default:	`5`

This setting allows you to configure the config check interval used by config_applied. By default it is set to 5 minutes.

`OPENWISP_MONITORING_AUTO_IPERF3`

type:	`bool`
default:	`False`

This setting allows you to choose whether iperf3 checks should be created automatically for newly registered devices. It's disabled by default.

OPENWISP_MONITORING_IPERF3_CHECK_CONFIG

type:	dict
default:	{}

This setting allows to override the default iperf3 check configuration defined in `openwisp_monitoring.check.classes.iperf3.DEFAULT_IPERF3_CHECK_CONFIG`.

For example, you can change the values of supported iperf3 check parameters.

```
OPENWISP_MONITORING_IPERF3_CHECK_CONFIG = {
    # 'org_pk' : {'host' : [], 'client_options' : {}}
    "a9734710-db30-46b0-a2fc-01f01046fe4f": {
        # Some public iperf3 servers
        # https://iperf.fr/iperf-servers.php#public-servers
        "host": ["iperf3.openwisp.io", "2001:db8::1", "192.168.5.2"],
        "client_options": {
            "port": 6209,
            # Number of parallel client streams to run
            # note that iperf3 is single threaded
            # so if you are CPU bound this will not
            # yield higher throughput
            "parallel": 5,
            # Set the connect_timeout (in milliseconds) for establishing
            # the initial control connection to the server, the lower the value
            # the faster the down iperf3 server will be detected (ex. 1000 ms (1 sec))
            "connect_timeout": 1000,
            # Window size / socket buffer size
            "window": "300K",
            # Only one reverse condition can be chosen,
            # reverse or bidirectional
            "reverse": True,
            # Only one test end condition can be chosen,
            # time, bytes or blockcount
            "blockcount": "1K",
            "udp": {"bitrate": "50M", "length": "1460K"},
            "tcp": {"bitrate": "20M", "length": "256K"},
        },
    }
}
```

OPENWISP_MONITORING_IPERF3_CHECK_DELETE_RSA_KEY

type:	bool
default:	True

This setting allows you to set whether iperf3 check RSA public key will be deleted after successful completion of the check or not.

OPENWISP_MONITORING_IPERF3_CHECK_LOCK_EXPIRE

type:	int
default:	600

This setting allows you to set a cache lock expiration time for the iperf3 check when running on multiple servers. Make sure it is always greater than the total iperf3 check time, i.e. greater than the TCP + UDP test time. By default, it is set to **600 seconds (10 mins)**.

OPENWISP_MONITORING_AUTO_CHARTS

type:	`list`
default:	`('traffic', 'wifi_clients', 'uptime', 'packet_loss', 'rtt')`

Automatically created charts.

OPENWISP_MONITORING_CRITICAL_DEVICE_METRICS

type:	`list` of `dict` objects
default:	`[{'key': 'ping', 'field_name': 'reachable'}]`

Device metrics that are considered critical:

when a value crosses the boundary defined in the "threshold value" field of the alert settings related to one of these metric types, the health status of the device related to the metric moves into `CRITICAL`.

By default, if devices are not reachable by pings they are flagged as `CRITICAL`.

OPENWISP_MONITORING_HEALTH_STATUS_LABELS

type:	`dict`
default:	`{'unknown': 'unknown', 'ok': 'ok', 'problem': 'problem', 'critical': 'critical', 'deactivated': 'deactivated'}`

This setting allows to change the health status labels, for example, if we want to use `online` instead of `ok` and `offline` instead of `critical`, you can use the following configuration:

```
OPENWISP_MONITORING_HEALTH_STATUS_LABELS = {
    "ok": "online",
    "problem": "problem",
    "critical": "offline",
}
```

OPENWISP_MONITORING_WIFI_SESSIONS_ENABLED

type:	`bool`
default:	`True`

Setting this to `False` will disable Monitoring WiFi Sessions feature.

OPENWISP_MONITORING_MANAGEMENT_IP_ONLY

type:	`bool`
default:	`True`

By default, only the management IP will be used to perform active checks to the devices.

If the devices are connecting to your OpenWISP instance using a shared layer2 network, hence the OpenWSP server can reach the devices using the `last_ip` field, you can set this to `False`.

Note

If this setting is not configured, it will fallback to the value of OPENWISP_CONTROLLER_MANAGEMENT_IP_ONLY setting. If `OPENWISP_CONTROLLER_MANAGEMENT_IP_ONLY` also not configured, then it will fallback to `True`.

`OPENWISP_MONITORING_DEVICE_RECOVERY_DETECTION`

type:	`bool`
default:	`True`

When device recovery detection is enabled, recoveries are discovered as soon as a device contacts the openwisp system again (e.g.: to get the configuration checksum or to send monitoring metrics).

This feature is enabled by default.

If you use OpenVPN as the management VPN, you may want to check out a similar integration built in **openwisp-network-topology**: when the status of an OpenVPN link changes (detected by monitoring the status information of OpenVPN), the network topology module will trigger the monitoring checks. For more information see: Network Topology Device Integration.

`OPENWISP_MONITORING_MAC_VENDOR_DETECTION`

type:	`bool`
default:	`True`

Indicates whether mac addresses will be complemented with hardware vendor information by performing lookups on the OUI (Organization Unique Identifier) table.

This feature is enabled by default.

`OPENWISP_MONITORING_WRITE_RETRY_OPTIONS`

type:	`dict`
default:	see below

```
# default value of OPENWISP_MONITORING_RETRY_OPTIONS:

dict(
    max_retries=None,
    retry_backoff=True,
    retry_backoff_max=600,
    retry_jitter=True,
)
```

Retry settings for recoverable failures during metric writes.

By default if a metric write fails (e.g.: due to excessive load on timeseries database at that moment) then the operation will be retried indefinitely with an exponential random backoff and a maximum delay of 10 minutes.

This feature makes the monitoring system resilient to temporary outages and helps to prevent data loss.

For more information regarding these settings, consult the celery documentation regarding automatic retries for known errors.

Note

The retry mechanism does not work when using UDP for writing data to the timeseries database. It is due to the nature of UDP protocol which does not acknowledge receipt of data packets.

OPENWISP_MONITORING_TIMESERIES_RETRY_OPTIONS

type:	dict
default:	see below

```
# default value of OPENWISP_MONITORING_RETRY_OPTIONS:

dict(max_retries=6, delay=2)
```

On busy systems, communication with the timeseries DB can occasionally fail. The timeseries DB backend will retry on any exception according to these settings. The delay kicks in only after the third consecutive attempt.

This setting shall not be confused with OPENWISP_MONITORING_WRITE_RETRY_OPTIONS, which is used to configure the infinite retrying of the celery task which writes metric data to the timeseries DB, while OPENWISP_MONITORING_TIMESERIES_RETRY_OPTIONS deals with any other read/write operation on the timeseries DB which may fail.

However these retries are not handled by celery but are simple python loops, which will eventually give up if a problem persists.

OPENWISP_MONITORING_TIMESERIES_RETRY_DELAY

type:	int
default:	2

This settings allow you to configure the retry delay time (in seconds) after 3 failed attempt in timeseries database.

This retry setting is used in retry mechanism to make the requests to the timeseries database resilient.

This setting is independent of celery retry settings.

OPENWISP_MONITORING_DASHBOARD_MAP

type:	bool
default:	True

Whether the geographic map in the dashboard is enabled or not. This feature provides a geographic map which shows the locations which have devices installed in and provides a visual representation of the monitoring status of the devices, this allows to get an overview of the network at glance.

This feature is enabled by default and depends on the setting OPENWISP_ADMIN_DASHBOARD_ENABLED from openwisp-utils being set to True (which is the default).

You can turn this off if you do not use the geographic features of OpenWISP.

OPENWISP_MONITORING_DASHBOARD_TRAFFIC_CHART

type:	dict

default:	`{'__all__': ['wan', 'eth1', 'eth0.2']}`

This settings allows to configure the interfaces which should be included in the **General Traffic** chart in the admin dashboard.

This setting should be defined in the following format:

E.g., if you want the **General Traffic** chart to show data from two interfaces for an organization, you need to configure this setting as follows:

Note

The value of `__all__` key is used if an organization does not have list of interfaces defined in `OPENWISP_MONITORING_DASHBOARD_TRAFFIC_CHART`.

Note

If a user can manage more than one organization (e.g. superusers), then the **General Traffic** chart will always show data from interfaces of `__all__` configuration.

`OPENWISP_MONITORING_METRICS`

type:	`dict`
default:	`{}`

This setting allows to define additional metric configuration or to override the default metric configuration defined in `openwisp_monitoring.monitoring.configuration.DEFAULT_METRICS`.

For example, if you want to change only the **field_name** of `clients` metric to `wifi_clients` you can use:

```
from django.utils.translation import gettext_lazy as _

OPENWISP_MONITORING_METRICS = {
    "clients": {
        "label": _("WiFi clients"),
        "field_name": "wifi_clients",
    },
}
```

For example, if you want to change only the default alert settings of `memory` metric you can use:

```
OPENWISP_MONITORING_METRICS = {
    "memory": {"alert_settings": {"threshold": 75, "tolerance": 10}},
}
```

For example, if you want to change only the notification of `config_applied` metric you can use:

```
from django.utils.translation import gettext_lazy as _

OPENWISP_MONITORING_METRICS = {
    "config_applied": {
        "notification": {
            "problem": {
                "verbose_name": "Configuration PROBLEM",
                "verb": _("has not been applied"),
                "email_subject": _(
```

```
                    "[{site.name}] PROBLEM: {notification.target} configuration "
                    "status issue"
                ),
                "message": _(
                    "The configuration for device [{notification.target}]"
                    "({notification.target_link}) {notification.verb} in a timely manner."
                ),
            },
            "recovery": {
                "verbose_name": "Configuration RECOVERY",
                "verb": _("configuration has been applied again"),
                "email_subject": _(
                    "[{site.name}] RECOVERY: {notification.target} {notification.verb} "
                    "successfully"
                ),
                "message": _(
                    "The device [{notification.target}]({notification.target_link}) "
                    "{notification.verb} successfully."
                ),
            },
        },
    },
}
```

Or if you want to define a new metric configuration, which you can then call in your custom code (e.g.: a custom check class), you can do so as follows:

```
from django.utils.translation import gettext_lazy as _

OPENWISP_MONITORING_METRICS = {
    "top_fields_mean": {
        "name": "Top Fields Mean",
        "key": "{key}",
        "field_name": "{field_name}",
        "label": "_(Top fields mean)",
        "related_fields": ["field1", "field2", "field3"],
    },
}
```

OPENWISP_MONITORING_CHARTS

type:	dict
default:	{}

This setting allows to define additional charts or to override the default chart configuration defined in `openwisp_monitoring.monitoring.configuration.DEFAULT_CHARTS`.

In the following example, we modify the description of the traffic chart:

```
OPENWISP_MONITORING_CHARTS = {
    "traffic": {
        "description": (
            "Network traffic, download and upload, measured on "
            'the interface "{metric.key}", custom message here.'
        ),
    }
}
```

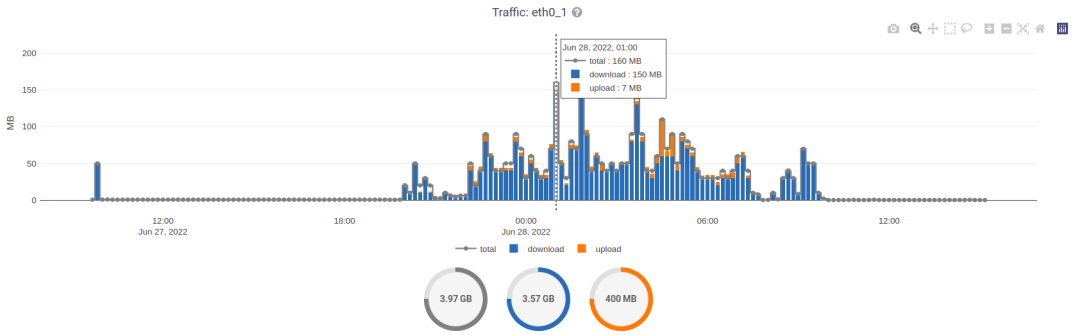
Or if you want to define a new chart configuration, which you can then call in your custom code (e.g.: a custom check class), you can do so as follows:

```
from django.utils.translation import gettext_lazy as _

OPENWISP_MONITORING_CHARTS = {
    "ram": {
        "type": "line",
        "title": "RAM usage",
        "description": "RAM usage",
        "unit": "bytes",
        "order": 100,
        "query": {
            "influxdb": (
                "SELECT MEAN(total) AS total, MEAN(free) AS free, "
                "MEAN(buffered) AS buffered FROM {key} WHERE time >= '{time}' AND "
                "content_type = '{content_type}' AND object_id = '{object_id}' "
                "GROUP BY time(1d)"
            )
        },
    }
}
```

In case you just want to change the colors used in a chart here's how to do it:

```
OPENWISP_MONITORING_CHARTS = {
    "traffic": {"colors": ["#000000", "#cccccc", "#111111"]}
}
```

Adaptive Size Charts



When configuring charts, it is possible to flag their unit as `adaptive_prefix`, this allows to make the charts more readable because the units are shown in either *KB*, *MB*, *GB* and *TB* depending on the size of each point, the summary values and Y axis are also resized.

Example taken from the default configuration of the traffic chart:

```
OPENWISP_MONITORING_CHARTS = {
    "traffic": {
        # other configurations for this chart
        # traffic measured in 'B' (bytes)
        # unit B, KB, MB, GB, TB
        "unit": "adaptive_prefix+B",
    },
    "bandwidth": {
        # other configurations for this chart
        # adaptive unit for bandwidth related charts
        # bandwidth measured in 'bps'(bits/sec)
        # unit bps, Kbps, Mbps, Gbps, Tbps
        "unit": "adaptive_prefix+bps",
    },
}
```

`OPENWISP_MONITORING_DEFAULT_CHART_TIME`

type:	`str`
default:	`7d`
possible values	`1d`, `3d`, `7d`, `30d` or `365d`

Allows to set the default time period of the time series charts.

`OPENWISP_MONITORING_AUTO_CLEAR_MANAGEMENT_IP`

type:	`bool`
default:	`True`

This setting allows you to automatically clear management_ip of a device when it goes offline. It is enabled by default.

`OPENWISP_MONITORING_API_URLCONF`

type:	`string`
default:	`None`

Changes the `urlconf` option of django URLs to point the monitoring API URLs to another installed module, example, `myapp.urls`. (Useful when you have a separate API instance.)

`OPENWISP_MONITORING_API_BASEURL`

type:	`string`
default:	`None`

If you have a separate instance of the OpenWISP Monitoring API on a different domain, you can use this option to change the base of the URL, this will enable you to point all the API URLs to your API server's domain, example: `https://api.myservice.com`.

`OPENWISP_MONITORING_CACHE_TIMEOUT`

type:	`int`
default:	`86400` (24 hours in seconds)

This setting allows to configure timeout (in seconds) for monitoring data cache.

Management Commands

`run_checks`

This command will execute all the available checks for all the devices. By default checks are run periodically by *celery beat*.

Example usage:

```
cd tests/
./manage.py run_checks
```

```
migrate_timeseries
```

This command triggers asynchronous migration of the time-series database.

Example usage:

```
cd tests/
./manage.py migrate_timeseries
```

Developer Docs

Note

This page is for developers who want to customize or extend OpenWISP Monitoring, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Monitoring User Docs

Developer Installation Instructions

Note

This page is for developers who want to customize or extend OpenWISP Monitoring, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Monitoring User Docs

Dependencies

- Python >= 3.8
- InfluxDB 1.8
- fping
- OpenSSL

Installing for Development

Install the system dependencies:

Install system packages:

```
sudo apt update
sudo apt install -y sqlite3 libsqlite3-dev openssl libssl-dev
sudo apt install -y gdal-bin libproj-dev libgeos-dev libspatialite-dev libsqlite3-mod-spatia
sudo apt install -y fping
sudo apt install -y chromium
```

Fork and clone the forked repository:

```
git clone git://github.com/<your_fork>/openwisp-monitoring
```

Navigate into the cloned repository:

```
cd openwisp-monitoring/
```

Start Redis and InfluxDB using Docker:

```
docker-compose up -d redis influxdb
```

Setup and activate a virtual-environment. (we'll be using virtualenv)

```
python -m virtualenv env
source env/bin/activate
```

Make sure that you are using pip version 20.2.4 before moving to the next step:

```
pip install -U pip wheel setuptools
```

Install development dependencies:

```
pip install -e .
pip install -r requirements-test.txt
npm install -g jshint stylelint
```

Install WebDriver for Chromium for your browser version from https://chromedriver.chromium.org/home and extract `chromedriver` to one of directories from your `$PATH` (example: `~/.local/bin/`).

Create database:

```
cd tests/
./manage.py migrate
./manage.py createsuperuser
```

Run celery and celery-beat with the following commands (separate terminal windows are needed):

```
cd tests/
celery -A openwisp2 worker -l info
celery -A openwisp2 beat -l info
```

Launch development server:

```
./manage.py runserver 0.0.0.0:8000
```

You can access the admin interface at `http://127.0.0.1:8000/admin/`.

Run tests with:

```
./runtests.py  # using --parallel is not supported in this module
```

Run quality assurance tests with:

```
./run-qa-checks
```

Alternative Sources

PyPI

To install the latest Pypi:

```
pip install openwisp-monitoring
```

Github

To install the latest development version tarball via HTTPs:

```
pip install https://github.com/openwisp/openwisp-monitoring/tarball/master
```

Alternatively you can use the git protocol:

```
pip install -e git+git://github.com/openwisp/openwisp-monitoring#egg=openwisp_monitoring
```

Install and Run on Docker

Warning

This Docker image is for development purposes only.

For the official OpenWISP Docker images, see: Docker OpenWISP.

Build from the Dockerfile:

```
docker-compose build
```

Run the docker container:

```
docker-compose up
```

Code Utilities

Note

This page is for developers who want to customize or extend OpenWISP Monitoring, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Monitoring User Docs

Registering / Unregistering Metric Configuration

OpenWISP Monitoring provides registering and unregistering metric configuration through utility functions `openwisp_monitoring.monitoring.configuration.register_metric` and `openwisp_monitoring.monitoring.configuration.unregister_metric`. Using these functions you can register or unregister metric configurations from anywhere in your code.

`register_metric`

This function is used to register a new metric configuration from anywhere in your code.

Parameter	Description
metric_name:	A `str` defining name of the metric configuration.
metric_configuration:	A `dict` defining configuration of the metric.

An example usage has been shown below.

```
from django.utils.translation import gettext_lazy as _
from openwisp_monitoring.monitoring.configuration import register_metric

# Define configuration of your metric
metric_config = {
    "label": _("Ping"),
    "name": "Ping",
    "key": "ping",
    "field_name": "reachable",
    "related_fields": ["loss", "rtt_min", "rtt_max", "rtt_avg"],
    "charts": {
        "uptime": {
            "type": "bar",
            "title": _("Ping Success Rate"),
            "description": _(
                "A value of 100% means reachable, 0% means unreachable, values in "
                "between 0% and 100% indicate the average reachability in the "
                "period observed. Obtained with the fping linux program."
            ),
            "summary_labels": [_("Average Ping Success Rate")],
            "unit": "%",
            "order": 200,
            "colorscale": {
                "max": 100,
                "min": 0,
                "label": _("Rate"),
                "scale": [
                    [
                        [0, "#c13000"],
                        [0.1, "cb7222"],
                        [0.5, "#deed0e"],
                        [0.9, "#7db201"],
                        [1, "#498b26"],
                    ],
                ],
                "map": [
                    [100, "#498b26", _("Flawless")],
                    [90, "#7db201", _("Mostly Reachable")],
                    [50, "#deed0e", _("Partly Reachable")],
                    [10, "#cb7222", _("Mostly Unreachable")],
                    [None, "#c13000", _("Unreachable")],
                ],
```

```
                "fixed_value": 100,
            },
            "query": chart_query["uptime"],
        },
        "packet_loss": {
            "type": "bar",
            "title": _("Packet loss"),
            "description": _(
                "Indicates the percentage of lost packets observed in ICMP probes. "
                "Obtained with the fping linux program."
            ),
            "summary_labels": [_("Average packet loss")],
            "unit": "%",
            "colors": "#d62728",
            "order": 210,
            "query": chart_query["packet_loss"],
        },
        "rtt": {
            "type": "scatter",
            "title": _("Round Trip Time"),
            "description": _(
                "Round trip time observed in ICMP probes, measuered in milliseconds."
            ),
            "summary_labels": [
                _("Average RTT"),
                _("Average Max RTT"),
                _("Average Min RTT"),
            ],
            "unit": _(" ms"),
            "order": 220,
            "query": chart_query["rtt"],
        },
    },
    "alert_settings": {"operator": "<", "threshold": 1, "tolerance": 0},
    "notification": {
        "problem": {
            "verbose_name": "Ping PROBLEM",
            "verb": "cannot be reached anymore",
            "level": "warning",
            "email_subject": _(
                "[{site.name}] {notification.target} is not reachable"
            ),
            "message": _(
                "The device [{notification.target}] {notification.verb} anymore by our ping
                "messages."
            ),
        },
        "recovery": {
            "verbose_name": "Ping RECOVERY",
            "verb": "has become reachable",
            "level": "info",
            "email_subject": _(
                "[{site.name}] {notification.target} is reachable again"
            ),
            "message": _(
                "The device [{notification.target}] {notification.verb} again by our ping "
                "messages."
            ),
        },
    },
```

```
}

# Register your custom metric configuration
register_metric("ping", metric_config)
```

The above example will register one metric configuration (named `ping`), three chart configurations (named `rtt`, `packet_loss`, `uptime`) as defined in the **charts** key, two notification types (named `ping_recovery`, `ping_problem`) as defined in **notification** key.

The `AlertSettings` of `ping` metric will by default use `threshold` and `tolerance` defined in the `alert_settings` key. You can always override them and define your own custom values via the *admin*.

You can also use the `alert_field` key in metric configuration which allows `AlertSettings` to check the `threshold` on `alert_field` instead of the default `field_name` key.

Note

It will raise `ImproperlyConfigured` exception if a metric configuration is already registered with same name (not to be confused with verbose_name).

If you don't need to register a new metric but need to change a specific key of an existing metric configuration, you can use OPENWISP_MONITORING_METRICS.

`unregister_metric`

This function is used to unregister a metric configuration from anywhere in your code.

Parameter	Description
metric_name:	A `str` defining name of the metric configuration.

An example usage is shown below.

```
from openwisp_monitoring.monitoring.configuration import unregister_metric

# Unregister previously registered metric configuration
unregister_metric("metric_name")
```

Note

It will raise `ImproperlyConfigured` exception if the concerned metric configuration is not registered.

Registering / Unregistering Chart Configuration

OpenWISP Monitoring provides registering and unregistering chart configuration through utility functions `openwisp_monitoring.monitoring.configuration.register_chart` and `openwisp_monitoring.monitoring.configuration.unregister_chart`. Using these functions you can register or unregister chart configurations from anywhere in your code.

`register_chart`

This function is used to register a new chart configuration from anywhere in your code.

Parameter	Description

chart_name:	A `str` defining name of the chart configuration.
chart_configuration:	A `dict` defining configuration of the chart.

An example usage has been shown below.

```
from openwisp_monitoring.monitoring.configuration import register_chart

# Define configuration of your chart
chart_config = {
    "type": "histogram",
    "title": "Histogram",
    "description": "Histogram",
    "top_fields": 2,
    "order": 999,
    "query": {
        "influxdb": (
            "SELECT {fields|SUM|/ 1} FROM {key} "
            "WHERE time >= '{time}' AND content_type = "
            "'{content_type}' AND object_id = '{object_id}'"
        )
    },
}

# Register your custom chart configuration
register_chart("chart_name", chart_config)
```

Note

It will raise `ImproperlyConfigured` exception if a chart configuration is already registered with same name (not to be confused with verbose_name).

If you don't need to register a new chart but need to change a specific key of an existing chart configuration, you can use OPENWISP_MONITORING_CHARTS.

`unregister_chart`

This function is used to unregister a chart configuration from anywhere in your code.

Parameter	Description
chart_name:	A `str` defining name of the chart configuration.

An example usage is shown below.

```
from openwisp_monitoring.monitoring.configuration import unregister_chart

# Unregister previously registered chart configuration
unregister_chart("chart_name")
```

Note

It will raise `ImproperlyConfigured` exception if the concerned chart configuration is not registered.

Monitoring Notifications

OpenWISP Monitoring registers and uses the following notification types:

- `threshold_crossed`: Fires when a metric crosses the boundary defined in the threshold value of the alert settings.
- `threhold_recovery`: Fires when a metric goes back within the expected range.
- `connection_is_working`: Fires when the connection to a device is working.
- `connection_is_not_working`: Fires when the connection (e.g.: SSH) to a device stops working (e.g.: credentials are outdated, management IP address is outdated, or device is not reachable).

Registering Notification Types

You can define your own notification types using `register_notification_type` function from OpenWISP Notifications.

For more information, see the relevant documentation section about registering notification types in the Notifications module.

Once a new notification type is registered, you have to use the "notify" signal provided the Notifications module to send notifications for this type.

Signals

Note

If you're not familiar with signals, please refer to the Django Signals documentation.

`device_metrics_received`

Full Python path: `openwisp_monitoring.device.signals.device_metrics_received`

Arguments:

- `instance`: instance of `Device` whose metrics have been received
- `request`: the HTTP request object
- `time`: time with which metrics will be saved. If none, then server time will be used
- `current`: whether the data has just been collected or was collected previously and sent now due to network connectivity issues

This signal is emitted when device metrics are received to the `DeviceMetric` view (only when using HTTP POST).

The signal is emitted just before a successful response is returned, it is not sent if the response was not successful.

`health_status_changed`

Full Python path: `openwisp_monitoring.device.signals.health_status_changed`

Arguments:

- `instance`: instance of `DeviceMonitoring` whose status has been changed
- `status`: the status by which DeviceMonitoring's existing status has been updated with

This signal is emitted only if the health status of DeviceMonitoring object gets updated.

`threshold_crossed`

Full Python path: `openwisp_monitoring.monitoring.signals.threshold_crossed`

Arguments:

- `metric`: `Metric` object whose threshold defined in related alert settings was crossed
- `alert_settings`: `AlertSettings` related to the `Metric`
- `target`: related `Device` object
- `first_time`: it will be set to true when the metric is written for the first time. It shall be set to false afterwards.
- `tolerance_crossed`: it will be set to true if the metric has crossed the threshold for tolerance configured in alert settings. Otherwise, it will be set to false.

`first_time` parameter can be used to avoid initiating unneeded actions. For example, sending recovery notifications.

This signal is emitted when the threshold value of a `Metric` defined in alert settings is crossed.

`pre_metric_write`

Full Python path: `openwisp_monitoring.monitoring.signals.pre_metric_write`

Arguments:

- `metric`: `Metric` object whose data shall be stored in timeseries database
- `values`: metric data that shall be stored in the timeseries database
- `time`: time with which metrics will be saved
- `current`: whether the data has just been collected or was collected previously and sent now due to network connectivity issues

This signal is emitted for every metric before the write operation is sent to the timeseries database.

`post_metric_write`

Full Python path: `openwisp_monitoring.monitoring.signals.post_metric_write`

Arguments:

- `metric`: `Metric` object whose data is being stored in timeseries database
- `values`: metric data that is being stored in the timeseries database
- `time`: time with which metrics will be saved
- `current`: whether the data has just been collected or was collected previously and sent now due to network connectivity issues

This signal is emitted for every metric after the write operation is successfully executed in the background.

Exceptions

`TimeseriesWriteException`

Full Python path: `openwisp_monitoring.db.exceptions.TimeseriesWriteException`

If there is any failure due while writing data in timeseries database, this exception will be raised with a helpful error message explaining the cause of the failure. This exception will normally be caught and the failed write task will be retried in the background so that there is no loss of data if failures occur due to overload of Timeseries server. You can read more about this retry mechanism at OPENWISP_MONITORING_WRITE_RETRY_OPTIONS.

`InvalidMetricConfigException`

Full Python path: `openwisp_monitoring.monitoring.exceptions.InvalidMetricConfigException`

This exception will be raised if the metric configuration is broken.

`InvalidChartConfigException`

Full Python path: `openwisp_monitoring.monitoring.exceptions.InvalidChartConfigException`

This exception will be raised if the chart configuration is broken.

Extending OpenWISP Monitoring

Note

This page is for developers who want to customize or extend OpenWISP Monitoring, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Monitoring User Docs

One of the core values of the OpenWISP project is Software Reusability, for this reason *openwisp-monitoring* provides a set of base classes which can be imported, extended and reused to create derivative apps.

In order to implement your custom version of *openwisp-monitoring*, you need to perform the steps described in the rest of this section.

When in doubt, the code in the test project and the `sample apps` namely sample_check, sample_monitoring, sample_device_monitoring will guide you in the correct direction: just replicate and adapt that code to get a basic derivative of *openwisp-monitoring* working.

Important

If you plan on using a customized version of this module, we suggest to start with it since the beginning, because migrating your data from the default module to your extended version may be time consuming.

1. Initialize your Custom Module

The first thing you need to do in order to extend any *openwisp-monitoring* app is create a new django app which will contain your custom version of that *openwisp-monitoring* app.

A django app is nothing more than a python package (a directory of python scripts), in the following examples we'll call these django apps as `mycheck`, `mydevicemonitoring`, `mymonitoring` but you can name it how you want:

```
django-admin startapp mycheck
django-admin startapp mydevicemonitoring
django-admin startapp mymonitoring
```

Keep in mind that the command mentioned above must be called from a directory which is available in your PYTHON_PATH so that you can then import the result into your project.

Now you need to add `mycheck` to `INSTALLED_APPS` in your `settings.py`, ensuring also that `openwisp_monitoring.check` has been removed:

```
INSTALLED_APPS = [
    # ... other apps ...
    # 'openwisp_monitoring.check',        <-- comment out or delete this line
    # 'openwisp_monitoring.device',       <-- comment out or delete this line
    # 'openwisp_monitoring.monitoring'    <-- comment out or delete this line
    "mycheck",
    "mydevicemonitoring",
    "mymonitoring",
    "nested_admin",
]
```

For more information about how to work with django projects and django apps, please refer to the "Tutorial: Writing your first Django app" in the django documentation.

2. Install `openwisp-monitoring`

Install (and add to the requirement of your project) *openwisp-monitoring*:

```
pip install --U https://github.com/openwisp/openwisp-monitoring/tarball/master
```

3. Add `EXTENDED_APPS`

Add the following to your `settings.py`:

```
EXTENDED_APPS = ["device_monitoring", "monitoring", "check"]
```

4. Add `openwisp_utils.staticfiles.DependencyFinder`

Add `openwisp_utils.staticfiles.DependencyFinder` to `STATICFILES_FINDERS` in your `settings.py`:

```
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
```

```
    "openwisp_utils.staticfiles.DependencyFinder",
]
```

5. Add `openwisp_utils.loaders.DependencyLoader`

Add `openwisp_utils.loaders.DependencyLoader` to `TEMPLATES` in your `settings.py`:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "OPTIONS": {
            "loaders": [
                "django.template.loaders.filesystem.Loader",
                "django.template.loaders.app_directories.Loader",
                "openwisp_utils.loaders.DependencyLoader",
            ],
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    }
]
```

6. Inherit the AppConfig Class

Please refer to the following files in the sample app of the test project:

- sample_check/__init__.py.
- sample_check/apps.py.
- sample_monitoring/__init__.py.
- sample_monitoring/apps.py.
- sample_device_monitoring/__init__.py.
- sample_device_monitoring/apps.py.

For more information regarding the concept of `AppConfig` please refer to the "Applications" section in the django documentation.

7. Create your Custom Models

To extend `check` app, refer to sample_check models.py file.

To extend `monitoring` app, refer to sample_monitoring models.py file.

To extend `device_monitoring` app, refer to sample_device_monitoring models.py file.

Note

- For doubts regarding how to use, extend or develop models please refer to the "Models" section in the django documentation.
- For doubts regarding proxy models please refer to proxy models.

8. Add Swapper Configurations

Add the following to your `settings.py`:

```
# Setting models for swapper module
# For extending check app
CHECK_CHECK_MODEL = "YOUR_MODULE_NAME.Check"
# For extending monitoring app
MONITORING_CHART_MODEL = "YOUR_MODULE_NAME.Chart"
MONITORING_METRIC_MODEL = "YOUR_MODULE_NAME.Metric"
MONITORING_ALERTSETTINGS_MODEL = "YOUR_MODULE_NAME.AlertSettings"
# For extending device_monitoring app
DEVICE_MONITORING_DEVICEDATA_MODEL = "YOUR_MODULE_NAME.DeviceData"
DEVICE_MONITORING_DEVICEMONITORING_MODEL = (
    "YOUR_MODULE_NAME.DeviceMonitoring"
)
DEVICE_MONITORING_WIFICLIENT_MODEL = "YOUR_MODULE_NAME.WifiClient"
DEVICE_MONITORING_WIFISESSION_MODEL = "YOUR_MODULE_NAME.WifiSession"
```

Substitute `<YOUR_MODULE_NAME>` with your actual django app name (also known as `app_label`).

9. Create Database Migrations

Create and apply database migrations:

```
./manage.py makemigrations
./manage.py migrate
```

For more information, refer to the "Migrations" section in the django documentation.

10. Create your Custom Admin

To extend `check` app, refer to sample_check admin.py file.

To extend `monitoring` app, refer to sample_monitoring admin.py file.

To extend `device_monitoring` app, refer to sample_device_monitoring admin.py file.

To introduce changes to the admin, you can do it in the two ways described below.

Note

For doubts regarding how the django admin works, or how it can be customized, please refer to "The django admin site" section in the django documentation.

1. Monkey Patching

If the changes you need to add are relatively small, you can resort to monkey patching.

For example, for `check` app you can do it as:

```
from openwisp_monitoring.check.admin import CheckAdmin

CheckAdmin.list_display.insert(1, "my_custom_field")
CheckAdmin.ordering = ["-my_custom_field"]
```

Similarly for `device_monitoring` app, you can do it as:

```
from openwisp_monitoring.device.admin import DeviceAdmin, WifiSessionAdmin
```

```
DeviceAdmin.list_display.insert(1, "my_custom_field")
DeviceAdmin.ordering = ["-my_custom_field"]
WifiSessionAdmin.fields += ["my_custom_field"]
```

Similarly for `monitoring` app, you can do it as:

```
from openwisp_monitoring.monitoring.admin import (
    MetricAdmin,
    AlertSettingsAdmin,
)

MetricAdmin.list_display.insert(1, "my_custom_field")
MetricAdmin.ordering = ["-my_custom_field"]
AlertSettingsAdmin.list_display.insert(1, "my_custom_field")
AlertSettingsAdmin.ordering = ["-my_custom_field"]
```

2. Inheriting Admin Classes

If you need to introduce significant changes and/or you don't want to resort to monkey patching, you can proceed as follows:

For `check` app,

```
from django.contrib import admin

from openwisp_monitoring.check.admin import CheckAdmin as BaseCheckAdmin
from swapper import load_model

Check = load_model("check", "Check")

admin.site.unregister(Check)


@admin.register(Check)
class CheckAdmin(BaseCheckAdmin):
    # add your changes here
    pass
```

For `device_monitoring` app,

```
from django.contrib import admin

from openwisp_monitoring.device_monitoring.admin import (
    DeviceAdmin as BaseDeviceAdmin,
)
from openwisp_monitoring.device_monitoring.admin import (
    WifiSessionAdmin as BaseWifiSessionAdmin,
)
from swapper import load_model

Device = load_model("config", "Device")
WifiSession = load_model("device_monitoring", "WifiSession")

admin.site.unregister(Device)
admin.site.unregister(WifiSession)


@admin.register(Device)
class DeviceAdmin(BaseDeviceAdmin):
    # add your changes here
    pass
```

```
@admin.register(WifiSession)
class WifiSessionAdmin(BaseWifiSessionAdmin):
    # add your changes here
    pass
```

For `monitoring` app,

```
from django.contrib import admin

from openwisp_monitoring.monitoring.admin import (
    AlertSettingsAdmin as BaseAlertSettingsAdmin,
    MetricAdmin as BaseMetricAdmin,
)
from swapper import load_model

Metric = load_model("Metric")
AlertSettings = load_model("AlertSettings")

admin.site.unregister(Metric)
admin.site.unregister(AlertSettings)


@admin.register(Metric)
class MetricAdmin(BaseMetricAdmin):
    # add your changes here
    pass


@admin.register(AlertSettings)
class AlertSettingsAdmin(BaseAlertSettingsAdmin):
    # add your changes here
    pass
```

11. Create Root URL Configuration

Please refer to the urls.py file in the test project.

For more information about URL configuration in django, please refer to the "URL dispatcher" section in the django documentation.

12. Create `celery.py`

Please refer to the celery.py file in the test project.

For more information about the usage of celery in django, please refer to the "First steps with Django" section in the celery documentation.

13. Import Celery Tasks

Add the following in your `settings.py` to import celery tasks from `device_monitoring` app.

```
CELERY_IMPORTS = ("openwisp_monitoring.device.tasks",)
```

14. Create the Custom Command `run_checks`

Please refer to the run_checks.py file in the test project.

For more information about the usage of custom management commands in django, please refer to the "Writing custom django-admin commands" section in the django documentation.

15. Import the Automated Tests

When developing a custom application based on this module, it's a good idea to import and run the base tests too, so that you can be sure the changes you're introducing are not breaking some of the existing features of openwisp-monitoring.

In case you need to add breaking changes, you can overwrite the tests defined in the base classes to test your own behavior.

For, extending `check` app see the tests of sample_check app to find out how to do this.

For, extending `device_monitoring` app see the tests of sample_device_monitoring app to find out how to do this.

For, extending `monitoring` app see the tests of sample_monitoring app to find out how to do this.

Other Base Classes that can be Inherited and Extended

The following steps are not required and are intended for more advanced customization.

`DeviceMetricView`

This view is responsible for displaying `Charts` and `Status` primarily.

The full python path is: `openwisp_monitoring.device.api.views.DeviceMetricView`.

If you want to extend this view, you will have to perform the additional steps below.

Step 1. Import and extend view:

```
# mydevice/api/views.py
from openwisp_monitoring.device.api.views import (
    DeviceMetricView as BaseDeviceMetricView,
)


class DeviceMetricView(BaseDeviceMetricView):
    # add your customizations here ...
    pass
```

Step 2: remove the following line from your root `urls.py` file:

```
re_path(
    "api/v1/monitoring/device/(?P<pk>[^/]+)/$",
    views.device_metric,
    name="api_device_metric",
),
```

Step 3: add an URL route pointing to your custom view in `urls.py` file:

```
# urls.py
from mydevice.api.views import DeviceMetricView

urlpatterns = [
    # ... other URLs
    re_path(
        r"^(?P<path>.*)$",
        DeviceMetricView.as_view(),
        name="api_device_metric",
    ),
]
```

Other useful resources:

- REST API Reference
- Settings

Network Topology

Seealso

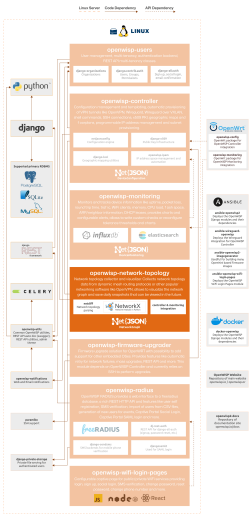
Source code: github.com/openwisp/openwisp-network-topology.

OpenWISP Network Topology is a network topology collector and visualizer web application and API, it allows to collect network topology data from different networking software (dynamic mesh routing protocols, OpenVPN), store it, visualize it, edit its details, it also provides hooks (a.k.a Django signals) to execute code when the status of a link changes.

When used in conjunction with OpenWISP Controller and OpenWISP Monitoring, it makes the monitoring system faster in detecting change to the network.

For a comprehensive overview of features, please refer to the Network Topology: Features page.

The following diagram illustrates the role of the Network Topology module within the OpenWISP architecture.



OpenWISP Architecture: highlighted network topology module

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

Network Topology: Features

OpenWISP Network Topology module offers robust features for managing, visualizing, and monitoring network topologies. Key features include:

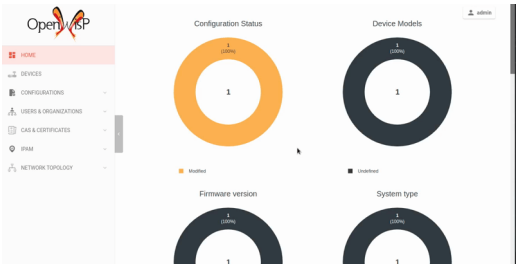
- **network topology collector supporting different formats:**
    - NetJSON NetworkGraph
    - OLSR (jsoninfo/txtinfo)
    - batman-adv (jsondoc/txtinfo)
    - BMX6 (q6m)

    - CNML 1.0
    - OpenVPN
    - Wireguard
    - ZeroTier
    - additional formats can be added by writing custom netdiff parsers
- **network topology visualizer** based on netjsongraph.js
- Rest API that exposes data in NetJSON *NetworkGraph* format
- **admin interface** that allows to easily manage, audit, visualize and debug topologies and their relative data (nodes, links)
- RECEIVE network topology data from multiple nodes
- **topology history**: allows saving daily snapshots of each topology that can be viewed in the frontend
- **faster monitoring**: integrates with OpenWISP Controller and OpenWISP Monitoring for faster detection of critical events in the network

Quick Start Guide

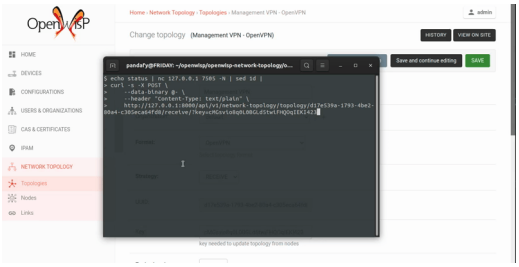
This module works by periodically collecting the network topology graph data of the supported networking software or formats. The data has to be either fetched by the application or received in POST API requests, therefore after deploying the application, additional steps are required to make the data collection and visualization work, read on to find out how.

Creating a Topology



1. Create a topology object by going to *Network Topology* > *Topologies* > *Add topology*.
2. Give an appropriate label to the topology.
3. Select the *topology format* from the dropdown menu. The *topology format* determines which parser should be used to process topology data.
4. Select the *Strategy* for updating this topology.
    - If you are using FETCH strategy, then enter the URL for fetching topology data in the *Url* field.
    - If you are using RECEIVE strategy, you will get the *URL* for sending topology data. The *RECEIVE* strategy provides an additional field *expiration time*. This can be used to add delay in marking missing links as down.

Sending Data for Topology with RECEIVE Strategy



1. Copy the *URL* generated by OpenWISP for sending the topology data.

   E.g., in our case the URL is `http://127.0.0.1:8000/api/v1/network-topology/topology/d17e539a-1793-4be2-80a4-c305eca64fd8/receive/?key=cMGsvio8q0L0BGLd5twiFHQOqIEKI423`.

   > **Note**
   >
   > The topology receive URL is shown only after the topology object is created.

2. Create a script (e.g.: `/opt/send-topology.sh`) which sends the topology data using `POST`, in the example script below we are sending the status log data of OpenVPN but the same code can be applied to other formats by replacing `cat /var/log/openvpn/tun0.stats` with the actual command which returns the network topology output:

```
#!/bin/bash
# replace COMMAND with the command used to fetch the topology data
COMMAND="cat /var/log/openvpn/tun0.stats"
UUID="<TOPOLOGY-UUID-HERE>"
KEY="<TOPOLOGY-KEY-HERE>"
OPENWISP_URL="https://<OPENWISP_DOMAIN_HERE>"
$COMMAND |
    # Upload the topology data to OpenWISP
    curl -X POST \
        --data-binary @- \
        --header "Content-Type: text/plain" \
        $OPENWISP_URL/api/v1/network-topology/topology/$UUID/receive/?key=$KEY
```

3. Add the `/opt/send-topology.sh` script created in the previous step to the crontab, here's an example which sends the topology data every 5 minutes:

```
# flag script as executable
chmod +x /opt/send-topology.sh
# open crontab
crontab -e

## Add the following line and save

echo */5 * * * * /opt/send-topology.sh
```

4. Once the steps above are completed, you should see nodes and links being created automatically, you can see the network topology graph from the admin page of the topology change page (you have to click on the *View topology graph* button in the upper right part of the page) or, alternatively, a non-admin visualizer page is also available at the URL `/topology/topology/<TOPOLOGY-UUID>/`.

Sending Data for ZeroTier Topology with RECEIVE Strategy

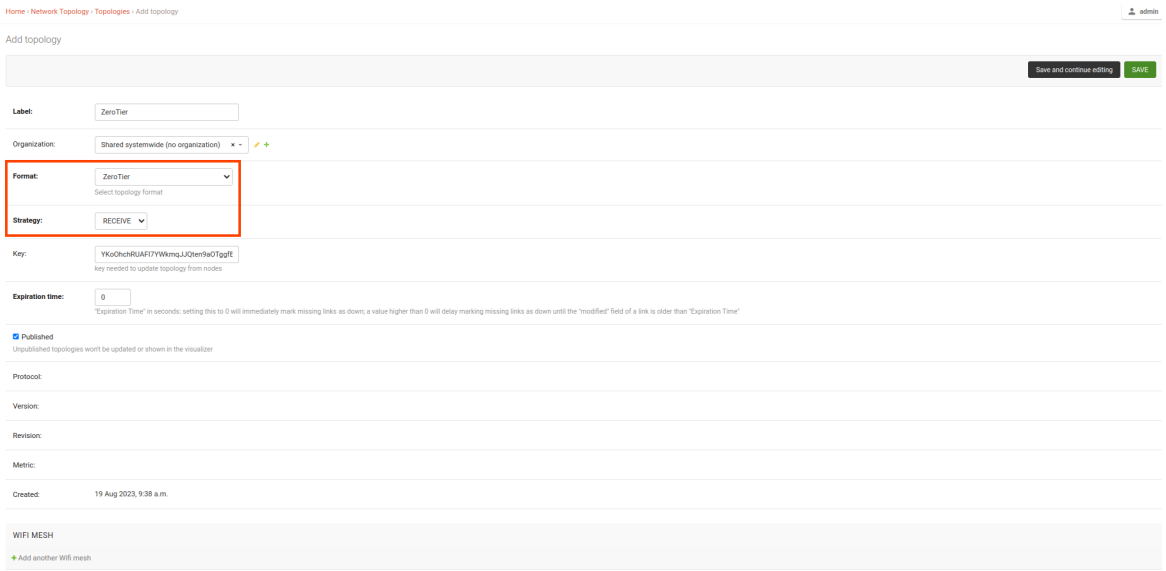
Follow the procedure described below to setup ZeroTier topology with RECEIVE strategy.

Note

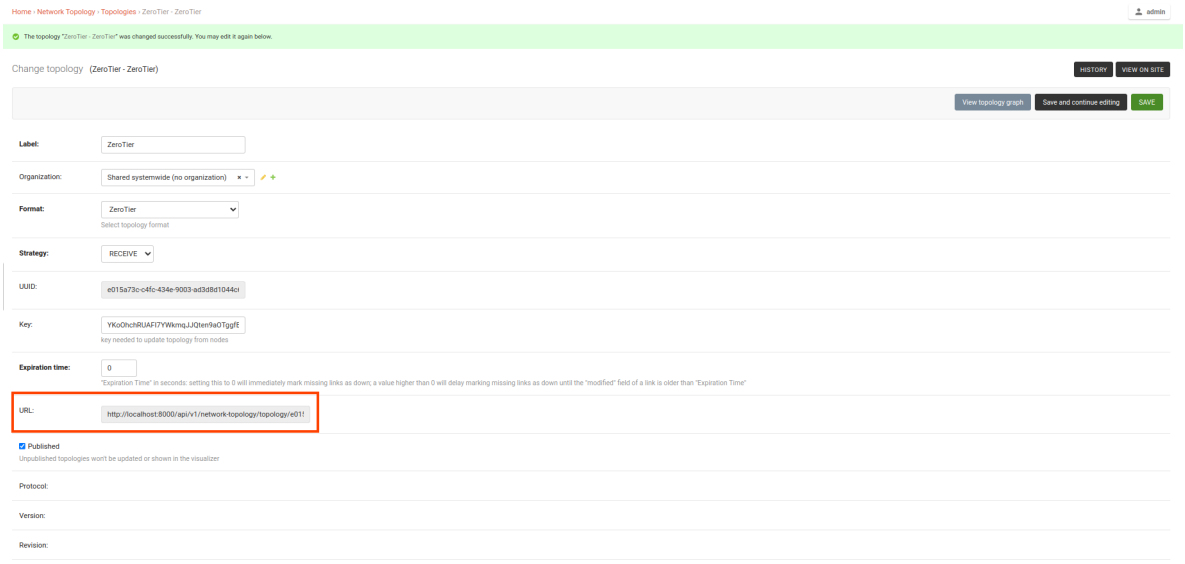
In this example, the **Shared systemwide (no organization)** option is used for the ZeroTier topology organization. You are free to opt for any organization, as long as both the topology and the device share the same organization, assuming the OpenWISP controller integration feature is enabled.

1. Create Topology for ZeroTier

1. Visit `admin/topology/topology/add` to add a new topology.
2. We will set the **Label** of this topology to `ZeroTier` and select the topology **Format** from the dropdown as `ZeroTier`.
3. Select the strategy as `RECEIVE` from the dropdown.



4. Let use default **Expiration time** `0` and make sure **Published** option is checked.
5. After clicking on the **Save and continue editing** button, a topology receive URL is generated. Make sure you copy that URL for later use in the topology script.



2. Create a Script for Sending ZeroTier Topology Data

1. Now, create a script (e.g: `/opt/send-zt-topology.sh`) that sends the ZeroTier topology data using a POST request. In the example script below, we are sending the ZeroTier self-hosted controller peers data:

```
#!/bin/bash
# command to fetch zerotier controller peers data in json format
COMMAND="zerotier-cli peers -j"
UUID="<TOPOLOGY-UUID-HERE>"
KEY="<TOPOLOGY-KEY-HERE>"
OPENWISP_URL="https://<OPENWISP_DOMAIN_HERE>"
$COMMAND |
    # Upload the topology data to OpenWISP
    curl -X POST \
        --data-binary @- \
        --header "Content-Type: text/plain" \
        $OPENWISP_URL/api/v1/network-topology/topology/$UUID/receive/?key=$KEY
```

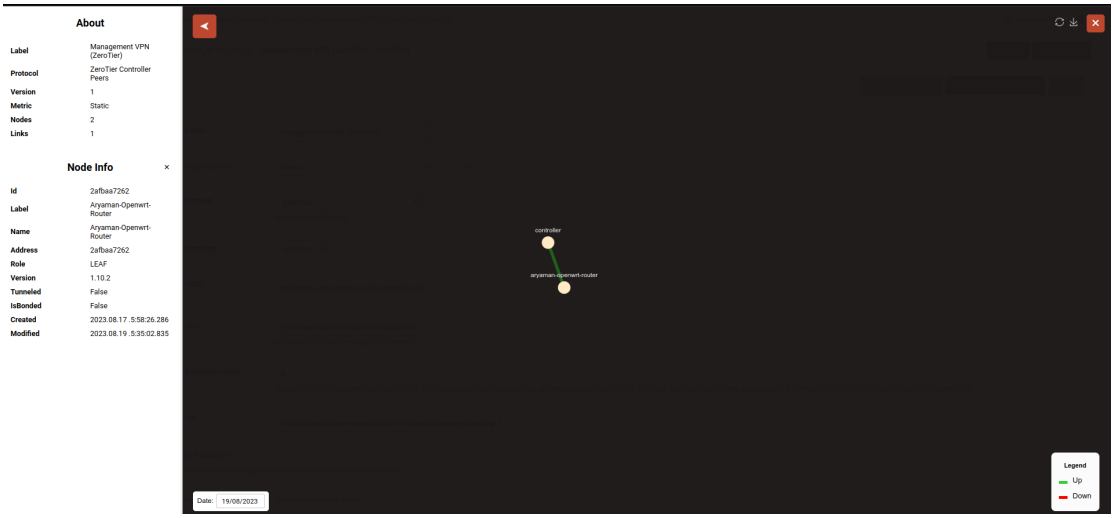
2. Add the `/opt/send-zt-topology.sh` script created in the previous step to the root crontab, here's an example which sends the topology data every **5 minutes**:

```
# flag script as executable
chmod +x /opt/send-zt-topology.sh

# open rootcrontab
sudo crontab -e

## Add the following line and save

echo */5 * * * * /opt/send-zt-topology.sh
```

Note

When using the **ZeroTier** topology, ensure that you use `sudo crontab -e` to edit the **root crontab**. This step is essential because the `zerotier-cli peers -j` command requires **root privileges** for kernel interaction, without which the command will not function correctly.

3. Once the steps above are completed, you should see nodes and links being created automatically, you can see the network topology graph from the admin page of the topology change page (you have to click on the **View topology graph** button in the upper right part of the page) or, alternatively, a non-admin visualizer page is also available at the URL `/topology/topology/<TOPOLOGY-UUID>/`.



Topology Collection Strategies

There are mainly two ways of collecting topology information:

- **FETCH** strategy
- **RECEIVE** strategy

Each `Topology` instance has a `strategy` field which can be set to the desired setting.

FETCH Strategy

Topology data will be fetched from a URL.

When some links are not detected anymore they will be flagged as "down" straightaway.

RECEIVE Strategy

Topology data is sent directly from one or more nodes of the network.

The collector waits to receive data in the payload of a POST HTTP request; when such a request is received, a `key` parameter it's first checked against the `Topology` key.

If the request is authorized the collector proceeds to update the topology.

If the data is sent from one node only, it's highly advised to set the `expiration_time` of the `Topology` instance to `0` (seconds), this way the system works just like in the **FETCH strategy**, with the only difference that the data is sent by one node instead of fetched by the collector.

If the data is sent from multiple nodes, you **SHOULD** set the `expiration_time` of the `Topology` instance to a value slightly higher than the interval used by nodes to send the topology, this way links will be flagged as "down" only if they haven't been detected for a while. This mechanism allows to visualize the topology even if the network has been split in several parts, the disadvantage is that it will take a bit more time to detect links that go offline.

Integrations with other OpenWISP modules

If you use OpenWISP Controller or OpenWISP Monitoring and you use OpenVPN, Wireguard or ZeroTier for the management VPN, you can use the integration available in `openwisp_network_topology.integrations.device`.

This additional and optional module provides the following features:

- whenever the status of a link changes:
  - the management IP address of the related device is updated straightaway
  - if OpenWISP Monitoring is enabled, the device checks are triggered (e.g.: ping)
- if OpenWISP Monitoring is installed and enabled, the system can automatically create topology for the WiFi Mesh (802.11s) interfaces using the monitoring data provided by the agent. You can enable this by setting OPENWISP_NETWORK_TOPOLOGY_WIFI_MESH_INTEGRATION to `True`.

This integration makes the whole system a lot faster in detecting important events in the network.

Note

If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

In order to use this module simply add `openwisp_network_topology.integrations.device` to `INSTALLED_APPS` in the Django project settings, e.g.:

```
INSTALLED_APPS.append("openwisp_network_topology.integrations.device")
```

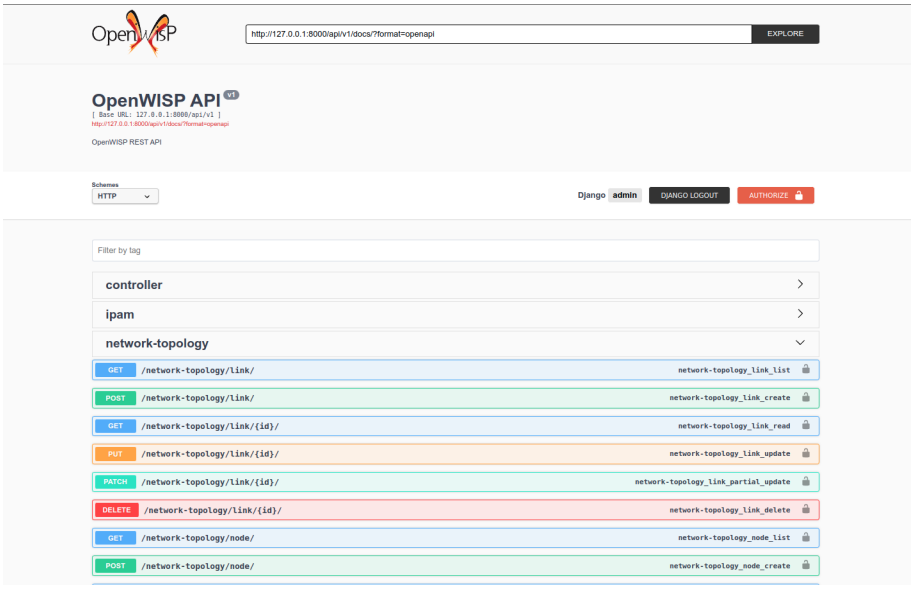
If you have enabled WiFI Mesh integration, you will also need to update the `CELERY_BEAT_SCHEDULE` as follow:

```
CELERY_BEAT_SCHEDULE.update(
    {
        "create_mesh_topology": {
            # This task generates the mesh topology from monitoring data
            "task": "openwisp_network_topology.integrations.device.tasks.create_mesh_topolog
            # Execute this task every 5 minutes
            "schedule": timedelta(minutes=5),
            "args": (
                # List of organization UUIDs. The mesh topology will be
                # created only for devices belonging these organizations.
                [
                    "4e002f97-eb01-4371-a4a8-857faa22fe5c",
                    "be88d4c4-599a-4ca2-a1c0-3839b4fdc315",
                ],
                # The task won't use monitoring data reported
                # before this time (in seconds)
                6 * 60,  # 6 minutes
            ),
        },
    }
)
```

If you are enabling this integration on a preexisting system, use the create_device_nodes management command to create the relationship between devices and nodes.

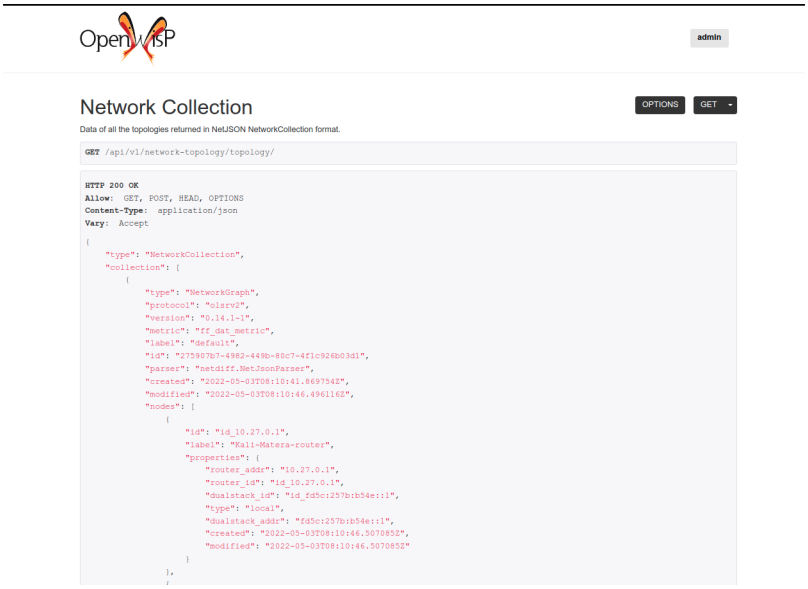
Rest API

Live Documentation



A general live API documentation (following the OpenAPI specification) at `/api/v1/docs/`.

Browsable Web Interface



Additionally, opening any of the endpoints listed below directly in the browser will show the browsable API interface of Django-REST-Framework, which makes it even easier to find out the details of each endpoint.

List of Endpoints

Since the detailed explanation is contained in the Live Documentation and in the Browsable Web Interface of each point, here we'll provide just a list of the available endpoints, for further information please open the URL of the endpoint in your browser.

List Topologies

```
GET /api/v1/network-topology/topology/
```

Available filters:

- `strategy`: Filter topologies based on their strategy (`fetch` or `receive`). E.g. `?strategy=<topology_strategy>`.
- `parser`: Filter topologies based on their parser. E.g. `?parser=<topology_parsers>`.
- `organization`: Filter topologies based on their organization. E.g. `?organization=<topology_organization_id>`.
- `organization_slug`: Filter topologies based on their organization slug. E.g. `?organization_slug=<topology_organization_slug>`.

You can use multiple filters in one request, e.g.:

```
/api/v1/network-topology/topology/?organization=371791ec-e3fe-4c9a-8972-3e8b882416f6&strateg
```

Note

By default, `/api/v1/network-topology/topology/` does not include unpublished topologies. If you want to include unpublished topologies in the response, use `?include_unpublished=true` filter as following:

```
GET /api/v1/network-topology/topology/?include_unpublished=true
```

Create Topology

```
POST /api/v1/network-topology/topology/
```

Detail of a Topology

```
GET /api/v1/network-topology/topology/{id}/
```

> **Note**
>
> By default, `/api/v1/network-topology/topology/{id}/` will return `HTTP 404 Not Found` for unpublished topologies. If you want to retrieve an unpublished topology, use `?include_unpublished=true` filter as following:

```
GET /api/v1/network-topology/topology/{id}/?include_unpublished=true
```

Change Topology Detail

```
PUT /api/v1/network-topology/topology/{id}/
```

Patch Topology Detail

```
PATCH /api/v1/network-topology/topology/{id}/
```

Delete Topology

```
DELETE /api/v1/network-topology/topology/{id}/
```

View Topology History

This endpoint is used to go back in time to view previous topology snapshots. For it to work, snapshots need to be saved periodically as described in save_snapshot section above.

For example, we could use the endpoint to view the snapshot of a topology saved on `2020-08-08` as follows.

```
GET /api/v1/network-topology/topology/{id}/history/?date=2020-08-08
```

Send Topology Data

```
POST /api/v1/network-topology/topology/{id}/receive/
```

List Links

```
GET /api/v1/network-topology/link/
```

Available filters:

- `topology`: Filter links belonging to a topology. E.g. `?topology=<topology_id>`.
- `organization`: Filter links belonging to an organization. E.g. `?organization=<organization_id>`.

- `organization_slug`: Filter links based on their organization slug. E.g. `?organization_slug=<organization_slug>`.
- `status`: Filter links based on their status (up or down). E.g. `?status=<link_status>`.

You can use multiple filters in one request, e.g.:

```
/api/v1/network-topology/link/?status=down&topology=7fce01bd-29c0-48b1-8fce-0508f2d75d36
```

Create Link

```
POST /api/v1/network-topology/link/
```

Get Link Detail

```
GET /api/v1/network-topology/link/{id}/
```

Change Link Detail

```
PUT /api/v1/network-topology/link/{id}/
```

Patch Link Detail

```
PATCH /api/v1/network-topology/link/{id}/
```

Delete Link

```
DELETE /api/v1/network-topology/link/{id}/
```

List Nodes

```
GET /api/v1/network-topology/node/
```

Available filters:

- `topology`: Filter nodes belonging to a topology. E.g. `?topology=<topology_id>`.
- `organization`: Filter nodes belonging to an organization. E.g. `?organization=<organization_id>`.
- `organization_slug`: Filter nodes based on their organization slug. E.g. `?organization_slug=<organization_slug>`.

You can use multiple filters in one request, e.g.:

```
/api/v1/network-topology/node/?organization=371791ec-e3fe-4c9a-8972-3e8b882416f6&topology=7f
```

Create Node

```
POST /api/v1/network-topology/node/
```

Get Node Detail

```
GET /api/v1/network-topology/node/{id}/
```

Change Node Detail

```
PUT /api/v1/network-topology/node/{id}/
```

Patch Node Detail

```
PATCH /api/v1/network-topology/node/{id}/
```

Delete Node

```
DELETE /api/v1/network-topology/node/{id}/
```

Settings

> **Note**
>
> If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

`OPENWISP_NETWORK_TOPOLOGY_PARSERS`

type:	`list`
default:	`[]`

Additional custom netdiff parsers.

`OPENWISP_NETWORK_TOPOLOGY_SIGNALS`

type:	`str`
default:	`None`

String representing Python module to import on initialization.

Useful for loading Django signals or to define custom behavior.

`OPENWISP_NETWORK_TOPOLOGY_TIMEOUT`

type:	`int`
default:	`8`

Timeout when fetching topology URLs.

`OPENWISP_NETWORK_TOPOLOGY_LINK_EXPIRATION`

type:	`int`

default:	`60`

If a link is down for more days than this number, it will be deleted by the `update_topology` management command.

Setting this to `False` will disable this feature.

`OPENWISP_NETWORK_TOPOLOGY_NODE_EXPIRATION`

type:	`int`
default:	`False`

If a node has not been modified since the days specified and if it has no links, it will be deleted by the `update_topology` management command. This depends on `OPENWISP_NETWORK_TOPOLOGY_LINK_EXPIRATION` being enabled. Replace `False` with an integer to enable the feature.

`OPENWISP_NETWORK_TOPOLOGY_VISUALIZER_CSS`

type:	`str`
default:	`netjsongraph/css/style.css`

Path of the visualizer css file. Allows customization of css according to user's preferences.

`OPENWISP_NETWORK_TOPOLOGY_API_URLCONF`

type:	`string`
default:	`None`

Use the `urlconf` option to change receive API URL to point to another module, example, `myapp.urls`.

`OPENWISP_NETWORK_TOPOLOGY_API_BASEURL`

type:	`string`
default:	`None`

If you have a separate instance of the OpenWISP Network Topology API on a different domain, you can use this option to change the base of the URL, this will enable you to point all the API URLs to your API server's domain, example value: `https://api.myservice.com`.

`OPENWISP_NETWORK_TOPOLOGY_API_AUTH_REQUIRED`

type:	`boolean`
default:	`True`

When enabled, the API endpoints will only allow authenticated users who have the necessary permissions to access the objects which belong to the organizations the user manages.

`OPENWISP_NETWORK_TOPOLOGY_WIFI_MESH_INTEGRATION`

type:	`boolean`
default:	`False`

When enabled, network topology objects will be automatically created and updated based on the WiFi mesh interfaces peer information supplied by the monitoring agent.

Note

The network topology objects are created using the device monitoring data collected by OpenWISP Monitoring. Thus, it requires integration with OpenWISP Controller and OpenWISP Monitoring to be enabled in the Django project.

Management Commands

`update_topology`

After topology URLs (URLs exposing the files that the topology of the network) have been added in the admin, the `update_topology` management command can be used to collect data and start playing with the network graph:

```
./manage.py update_topology
```

The management command accepts a `--label` argument that will be used to search in topology labels, e.g.:

```
./manage.py update_topology --label mytopology
```

Logging

The `update_topology` management command will automatically try to log errors.

For a good default `LOGGING` configuration refer to the test settings.

`save_snapshot`

The `save_snapshot` management command can be used to save the topology graph data which could be used to view the network topology graph sometime in future:

```
./manage.py save_snapshot
```

The management command accepts a `--label` argument that will be used to search in topology labels, e.g.:

```
./manage.py save_snapshot --label mytopology
```

`upgrade_from_django_netjsongraph`

If you are upgrading from django-netjsongraph to openwisp-network-topology, there is an easy migration script that will import your topologies, users & groups to openwisp-network-topology instance:

```
./manage.py upgrade_from_django_netjsongraph
```

The management command accepts an argument `--backup`, that you can pass to give the location of the backup files, by default it looks in the `tests/` directory, e.g.:

```
./manage.py upgrade_from_django_netjsongraph --backup /home/user/django_netjsongraph/
```

The management command accepts another argument `--organization`, if you want to import data to a specific organization, you can give its UUID for the same, by default the data is added to the first found organization, e.g.:

```
./manage.py upgrade_from_django_netjsongraph --organization 900856da-c89a-412d-8fee-45a9c763
```

Note

you can follow the tutorial to migrate database from django-netjsongraph.

`create_device_nodes`

This management command can be used to create the initial `DeviceNode` relationships when the integration with OpenWISP Controller is enabled in a preexisting system which already has some devices and topology objects in its database.

```
./manage.py create_device_nodes
```

Developer Docs

Note

This page is for developers who want to customize or extend OpenWISP Network Topology, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Network Topology User Docs

Installation Instructions

Note

This page is for developers who want to customize or extend OpenWISP Network Topology, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Network Topology User Docs

Installing for Development

Install the system dependencies:

```
sudo apt install -y sqlite3 libsqlite3-dev
# Install system dependencies for spatialite which is required
# to run tests for openwisp-network-topology integrations with
# openwisp-network-topology and openwisp-monitoring.
sudo apt install libspatialite-dev libsqlite3-mod-spatialite
```

Fork and clone the forked repository:

```
git clone git://github.com/<your_fork>/openwisp-network-topology
```

Navigate into the cloned repository:

```
cd openwisp-network-topology/
```

Start InfluxDB and Redis using Docker (required by the test project to run tests for WiFi Mesh Integration):

```
docker-compose up -d influxdb redis
```

Setup and activate a virtual-environment (we'll be using virtualenv):

```
python -m virtualenv env
source env/bin/activate
```

Make sure that your base python packages are up to date before moving to the next step:

```
pip install -U pip wheel setuptools
```

Install development dependencies:

```
pip install -e .
pip install -r requirements-test.txt
```

Create database:

```
cd tests/
./manage.py migrate
./manage.py createsuperuser
```

You can access the admin interface at `http://127.0.0.1:8000/admin/`.

Run tests with:

```
# Running tests without setting the "WIFI_MESH" environment
# variable will not run tests for WiFi Mesh integration.
# This is done to avoid slowing down the test suite by adding
# dependencies which are only used by the integration.
./runtests.py
# You can run the tests only for WiFi mesh integration using
# the following command
WIFI_MESH=1 ./runtests.py
```

Run QA tests:

```
./run-qa-checks
```

Alternative Sources

Pypi

To install the latest Pypi:

```
pip install openwisp-network-topology
```

Github

To install the latest development version tarball via HTTPs:

```
pip install https://github.com/openwisp/openwisp-network-topology/tarball/master
```

Alternatively you can use the git protocol:

```
pip install -e git+git://github.com/openwisp/openwisp-network-topology#egg=openwisp_network-
```

Overriding Visualizer Templates

Note

This page is for developers who want to customize or extend OpenWISP Network Topology, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Network Topology User Docs

Follow these steps to override and customize the visualizer's default templates:

- create a directory in your django project and put its full path in `TEMPLATES['DIRS']`, which can be found in the django `settings.py` file
- create a sub directory named `netjsongraph` and add all the templates which shall override the default `netjsongraph/*` templates
- create a template file with the same name of the template file you want to override

More information about the syntax used in django templates can be found in the django templates documentation.

Example: Overriding the `<script>` Tag

Here's a step by step guide on how to change the javascript options passed to netjsongraph.js, remember to replace `<project_path>` with the absolute file system path of your project.

Step 1: create a directory in `<project_path>/templates/netjsongraph`

Step 2: open your `settings.py` and edit the `TEMPLATES['DIRS']` setting so that it looks like the following example:

```
# settings.py
TEMPLATES = [
    {
        "DIRS": [os.path.join(BASE_DIR, "templates")],
        # ... all other lines have been omitted for brevity ...
    }
]
```

Step 3: create a new file named `netjsongraph-script.html` in the new `<project_path>/templates/netjsongraph/` directory, e.g.:

```
<!-- <project_path>/templates/netjsongraph/netjsongraph-script.html -->
<script>
```

```
    // custom JS code here
</script>
```

Extending OpenWISP Network Topology

Note

This page is for developers who want to customize or extend OpenWISP Network Topology, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Network Topology User Docs

One of the core values of the OpenWISP project is Software Reusability, for this reason *openwisp-network-topology* provides a set of base classes which can be imported, extended and reused to create derivative apps.

In order to implement your custom version of *openwisp-network-topology*, you need to perform the steps described in this section.

When in doubt, the code in the test project and the sample app will serve you as source of truth: just replicate and adapt that code to get a basic derivative of *openwisp-network-topology* working.

Important

If you plan on using a customized version of this module, we suggest to start with it since the beginning, because migrating your data from the default module to your extended version may be time consuming.

1. Initialize your Custom Module

The first thing you need to do is to create a new django app which will contain your custom version of *openwisp-network-topology*.

A django app is nothing more than a python package (a directory of python scripts), in the following examples we'll call this django app `sample_network_topology`, but you can name it how you want:

```
django-admin startapp sample_network_topology
```

If you use the integration with openwisp-controller, you may want to extend also the integration app if you need:

```
django-admin startapp sample_integration_device
```

Keep in mind that the command mentioned above must be called from a directory which is available in your PYTHON_PATH so that you can then import the result into your project.

Now you need to add `sample_network_topology` to `INSTALLED_APPS` in your `settings.py`, ensuring also that `openwisp_network_topology` has been removed:

```
INSTALLED_APPS = [
    # ... other apps ...
    "openwisp_utils.admin_theme",
    # all-auth
    "django.contrib.sites",
    "openwisp_users.accounts",
    "allauth",
    "allauth.account",
    "allauth.socialaccount",
    # (optional) openwisp_controller - required only if you are using the integration app
    "openwisp_controller.pki",
    "openwisp_controller.config",
    "reversion",
    "sortedm2m",
    # network topology
    # 'sample_network_topology' <-- uncomment and replace with your app-name here
    # (optional) required only if you need to extend the integration app
    # 'sample_integration_device' <-- uncomment and replace with your integration-app-name h
    "openwisp_users",
    # admin
    "django.contrib.admin",
    # rest framework
    "rest_framework",
]
```

For more information about how to work with django projects and django apps, please refer to the django documentation.

2. Install `openwisp-network-topology`

Install (and add to the requirement of your project) openwisp-network-topology:

```
pip install openwisp-network-topology
```

3. Add `EXTENDED_APPS`

Add the following to your `settings.py`:

```
EXTENDED_APPS = ("openwisp_network_topology",)
```

4. Add `openwisp_utils.staticfiles.DependencyFinder`

Add `openwisp_utils.staticfiles.DependencyFinder` to `STATICFILES_FINDERS` in your `settings.py`:

```
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
    "openwisp_utils.staticfiles.DependencyFinder",
]
```

5. Add `openwisp_utils.loaders.DependencyLoader`

Add `openwisp_utils.loaders.DependencyLoader` to `TEMPLATES` in your `settings.py`:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "OPTIONS": {
            "loaders": [
                "django.template.loaders.filesystem.Loader",
                "django.template.loaders.app_directories.Loader",
                "openwisp_utils.loaders.DependencyLoader",
            ],
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    }
]
```

6. Inherit the AppConfig Class

Please refer to the following files in the sample app of the test project:

- sample_network_topology/__init__.py.
- sample_network_topology/apps.py.

For the integration with openwisp-controller, see:

- sample_integration_device/__init__.py.
- sample_integration_device/apps.py.

You have to replicate and adapt that code in your project.

For more information regarding the concept of `AppConfig` please refer to the "Applications" section in the django documentation.

7. Create your Custom Models

Please refer to sample_app models file use in the test project.

You have to replicate and adapt that code in your project.

Note

If you have questions about using, extending, or developing models, refer to the "Models" section of the Django documentation.

8. Add Swapper Configurations

Once you have created the models, add the following to your `settings.py`:

```
# Setting models for swapper module
TOPOLOGY_LINK_MODEL = "sample_network_topology.Link"
TOPOLOGY_NODE_MODEL = "sample_network_topology.Node"
TOPOLOGY_SNAPSHOT_MODEL = "sample_network_topology.Snapshot"
TOPOLOGY_TOPOLOGY_MODEL = "sample_network_topology.Topology"
# if you use the integration with OpenWISP Controller and/or OpenWISP Monitoring
TOPOLOGY_DEVICE_DEVICENODE_MODEL = "sample_integration_device.DeviceNode"
TOPOLOGY_DEVICE_WIFIMESH_MODEL = "sample_integration_device.WifiMesh"
```

Substitute `sample_network_topology` with the name you chose in step 1.

9. Create Database Migrations

Create and apply database migrations:

```
./manage.py makemigrations
./manage.py migrate
```

For more information, refer to the "Migrations" section in the django documentation.

10. Create the Admin

Refer to the admin.py file of the sample app.

To introduce changes to the admin, you can do it in two main ways which are described below.

Note

For more information regarding how the django admin works, or how it can be customized, please refer to "The django admin site" section in the django documentation.

1. Monkey Patching

If the changes you need to add are relatively small, you can resort to monkey patching.

For example:

```
from openwisp_network_topology.admin import (
    TopologyAdmin,
    LinkAdmin,
    NodeAdmin,
)

# TopologyAdmin.list_display.insert(1, 'my_custom_field') <-- your custom change example
# LinkAdmin.list_display.insert(1, 'my_custom_field') <-- your custom change example
# NodeAdmin.list_display.insert(1, 'my_custom_field') <-- your custom change example
```

2. Inheriting Admin Classes

If you need to introduce significant changes and/or you don't want to resort to monkey patching, you can proceed as follows:

```
from django.contrib import admin
from swapper import load_model

from openwisp_network_topology.admin import (
    TopologyAdmin as BaseTopologyAdmin,
    LinkAdmin as BaseLinkAdmin,
    NodeAdmin as BaseNodeAdmin,
)

Node = load_model("topology", "Node")
Link = load_model("topology", "Link")
Topology = load_model("topology", "Topology")

admin.site.unregister(Topology)
admin.site.unregister(Link)
admin.site.unregister(Node)


@admin.register(Topology, TopologyAdmin)
class TopologyAdmin(BaseTopologyAdmin):
    # add your changes here
    pass


@admin.register(Link, LinkAdmin)
class LinkAdmin(BaseLinkAdmin):
    # add your changes here
    pass


@admin.register(Node, NodeAdmin)
class NodeAdmin(BaseNodeAdmin):
    # add your changes here
    pass
```

11. Create Root URL Configuration

The following can be used to register all the URLs in your `urls.py`.

Please read and replicate according to your project needs:

```
# If you've extended visualizer views (discussed below).
# Import visualizer views & function to add it.
# from openwisp_network_topology.utils import get_visualizer_urls
# from .sample_network_topology.visualizer import views

urlpatterns = [
    # If you've extended visualizer views (discussed below).
    # Add visualizer views in urls.py
    # path('topology/', include(get_visualizer_urls(views))),
    path("", include("openwisp_network_topology.urls")),
    path("admin/", admin.site.urls),
]
```

For more information about URL configuration in django, please refer to the "URL dispatcher" section in the django documentation.

12. Setup API URLs

You need to create a file `api/urls.py` (the name & path of the file must match) inside your app, which contains the following:

```
from openwisp_network_topology.api import views

# When you want to modify views, please change views location
# from . import views
from openwisp_network_topology.utils import get_api_urls

urlpatterns = get_api_urls(views)
```

13. Extending Management Commands

To extend the management commands, create *sample_network_topology/management/commands* directory and two files in it:

- save_snapshot.py
- update_topology.py

14. Import the Automated Tests

When developing a custom application based on this module, it's a good idea to import and run the base tests too, so that you can be sure the changes you're introducing are not breaking some of the existing features of *openwisp-network-topology*.

Refer to the tests.py file of the sample app.

In case you need to add breaking changes, you can overwrite the tests defined in the base classes to test your own behavior.

For testing you also need to extend the fixtures, you can copy the file `openwisp_network_topology/fixtures/test_users.json` in your sample app's `fixtures/` directory.

Now, you can then run tests with:

```
# the --parallel flag is optional
./manage.py test --parallel sample_network_topology
```

Substitute `sample_network_topology` with the name you chose in step 1.

For more information about automated tests in django, please refer to "Testing in Django".

Other Base Classes that can be Inherited and Extended

The following steps are not required and are intended for more advanced customization.

1. Extending API Views

Extending the views is only required when you want to make changes in the behavior of the API. Please refer to sample_network_topology/api/views.py and replicate it in your application.

If you extend these views, remember to use these views in the `api/urls.py`.

2. Extending the Visualizer Views

Similar to API views, visualizer views are only required to be extended when you want to make changes in the Visualizer. Please refer to sample_network_topology/visualizer/views.py and replicate it in your application.

If you extend these views, remember to use these views in the `urls.py`.

Other useful resources:

- Rest API
- Settings

Firmware Upgrader

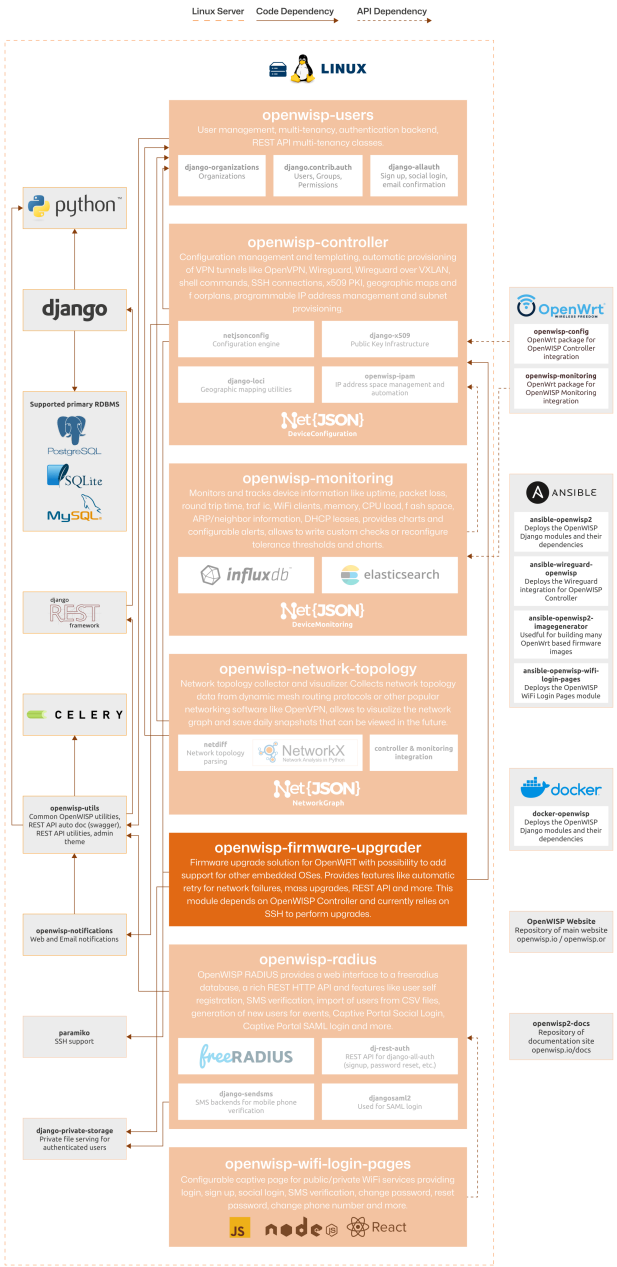
Seealso

Source code: github.com/openwisp/openwisp-firmware-upgrader.

A firmware upgrade solution designed specifically for OpenWrt devices, with the potential to support other embedded operating systems in the future. It offers a robust and automated upgrade process, featuring functionalities such as automatic device detection, retry mechanisms for network failures, mass upgrades, and a REST API for integration.

For a comprehensive overview of features, please refer to the Firmware Upgrader: Features page.

The following diagram illustrates the role of the Firmware Upgrader module within the OpenWISP architecture.



OpenWISP Architecture: highlighted firmware upgrader module

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

Firmware Upgrader: Features

- Stores information of each upgrade operation which can be seen from the device page
- Automatic retries for recoverable failures (e.g.: firmware image upload issues because of intermittent internet connection)
- Performs a final check to find out if the upgrade completed successfully or not
- Prevents accidental multiple upgrades using the same firmware image
- Single device upgrade
- Mass upgrades
- Possibility to divide firmware images in categories
- REST API
- Possibility of writing custom upgraders for other firmware OSes or for custom OpenWrt based firmwares
- Configurable timeouts

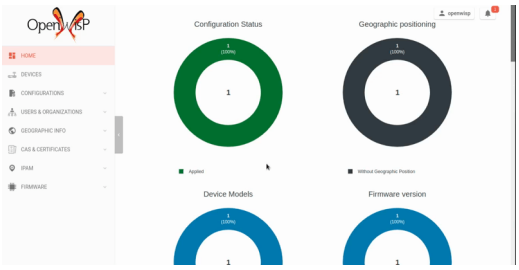
Quick Start Guide

Requirements

- Devices running at least OpenWrt 12.09 Attitude Adjustment, older versions of OpenWrt have not worked at all in our tests
- Devices must have enough free RAM to be able to upload the new image to `/tmp`

1. Create a Category

Create a category for your firmware images by going to *Firmware management* > *Firmware categories* > *Add firmware category*, if you use only one firmware type in your network, you could simply name the category "default" or "standard".



If you use multiple firmware images with different features, create one category for each firmware type, e.g.:

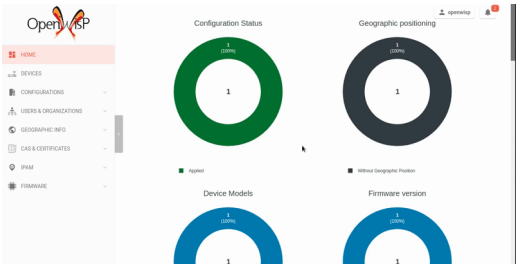
- WiFi
- SDN router
- LoRa Gateway

This is necessary in order to perform mass upgrades only on specific firmware categories when, for example, a new *LoRa Gateway* firmware becomes available.

2. Create the Build Object

Create a build object by going to *Firmware management* > *Firmware builds* > *Add firmware build*, the build object is related to a firmware category and is the collection of the different firmware images which have been compiled for the different hardware models supported by the system.

The version field indicates the firmware version, the change log field is optional but we recommend filling it to help operators know the differences between each version.



An important but optional field of the build model is **OS identifier**, this field should match the value of the **Operating System** field which gets automatically filled during device registration, e.g.: `OpenWrt 19.07-SNAPSHOT r11061-6ffd4d8a4d`. It is used by the firmware-upgrader module to automatically create `DeviceFirmware` objects for existing devices or when new devices register. A `DeviceFirmware` object represent the relationship between a device and a firmware image, it basically tells us which firmware image is installed on the device.

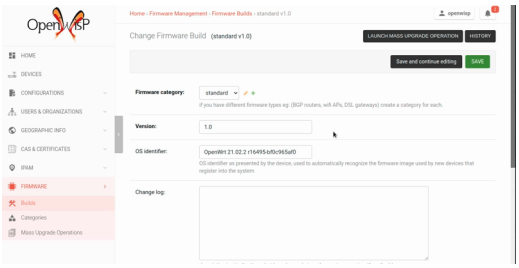
To find out the exact value to use, you should either do a test flash on a device and register it to the system or you should inspect the firmware image by decompressing it and find the generated value in the firmware image.

If you're not sure about what **OS identifier** to use, just leave it empty, you can fill it later on when you find out.

Now save the build object to create it.

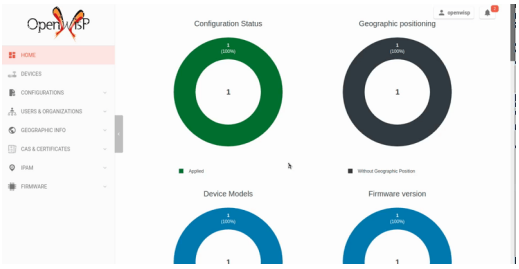
3. Upload Images to the Build

Now is time to add images to the build, we suggest adding one image at time. Alternatively the REST API can be used to automate this step.



If you use a hardware model which is not listed in the image types, if the hardware model is officially supported by OpenWrt, you can send us a pull-request to add it, otherwise you can use the setting OPENWISP_CUSTOM_OPENWRT_IMAGES to add it.

4. Perform a Firmware Upgrade to a Specific Device



Once a new build is ready, has been created in the system and its image have been uploaded, it will be the time to finally upgrade our devices.

To perform the upgrade of a single device, navigate to the device details, then go to the "Firmware" tab.

If you correctly filled **OS identifier** in step 2, you should have a situation similar to the one above: in this example, the device is using version `1.0` and we want to upgrade it to version `2.0`, once the new firmware image is selected we just have to hit save, then a new tab will appear in the device page which allows us to see what's going on during the upgrade.

Right now, the update of the upgrade information is not asynchronous yet, so you will have to reload the page periodically to find new information. This will be addressed in a future release.

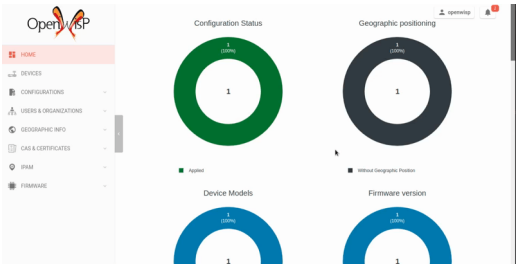
5. Performing Mass Upgrades

Before proceeding, please ensure the following preconditions are met:

- the system is configured correctly
- the new firmware images are working as expected
- you already tried the upgrade of single devices several times.

At this stage you can try a mass upgrade by doing the following:

- go to the build list page
- select the build which contains the latest firmware images you want the devices to be upgraded with
- click on "Mass-upgrade devices related to the selected build".



At this point you should see a summary page which will inform you of which devices are going to be upgraded, you can either confirm the operation or cancel.

Once the operation is confirmed you will be redirected to a page in which you can monitor the progress of the upgrade operations.

Right now, the update of the upgrade information is not asynchronous yet, so you will have to reload the page periodically to find new information. This will be addressed in a future release.

Automatic Device Firmware Detection

OpenWISP Firmware Upgrader maintains a data structure for mapping the firmware image files to board names called `OPENWRT_FIRMWARE_IMAGE_MAP`.

Here is an example firmware image item from `OPENWRT_FIRMWARE_IMAGE_MAP`

```
{
    # Firmware image file name.
    "ar71xx-generic-cf-e320n-v2-squashfs-sysupgrade.bin": {
        # Human readable name of the model which is displayed on
        # the UI
        "label": "COMFAST CF-E320N v2 (OpenWrt 19.07 and earlier)",
        # Tupe of board names with which the different versions
        # of the hardware are identified on OpenWrt
        "boards": ("COMFAST CF-E320N v2",),
    }
}
```

When a device registers on OpenWISP, the openwisp-config agent reads the device board name from */tmp/sysinfo/model* and sends it to OpenWISP. This value is then saved in the `Device.model` field. *OpenWISP Firmware Upgrader* uses this field to automatically detect the correct firmware image for the device.

Use the OPENWISP_CUSTOM_OPENWRT_IMAGES setting to add additional firmware image in your project.

Writing Custom Firmware Upgrader Classes

You can write custom upgraders for other firmware OSes or for custom OpenWrt based firmwares.

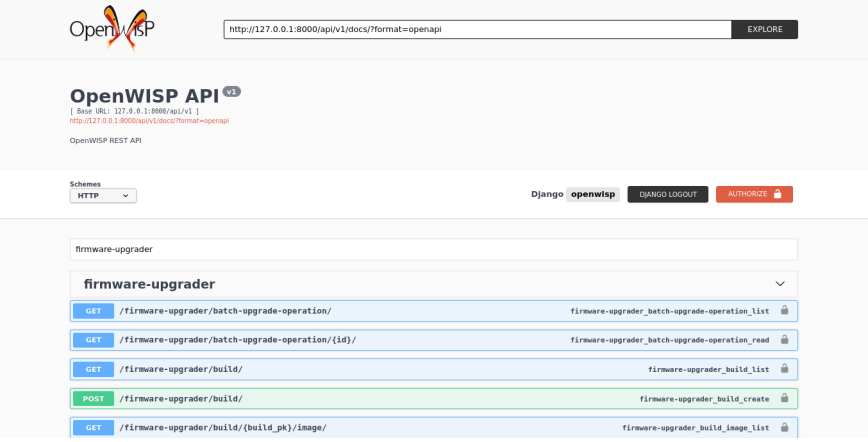
Here is an example custom OpenWrt firmware upgrader class:

```
from openwisp_firmware_upgrader.upgraders.openwrt import OpenWrt


class CustomOpenWrtBasedFirmware(OpenWrt):
    # this firmware uses a custom upgrade command
    UPGRADE_COMMAND = "upgrade_firmware.sh --keep-config"
    # it takes somewhat more time to boot so it needs more time
    RECONNECT_DELAY = 150
    RECONNECT_RETRY_DELAY = 5
    RECONNECT_MAX_RETRIES = 20

    def get_remote_path(self, image):
        return "/tmp/firmware.img"

    def get_upgrade_command(self, path):
        return self.UPGRADE_COMMAND
```

You will need to place your custom upgrader class on the python path of your application and then add this path to the OPENWISP_FIRMWARE_UPGRADERS_MAP setting.

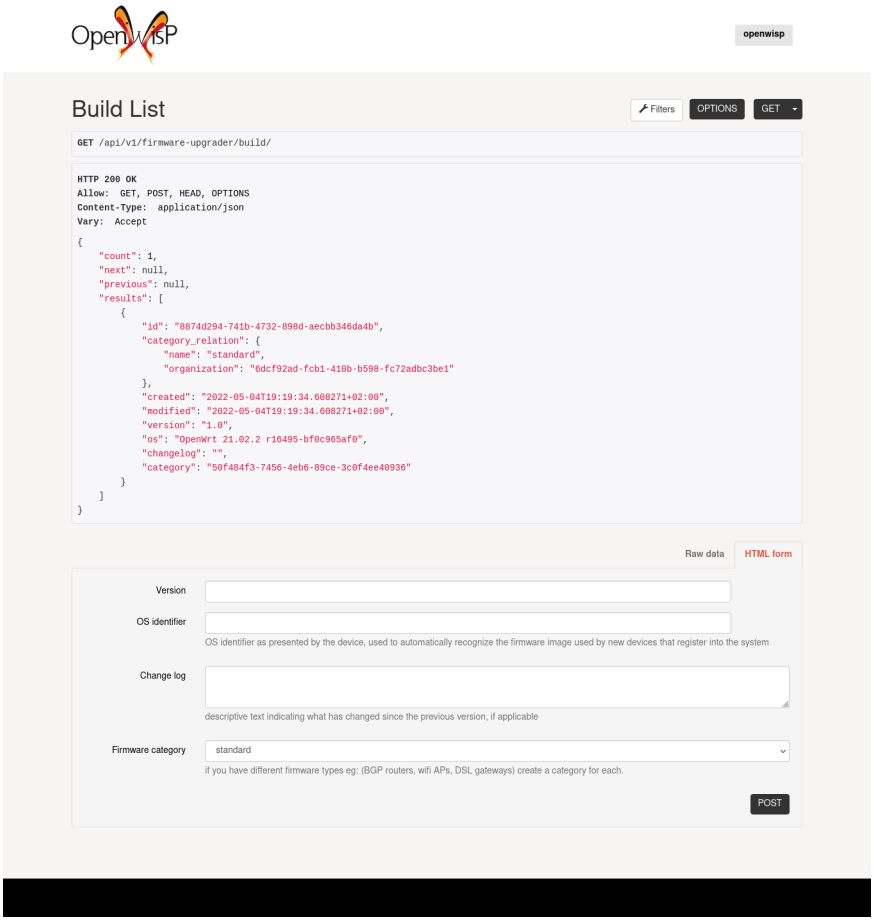
REST API Reference

Live Documentation



A general live API documentation (following the OpenAPI specification) at `/api/v1/docs/`.

Browsable Web Interface



Additionally, opening any of the endpoints listed below directly in the browser will show the browsable API interface of Django-REST-Framework, which makes it even easier to find out the details of each endpoint.

Authentication

See openwisp-users: authenticating with the user token.

When browsing the API via the Live Documentation or the Browsable Web Interface, you can also use the session authentication by logging in the django admin.

Pagination

All *list* endpoints support the `page_size` parameter that allows paginating the results in conjunction with the `page` parameter.

```
GET /api/v1/firmware-upgrader/build/?page_size=10
GET /api/v1/firmware-upgrader/build/?page_size=10&page=2
```

Filtering by Organization Slug

Most endpoints allow to filter by organization slug, e.g.:

```
GET /api/v1/firmware-upgrader/build/?organization=org-slug
```

List of Endpoints

Since the detailed explanation is contained in the Live Documentation and in the Browsable Web Interface of each point, here we'll provide just a list of the available endpoints, for further information please open the URL of the endpoint in your browser.

List Mass Upgrade Operations

```
GET /api/v1/firmware-upgrader/batch-upgrade-operation/
```

Available filters

The list of batch upgrade operations provides the following filters:

- `build` (Firmware build ID)
- `status` (One of: idle, in-progress, success, failed)

Here's a few examples:

```
GET /api/v1/firmware-upgrader/batch-upgrade-operation/?build={build_id}
GET /api/v1/firmware-upgrader/batch-upgrade-operation/?status={status}
```

Get Mass Upgrade Operation Detail

```
GET /api/v1/firmware-upgrader/batch-upgrade-operation/{id}/
```

List Firmware Builds

```
GET /api/v1/firmware-upgrader/build/
```

Available filters

The list of firmware builds provides the following filters:

- `category` (Firmware category ID)
- `version` (Firmware build version)
- `os` (Firmware build os identifier)

Here's a few examples:

```
GET /api/v1/firmware-upgrader/build/?category={category_id}
GET /api/v1/firmware-upgrader/build/?version={version}
GET /api/v1/firmware-upgrader/build/?os={os}
```

Create Firmware Build

```
POST /api/v1/firmware-upgrader/build/
```

Get Firmware Build Details

```
GET /api/v1/firmware-upgrader/build/{id}/
```

Change Details of Firmware Build

```
PUT /api/v1/firmware-upgrader/build/{id}/
```

Patch Details of Firmware Build

```
PATCH /api/v1/firmware-upgrader/build/{id}/
```

Delete Firmware Build

```
DELETE /api/v1/firmware-upgrader/build/{id}/
```

Get List of Images of a Firmware Build

```
GET /api/v1/firmware-upgrader/build/{id}/image/
```

Available filters

The list of images of a firmware build can be filtered by using `type` (any one of the available firmware image types).

```
GET /api/v1/firmware-upgrader/build/{id}/image/?type={type}
```

Upload New Firmware Image to the Build

```
POST /api/v1/firmware-upgrader/build/{id}/image/
```

Get Firmware Image Details

```
GET /api/v1/firmware-upgrader/build/{build_id}/image/{id}/
```

Delete Firmware Image

```
DELETE /api/v1/firmware-upgrader/build/{build_id}/image/{id}/
```

Download Firmware Image

```
GET /api/v1/firmware-upgrader/build/{build_id}/image/{id}/download/
```

Perform Batch Upgrade

Upgrades all the devices related to the specified build ID.

```
POST /api/v1/firmware-upgrader/build/{id}/upgrade/
```

Dry-run Batch Upgrade

Returns a list representing the `DeviceFirmware` and `Device` instances that would be upgraded if POST is used.

`Device` objects are indicated only when no `DeviceFirmware` object exists for a device which would be upgraded.

```
GET /api/v1/firmware-upgrader/build/{id}/upgrade/
```

List Firmware Categories

```
GET /api/v1/firmware-upgrader/category/
```

Create New Firmware Category

```
POST /api/v1/firmware-upgrader/category/
```

Get Firmware Category Details

```
GET /api/v1/firmware-upgrader/category/{id}/
```

Change the Details of a Firmware Category

```
PUT /api/v1/firmware-upgrader/category/{id}/
```

Patch the Details of a Firmware Category

```
PATCH /api/v1/firmware-upgrader/category/{id}/
```

Delete a Firmware Category

```
DELETE /api/v1/firmware-upgrader/category/{id}/
```

List Upgrade Operations

```
GET /api/v1/firmware-upgrader/upgrade-operation/
```

Available filters

The list of upgrade operations provides the following filters:

- `device__organization` (Organization ID of the device)
- `device__organization_slug` (Organization slug of the device)
- `device` (Device ID)
- `image` (Firmware image ID)
- `status` (One of: in-progress, success, failed, aborted)

Here's a few examples:

```
GET /api/v1/firmware-upgrader/upgrade-operation/?device__organization={organization_id}
GET /api/v1/firmware-upgrader/upgrade-operation/?device__organization__slug={organization_sl
```

```
GET /api/v1/firmware-upgrader/upgrade-operation/?device={device_id}
GET /api/v1/firmware-upgrader/upgrade-operation/?image={image_id}
GET /api/v1/firmware-upgrader/upgrade-operation/?status={status}
```

Get Upgrade Operation Details

```
GET /api/v1/firmware-upgrader/upgrade-operation/{id}
```

List Device Upgrade Operations

```
GET /api/v1/firmware-upgrader/device/{device_id}/upgrade-operation/
```

Available filters

The list of device upgrade operations can be filtered by `status` (one of: in-progress, success, failed, aborted).

```
GET /api/v1/firmware-upgrader/device/{device_id}/upgrade-operation/?status={status}
```

Create Device Firmware

Sending a PUT request to the endpoint below will create a new device firmware if it does not already exist.

```
PUT /api/v1/firmware-upgrader/device/{device_id}/firmware/
```

Get Device Firmware Details

```
GET /api/v1/firmware-upgrader/device/{device_id}/firmware/
```

Change Details of Device Firmware

```
PUT /api/v1/firmware-upgrader/device/{device_id}/firmware/
```

Patch Details of Device Firmware

```
PATCH /api/v1/firmware-upgrader/device/{device_id}/firmware/
```

Delete Device Firmware

```
DELETE /api/v1/firmware-upgrader/device/{device_pk}/firmware/
```

Settings

> **Note**
>
> If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

OPENWISP_FIRMWARE_UPGRADER_RETRY_OPTIONS

type:	dict
default:	see below

```
# default value of OPENWISP_FIRMWARE_UPGRADER_RETRY_OPTIONS:

dict(
    max_retries=4,
    retry_backoff=60,
    retry_backoff_max=600,
    retry_jitter=True,
)
```

Retry settings for recoverable failures during firmware upgrades.

By default if an upgrade operation fails before the firmware is flashed (e.g.: because of a network issue during the upload of the image), the upgrade operation will be retried 4 more times with an exponential random backoff and a maximum delay of 10 minutes.

For more information regarding these settings, consult the celery documentation regarding automatic retries for known errors.

OPENWISP_FIRMWARE_UPGRADER_TASK_TIMEOUT

type:	int
default:	600

Timeout for the background tasks which perform firmware upgrades.

If for some unexpected reason an upgrade remains stuck for more than 10 minutes, the upgrade operation will be flagged as failed and the task will be killed.

This should not happen, but a global task time out is a best practice when using background tasks because it prevents the situation in which an unexpected bug causes a specific task to hang, which will quickly fill all the available slots in a background queue and prevent other tasks from being executed, which will end up affecting negatively the rest of the application.

OPENWISP_CUSTOM_OPENWRT_IMAGES

type:	tuple
default:	None

This setting can be used to extend the list of firmware image types included in *OpenWISP Firmware Upgrader*. This setting is suited to add support for custom OpenWrt images.

```
OPENWISP_CUSTOM_OPENWRT_IMAGES = (
    (
        # Firmware image file name.
        "customimage-squashfs-sysupgrade.bin",
        {
            # Human readable name of the model which is displayed on
            # the UI
            "label": "Custom WAP-1200",
            # Tuple of board names with which the different versions of
            # the hardware are identified on OpenWrt
            "boards": ("CWAP1200",),
        },
```

```
    ),
)
```

Kindly read Automatic Device Firmware Detection section of this documentation to know how *OpenWISP Firmware Upgrader* uses this setting in upgrades.

OPENWISP_FIRMWARE_UPGRADER_MAX_FILE_SIZE

type:	`int`
default:	`30 * 1024 * 1024` (30 MB)

This setting can be used to set the maximum size limit for firmware images, e.g.:

```
OPENWISP_FIRMWARE_UPGRADER_MAX_FILE_SIZE = 40 * 1024 * 1024  # 40MB
```

Notes:

- Value must be specified in bytes. `None` means unlimited.

OPENWISP_FIRMWARE_UPGRADER_API

type:	`bool`
default:	`True`

Indicates whether the API for Firmware Upgrader is enabled or not.

OPENWISP_FIRMWARE_UPGRADER_OPENWRT_SETTINGS

type:	`dict`
default:	`{}`

Allows changing the default OpenWrt upgrader settings, e.g.:

```
OPENWISP_FIRMWARE_UPGRADER_OPENWRT_SETTINGS = {
    "reconnect_delay": 120,
    "reconnect_retry_delay": 20,
    "reconnect_max_retries": 15,
    "upgrade_timeout": 90,
}
```

- `reconnect_delay`: amount of seconds to wait before trying to connect again to the device after the upgrade command has been launched; the re-connection step is necessary to verify the upgrade has completed successfully; defaults to `120` seconds
- `reconnect_retry_delay`: amount of seconds to wait after a re-connection attempt has failed; defaults to `20` seconds
- `reconnect_max_retries`: maximum re-connection attempts defaults to `15` attempts
- `upgrade_timeout`: amount of seconds before the shell session is closed after the upgrade command is launched on the device, useful in case the upgrade command hangs (it happens on older OpenWrt versions); defaults to `90` seconds

OPENWISP_FIRMWARE_API_BASEURL

type:	`dict`

default:	/ (points to same server)

If you have a separate instance of OpenWISP Firmware Upgrader API on a different domain, you can use this option to change the base of the image download URL, this will enable you to point to your API server's domain, e.g.: `https://api.myservice.com`.

`OPENWISP_FIRMWARE_UPGRADERS_MAP`

type:	`dict`
default:	`{ "openwisp_controller.connection.connectors.openwrt.ssh.OpenWrt": "openwisp_firmw }`

A dictionary that maps update strategies to upgraders.

If you want to use a custom update strategy you will need to use this setting to provide an entry with the class path of your update strategy as the key.

If you need to use a custom upgrader class you will need to use this setting to provide an entry with the class path of your upgrader as the value.

`OPENWISP_FIRMWARE_PRIVATE_STORAGE_INSTANCE`

type:	`str`
default:	`openwisp_firmware_upgrader.private_storage.storage.file_system_private_storage`

Dotted path to an instance of any one of the storage classes in private_storage. This instance is used to store firmware image files.

By default, an instance of `private_storage.storage.files.PrivateFileSystemStorage` is used.

Developer Docs

Note

This page is for developers who want to customize or extend OpenWISP Firmware Upgrader, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- Firmware Upgrader User Docs

Developer Installation Instructions

Note

This page is for developers who want to customize or extend OpenWISP Firmware Upgrader, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- Firmware Upgrader User Docs

Requirements

- Python >= 3.8
- OpenWISP Controller (and its dependencies) >= 1.0.0

Install Dependencies

Install spatialite and sqlite:

```
sudo apt-get install sqlite3 libsqlite3-dev openssl libssl-dev
sudo apt-get install gdal-bin libproj-dev libgeos-dev libspatialite-dev
```

Installing for Development

Fork and clone the forked repository:

```
git clone git://github.com/<your_fork>/openwisp-firmware-upgrader
```

Navigate into the cloned repository:

```
cd openwisp-firmware-upgrader/
```

Launch Redis:

```
docker-compose up -d redis
```

Setup and activate a virtual-environment (we'll be using virtualenv):

```
python -m virtualenv env
source env/bin/activate
```

Make sure that your base python packages are up to date before moving to the next step:

```
pip install -U pip wheel setuptools
```

Install development dependencies:

```
pip install -e .
pip install -r requirements-test.txt
sudo npm install -g jshint stylelint
```

Install WebDriver for Chromium for your browser version from https://chromedriver.chromium.org/home and Extract `chromedriver` to one of directories from your `$PATH` (example: `~/.local/bin/`).

Create database:

```
cd tests/
./manage.py migrate
./manage.py createsuperuser
```

Launch development server:

```
./manage.py runserver 0.0.0.0:8000
```

You can access the admin interface at `http://127.0.0.1:8000/admin/`.

Run celery and celery-beat with the following commands (separate terminal windows are needed):

```
# (cd tests)
celery -A openwisp2 worker -l info
celery -A openwisp2 beat -l info
```

Run tests with:

```
# run qa checks
./run-qa-checks

# standard tests
./runtests.py

# tests for the sample app
SAMPLE_APP=1 ./runtests.py --keepdb --failfast
```

When running the last line of the previous example, the environment variable `SAMPLE_APP` activates the app in `/tests/openwisp2/sample_firmware_upgrader/` which is a simple django app that extends `openwisp-firmware-upgrader` with the sole purpose of testing its extensibility, for more information regarding this concept, read Extending OpenWISP Firmware Upgrader.

Important

If you want to add `openwisp-firmware-upgrader` to an existing Django project, then you can take reference from the test project in openwisp-firmware-upgrader repository

Extending OpenWISP Firmware Upgrader

Note

This page is for developers who want to customize or extend OpenWISP Firmware Upgrader, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- Firmware Upgrader User Docs

One of the core values of the OpenWISP project is Software Reusability, for this reason *OpenWISP Firmware Upgrader* provides a set of base classes which can be imported, extended and reused to create derivative apps.

In order to implement your custom version of *OpenWISP Firmware Upgrader*, you need to perform the steps described in this section.

When in doubt, the code in the test project and the sample app will serve you as source of truth: just replicate and adapt that code to get a basic derivative of *OpenWISP Firmware Upgrader* working.

Important

If you plan on using a customized version of this module, we suggest to start with it since the beginning, because migrating your data from the default module to your extended version may be time consuming.

1. Initialize your Custom Module

The first thing you need to do is to create a new django app which will contain your custom version of *OpenWISP Firmware Upgrader*.

A django app is nothing more than a python package (a directory of python scripts), in the following examples we'll call this django app `myupgrader`, but you can name it how you want:

```
django-admin startapp myupgrader
```

Keep in mind that the command mentioned above must be called from a directory which is available in your PYTHON_PATH so that you can then import the result into your project.

Now you need to add `myupgrader` to `INSTALLED_APPS` in your `settings.py`, ensuring also that `openwisp_firmware_upgrader` has been removed:

```
INSTALLED_APPS = [
    # ... other apps ...
    # 'openwisp_firmware_upgrader'  <-- comment out or delete this line
    "myupgrader"
]
```

For more information about how to work with django projects and django apps, please refer to the django documentation.

2. Install `openwisp-firmware-upgrader`

Install (and add to the requirement of your project) `openwisp-firmware-upgrader`:

```
pip install openwisp-firmware-upgrader
```

3. Add `EXTENDED_APPS`

Add the following to your `settings.py`:

```
EXTENDED_APPS = ["openwisp_firmware_upgrader"]
```

4. Add `openwisp_utils.staticfiles.DependencyFinder`

Add `openwisp_utils.staticfiles.DependencyFinder` to `STATICFILES_FINDERS` in your `settings.py`:

```
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
    "openwisp_utils.staticfiles.DependencyFinder",
]
```

5. Add `openwisp_utils.loaders.DependencyLoader`

Add `openwisp_utils.loaders.DependencyLoader` to `TEMPLATES` in your `settings.py`:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "OPTIONS": {
            "loaders": [
                "django.template.loaders.filesystem.Loader",
                "django.template.loaders.app_directories.Loader",
                "openwisp_utils.loaders.DependencyLoader",
            ],
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    }
]
```

6. Inherit the AppConfig Class

Please refer to the following files in the sample app of the test project:

- sample_firmware_upgrader/__init__.py.
- sample_firmware_upgrader/apps.py.

You have to replicate and adapt that code in your project.

For more information regarding the concept of `AppConfig` please refer to the "Applications" section in the django documentation.

7. Create your Custom Models

For the purpose of showing an example, we added a simple "details" field to the models of the sample app in the test project.

You can add fields in a similar way in your `models.py` file.

Note

If you have questions about using, extending, or developing models, refer to the "Models" section of the Django documentation.

8. Add Swapper Configurations

Once you have created the models, add the following to your `settings.py`:

```
# Setting models for swapper module
FIRMWARE_UPGRADER_CATEGORY_MODEL = "myupgrader.Category"
FIRMWARE_UPGRADER_BUILD_MODEL = "myupgrader.Build"
FIRMWARE_UPGRADER_FIRMWAREIMAGE_MODEL = "myupgrader.FirmwareImage"
FIRMWARE_UPGRADER_DEVICEFIRMWARE_MODEL = "myupgrader.DeviceFirmware"
FIRMWARE_UPGRADER_BATCHUPGRADEOPERATION_MODEL = (
    "myupgrader.BatchUpgradeOperation"
)
FIRMWARE_UPGRADER_UPGRADEOPERATION_MODEL = "myupgrader.UpgradeOperation"
```

Substitute `myupgrader` with the name you chose in step 1.

9. Create Database Migrations

Create and apply database migrations:

```
./manage.py makemigrations
./manage.py migrate
```

For more information, refer to the "Migrations" section in the django documentation.

10. Create the Admin

Refer to the admin.py file of the sample app.

To introduce changes to the admin, you can do it in two main ways which are described below.

For more information regarding how the django admin works, or how it can be customized, please refer to "The django admin site" section in the django documentation.

1. Monkey Patching

If the changes you need to add are relatively small, you can resort to monkey patching.

For example:

```
from openwisp_firmware_upgrader.admin import (
    BatchUpgradeOperationAdmin,
    BuildAdmin,
    CategoryAdmin,
)

BuildAdmin.list_display.insert(1, "my_custom_field")
BuildAdmin.ordering = ["-my_custom_field"]
```

2. Inheriting Admin Classes

If you need to introduce significant changes and/or you don't want to resort to monkey patching, you can proceed as follows:

```
from django.contrib import admin
from openwisp_firmware_upgrader.admin import (
    BatchUpgradeOperationAdmin as BaseBatchUpgradeOperationAdmin,
    BuildAdmin as BaseBuildAdmin,
    CategoryAdmin as BaseCategoryAdmin,
)
from openwisp_firmware_upgrader.swapper import load_model

BatchUpgradeOperation = load_model("BatchUpgradeOperation")
Build = load_model("Build")
Category = load_model("Category")
DeviceFirmware = load_model("DeviceFirmware")
FirmwareImage = load_model("FirmwareImage")
UpgradeOperation = load_model("UpgradeOperation")

admin.site.unregister(BatchUpgradeOperation)
admin.site.unregister(Build)
admin.site.unregister(Category)


class BatchUpgradeOperationAdmin(BaseBatchUpgradeOperationAdmin):
    # add your changes here
    pass


class BuildAdmin(BaseBuildAdmin):
    # add your changes here
    pass


class CategoryAdmin(BaseCategoryAdmin):
    # add your changes here
    pass
```

11. Create Root URL Configuration

Please refer to the urls.py file in the test project.

For more information about URL configuration in django, please refer to the "URL dispatcher" section in the django documentation.

12. Create `celery.py`

Please refer to the celery.py file in the test project.

For more information about the usage of celery in django, please refer to the "First steps with Django" section in the celery documentation.

13. Import the Automated Tests

When developing a custom application based on this module, it's a good idea to import and run the base tests too, so that you can be sure the changes you're introducing are not breaking some of the existing features of *OpenWISP Firmware Upgrader*.

In case you need to add breaking changes, you can overwrite the tests defined in the base classes to test your own behavior.

See the tests of the sample app to find out how to do this.

You can then run tests with:

```
# the --parallel flag is optional
./manage.py test --parallel myupgrader
```

Substitute `myupgrader` with the name you chose in step 1.

For more information about automated tests in django, please refer to "Testing in Django".

Other Base Classes That Can be Inherited and Extended

The following steps are not required and are intended for more advanced customization.

`FirmwareImageDownloadView`

This view controls how the firmware images are stored and who has permission to download them.

The full python path is: `openwisp_firmware_upgrader.private_storage.FirmwareImageDownloadView`.

If you want to extend this view, you will have to perform the additional steps below.

Step 1. import and extend view:

```
# myupgrader/views.py
from openwisp_firmware_upgrader.private_storage import (
    FirmwareImageDownloadView as BaseFirmwareImageDownloadView,
)


class FirmwareImageDownloadView(BaseFirmwareImageDownloadView):
    # add your customizations here ...
    pass
```

Step 2: remove the following line from your root `urls.py` file:

```
path(
    "firmware/",
    include("openwisp_firmware_upgrader.private_storage.urls"),
),
```

Step 3: add an URL route pointing to your custom view in `urls.py` file:

```
# urls.py
from myupgrader.views import FirmwareImageDownloadView

urlpatterns = [
    # ... other URLs
    path(
        "<your-custom-path>",
        FirmwareImageDownloadView.as_view(),
        name="serve_private_file",
    ),
]
```

For more information regarding django views, please refer to the "Class based views" section in the django documentation.

API Views

If you need to customize the behavior of the API views, the procedure to follow is similar to the one described in FirmwareImageDownloadView, with the difference that you may also want to create your own serializers if needed.

The API code is stored in openwisp_firmware_upgrader.api and is built using django-rest-framework

For more information regarding Django REST Framework API views, please refer to the "Generic views" section in the Django REST Framework documentation.

Other useful resources:

- REST API Reference
- Settings

RADIUS

Seealso

Source code: github.com/openwisp/openwisp-radius.

OpenWISP RADIUS is available since OpenWISP 22.05 and provides many features aimed at public WiFi services.

For a full introduction please refer to RADIUS: Features.

The following diagram illustrates the role of the RADIUS module within the OpenWISP architecture.

OpenWISP Architecture: highlighted radius module

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

RADIUS: Features

The RADIUS module provides the following features:

- Registration of new users
- SMS verification
- Importing users
- Generating users
- Social Login
- Single Sign-On (SAML)
- Enforcing Session Limits
- Change of Authorization (CoA)
- REST API Reference

Registration of new users

openwisp-radius uses django-rest-auth which provides registration of new users via REST API so you can implement registration and password reset directly from your captive page.

The registration API endpoint is described in API: User Registration.

If you need users to self-register to a public wifi service, we suggest to take a look at OpenWISP WiFi Login Pages, which is built to work with openwisp-radius.

Generating users

Many a times, a network admin might need to generate temporary users (e.g.: events).

This feature can be used for generating users by specifying a prefix and the number of users to be generated.

There are many features included in it such as:

- **Generating users in batches**: all of the users of a particular **prefix** would be stored in batches and can be retrieved/deleted easily using the batch functions.
- **Download user credentials in PDF format**: get the usernames and passwords generated outputted into a PDF.
- **Set an expiration date**: an expiration date can be set for a batch after which the users would not able to authenticate to the RADIUS Server.

This operation can be performed via the admin interface, with a management command or via the REST API.

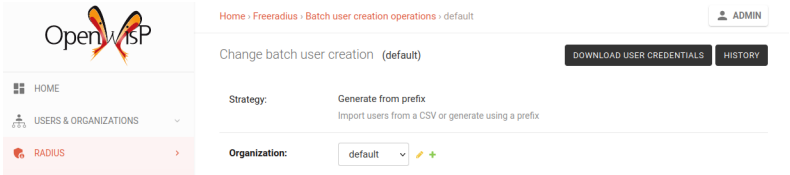
Note

Users imported or generated through this form will be flagged as verified if the organization requires identity verification, otherwise the generated users would not be able to log in. If this organization requires identity verification, make sure the identity of the users is verified before giving out the credentials.

Using the admin interface

To generate users from the admin interface, go to `Home > Batch user creation operations > Add` (URL: `/admin/openwisp_radius/radiusbatch/add`), set `Strategy` to `Generate from prefix`, fill in the remaining fields that are shown after the selection of the strategy and save.

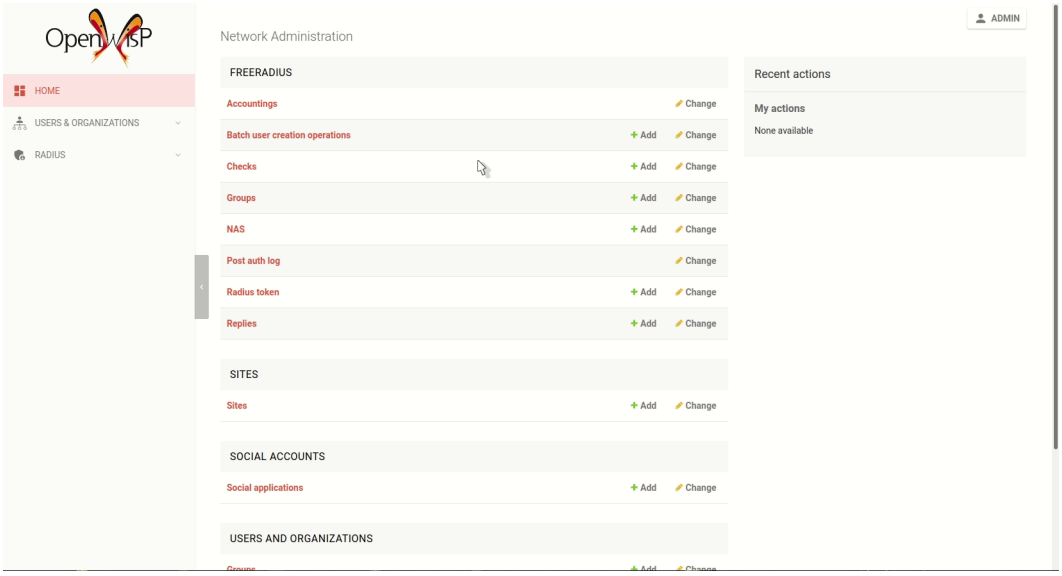
Once the batch object has been created, a PDF containing the user credentials can be downloaded by using the "Download user credentials" button in the upper right corner of the page:



The contents of the PDF is in format of a table of users & their passwords:

Username	Password
sample1	ygJDRWt1
sample2	Ar1l2tZY

Usage Demonstration:



Management command: `prefix_add_users`

This command generates users whose usernames start with a particular prefix. Usage is as shown below.

```
./manage.py prefix_add_users --name <name_of_batch> \
                             --organization=<organization-slug> \
                             --prefix <prefix> \
                             --n <number_of_users> \
                             --expiration <expiration_date> \
                             --password-length <password_length> \
                             --output <path_to_pdf_file>
```

Note

The expiration, password-length and output are optional parameters. The options expiration and password-length default to never and 8 respectively. If output parameter is not provided, PDF file is not created on the server and can be accessed from the admin interface.

REST API: Batch user creation

See API documentation: Batch user creation.

Importing users

This feature can be used for importing users from a csv file. There are many features included in it such as:

- Importing users in batches: all of the users of a particular csv file would be stored in batches and can be retrieved/ deleted easily using the batch functions.
- Set an expiration date: Expiration date can be set for a batch after which the users would not able to authenticate to the RADIUS Server.
- Auto-generate usernames and passwords: The usernames and passwords are automatically generated if they aren't provided in the csv file. Usernames are generated from the email address whereas passwords are generated randomly and their lengths can be customized.
- Passwords are accepted in both clear-text and hash formats from the CSV.
- Send mails to users whose passwords have been generated automatically.

This operation can be performed via the admin interface, with a management command or via the REST API.

CSV Format

The CSV shall be of the format:

```
username,password,email,firstname,lastname
```

Imported users with hashed passwords

The hashes are directly stored in the database if they are of the django hash format.

For example, a password `myPassword123`, hashed using salted SHA1 algorithm, will look like:

```
pbkdf2_sha256$100000$cKdP39chT3pW$2EtVk4Hhm1V65GNfYAA5AHj0uyD60f2CmqumqiB/gRk=
```

So a full CSV line containing that password would be:

```
username,pbkdf2_sha256$100000$cKdP39chT3pW$2EtVk4Hhm1V65GNfYAA5AHj0uyD60f2CmqumqiB/gRk=,emai
```

Importing users with clear-text passwords

Clear-text passwords must be flagged with the prefix `cleartext$`.

For example, if we want to use the password `qwerty`, we must use: `cleartext$qwerty`.

Auto-generation of usernames and passwords

Email is the only mandatory field of the CSV file.

Other fields like username and password will be auto-generated if omitted.

Emails will be sent to users whose usernames or passwords have been auto-generated and contents of these emails can be customized too.

Here are some defined settings for doing that:

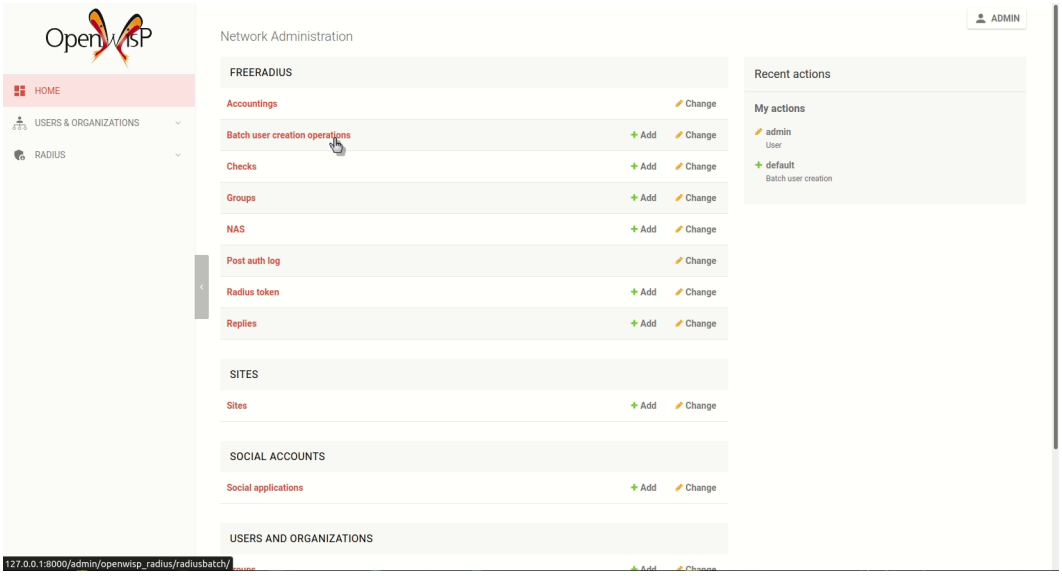
- OPENWISP_RADIUS_BATCH_MAIL_SUBJECT
- OPENWISP_RADIUS_BATCH_MAIL_MESSAGE
- OPENWISP_RADIUS_BATCH_MAIL_SENDER

Using the admin interface

Note

The CSV uploaded must follow the CSV format described above.

To generate users from the admin interface, go to `Home > Batch user creation operations > Add` (URL: `/admin/openwisp_radius/radiusbatch/add`), set `Strategy` to `Import from CSV`, choose the CSV file to upload and save.



Management command: `batch_add_users`

This command imports users from a csv file. Usage is as shown below.

```
./manage.py batch_add_users --name <name_of_batch> \
                            --organization=<organization-slug> \
                            --file <filepath> \
                            --expiration <expiration_date> \
                            --password-length <password_length>
```

Note

The expiration and password-length are optional parameters which default to never and 8 respectively.

REST API: Batch user creation

See API documentation: Batch user creation.

Social Login

Important

The social login feature is disabled by default.

In order to enable this feature you have to follow the setup instructions below and then activate it via global setting or from the admin interface.

Social login is supported by generating an additional temporary token right after users perform the social sign-in, the user is then redirected to the captive page with two querystring parameters: `username` and `token`.

The captive page must recognize these two parameters and automatically perform the submit action of the login form: `username` should obviously used for the username field, while `token` should be used for the password field.

The internal REST API of OpenWISP RADIUS will recognize the token and authorize the user.

This kind of implementation allows to implement the social login with any captive portal which already supports the RADIUS protocol because it's totally transparent for it, that is, the captive portal doesn't even know the user is signing-in with a social network.

Note

If you're building a public wifi service, we suggest to take a look at OpenWISP WiFi Login Pages, which is built to work with openwisp-radius.

Setup

Ensure the your project `settings.py` contains the instructions shown in the example below, which shows how to configure the Facebook social login provider.

Note

If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

```
INSTALLED_APPS = [
    # ... other apps ..
    # apps needed for social login
    "rest_framework.authtoken",
    "django.contrib.sites",
    "allauth",
    "allauth.account",
    "allauth.socialaccount",
    # showing facebook as an example
    # to configure social login with other social networks
    # refer to the django-allauth documentation
    "allauth.socialaccount.providers.facebook",
]

SITE_ID = 1

# showing facebook as an example
# to configure social login with other social networks
# refer to the django-allauth documentation
SOCIALACCOUNT_PROVIDERS = {
    "facebook": {
        "METHOD": "oauth2",
        "SCOPE": ["email", "public_profile"],
        "AUTH_PARAMS": {"auth_type": "reauthenticate"},
        "INIT_PARAMS": {"cookie": True},
        "FIELDS": [
            "id",
            "email",
            "name",
            "first_name",
            "last_name",
            "verified",
        ],
        "VERIFIED_EMAIL": True,
```

```
    }
}
```

Ensure your main `urls.py` contains the `allauth.urls`:

```
urlpatterns = [
    # .. other urls ...
    path("accounts/", include("allauth.urls")),
]
```

Configure the social account application

Refer to the django-allauth documentation to find out how to complete the configuration of a sample Facebook login app.

Captive page button example

Following the previous example configuration with Facebook, in your captive page you will need an HTML button similar to the ones in the following examples.

This example needs the slug of the organization to assign the new user to the right organization:

```
<a href="https://openwisp2.mywifiproject.com/accounts/facebook/login/?next=%2Fradius%2Fsocia
   class="button">Log in with Facebook
</a>
```

Substitute `openwisp2.mywifiproject.com`, `captivepage.mywifiproject.com` and `default` with the hostname of your openwisp-radius instance, your captive page and the organization slug respectively.

Alternatively, you can take a look at OpenWISP WiFi Login Pages, which provides buttons for Facebook, Google and Twitter by default.

Settings

See social login related settings.

Single Sign-On (SAML)

Important

The SAML registration method is disabled by default.

In order to enable this feature you have to follow the SAML setup instructions below and then activate it via global setting or from the admin interface.

SAML is supported by generating an additional temporary token right after users authenticates via SSO, the user is then redirected to the captive page with 3 querystring parameters:

- `username`
- `token` (REST auth token)
- `login_method=saml`

The captive page must recognize these two parameters, validate the token and automatically perform the submit action of the captive portal login form: `username` should obviously used for the username field, while `token` should be used for the password field.

The third parameter, `login_method=saml`, is needed because it allows the captive page to remember that the user logged in via SAML. This information will be used later on when performing the SAML logout.

The internal REST API of openwisp-radius will recognize the token and authorize the user.

This kind of implementation allows to support SAML with any captive portal which already supports the RADIUS protocol because it's totally transparent for it, that is, the captive portal doesn't even know the user is signing-in with a SSO.

Note

If you're building a public wifi service, we suggest to take a look at OpenWISP WiFi Login Pages, which is built to work with openwisp-radius.

Setup

Install required system dependencies:

```
sudo apt install xmlsec1
```

Install the SAML dependencies in the python environment used by OpenWISP:

```
# by default, in instances deployed
# via ansible-openwisp2, the python env
# is in /opt/openwisp2/env/
source /opt/openwisp2/env/bin/activate

pip install openwisp-radius[saml]
```

Note

If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

Ensure your `settings.py` looks like the following:

```
INSTALLED_APPS = [
    # ... other apps ..
    # apps needed for SAML login
    "rest_framework.authtoken",
    "django.contrib.sites",
    "allauth",
    "allauth.account",
    "djangosaml2",
]

SITE_ID = 1

# Update AUTHENTICATION_BACKENDS
AUTHENTICATION_BACKENDS = (
    "openwisp_users.backends.UsersAuthenticationBackend",
    "openwisp_radius.saml.backends.OpenwispRadiusSaml2Backend",  # <- add for SAML login
)

# Update MIDDLEWARE
MIDDLEWARE = [
```

```
    # ... other middlewares ...
    "djangosaml2.middleware.SamlSessionMiddleware",
]
```

Ensure your main `urls.py` contains the `openwisp_users.accounts.urls`:

```
urlpatterns = [
    # .. other urls ...
    path("accounts/", include("openwisp_users.accounts.urls")),
]
```

Configure the djangosaml2 settings

Refer to the djangosaml2 documentation to find out how to configure required settings for SAML.

Captive page button example

After successfully configuring SAML settings for your Identity Provider, you will need an HTML button similar to the one in the following example.

This example needs the slug of the organization to assign the new user to the right organization:

```
<a href="https://openwisp2.mywifiproject.com/radius/saml2/login/?RelayState=https://captivep
   class="button">
   Log in with SSO
</a>
```

Substitute `openwisp2.mywifiproject.com`, `https://captivepage.mywifiproject.com` and `default` with the hostname of your openwisp-radius instance, your captive page and the organization slug respectively.

Alternatively, you can take a look at OpenWISP WiFi Login Pages, which provides buttons for Single Sign-On (SAML) by default.

Logout

When logging out a user which logged in via SAML, the captive page should also call the SAML logout URL: `/radius/saml2/logout/`.

The OpenWISP WiFi Login Pages app supports this with minimal configuration, refer to the OpenWISP WiFi Login Pages section.

Settings

See SAML related settings.

FAQs

Preventing change in username of a registered user

The `djangosaml2` library requires configuring `SAML_DJANGO_USER_MAIN_ATTRIBUTE` setting which serves as the primary lookup value for User objects. Whenever a user logs in or registers through the SAML method, a database query is made to check whether such a user already exists. This lookup is done using the value of `SAML_DJANGO_USER_MAIN_ATTRIBUTE` setting. If a match is found, the details of the user are updated with the information received from SAML Identity Provider.

If a user (who has registered on OpenWISP with a different method from SAML) logs into OpenWISP with SAML, then the default behavior of OpenWISP RADIUS prevents updating username of this user. Because, this operation could render the user's old credentials useless. If you want to update the username in such scenarios with details

received from Identity Provider, set OPENWISP_RADIUS_SAML_UPDATES_PRE_EXISTING_USERNAME to `True`.

Enforcing Session Limits

The default freeradius schema does not include a table where groups are stored, but openwisp-radius adds a model called `RadiusGroup` and alters the default freeradius schema to add some optional foreign-keys from other tables like:

- `radgroupcheck`
- `radgroupreply`
- `radusergroup`

These foreign keys make it easier to automate many synchronization and integrity checks between the `RadiusGroup` table and its related tables but they are not strictly mandatory from the database point of view: their value can be `NULL` and their presence and validation is handled at application level, this makes it easy to use existing freeradius databases.

For each group, checks and replies can be specified directly in the edit page of a Radius Group (`admin` > `groups` > `add group` or `change group`).

Default Groups

Some groups are created automatically by **openwisp-radius** during the initial migrations:

- `users`: this is the default group which limits users sessions to 3 hours and 300 MB (daily)
- `power-users`: this group does not have any check, therefore users who are members of this group won't be limited in any way

You can customize the checks and the replies of these groups, as well as create new groups according to your needs and preferences.

Note on the default group: keep in mind that the group flagged as default will by automatically assigned to new users, it cannot be deleted nor it can be flagged as non-default: to set another group as default simply check that group as the default one, save and **openwisp-radius** will remove the default flag from the old default group.

How Limits are Enforced: Counters

In Freeradius, this kind of feature is implemented with the rlm_sqlcounter.

The problem with this FreeRADIUS module is that it doesn't know about OpenWISP, so it does not support multi-tenancy. This means that if multiple organizations are using the OpenWISP instance, it's possible that a user may be an end user of multiple organizations and hence have one radius group assigned for each, but the *sqlcounter* module will not understand the right group to choose when enforcing limits, with the result that the enforcing of limits will not work as expected, unless one FreeRADIUS site with different *sqlcounter* configurations is created for each organization using the system, which is doable but cumbersome to maintain.

For the reasons explained above, an alternative counter feature has been implemented in the authorize API endpoint of OpenWISP RADIUS.

The default counters available are described below.

`DailyCounter`

This counter is used to limit the amount of time users can use the network every day. It works by checking whether the total session time of a user during a specific day is below the value indicated in the `Max-Daily-Session` group check attribute, sending the remaining session time with a `Session-Timeout` reply message or rejecting the authorization if the limit has been passed.

`DailyTrafficCounter`

This counter is used to limit the amount of traffic users can consume every day. It works by checking whether the total amount of download plus upload octets (bytes consumed) is below the value indicated in the `Max-Daily-Session-Traffic` group check attribute, sending the remaining octets with a reply message or rejecting the authorization if the limit has been passed.

The attributes used for the check and or the reply message are configurable because it can differ from NAS to NAS, see OPENWISP_RADIUS_TRAFFIC_COUNTER_CHECK_NAME OPENWISP_RADIUS_TRAFFIC_COUNTER_REPLY_NAME for more information.

`MonthlyTrafficCounter`

This counter is used to limit the amount of traffic users can consume every solar month. It works by checking whether the total amount of download plus upload octets (bytes consumed) is below the value indicated in the `Max-Monthly-Session-Traffic` group check attribute, sending the remaining octets with a reply message or rejecting the authorization if the limit has been passed.

The reply message is configurable because it can differ from NAS to NAS, OPENWISP_RADIUS_TRAFFIC_COUNTER_REPLY_NAME for more information.

`MonthlySubscriptionTrafficCounter`

Important

This counter is not enabled by default. It can be enabled via the Counter related settings.

Same as `MonthlyTrafficCounter`, but with the difference that the reset period depends on the day in which the user subscribed to the service: if the user signed up (or their account was created by an admin) on a date like November 15 2022, the reset period will start on the *15th day* of every month.

Database Support

The counters described above are available for PostgreSQL, MySQL, SQLite and are enabled by default.

There's a different class of each counter for each database, because the query is executed with raw SQL defined on each class, instead of the classic django-ORM approach which is database agnostic.

It was implemented this way to ensure maximum flexibility and adherence to the FreeRADIUS *sqlcounter* implementation.

Django Settings

The settings available to control the behavior of counters are described in Counter related settings.

Writing Custom Counter Classes

It is possible to write custom counter classes to satisfy any need.

The easiest way is to subclass `openwisp_radius.counters.base.BaseCounter`, then implement at least the following attributes:

- `counter_name`: name of the counter, used internally for debugging;
- `check_name`: attribute name used in the database lookup to the group check table;
- `reply_name`: attribute name sent in the reply message;

- `reset`: reset period, either `daily`, `weekly`, `monthly`, `monthly_subscription` or `never`;
- `sql`: the raw SQL query to execute;
- `get_sql_params`: a method which returns a list of the arguments passed to the interpolation of the raw SQL query.

Please look at the source code of OpenWISP RADIUS to find out more.

- openwisp_radius.counters.base
- openwisp_radius.counters.postgresql

Once the new class is ready, you will need to add it to OPENWISP_RADIUS_COUNTERS.

It is also possible to implement a check class in a completely custom fashion (that is, not inheriting from `BaseCounter`), the only requirements are:

- the class must have a constructor (`__init__` method) identical to the one used in the `BaseCounter` class;
- the class must have a `check` method which doesn't need any required argument and returns the remaining counter value or raises `MaxQuotaReached` if the limit has been reached and the authorization should be rejected; This method may return `None` if no additional RADIUS attribute needs to be added to the response.

Change of Authorization (CoA)

Important

The *Change of Authorization (CoA)* is disabled by default.

In order to enable this feature you have it enable it via global setting or from the admin interface.

The openwisp-radius module supports the Change of Authorization (CoA) specification of the RADIUS protocol described in RFC 5176.

Whenever the *RADIUS Group* of a user is changed, openwisp-radius updates the NAS with the user's latest RADIUS Attributes. This is achieved by sending CoA RADIUS packet to NAS for all open RADIUS sessions of the user. This allows enforcing RADIUS limits without requiring the user to re-authenticate with the NAS.

The CoA RADIUS packet contains the RADIUS Attributes defined in the new *RADIUS Group* of the user. If the new *RADIUS Group* does not specify any attributes, the CoA RADIUS packet will unset the attributes set by the previous *RADIUS Group*.

Consider the following example with two *RADIUS Groups*:

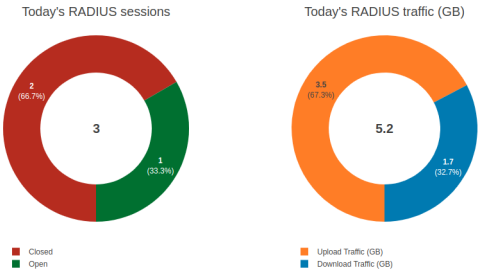
<table>
<tr><th>RADIUS Group Name</th><th>RADIUS Group Checks</th></tr>
<tr><td>users</td><td>
<table>
<tr><th>Attribute</th><th>Value</th></tr>
<tr><td><code>Max-Daily-Session-Traffic</code></td><td><code>:=3000000000</code></td></tr>
<tr><td><code>Max-Daily-Session</code></td><td><code>:=10800</code></td></tr>
</table>
</td></tr>
<tr><td>power-users</td><td>Note: This group intentionally does not define any limits.</td></tr>
</table>

A user, Jane is assigned `users` *RADIUS Group* and is currently using the network, i.e. has an open RADIUS session. The administrator of the system decided to upgrade the *RADIUS Group* of Jane to `power-users`, allowing Jane to use the network without any limits. Without CoA, Jane will have to logout of the captive portal (NAS) and log-in again to browse the network without any limits. But when CoA is enabled in openwisp-radius, openwisp-radius will update the NAS with the limits defined in Jane's new RADIUS Group. In this case, openwisp-radius will tell the NAS to unset the limits that were configured by the previous RADIUS Group.

If the system administrators later decided to downgrade the *RADIUS Group* of Jane to `users`, hence enforcing limits to the usage of the network, openwisp-radius will update the NAS with the limits defined for the `users` group for all active RADIUS sessions if CoA is enabled in openwisp-radius.

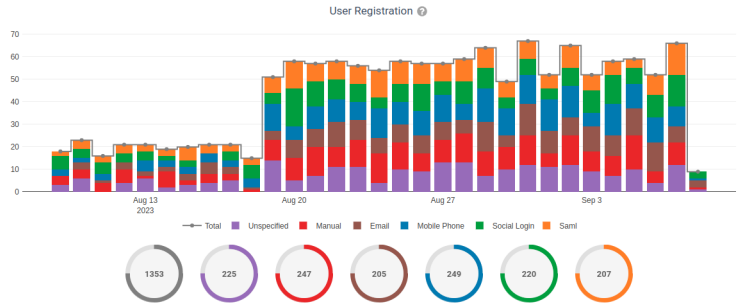
Integration with OpenWISP Monitoring

OpenWISP RADIUS includes an optional Django sub-app that adds integration with OpenWISP Monitoring to provide RADIUS metrics.



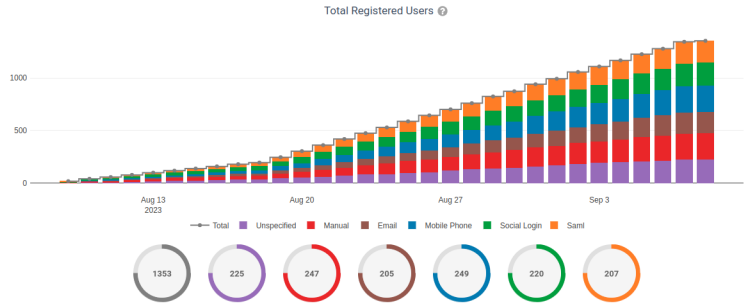
RADIUS metrics

1. User registrations



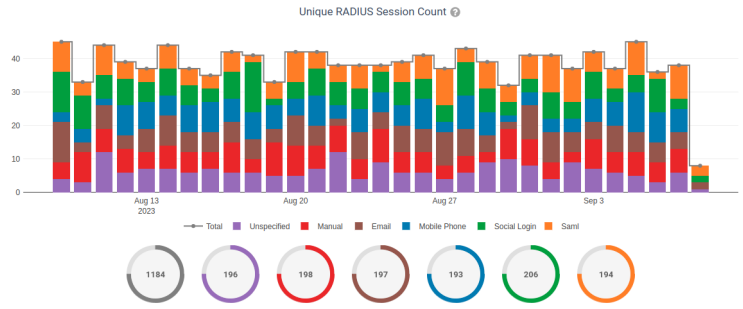
This chart shows number of users signed up using different registration methods.

2. Total user registrations



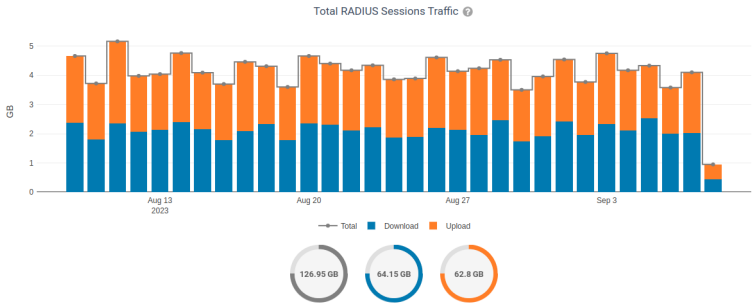
This chart shows total users registered using different registration methods in the system on a given date.

3. Unique RADIUS Sessions



This chart shows unique RADIUS sessions. It is helpful to know how many unique users has used the system in a given time.

4. RADIUS traffic



This chart shows the RADIUS traffic generated by user sessions.

Enabling RADIUS metrics in Django project

> **Note**
>
> If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

You can enable the monitoring integration by including `openwisp_radius.integrations.monitoring` in `INSTALLED_APPS` of your Django project's settings as following:

```
# In your_project/settings.py

INSTALLED_APPS.append("openwisp_radius.integrations.monitoring")
CELERY_BEAT_SCHEDULE.update(
    {
        "write_user_registration_metrics": {
            "task": "openwisp_radius.integrations.monitoring.tasks.write_user_registration_m
            "schedule": timedelta(hours=1),
            "args": None,
            "relative": True,
        }
    }
)
```

> **Note**
>
> Ensure your Django project is correctly configured to utilize OpenWISP Monitoring. For production environments, it is advisable to deploy OpenWISP using Ansible OpenWISP or Docker OpenWISP, as they simplify the deployment process considerably.

Important

If you are registering a "registration method" in any other Django application, then `openwisp_radius.integrations.monitoring` should come after that app in the `INSTALLED_APPS`. Otherwise, the registration method would not appear in the chart.

Management commands

These management commands are necessary for enabling certain features and for database cleanup.

Example usage:

```
cd tests/
./manage.py <command> <args>
```

In this page we list the management commands currently available in **openwisp-radius**.

`delete_old_radacct`

This command deletes RADIUS accounting sessions older than `<days>`.

```
./manage.py delete_old_radacct <days>
```

For example:

```
./manage.py delete_old_radacct 365
```

`delete_old_postauth`

This command deletes RADIUS post-auth logs older than `<days>`.

```
./manage.py delete_old_postauth <days>
```

For example:

```
./manage.py delete_old_postauth 365
```

`cleanup_stale_radacct`

This command closes stale RADIUS sessions that have remained open for the number of specified `<days>`.

```
./manage.py cleanup_stale_radacct <days>
```

For example:

```
./manage.py cleanup_stale_radacct 15
```

`deactivate_expired_users`

Note

Find out more about this feature in its dedicated page

This command deactivates expired user accounts which were created with batch operation temporarily (e.g.: for en event) and have an expiration date set.

```
./manage.py deactivate_expired_users
```

`delete_old_radiusbatch_users`

This command deletes users created using batch operation that have expired (and should have been deactivated by `deactivate_expired_users`) for more than the specified `<duration_in_days>`.

```
./manage.py delete_old_radiusbatch_users --older-than-days <duration_in_days>
```

Note that the default duration is set to **540 days** (18 months).

For backward compatibility, the command also accepts the argument `--older-than-months`:

```
./manage.py delete_old_radiusbatch_users --older-than-months <duration_in_months>
```

If both `--older-than-days` and `--older-than-months` are provided, preference is given to `--older-than-days`.

`delete_unverified_users`

This command deletes unverified users that have been registered for more than specified duration and have no associated radius session. This feature is needed to delete users who have registered but never completed the verification process. **Staff users will not be deleted by this management command.**

```
./manage.py delete_unverified_users --older-than-days <duration_in_days>
```

Note that the default duration is set to **1 day**.

It is also possible to exclude users that have registered using specified methods. You can specify multiple methods separated by comma(,). Following is an example:

```
./manage.py delete_unverified_users --older-than-days 1 --exclude-methods mobile_phone,email
```

`upgrade_from_django_freeradius`

If you are upgrading from django-freeradius to openwisp-radius, there is an easy migration script that will import your freeradius database, sites, social website account users, users & groups to openwisp-radius instance:

```
./manage.py upgrade_from_django_freeradius
```

The management command accepts an argument `--backup`, that you can pass to give the location of the backup files, by default it looks in the `tests/` directory, e.g.:

```
./manage.py upgrade_from_django_freeradius --backup /home/user/django_freeradius/
```

The management command accepts another argument `--organization`, if you want to import data to a specific organization, you can give its UUID for the same, by default the data is added to the first found organization, e.g.:

```
./manage.py upgrade_from_django_freeradius --organization 900856da-c89a-412d-8fee-45a9c763ca
```

Note

You can follow the tutorial to migrate database from django-freeradius.

Warning

It is not possible to export user credential data for `RadiusBatch` created using prefix, please manually preserve the PDF files if you want to access the data in the future.

`convert_called_station_id`

If an installation uses a centralized captive portal, the value of "Called Station ID" of RADIUS Sessions will always show the MAC address of the captive portal instead of the access points.

This command will update the "Called Station ID" to reflect the MAC address of the access points using information from OpenVPN. It requires installing `openvpn_status`, which can be installed using the following command

```
pip install openwisp-radius[openvpn_status]
```

In order to work, this command requires to be configured via the OPENWISP_RADIUS_CALLED_STATION_IDS setting.

Use the following command if you want to perform this operation for all RADIUS sessions that meet criteria of `OPENWISP_RADIUS_CALLED_STATION_IDS` setting.

```
./manage.py convert_called_station_id
```

You can also convert the "Called Station ID" of a particular RADIUS session by replacing session's `unique_id` in the following command:

```
./manage.py convert_called_station_id --unique_id=<session_unique_id>
```

Note

If you encounter `ParseError` for datetime data, you can set the datetime format of the parser using OPENWISP_RADIUS_OPENVPN_DATETIME_FORMAT setting.

Note

`convert_called_station_id` command will only operate on open RADIUS sessions, i.e. the "stop_time" field is None.

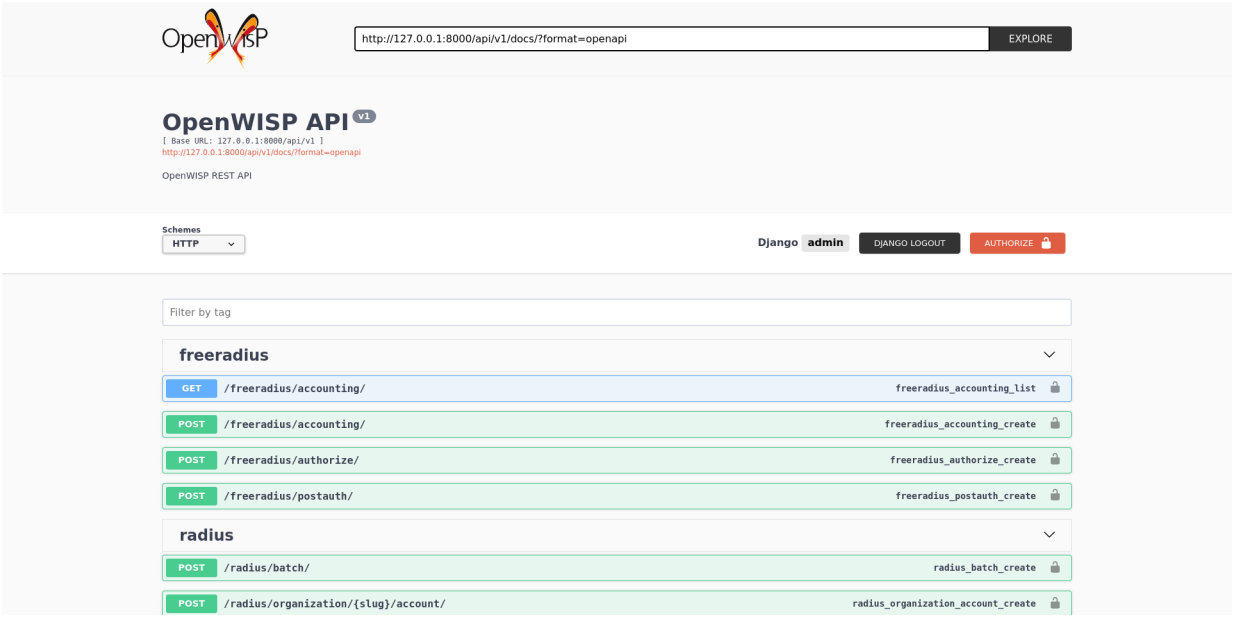
But if you are converting a single RADIUS session, it will operate on it even if the session is closed.

REST API Reference

Important

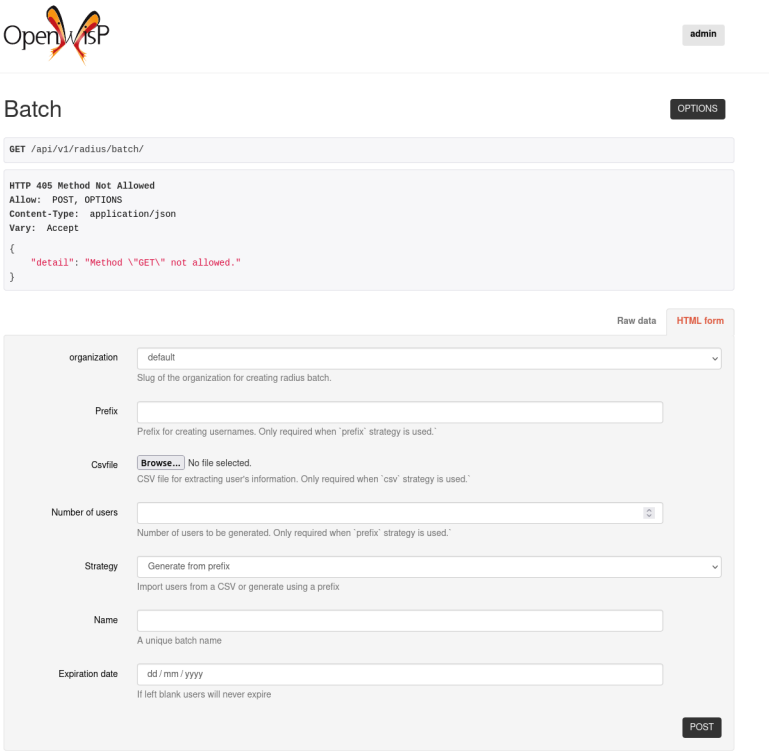
The REST API of openwisp-radius is enabled by default and may be turned off by setting OPENWISP_RADIUS_API to `False`.

Live documentation



A general live API documentation (following the OpenAPI specification) at `/api/v1/docs/`.

Browsable web interface



Additionally, opening any of the endpoints listed below directly in the browser will show the browsable API interface of Django-REST-Framework, which makes it even easier to find out the details of each endpoint.

FreeRADIUS API Endpoints

The following section is dedicated to API endpoints that are designed to be consumed by FreeRADIUS (Authorize, Post Auth, Accounting).

> **Important**
>
> These endpoints can be consumed only by hosts which have been added to the freeradius allowed hosts list.

FreeRADIUS API Authentication

There are 3 different methods with which the FreeRADIUS API endpoints can authenticate incoming requests and understand to which organization these requests belong.

Radius User Token

This method relies on the presence of a special token which was obtained by the user when authenticating via the Obtain Auth Token View, this means the user would have to log in through something like a web form first.

The flow works as follows:

1. the user enters credentials in a login form belonging to a specific organization and submits, the credentials are then sent to the Obtain Auth Token View;
2. if credentials are correct, a **radius user token** associated to the user and organization is created and returned in the response;
3. the login page or app must then initiate the HTTP request to the web server of the captive portal, (the URL of the form action of the default captive login page) using the radius user token as password, example:

```
curl -X POST http://captive.projcect.com:8005/index.php?zone=myorg \
     -d "auth_user=<username>&auth_pass=<radius_token>"
```

This method is recommended if you are using multiple organizations in the same OpenWISP instance.

> **Note**
>
> By default, `<radius_token>` is valid for authentication for one request only and a new `<radius_token>` needs to be obtained for each request. However, if OPENWISP_RADIUS_DISPOSABLE_RADIUS_USER_TOKEN is set to `False`, the `<radius_token>` is valid for authentication as long as freeradius accounting `Stop` request is not sent or the token is not deleted.

> **Warning**
>
> If you are using Radius User token method, keep in mind that one user account can only authenticate with one organization at a time, i.e a single user account cannot consume services from multiple organizations simultaneously.

Bearer token

This other method allows to use the system without the need for a user to obtain a token first, the drawback is that one FreeRADIUS site has to be configured for each organization, the authorization credentials for the specific organization is sent in each request, see Configure the site for more information on the FreeRADIUS site configuration.

The (Organization UUID and Organization RADIUS token) are sent in the authorization header of the HTTP request in the form of a Bearer token, e.g.:

```
curl -X POST http://localhost:8000/api/v1/freeradius/authorize/ \
     -H "Authorization: Bearer <org-uuid> <token>" \
     -d "username=<username>&password=<password>"
```

This method is recommended if you are using only one organization and you have no need nor intention of adding more organizations in the future.

Querystring

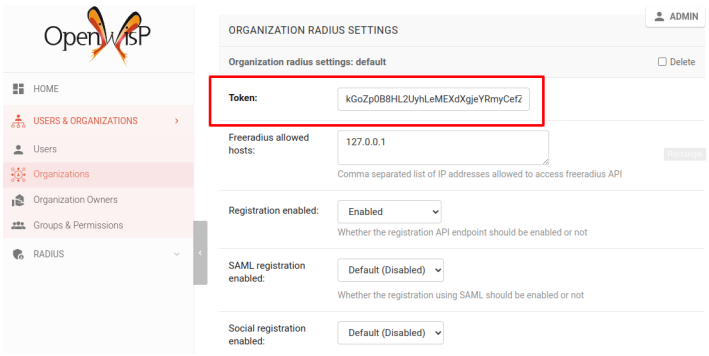
This method is identical to the previous one, but the credentials are sent in querystring parameters, e.g.:

```
curl -X POST http://localhost:8000/api/v1/freeradius/authorize/?uuid=<org-uuid>&token=<token
     -d "username=<username>&password=<password>"
```

This method is not recommended for production usage, it should be used for testing and debugging only (because webservers can include the querystring parameters in their logs).

Organization UUID & RADIUS API Token

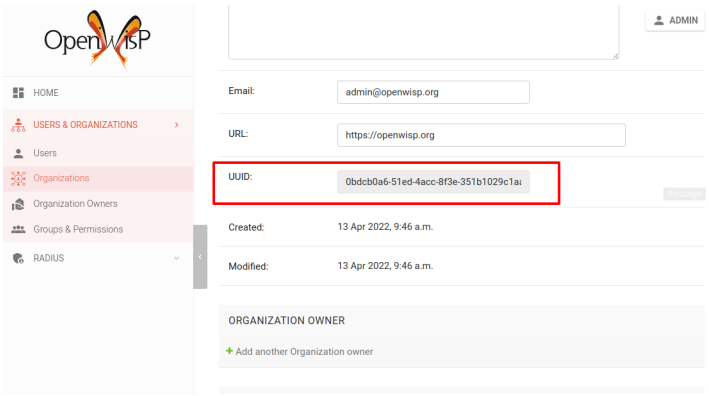
You can get (and set) the value of the OpenWISP RADIUS API token in the organization configuration page on the OpenWISP dashboard (select your organization in `/admin/openwisp_users/organization/`):



Note

It is highly recommended that you use a hard to guess value, longer than 15 characters containing both letters and numbers. E.g.: `165f9a790787fc38e5cc12c1640db2300648d9a2`.

You will also need the UUID of your organization from the organization change page (select your organization in `/admin/openwisp_users/organization/`):



Requests authorizing with bearer-token or querystring method **must** contain organization UUID & token. If the tokens are missing or invalid, the request will receive a `403` HTTP error.

For information on how to configure FreeRADIUS to send the bearer tokens, see Configure the site.

API Throttling

To override the default API throttling settings, add the following to your `settings.py` file:

```
REST_FRAMEWORK = {
    "DEFAULT_THROTTLE_CLASSES": [
        "rest_framework.throttling.ScopedRateThrottle",
    ],
    "DEFAULT_THROTTLE_RATES": {
        # None by default
        "authorize": None,
        "postauth": None,
        "accounting": None,
        "obtain_auth_token": None,
        "validate_auth_token": None,
        "create_phone_token": None,
        "phone_token_status": None,
        "validate_phone_token": None,
        # Relaxed throttling Policy
        "others": "400/hour",
    },
}
```

The rate descriptions used in `DEFAULT_THROTTLE_RATES` may include `second`, `minute`, `hour` or `day` as the throttle period, setting it to `None` will result in no throttling.

List of Endpoints

Authorize

Use by FreeRADIUS to perform the `authorization` phase.

It's triggered when a user submits the form to login into the captive portal. The captive portal has to be configured to send the password to freeradius in clear text (will be encrypted with the freeradius shared secret, can be tunneled via TLS for increased security if needed).

FreeRADIUS in turn will send the username and password via HTTPs to this endpoint.

Responds to only **POST**.

```
/api/v1/freeradius/authorize/
```

Example:

```
POST /api/v1/freeradius/authorize/ HTTP/1.1 username=testuser&password=testpassword
```

Param	Description
username	Username for the given user
password	Password for the given user

If the authorization is successful, the API will return all group replies related to the group with highest priority assigned to the user.

If the authorization is unsuccessful, the response body can either be empty or it can contain an explicit rejection, depending on how the OPENWISP_RADIUS_API_AUTHORIZE_REJECT setting is configured.

Post Auth

API endpoint designed to be used by FreeRADIUS `postauth`.

Responds only to **POST**.

```
/api/v1/freeradius/postauth/
```

Param	Description
username	Username
password	Password (*)
reply	Radius reply received by freeradius
called_station_id	Called Station ID
calling_station_id	Calling Station ID

(*): the `password` is stored only on unsuccessful authorizations.

Returns an empty response body in order to instruct FreeRADIUS to avoid processing the response body.

Accounting

```
/api/v1/freeradius/accounting/
```

GET

Returns a list of accounting objects

```
GET /api/v1/freeradius/accounting/
```


```
[
  {
      "called_station_id": "00-27-22-F3-FA-F1:hostname",
      "nas_port_type": "Async",
      "groupname": null,
      "id": 1,
      "realm": "",
      "terminate_cause": "User_Request",
      "nas_ip_address": "172.16.64.91",
      "authentication": "RADIUS",
      "stop_time": null,
      "nas_port_id": "1",
      "service_type": "Login-User",
      "username": "admin",
      "update_time": null,
      "connection_info_stop": null,
      "start_time": "2018-03-10T14:44:17.234035+01:00",
      "output_octets": 1513075509,
```

```
        "calling_station_id": "5c:7d:c1:72:a7:3b",
        "input_octets": 9900909,
        "interval": null,
        "session_time": 261,
        "session_id": "35000006",
        "connection_info_start": null,
        "framed_protocol": "test",
        "framed_ip_address": "127.0.0.1",
        "unique_id": "75058e50"
    }
]
```


POST

Add or update accounting information (start, interim-update, stop); does not return any JSON response so that freeradius will avoid processing the response without generating warnings

Param	Description
session_id	Session ID
unique_id	Accounting unique ID
username	Username
groupname	Group name
realm	Realm
nas_ip_address	NAS IP address
nas_port_id	NAS port ID
nas_port_type	NAS port type
start_time	Start time
update_time	Update time
stop_time	Stop time
interval	Interval
session_time	Session Time
authentication	Authentication
connection_info_start	Connection Info Start
connection_info_stop	Connection Info Stop
input_octets	Input Octets
output_octets	Output Octets
called_station_id	Called station ID
calling_station_id	Calling station ID
terminate_cause	Termination Cause
service_type	Service Type
framed_protocol	Framed protocol
framed_ip_address	framed IP address

Pagination

Pagination is provided using a Link header pagination. Check here for more information about traversing with pagination.

```
{
  ....
  ....
  link: <http://testserver/api/v1/freeradius/accounting/?page=2&page_size=1>; rel=\"next\",
        <http://testserver/api/v1/freeradius/accounting/?page=3&page_size=1>; rel=\"last\"
  ....
  ....
}
```

Note

Default page size is 10, which can be overridden using the *page_size* parameter.

Filters

The JSON objects returned using the GET endpoint can be filtered/queried using specific parameters.

Filter Parameters	Description
username	Username
called_station_id	Called Station ID
calling_station_id	Calling Station ID
start_time	Start time (greater or equal to)
stop_time	Stop time (less or equal to)
is_open	If stop_time is null

User API Endpoints

These API endpoints are designed to be used by users (e.g.: creating an account, changing their password, obtaining access tokens, validating their phone number, etc.).

Note

The API endpoints described below do not require the Organization API Token described in the beginning of this document.

Some endpoints require the sending of the user API access token sent in the form of a "Bearer Token", example:

```
curl -H "Authorization: Bearer <user-token>" \
     'http://localhost:8000/api/v1/radius/organization/default/account/session/'
```

List of Endpoints

User Registration

Important

This endpoint is enabled by default but can be disabled either via a global setting or from the admin interface.

```
/api/v1/radius/organization/<organization-slug>/account/
```

Responds only to **POST**.

Parameters:

Param	Description
username	string
phone_number	string (*)
email	string
password1	string
password2	string
first_name	string (**)
last_name	string (**)
birth_date	string (**)
location	string (**)
method	string (***)

(*) `phone_number` is required only when the organization has enabled SMS verification in its "Organization RADIUS Settings".

(**) `first_name`, `last_name`, `birth_date` and `location` are optional fields which are disabled by default to make the registration simple, but can be enabled through configuration.

(**) `method` must be one of the available registration/verification methods; if identity verification is disabled for a particular org, an empty string will be acceptable.

Registering to Multiple Organizations

An **HTTP 409** response will be returned if an existing user tries to register on a URL of a different organization (because the account already exists). The response will contain a list of organizations with which the user has already registered to the system which may be shown to the user in the UI. E.g.:

```
{
    "details": "A user like the one being registered already exists.",
    "organizations":[
        {"slug":"default","name":"default"}
    ]
}
```

The existing user can register with a new organization using the login endpoint. The user will also get membership of the new organization only if the organization has user registration enabled.

Reset password

This is the classic "password forgotten recovery feature" which sends a reset password token to the email of the user.

```
/api/v1/radius/organization/<organization-slug>/account/password/reset/
```

Responds only to **POST**.

Parameters:

Param	Description
input	string that can be an email, phone_number or username.

Confirm reset password

Allows users to confirm their reset password after having it requested via the Reset password endpoint.

`/api/v1/radius/organization/<organization-slug>/account/password/reset/confirm/`

Responds only to **POST**.

Parameters:

Param	Description
new_password1	string
new_password2	string
uid	string
token	string

Change password

Requires the user auth token (Bearer Token).

Allows users to change their password after using the Reset password endpoint.

`/api/v1/radius/organization/<organization-slug>/account/password/change/`

Responds only to **POST**.

Parameters:

Param	Description
current_password	string
new_password	string
confirm_password	string

Login (Obtain User Auth Token)

`/api/v1/radius/organization/<organization-slug>/account/token/`

Responds only to **POST**.

Returns:

- `radius_user_token`: the user radius token, which can be used to authenticate the user in the captive portal by sending it in place of the user password (it will be passed to freeradius which in turn will send it to the authorize API endpoint which will recognize the token as the user password)
- `key`: the user API access token, which will be needed to authenticate the user to eventual subsequent API requests (e.g.: change password)
- `is_active` if it's `false` it means the user has been banned
- `is_verified` when identity verification is enabled, it indicates whether the user has completed an indirect identity verification process like confirming their mobile phone number
- `method` registration/verification method used by the user to register, e.g.: `mobile_phone`, `social_login`, etc.

- `username`
- `email`
- `phone_number`
- `first_name`
- `last_name`
- `birth_date`
- `location`

If the user account is inactive or unverified the endpoint will send the data anyway but using the HTTP status code 401, this way consumers can recognize these users and trigger the appropriate response needed (e.g.: reject them or initiate account verification).

If an existing user account tries to authenticate to an organization of which they're not member of, then they would be automatically added as members (if registration is enabled for that org). Please refer to "Registering to Multiple Organizations".

This endpoint updates the user language preference field according to the `Accept-Language` HTTP header.

Parameters:

Param	Description
username	string
password	string

Validate user auth token

Used to check whether the auth token of a user is valid or not.

Return also the radius user token and username in the response.

```
/api/v1/radius/organization/<organization-slug>/account/token/validate/
```

Responds only to **POST**.

Parameters:

Param	Description
token	the rest auth token to validate

The user information is returned in the response (similarly to Obtain User Auth Token), along with the following additional parameter:

- `response_code`: string indicating whether the result is successful or not, to be used for translation.

This endpoint updates the user language preference field according to the `Accept-Language` HTTP header.

User Radius Sessions

Requires the user auth token (Bearer Token).

Returns the radius sessions of the logged-in user and the organization specified in the URL.

```
/api/v1/radius/organization/<organization-slug>/account/session/
```

Responds only to **GET**.

User Radius Usage

Requires the user auth token (Bearer Token).

Returns the radius usage of the logged-in user and the organization specified in the URL.

It executes the relevant RADIUS counters and returns information that shows how much time and/or traffic the user has consumed.

`/api/v1/radius/organization/<organization-slug>/account/usage/`

Responds only to **GET**.

Create SMS token

> **Note**
>
> This API endpoint will work only if the organization has enabled SMS verification.

Requires the user auth token (Bearer Token).

Used for SMS verification, sends a code via SMS to the phone number of the user.

`/api/v1/radius/organization/<organization-slug>/account/phone/token/`

Responds only to **POST**.

No parameters required.

Get active SMS token status

> **Note**
>
> This API endpoint will work only if the organization has enabled SMS verification.

Requires the user auth token (Bearer Token).

Used for SMS verification, allows checking whether an active SMS token was already requested for the mobile phone number of the logged in account.

`/api/v1/radius/organization/<organization-slug>/account/phone/token/active/`

Responds only to **GET**.

No parameters required.

Verify/Validate SMS token

> **Note**
>
> This API endpoint will work only if the organization has enabled SMS verification.

Requires the user auth token (Bearer Token).

Used for SMS verification, allows users to validate the code they receive via SMS.

`/api/v1/radius/organization/<organization-slug>/account/phone/verify/`

Responds only to **POST**.

Parameters:

Param	Description
code	string

Change phone number

Note

This API endpoint will work only if the organization has enabled SMS verification.

Requires the user auth token (Bearer Token).

Allows users to change their phone number, will flag the user as inactive and send them a verification code via SMS. The phone number of the user is updated only after this verification code has been validated.

`/api/v1/radius/organization/<organization-slug>/account/phone/change/`

Responds only to **POST**.

Parameters:

Param	Description
phone_number	string

Batch user creation

This API endpoint allows to use the features described in Importing users and Generating users.

`/api/v1/radius/batch/`

Note

This API endpoint allows to use the features described in Importing users and Generating users.

Responds only to **POST**, used to save a `RadiusBatch` instance.

It is possible to generate the users of the `RadiusBatch` with two different strategies: csv or prefix.

The csv method needs the following parameters:

Param	Description
name	Name of the operation
strategy	csv
csvfile	file with the users
expiration_date	date of expiration of the users
organization_slug	slug of organization of the users

These others are for the prefix method:

Param	Description
name	name of the operation

strategy	prefix
prefix	prefix for the generation of users
number_of_users	number of users
expiration_date	date of expiration of the users
organization_slug	slug of organization of the users

When using this strategy, in the response you can find the field `user_credentials` containing the list of users created (example: `[['username', 'password'], ['sample_user', 'BBuOb5sN']]`) and the field `pdf_link` which can be used to download a PDF file containing the user credentials.

Batch CSV Download

```
/api/v1/radius/organization/<organization-slug>/batch/<id>/csv/<filename>
```

Responds only to **GET**.

Parameters:

Param	Description
slug	string
id	string
filename	string

Settings

Note

If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

Admin related settings

These settings control details of the administration interface of openwisp-radius.

Note

The values of overridden settings fields do not change even when the global defaults are changed.

`OPENWISP_RADIUS_EDITABLE_ACCOUNTING`

Default: `False`

Whether `radacct` entries are editable from the django admin or not.

`OPENWISP_RADIUS_EDITABLE_POSTAUTH`

Default: `False`

Whether `postauth` logs are editable from the django admin or not.

`OPENWISP_RADIUS_GROUPCHECK_ADMIN`

Default: `False`

Direct editing of group checks items is disabled by default because these can be edited through inline items in the Radius Group admin (Freeradius > Groups).

This is done with the aim of simplifying the admin interface and avoid overwhelming users with too many options.

If for some reason you need to enable direct editing of group checks you can do so by setting this to `True`.

`OPENWISP_RADIUS_GROUPREPLY_ADMIN`

Default: `False`

Direct editing of group reply items is disabled by default because these can be edited through inline items in the Radius Group admin (Freeradius > Groups).

This is done with the aim of simplifying the admin interface and avoid overwhelming users with too many options.

If for some reason you need to enable direct editing of group replies you can do so by setting this to `True`.

`OPENWISP_RADIUS_USERGROUP_ADMIN`

Default: `False`

Direct editing of user group items (`radusergroup`) is disabled by default because these can be edited through inline items in the User admin (Users and Organizations > Users).

This is done with the aim of simplifying the admin interface and avoid overwhelming users with too many options.

If for some reason you need to enable direct editing of user group items you can do so by setting this to `True`.

`OPENWISP_RADIUS_USER_ADMIN_RADIUSTOKEN_INLINE`

Default: `False`

The functionality of editing a user's `RadiusToken` directly through an inline from the user admin page is disabled by default.

This is done with the aim of simplifying the admin interface and avoid overwhelming users with too many options.

If for some reason you need to enable editing user's `RadiusToken` from the user admin page, you can do so by setting this to `True`.

Model related settings

These settings control details of the openwisp-radius model classes.

`OPENWISP_RADIUS_DEFAULT_SECRET_FORMAT`

Default: `NT-Password`

The default encryption format for storing radius check values.

`OPENWISP_RADIUS_DISABLED_SECRET_FORMATS`

Default: `[]`

A list of disabled encryption formats, by default all formats are enabled in order to keep backward compatibility with legacy systems.

`OPENWISP_RADIUS_BATCH_DEFAULT_PASSWORD_LENGTH`

Default: `8`

The default password length of the auto generated passwords while batch addition of users from the csv.

`OPENWISP_RADIUS_BATCH_DELETE_EXPIRED`

Default: `540` (18 months)

It is the number of days after which the expired users are deleted.

`OPENWISP_RADIUS_BATCH_PDF_TEMPLATE`

It is the template used to generate the PDF when users are being generated using the batch add users feature using the prefix.

The value should be the absolute path to the template of the PDF.

`OPENWISP_RADIUS_EXTRA_NAS_TYPES`

Default: `tuple()`

This setting can be used to add custom NAS types that can be used from the admin interface when managing NAS instances.

For example, you want a custom NAS type called `cisco`, you would add the following to your project `settings.py`:

```
OPENWISP_RADIUS_EXTRA_NAS_TYPES = (("cisco", "Cisco Router"),)
```

`OPENWISP_RADIUS_FREERADIUS_ALLOWED_HOSTS`

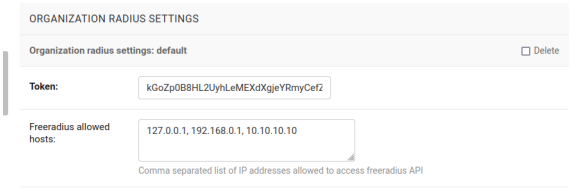
Default: `[]`

List of host IP addresses or subnets allowed to consume the freeradius API endpoints (Authorize, Accounting and Postauth), i.e the value of this option should be the IP address of your freeradius instance. Example: If your freeradius instance is running on the same host machine as OpenWISP, the value should be `127.0.0.1`. Similarly, if your freeradius instance is on a different host in the private network, the value should be the private IP of freeradius host like `192.0.2.50`. If your freeradius is on a public network, please use the public IP of your freeradius instance.

You can use subnets when freeradius is hosted on a variable IP, e.g.:

- `198.168.0.0/24` to allow the entire LAN.
- `0.0.0.0/0` to allow any address (useful for development / testing).

This value can be overridden per organization in the organization change page. You can skip setting this option if you intend to set it from organization change page for each organization.




```
OPENWISP_RADIUS_FREERADIUS_ALLOWED_HOSTS = [
    "127.0.0.1",
    "192.0.2.10",
    "192.168.0.0/24",
]
```

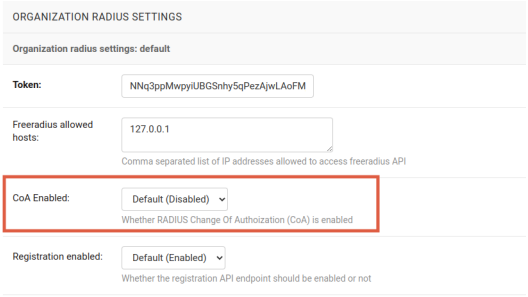
If this option and organization change page option are both empty, then all freeradius API requests for the organization will return `403`.

`OPENWISP_RADIUS_COA_ENABLED`

Default: `False`\`

If set to `True`, openwisp-radius will update the NAS with the user's current RADIUS attributes whenever the `RadiusGroup` of user is changed. This allow enforcing of rate limits on active RADIUS sessions without requiring users to re-authenticate. For more details, read the dedicated section for configuring openwisp-radius and NAS for using CoA.

This can be overridden for each organization separately via the organization radius settings section of the admin interface.



`RADCLIENT_ATTRIBUTE_DICTIONARIES`

type:	`list`
default:	`[]`

List of absolute file paths of additional RADIUS dictionaries used for RADIUS attribute mapping.

Note

A default dictionary is shipped with openwisp-radius. Any dictionary added using this setting will be used alongside the default dictionary.

`OPENWISP_RADIUS_MAX_CSV_FILE_SIZE`

type:	`int`
default:	*5 * 1024 * 1024* (5 MB)

This setting can be used to set the maximum size limit for firmware images, e.g.:

```
OPENWISP_RADIUS_MAX_CSV_FILE_SIZE = 10 * 1024 * 1024  # 10MB
```

Note

The numeric value represents the size of files in bytes. Setting this to `None` will mean there's no max size.

`OPENWISP_RADIUS_PRIVATE_STORAGE_INSTANCE`

type:	`str`
default:	`openwisp_radius.private_storage.storage.private_file_system_storage`

Dotted path to an instance of any one of the storage classes in private_storage. This instance is used for storing csv files of batch imports of users.

By default, an instance of `private_storage.storage.files.PrivateFileSystemStorage` is used.

`OPENWISP_RADIUS_CALLED_STATION_IDS`

Default: `{}`

This setting allows to specify the parameters to connect to the different OpenVPN management interfaces available for an organization. This setting is used by the convert_called_station_id command.

It should contain configuration in following format:

```
OPENWISP_RADIUS_CALLED_STATION_IDS = {
    # UUID of the organization for which settings are being specified
    # In this example 'default'
    "<organization_uuid>": {
        "openvpn_config": [
            {
                # Host address of OpenVPN management
                "host": "<host>",
                # Port of OpenVPN management interface. Defaults to 7505 (integer)
                "port": 7506,
                # Password of OpenVPN management interface (optional)
                "password": "<management_interface_password>",
            }
        ],
        # List of CALLED STATION IDs that has to be converted,
        # These look like: 00:27:22:F3:FA:F1:gw1.openwisp.org
        "unconverted_ids": ["<called_station_id>"],
    }
}
```

`OPENWISP_RADIUS_CONVERT_CALLED_STATION_ON_CREATE`

Default: `False`

If set to `True`, "Called Station ID" of a RADIUS session will be converted (as per configuration defined in OPENWISP_RADIUS_CALLED_STATION_IDS) just after the RADIUS session is created.

`OPENWISP_RADIUS_OPENVPN_DATETIME_FORMAT`

Default: `u'%a %b %d %H:%M:%S %Y'`

Specifies the datetime format of OpenVPN management status parser used by the convert_called_station_id command.

```
OPENWISP_RADIUS_UNVERIFY_INACTIVE_USERS
```

Default: `0` (disabled)

Number of days from user's `last_login` after which the user will be flagged as *unverified*.

When set to `0`, the feature would be disabled and the user will not be flagged as *unverified*.

```
OPENWISP_RADIUS_DELETE_INACTIVE_USERS
```

Default: `0` (disabled)

Number of days from user's `last_login` after which the user will be deleted.

When set to `0`, the feature would be disabled and the user will not be deleted.

API and user token related settings

These settings control details related to the API and the radius user token.

```
OPENWISP_RADIUS_API_URLCONF
```

Default: `None`

Changes the urlconf option of django URLs to point the RADIUS API URLs to another installed module, example, `myapp.urls` (useful when you have a separate API instance.)

```
OPENWISP_RADIUS_API_BASEURL
```

Default: `/` (points to same server)

If you have a separate instance of openwisp-radius API on a different domain, you can use this option to change the base of the image download URL, this will enable you to point to your API server's domain, example value: `https://myradius.myapp.com`.

```
OPENWISP_RADIUS_API
```

Default: `True`

Indicates whether the REST API of openwisp-radius is enabled or not.

```
OPENWISP_RADIUS_DISPOSABLE_RADIUS_USER_TOKEN
```

Default: `True`

Radius user tokens are used for authorizing users.

When this setting is `True` radius user tokens are deleted right after a successful authorization is performed. This reduces the possibility of attackers reusing the access tokens and posing as other users if they manage to intercept it somehow.

```
OPENWISP_RADIUS_API_AUTHORIZE_REJECT
```

Default: `False`

Indicates whether the Authorize API view will return `{"control:Auth-Type": "Reject"}` or not.

Rejecting an authorization request explicitly will prevent freeradius from attempting to perform authorization with other mechanisms (e.g.: radius checks, LDAP, etc.).

When set to `False`, if an authorization request fails, the API will respond with `None`, which will allow freeradius to keep attempting to authorize the request with other freeradius modules.

Set this to `True` if you are performing authorization exclusively through the REST API.

`OPENWISP_RADIUS_API_ACCOUNTING_AUTO_GROUP`

Default: `True`

When this setting is enabled, every accounting instance saved from the API will have its `groupname` attribute automatically filled in. The value filled in will be the `groupname` of the `RadiusUserGroup` of the highest priority among the RadiusUserGroups related to the user with the `username` as in the accounting instance. In the event there is no user in the database corresponding to the `username` in the accounting instance, the failure will be logged with `warning` level but the accounting will be saved as usual.

`OPENWISP_RADIUS_ALLOWED_MOBILE_PREFIXES`

Default: `[]`

This setting is used to specify a list of international mobile prefixes which should be allowed to register into the system via the user registration API.

That is, only users with phone numbers using the specified international prefixes will be allowed to register.

Leaving this unset or setting it to an empty list (`[]`) will effectively allow any international mobile prefix to register (which is the default setting).

For example:

```
OPENWISP_RADIUS_ALLOWED_MOBILE_PREFIXES = ["+44", "+237"]
```

Using the setting above will only allow phone numbers from the UK (+44) or Cameroon (+237).

Note

This setting is applicable only for organizations which have enabled the SMS verification option.

`OPENWISP_RADIUS_ALLOW_FIXED_LINE_OR_MOBILE`

Default: `False`

OpenWISP RADIUS only allow using mobile phone numbers for user registration. This can cause issues in regions where fixed line and mobile phone numbers uses the same pattern (e.g. USA). Setting the value to `True` would make phone number type checking less strict.

`OPENWISP_RADIUS_OPTIONAL_REGISTRATION_FIELDS`

Default:

```
{
    "first_name": "disabled",
    "last_name": "disabled",
    "birth_date": "disabled",
    "location": "disabled",
}
```

This global setting is used to specify if the optional user fields (`first_name`, `last_name`, `location` and `birth_date`) shall be disabled (hence ignored), allowed or required in the User Registration API.

The allowed values are:

- `disabled`: (**default**) the field is disabled.
- `allowed`: the field is allowed but not mandatory.
- `mandatory`: the field is mandatory.

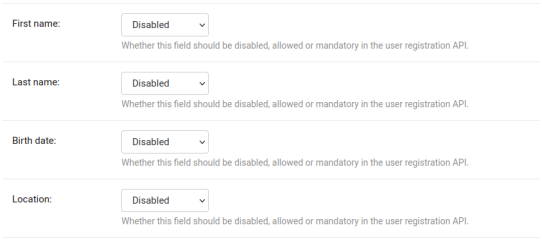
For example:

```
OPENWISP_RADIUS_OPTIONAL_REGISTRATION_FIELDS = {
    "first_name": "disabled",
    "last_name": "disabled",
    "birth_date": "mandatory",
    "location": "allowed",
}
```

Means:

- `first_name` and `last_name` fields are not required and their values if provided are ignored.
- `location` field is not required but its value will be saved to the database if provided.
- `birth_date` field is required and a `ValidationError` exception is raised if its value is not provided.

The setting for each field can also be overridden at organization level if needed, by going to `Home › Users and Organizations › Organizations > Edit organization` and then scrolling down to `ORGANIZATION RADIUS SETTINGS`.



By default the fields at organization level hold a `NULL` value, which means that the global setting specified in `settings.py` will be used.

OPENWISP_RADIUS_PASSWORD_RESET_URLS

> **Note**
>
> This setting can be overridden for each organization in the organization admin page, the setting implementation is left for backward compatibility but may be deprecated in the future.

Default:

```
{
    "__all__": "https://{site}/{organization}/password/reset/confirm/{uid}/{token}"
}
```

A dictionary representing the frontend URLs through which end users can complete the password reset operation.

The frontend could be OpenWISP WiFi Login Pages or another in-house captive page solution.

Keys of the dictionary must be either UUID of organizations or `__all__`, which is the fallback URL that will be used in case there's no customized URL for a specific organization.

The password reset URL must contain the "{token}" and "{uid}" placeholders.

The meaning of the variables in the string is the following:

- `{site}`: site domain as defined in the django site framework (defaults to example.com and an be changed through the django admin)
- `{organization}`: organization slug
- `{uid}`: uid of the password reset request
- `{token}`: token of the password reset request

If you're using OpenWISP WiFi Login Pages, the configuration is fairly simple, in case the NodeJS app is installed in the same domain of openwisp-radius, you only have to ensure the domain field in the main Site object is correct, if instead the NodeJS app is deployed on a different domain, say `login.wifiservice.com`, the configuration should be simply changed to:

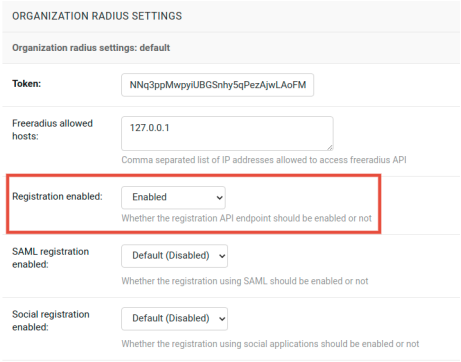
```
{
    "__all__": "https://login.wifiservice.com/{organization}/password/reset/confirm/{uid}/{t
}
```

OPENWISP_RADIUS_REGISTRATION_API_ENABLED

Default: `True`

Indicates whether the API registration view is enabled or not. When this setting is disabled (i.e. `False`), the registration API view is disabled.

This setting can be overridden in individual organizations via the admin interface, by going to *Organizations* then edit a specific organization and scroll down to *"Organization RADIUS settings"*, as shown in the screenshot below.



Note

We recommend using the override via the admin interface only when there are special organizations which need a different configuration, otherwise, if all the organization use the same configuration, we recommend changing the global setting.

OPENWISP_RADIUS_SMS_VERIFICATION_ENABLED

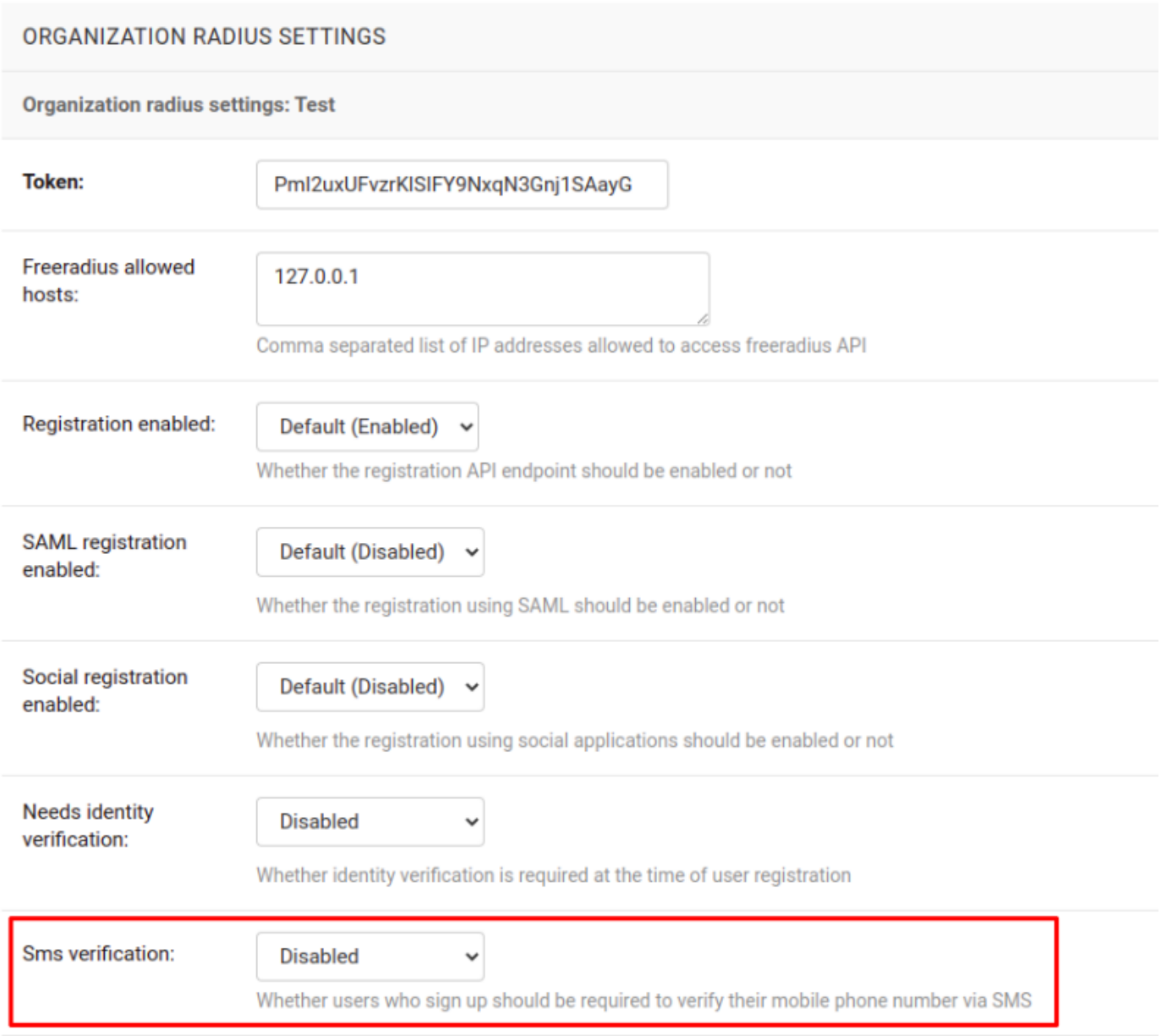
Default: `False`

Note

If you're looking for instructions on how to configure SMS sending, see SMS Token Related Settings.

If Identity verification is required, this setting indicates whether users who sign up should be required to verify their mobile phone number via SMS.

This can be overridden for each organization separately via the organization radius settings section of the admin interface.

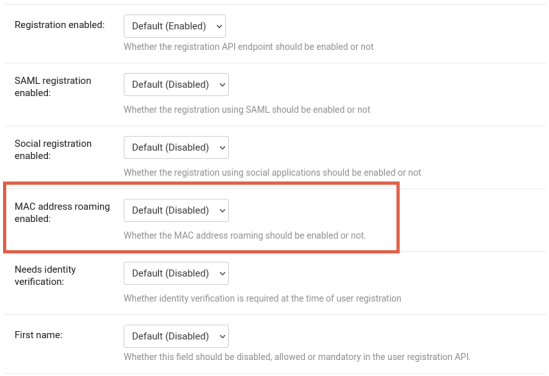


`OPENWISP_RADIUS_MAC_ADDR_ROAMING_ENABLED`

Default: `False`

Indicates whether MAC address roaming is supported. When this setting is enabled (i.e. `True`), MAC address roaming is enabled for all organizations.

This setting can be overridden in individual organizations via the admin interface, by going to *Organizations* then edit a specific organization and scroll down to *"Organization RADIUS settings"*, as shown in the screenshot below.



Note

We recommend using the override via the admin interface only when there are special organizations which need a different configuration, otherwise, if all the organization use the same configuration, we recommend changing the global setting.

`OPENWISP_RADIUS_NEEDS_IDENTITY_VERIFICATION`

Default: `False`

Indicates whether organizations require a user to be verified in order to login. This can be overridden globally or for each organization separately via the admin interface.

If this is enabled, each registered user should be verified using a verification method. The following choices are available by default:

- `''` (empty string): unspecified
- `manual`: Manually created
- `email`: Email
- `mobile_phone`: Mobile phone number verification via SMS
- `social_login`: social login feature

Note

Of the methods listed above, `mobile_phone` is generally accepted as a legal and valid form of indirect identity verification in those countries who require to provide a valid ID document before buying a SIM card.

Organizations which are required by law to identify their users before allowing them to access the network (e.g.: ISPs) can restrict users to register only through this method and can configure the system to only allow international mobile prefixes of countries which require a valid ID document to buy a SIM card.

Disclaimer: these are just suggestions on possible configurations of OpenWISP RADIUS and must not be considered as legal advice.

Adding support for more registration/verification methods

For those who need to implement additional registration and identity verification methods, such as supporting a National ID card, new methods can be added or an existing method can be removed using the `register_registration_method` and `unregister_registration_method` functions respectively.

For example:

```
from openwisp_radius.registration import (
    register_registration_method,
    unregister_registration_method,
)

# Enable registering via national digital ID
register_registration_method("national_id", "National Digital ID")

# Remove mobile verification method
unregister_registration_method("mobile_phone")
```

Note

Both functions will fail if a specific registration method is already registered or unregistered, unless the keyword argument `fail_loud` is passed as `False` (this useful when working with additional registration methods which are supported by multiple custom modules).

Pass `strong_identity` as `True` to to indicate that users who register using that method have indirectly verified their identity (e.g.: SMS verification, credit card, national ID card, etc).

Warning

If you need to implement a registration method that needs to grant limited internet access to unverified users so they can complete their verification process online on other websites which cannot be predicted and hence cannot be added to the walled garden, you can pass `authorize_unverified=True` to the `register_registration_method` function.

This is needed to implement payment flows in which users insert a specific 3D secure code in the website of their bank. Keep in mind that you should create a specific limited radius group for these unverified users.

Payment flows and credit/debit card verification are fully implemented in **OpenWISP Subscriptions**, a premium module available only to customers of the commercial support offering of OpenWISP.

Email related settings

Emails can be sent to users whose usernames or passwords have been auto-generated. The content of these emails can be customized with the settings explained below.

`OPENWISP_RADIUS_BATCH_MAIL_SUBJECT`

Default: `Credentials`

It is the subject of the mail to be sent to the users. E.g.: `Login Credentials`.

`OPENWISP_RADIUS_BATCH_MAIL_MESSAGE`

Default: `username: {}, password: {}`

The message should be a string in the format `Your username is {} and password is {}`.

The text could be anything but should have the format string operator `{}` for `.format` operations to work.

`OPENWISP_RADIUS_BATCH_MAIL_SENDER`

Default: `settings.DEFAULT_FROM_EMAIL`

It is the sender email which is also to be configured in the SMTP settings. The default sender email is a common setting from the Django core settings under `DEFAULT_FROM_EMAIL`. Currently, `DEFAULT_FROM_EMAIL` is set to to `webmaster@localhost`.

Counter related settings

`OPENWISP_RADIUS_COUNTERS`

Default: depends on the database backend in use, see How Limits are Enforced: Counters to find out what are the default counters enabled.

It's a list of strings, each representing the python path to a counter class.

It may be set to an empty list or tuple to disable the counter feature, e.g.:

```
OPENWISP_RADIUS_COUNTERS = []
```

If custom counters have been implemented, this setting should be changed to include the new classes, e.g.:

```
OPENWISP_RADIUS_COUNTERS = [
    # default counters for PostgreSQL, may be removed if not needed
    "openwisp_radius.counters.postgresql.daily_counter.DailyCounter",
    "openwisp_radius.counters.postgresql.radius_daily_traffic_counter.DailyTrafficCounter",
    # custom counters
    "myproject.counters.CustomCounter1",
    "myproject.counters.CustomCounter2",
]
```

`OPENWISP_RADIUS_TRAFFIC_COUNTER_CHECK_NAME`

Default: `Max-Daily-Session-Traffic`

Used by DailyTrafficCounter, it indicates the check attribute which is looked for in the database to find the maximum amount of daily traffic which users having the default `users` radius group assigned can consume.

`OPENWISP_RADIUS_TRAFFIC_COUNTER_REPLY_NAME`

Default: `CoovaChilli-Max-Total-Octets`

Used by DailyTrafficCounter, it indicates the reply attribute which is returned to the NAS to indicate how much remaining traffic users which users having the default `users` radius group assigned can consume.

It should be changed according to the NAS software in use, for example, if using PfSense, this setting should be set to `pfSense-Max-Total-Octets`.

`OPENWISP_RADIUS_RADIUS_ATTRIBUTES_TYPE_MAP`

Default: `{}`

Used by User Radius Usage API, it stores mapping of RADIUS attributes to the unit of value enforced by the attribute, e.g. `bytes` for traffic counters and `seconds` for session time counters.

In the following example, the setting is configured to return `bytes` type in the API response for `ChilliSpot-Max-Input-Octets` attribute:

```
OPENWISP_RADIUS_RADIUS_ATTRIBUTES_TYPE_MAP = {
    "ChilliSpot-Max-Input-Octets": "bytes"
}
```

Social Login related settings

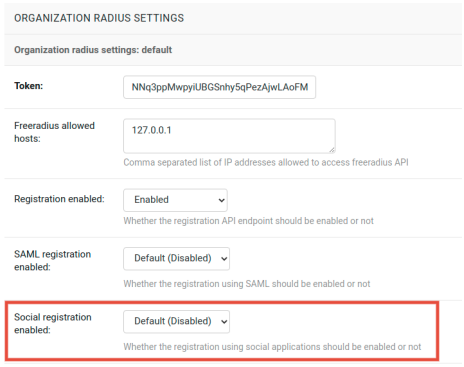
The following settings are related to the social login feature.

`OPENWISP_RADIUS_SOCIAL_REGISTRATION_ENABLED`

Default: `False`

Indicates whether the registration using social applications is enabled or not. When this setting is enabled (i.e. `True`), authentication using social applications is enabled for all organizations.

This setting can be overridden in individual organizations via the admin interface, by going to *Organizations* then edit a specific organization and scroll down to *"Organization RADIUS settings"*, as shown in the screenshot below.



Note

We recommend using the override via the admin interface only when there are special organizations which need a different configuration, otherwise, if all the organization use the same configuration, we recommend changing the global setting.

SAML related settings

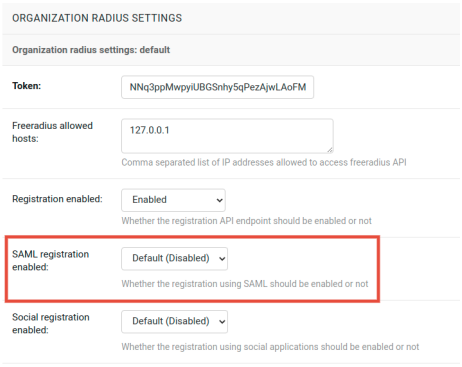
The following settings are related to the SAML feature.

`OPENWISP_RADIUS_SAML_REGISTRATION_ENABLED`

Default: `False`

Indicates whether registration using SAML is enabled or not. When this setting is enabled (i.e. `True`), authentication using SAML is enabled for all organizations.

This setting can be overridden in individual organizations via the admin interface, by going to *Organizations* then edit a specific organization and scroll down to *"Organization RADIUS settings"*, as shown in the screenshot below.



Note

We recommend using the override via the admin interface only when there are special organizations which need a different configuration, otherwise, if all the organization use the same configuration, we recommend changing the global setting.

`OPENWISP_RADIUS_SAML_REGISTRATION_METHOD_LABEL`

Default: `'Single Sign-On (SAML)'`

Sets the verbose name of SAML registration method.

`OPENWISP_RADIUS_SAML_IS_VERIFIED`

Default: `False`

Setting this to `True` will automatically flag user accounts created during SAML sign-in as verified users (`RegisteredUser.is_verified=True`).

This is useful when SAML identity providers can be trusted to be legally valid identity verifiers.

`OPENWISP_RADIUS_SAML_UPDATES_PRE_EXISTING_USERNAME`

Default: `False`

Allows updating username of a registered user with the value received from SAML Identity Provider. Read the FAQs in SAML integration documentation for details.

SMS token related settings

These settings allow to control aspects and limitations of the SMS tokens which are sent to users for the purpose of verifying their mobile phone number.

These settings are applicable only when SMS verification is enabled.

`SENDSMS_BACKEND`

This setting takes a python path which points to the django-sendsms backend which will be used by the system to send SMS messages.

The list of supported SMS services can be seen in the source code of the django-sendsms backends. Adding support for other SMS services can be done by sub-classing the `BaseSmsBackend` and implement the logic needed to talk to the SMS service.

The value of this setting can point to any class on the python path, so the backend doesn't have to be necessarily shipped in *django-sendsms* but can be deployed in any other location.

```
OPENWISP_RADIUS_SMS_TOKEN_DEFAULT_VALIDITY
```

Default: `30`

For how many minutes the SMS token is valid for.

```
OPENWISP_RADIUS_SMS_TOKEN_LENGTH
```

Default: `6`

The length of the SMS token.

```
OPENWISP_RADIUS_SMS_TOKEN_HASH_ALGORITHM
```

Default: `'sha256'`

The hashing algorithm used to generate the numeric code.

```
OPENWISP_RADIUS_SMS_COOLDOWN
```

Default: `30`

Seconds users needs to wait before being able to request a new SMS token.

```
OPENWISP_RADIUS_SMS_TOKEN_MAX_ATTEMPTS
```

Default: `5`

The max number of mistakes tolerated during verification, after this amount of mistaken attempts, it won't be possible to verify the token anymore and it will be necessary to request a new one.

```
OPENWISP_RADIUS_SMS_TOKEN_MAX_USER_DAILY
```

Default: `5`

The max number of SMS tokens a single user can request within a day.

```
OPENWISP_RADIUS_SMS_TOKEN_MAX_IP_DAILY
```

Default: `999`

The max number of tokens which can be requested from the same IP address during the same day.

```
OPENWISP_RADIUS_SMS_MESSAGE_TEMPLATE
```

Default: `{organization} verification code: {code}`

The template used for sending verification code to users via SMS.

Note

The template should always contain `{code}` placeholder. Otherwise, the sent SMS will not contain the verification code.

This value can be overridden per organization in the organization change page. You can skip setting this option if you intend to set it from organization change page for each organization. Keep in mind that the default value is translated in other languages. If the value is customized the translations will not work, so if you need this message to be translated in different languages you should either not change the default value or prepare the additional translations.

Developer Docs

Note

This page is for developers who want to customize or extend OpenWISP RADIUS, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP RADIUS User Docs

Developer Installation Instructions

Note

This page is for developers who want to customize or extend OpenWISP RADIUS, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP RADIUS User Docs

Dependencies

- Python >= 3.8

Installing for Development

Install the system dependencies:

```
sudo apt update
sudo apt install -y sqlite3 libsqlite3-dev libpq-dev
```

```
sudo apt install -y xmlsec1
sudo apt install -y chromium-browser
```

Fork and clone the forked repository:

```
git clone git://github.com/<your_fork>/openwisp-radius
```

Navigate into the cloned repository:

```
cd openwisp-radius/
```

Launch Redis:

```
docker-compose up -d redis
```

Setup and activate a virtual-environment (we'll be using virtualenv):

```
python -m virtualenv env
source env/bin/activate
```

Make sure that your base python packages are up to date before moving to the next step:

```
pip install -U pip wheel setuptools
```

Install development dependencies:

```
pip install -e .[saml,openvpn_status]
pip install -r requirements-test.txt
sudo npm install -g jshint stylelint
```

Install WebDriver for Chromium for your browser version from https://chromedriver.chromium.org/home and Extract `chromedriver` to one of directories from your `$PATH` (example: `~/.local/bin/`).

Create database:

```
cd tests/
./manage.py migrate
./manage.py createsuperuser
```

Launch celery worker (for background jobs):

```
celery -A openwisp2 worker -l info
```

Launch development server:

```
./manage.py runserver
```

You can access the admin interface at `http://127.0.0.1:8000/admin/`.

Run tests with:

```
./runtests.py --parallel
```

Run quality assurance tests with:

```
./run-qa-checks
```

Alternative Sources

Pypi

To install the latest Pypi:

```
pip install openwisp-radius
```

Github

To install the latest development version tarball via HTTPs:

```
pip install https://github.com/openwisp/openwisp-radius/tarball/master
```

Alternatively you can use the git protocol:

```
pip install -e git+git://github.com/openwisp/openwisp-radius#egg=openwisp_radius[saml,openvp
```

Migrating an existing freeradius database

If you already have a freeradius 3 database with the default schema, you should be able to use it with openwisp-radius (and extended apps) easily:

1. first of all, back up your existing database;
2. configure django to connect to your existing database;
3. fake the first migration (which only replicates the default freeradius schema) and then launch the rest of migrations normally, see the examples below to see how to do this.

```
./manage.py migrate --fake openwisp-radius 0001_initial_freeradius
./manage.py migrate
```

Troubleshooting Steps for Common Installation Issues

If you encounter any issue during installation, run:

```
pip install -e .[saml] -r requirements-test.txt
```

instead of `pip install -r requirements-test.txt`

Code Utilities

Note

This page is for developers who want to customize or extend OpenWISP RADIUS, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP RADIUS User Docs

Signals

`radius_accounting_success`

Path: `openwisp_radius.signals.radius_accounting_success`

Arguments:

- `sender` : `AccountingView`

- `accounting_data` (`dict`): accounting information
- `view`: instance of `AccountingView`

This signal is emitted every time the accounting REST API endpoint completes successfully, just before the response is returned.

The `view` argument can also be used to access the `request` object i.e. `view.request`.

Captive portal mock views

The development environment of openwisp-radius provides two URLs that mock the behavior of a captive portal, these URLs can be used when testing frontend applications like OpenWISP WiFi Login Pages during development.

> **Note**
>
> These views are meant to be used just for development and testing.

Captive Portal Login Mock View

- **URL**: `http://localhost:8000/captive-portal-mock/login/`.
- **POST fields**: `auth_pass` or `password`.

This view handles the captive portal login process by first checking for either an `auth_pass` or `password` in the POST request data. It then attempts to find a corresponding `RadiusToken` instance where the key matches the provided value. If a matching token is found and there are no active sessions (i.e., no open `RadiusAccounting` records), then it creates a new radius session for the user. If successful, the user is considered logged in.

Captive Portal Logout Mock View

- **URL**: `http://localhost:8000/captive-portal-mock/logout/`.
- **POST fields**: `logout_id`.

This view looks for an entry in the `radacct` table where `session_id` matches the value passed in the `logout_id` POST field. If such an entry is found, the view makes a `POST` request to the accounting view to mark the session as terminated, using `User-Request` as the termination cause.

Extending OpenWISP RADIUS

> **Note**
>
> This page is for developers who want to customize or extend OpenWISP RADIUS, whether for bug fixes, new features, or contributions.
>
> For user guides and general information, please see:
>
> - General OpenWISP Quickstart
> - OpenWISP RADIUS User Docs

One of the core values of the OpenWISP project is Software Reusability, for this reason *openwisp-radius* provides a set of base classes which can be imported, extended and reused to create derivative apps.

In order to implement your custom version of *openwisp-radius*, you need to perform the steps described in this section.

When in doubt, the code in the test project and the sample app will serve you as source of truth: just replicate and adapt that code to get a basic derivative of *openwisp-radius* working.

If you want to add new users fields, please follow the tutorial to extend the openwisp-users. As an example, we have extended *openwisp-users* to *sample_users* app and added a field `social_security_number` in the sample_users/models.py.

Important

If you plan on using a customized version of this module, we suggest to start with it since the beginning, because migrating your data from the default module to your extended version may be time consuming.

1. Initialize your custom module

The first thing you need to do is to create a new django app which will contain your custom version of *openwisp-radius*.

A django app is nothing more than a python package (a directory of python scripts), in the following examples we'll call this django app `myradius`, but you can name it how you want:

```
django-admin startapp myradius
```

Keep in mind that the command mentioned above must be called from a directory which is available in your PYTHON_PATH so that you can then import the result into your project.

2. Install `openwisp-radius`

Install (and add to the requirement of your project) openwisp-radius:

```
pip install openwisp-radius
```

3. Add your App to `INSTALLED_APPS`

Now you need to add `myradius` to `INSTALLED_APPS` in your `settings.py`, ensuring also that `openwisp_radius` has been removed:

```
import os

INSTALLED_APPS = [
    # ... other apps ...
    # openwisp admin theme
    "openwisp_utils.admin_theme",
    # all-auth
    "django.contrib.sites",
    "allauth",
    "allauth.account",
    "allauth.socialaccount",
    # admin
    "django.contrib.admin",
    # rest framework
    "rest_framework",
    "django_filters",
    # registration
    "rest_framework.authtoken",
    "dj_rest_auth",
    "dj_rest_auth.registration",
```

```
    # social login
    "allauth.socialaccount.providers.facebook",  # optional, can be removed if social login
    "allauth.socialaccount.providers.google",  # optional, can be removed if social login is
    # SAML login
    "djangosaml2",  # optional, can be removed if SAML login is not needed
    # openwisp
    # 'myradius', <-- replace with your app-name here
    "openwisp_users",
    "private_storage",
    "drf_yasg",
]

SITE_ID = 1
MEDIA_ROOT = os.path.join(BASE_DIR, "media")
PRIVATE_STORAGE_ROOT = os.path.join(MEDIA_ROOT, "private")

AUTHENTICATION_BACKENDS = (
    "openwisp_users.backends.UsersAuthenticationBackend",
    "openwisp_radius.saml.backends.OpenwispRadiusSaml2Backend",  # optional, can be removed
)
```

4. Add `EXTENDED_APPS`

Add the following to your `settings.py`:

```
EXTENDED_APPS = ("openwisp_radius",)
```

5. Add `openwisp_utils.staticfiles.DependencyFinder`

Add `openwisp_utils.staticfiles.DependencyFinder` to `STATICFILES_FINDERS` in your `settings.py`:

```
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
    "openwisp_utils.staticfiles.DependencyFinder",
]
```

6. Add `openwisp_utils.loaders.DependencyLoader`

Add `openwisp_utils.loaders.DependencyLoader` to `TEMPLATES` in your `settings.py`, but ensure it comes before `django.template.loaders.app_directories.Loader`:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "OPTIONS": {
            "loaders": [
                "django.template.loaders.filesystem.Loader",
                "openwisp_utils.loaders.DependencyLoader",
                "django.template.loaders.app_directories.Loader",
            ],
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
```

```
    }
]
```

7. Inherit the AppConfig class

Refer to the sample_radius/apps.py file in the sample app of the test project.

You have to replicate and adapt that code in your project.

For more information regarding the concept of `AppConfig` please refer to the "Applications" section in the django documentation.

8. Create your custom models

For the purpose of showing an example, we added a simple `details` field to the models of the sample app in the test project.

You can add fields in a similar way in your `models.py` file.

For doubts regarding how to use, extend or develop models please refer to the "Models" section in the django documentation.

9. Add swapper configurations

Once you have created the models, add the following to your `settings.py`:

```
# Setting models for swapper module
OPENWISP_RADIUS_RADIUSREPLY_MODEL = "myradius.RadiusReply"
OPENWISP_RADIUS_RADIUSGROUPREPLY_MODEL = "myradius.RadiusGroupReply"
OPENWISP_RADIUS_RADIUSCHECK_MODEL = "myradius.RadiusCheck"
OPENWISP_RADIUS_RADIUSGROUPCHECK_MODEL = "myradius.RadiusGroupCheck"
OPENWISP_RADIUS_RADIUSACCOUNTING_MODEL = "myradius.RadiusAccounting"
OPENWISP_RADIUS_NAS_MODEL = "myradius.Nas"
OPENWISP_RADIUS_RADIUSUSERGROUP_MODEL = "myradius.RadiusUserGroup"
OPENWISP_RADIUS_RADIUSPOSTAUTH_MODEL = "myradius.RadiusPostAuth"
OPENWISP_RADIUS_RADIUSBATCH_MODEL = "myradius.RadiusBatch"
OPENWISP_RADIUS_RADIUSGROUP_MODEL = "myradius.RadiusGroup"
OPENWISP_RADIUS_RADIUSTOKEN_MODEL = "myradius.RadiusToken"
OPENWISP_RADIUS_PHONETOKEN_MODEL = "myradius.PhoneToken"
OPENWISP_RADIUS_ORGANIZATIONRADIUSSETTINGS_MODEL = (
    "myradius.OrganizationRadiusSettings"
)
OPENWISP_RADIUS_REGISTEREDUSER_MODEL = "myradius.RegisteredUser"

# You will need to change AUTH_USER_MODEL if you are extending openwisp_users
AUTH_USER_MODEL = "openwisp_users.User"
```

Substitute `myradius` with the name you chose in step 1.

10. Create database migrations

Copy the migration files from the sample_radius's migration folder.

Now, create database migrations as per your custom application's requirements:

```
./manage.py makemigrations
```

If you are starting with a fresh database, you can apply the migrations:

```
./manage.py migrate
```

However, if you want migrate an existing freeradius database please read the guide in the setup.

For more information, refer to the "Migrations" section in the django documentation.

11. Create the admin

Refer to the admin.py file of the sample app.

To introduce changes to the admin, you can do it in two main ways which are described below.

For more information regarding how the django admin works, or how it can be customized, please refer to "The django admin site" section in the django documentation.

1. Monkey patching

If the changes you need to add are relatively small, you can resort to monkey patching.

For example:

```
from openwisp_radius.admin import (
    RadiusCheckAdmin,
    RadiusReplyAdmin,
    RadiusAccountingAdmin,
    NasAdmin,
    RadiusGroupAdmin,
    RadiusUserGroupAdmin,
    RadiusGroupCheckAdmin,
    RadiusGroupReplyAdmin,
    RadiusPostAuthAdmin,
    RadiusBatchAdmin,
)

# NasAdmin.fields += ['example_field'] <-- Monkey patching changes example
```

2. Inheriting admin classes

If you need to introduce significant changes and/or you don't want to resort to monkey patching, you can proceed as follows:

```
from django.contrib import admin
from openwisp_radius.admin import (
    RadiusCheckAdmin as BaseRadiusCheckAdmin,
    RadiusReplyAdmin as BaseRadiusReplyAdmin,
    RadiusAccountingAdmin as BaseRadiusAccountingAdmin,
    NasAdmin as BaseNasAdmin,
    RadiusGroupAdmin as BaseRadiusGroupAdmin,
    RadiusUserGroupAdmin as BaseRadiusUserGroupAdmin,
    RadiusGroupCheckAdmin as BaseRadiusGroupCheckAdmin,
    RadiusGroupReplyAdmin as BaseRadiusGroupReplyAdmin,
    RadiusPostAuthAdmin as BaseRadiusPostAuthAdmin,
    RadiusBatchAdmin as BaseRadiusBatchAdmin,
)
from swapper import load_model

Nas = load_model("openwisp_radius", "Nas")
RadiusAccounting = load_model("openwisp_radius", "RadiusAccounting")
RadiusBatch = load_model("openwisp_radius", "RadiusBatch")
RadiusCheck = load_model("openwisp_radius", "RadiusCheck")
RadiusGroup = load_model("openwisp_radius", "RadiusGroup")
RadiusPostAuth = load_model("openwisp_radius", "RadiusPostAuth")
RadiusReply = load_model("openwisp_radius", "RadiusReply")
PhoneToken = load_model("openwisp_radius", "PhoneToken")
RadiusGroupCheck = load_model("openwisp_radius", "RadiusGroupCheck")
```

```
RadiusGroupReply = load_model("openwisp_radius", "RadiusGroupReply")
RadiusUserGroup = load_model("openwisp_radius", "RadiusUserGroup")
OrganizationRadiusSettings = load_model(
    "openwisp_radius", "OrganizationRadiusSettings"
)
User = get_user_model()

admin.site.unregister(RadiusCheck)
admin.site.unregister(RadiusReply)
admin.site.unregister(RadiusAccounting)
admin.site.unregister(Nas)
admin.site.unregister(RadiusGroup)
admin.site.unregister(RadiusUserGroup)
admin.site.unregister(RadiusGroupCheck)
admin.site.unregister(RadiusGroupReply)
admin.site.unregister(RadiusPostAuth)
admin.site.unregister(RadiusBatch)


@admin.register(RadiusCheck)
class RadiusCheckAdmin(BaseRadiusCheckAdmin):
    pass
    # add your changes here


@admin.register(RadiusReply)
class RadiusReplyAdmin(BaseRadiusReplyAdmin):
    pass
    # add your changes here


@admin.register(RadiusAccounting)
class RadiusAccountingAdmin(BaseRadiusAccountingAdmin):
    pass
    # add your changes here


@admin.register(Nas)
class NasAdmin(BaseNasAdmin):
    pass
    # add your changes here


@admin.register(RadiusGroup)
class RadiusGroupAdmin(BaseRadiusGroupAdmin):
    pass
    # add your changes here


@admin.register(RadiusUserGroup)
class RadiusUserGroupAdmin(BaseRadiusUserGroupAdmin):
    pass
    # add your changes here


@admin.register(RadiusGroupCheck)
class RadiusGroupCheckAdmin(BaseRadiusGroupCheckAdmin):
    pass
    # add your changes here
```

```
@admin.register(RadiusGroupReply)
class RadiusGroupReplyAdmin(BaseRadiusGroupReplyAdmin):
    pass
    # add your changes here


@admin.register(RadiusPostAuth)
class RadiusPostAuthAdmin(BaseRadiusPostAuthAdmin):
    pass
    # add your changes here


@admin.register(RadiusBatch)
class RadiusBatchAdmin(BaseRadiusBatchAdmin):
    pass
    # add your changes here
```

12. Setup Freeradius API Allowed Hosts

Add allowed freeradius hosts in `settings.py`:

```
OPENWISP_RADIUS_FREERADIUS_ALLOWED_HOSTS = ["127.0.0.1"]
```

Read more about freeradius allowed hosts in settings page.

13. Setup Periodic tasks

Some periodic commands are required in production environments to enable certain features and facilitate database cleanup:

1. You need to create a celery configuration file as it's created in example file.

2. In the settings.py, configure the CELERY_BEAT_SCHEDULE. Some celery tasks take an argument, for instance `365` is given here for `delete_old_radacct` in the example settings. These arguments are passed to their respective management commands. More information about these parameters can be found at the management commands page.

3. Add the following in your settings.py file:

```
CELERY_IMPORTS = ("openwisp_radius.tasks",)
```

For more information about the usage of celery in django, please refer to the "First steps with Django" section in the celery documentation.

14. Create root URL configuration

The root `url.py` file should have the following paths (please read the comments):

```
from openwisp_radius.urls import get_urls

# Only imported when views are extended.
# from myradius.api.views import views as api_views
# from myradius.social.views import views as social_views
# from myradius.saml.views import views as saml_views

urlpatterns = [
    # ... other urls in your project ...
    path("admin/", admin.site.urls),
    # openwisp-radius urls
    path("accounts/", include("openwisp_users.accounts.urls")),
```

```
    path("api/v1/", include("openwisp_utils.api.urls")),
    # Use only when extending views (discussed below)
    # path('', include((get_urls(api_views, social_views, saml_views), 'radius'), namespace=
    # Remove when extending views
    path("", include("openwisp_radius.urls", namespace="radius")),
]
```

For more information about URL configuration in django, please refer to the "URL dispatcher" section in the django documentation.

15. Import the automated tests

When developing a custom application based on this module, it's a good idea to import and run the base tests too, so that you can be sure the changes you're introducing are not breaking some of the existing features of *openwisp-radius*.

In case you need to add breaking changes, you can overwrite the tests defined in the base classes to test your own behavior.

See the tests of the sample app to find out how to do this.

You can then run tests with:

```
# the --parallel flag is optional
./manage.py test --parallel myradius
```

Substitute `myradius` with the name you chose in step 1.

Other base classes that can be inherited and extended

The following steps are not required and are intended for more advanced customization.

1. Extending the API Views

The API view classes can be extended into other django applications as well. Note that it is not required for extending *openwisp-radius* to your app and this change is required only if you plan to make changes to the API views.

Create a view file as done in API views.py.

Remember to use these views in root URL configurations in point 14. If you want only extend the API views and not social views, you can use `get_urls(api_views, None)` to get social_views from *openwisp_radius*.

For more information about django views, please refer to the views section in the django documentation.

2. Extending the Social Views

The social view classes can be extended into other django applications as well. Note that it is not required for extending *openwisp-radius* to your app and this change is required only if you plan to make changes to the social views.

Create a view file as done in social views.py.

Remember to use these views in root URL configurations in point 14. If you want only extend the API views and not social views, you can use `get_urls(api_views, None)` to get social_views from *openwisp_radius*.

3. Extending the SAML Views

The SAML view classes can be extended into other django applications as well. Note that it is not required for extending *openwisp-radius* to your app and this change is required only if you plan to make changes to the SAML views.

Create a view file as done in saml views.py.

Remember to use these views in root URL configurations in point 14. If you want only extend the API views and social view but not SAML views, you can use `get_urls(api_views, social_views, None)` to get saml_views from *openwisp_radius*.

For more information about django views, please refer to the views section in the django documentation.

Other useful resources:

- REST API Reference
- Settings

Deploy instructions

See Enabling the RADIUS module on the OpenWISP ansible role documentation.

Alternatively you can set it up manually by following these guides:

Freeradius Setup for Captive Portal authentication

This guide explains how to install and configure freeradius 3 in order to make it work with OpenWISP RADIUS for Captive Portal authentication.

The guide is written for debian based systems, other linux distributions can work as well but the name of packages and files may be different.

Widely used solutions used with OpenWISP RADIUS are PfSense and Coova-Chilli, but other solutions can be used as well.

Note

Before users can authenticate through a captive portal, they will most likely need to sign up through a web page, or alternatively, they will need to perform social login or some other kind of Single Sign On (SSO).

The OpenWISP WiFi Login Pages web app is an open source solution which integrates with OpenWISP RADIUS to provide features like self user registration, social login, SSO/SAML login, SMS verification, simple username & password login using the Radius User Token method.

For more information see: OpenWISP WiFi Login Pages

How to install freeradius 3

First of all, become root:

```
sudo -s
```

In order to **install a recent version of FreeRADIUS**, we recommend using the freeradius packages provided by NetworkRADIUS.

After having updated the APT sources list to pull the NetworkRADIUS packages, let's proceed to update the list of available packages:

```
apt update
```

These packages are always needed:

```
apt install freeradius freeradius-rest
```

If you use MySQL:

```
apt install freeradius-mysql
```

If you use PostgreSQL:

```
apt install freeradius-postgresql
```

Warning

You have to install and configure an SQL database like PostgreSQL, MySQL (SQLite can also work, but we won't treat it here) and make sure both OpenWISP RADIUS and Freeradius point to it.

The steps outlined above may not be sufficient to get the DB of your choice to run, please consult the documentation of your database of choice for more information on how to get it to run properly.

In the rest of this document we will mention PostgreSQL often because that is the database generally preferred by the Django community.

Configuring Freeradius 3

For a complete reference on how to configure freeradius please read the Freeradius wiki, configuration files and their configuration tutorial.

Note

The path to freeradius configuration could be different on your system. This article use the `/etc/freeradius/` directory that ships with recent debian distributions and its derivatives

Refer to the mods-available documentation for the available configuration values.

Enable the configured modules

First of all enable the `rest` and optionally the `sql` module:

```
ln -s /etc/freeradius/mods-available/rest /etc/freeradius/mods-enabled/rest
# optional
ln -s /etc/freeradius/mods-available/sql /etc/freeradius/mods-enabled/sql
```

Configure the REST module

Configure the rest module by editing the file `/etc/freeradius/mods-enabled/rest`, substituting `<url>` with your django project's URL, (for example, if you are testing a development environment, the URL could be `http://127.0.0.1:8000`, otherwise in production could be something like `https://openwisp2.mydomain.org`)-

Warning

Remember you need to add your freeradius server IP address in openwisp freeradius allowed hosts settings. If the freeradius server IP is not in allowed hosts, all requests to openwisp radius API will return `403`.

Refer to the rest module documentation for the available configuration values.

```
# /etc/freeradius/mods-enabled/rest

connect_uri = "<url>"
```

```
authorize {
    uri = "${..connect_uri}/api/v1/freeradius/authorize/"
    method = 'post'
    body = 'json'
    data = '{"username": "%{User-Name}", "password": "%{User-Password}"}'
    tls = ${..tls}
}

# this section can be left empty
authenticate {}

post-auth {
    uri = "${..connect_uri}/api/v1/freeradius/postauth/"
    method = 'post'
    body = 'json'
    data = '{"username": "%{User-Name}", "password": "%{User-Password}", "reply": "%{reply:P
    tls = ${..tls}
}

accounting {
    uri = "${..connect_uri}/api/v1/freeradius/accounting/"
    method = 'post'
    body = 'json'
    data = '{"status_type": "%{Acct-Status-Type}", "session_id": "%{Acct-Session-Id}", "uniq
    tls = ${..tls}
}
```

Configure the SQL module

Note

The `sql` module is not extremely needed but we treat it here since it can be useful to implement custom behavior, moreover we treat it in this document also to show that OpenWISP RADIUS can integrate itself with other widely used FreeRADIUS modules.

Once you have configured properly an SQL server, e.g. PostgreSQL:, and you can connect with a username and password edit the file `/etc/freeradius/mods-available/sql` to configure Freeradius to use the relational database.

Change the configuration for `driver`, `dialect`, `server`, `port`, `login`, `password`, `radius_db` as you need to fit your SQL server configuration.

Refer to the sql module documentation for the available configuration values.

Example configuration using the PostgreSQL database:

```
# /etc/freeradius/mods-available/sql

driver = "rlm_sql_postgresql"
dialect = "postgresql"

# Connection info:
server = "localhost"
port = 5432
login = "<user>"
password = "<password>"
radius_db = "radius"
```

Configure the site

This section explains how to configure the FreeRADIUS site.

Please refer to FreeRADIUS API Authentication to understand the different possibilities with which FreeRADIUS can authenticate requests going to OpenWISP RADIUS so that OpenWISP RADIUS knows to which organization each request belongs.

If you are **not** using the method described in Radius User Token, you have to do the following:

- create one FreeRADIUS site for each organization
- uncomment the line which starts with `# api_token_header`
- substitute the occurrences of `<org_uuid>` and `<org_radius_api_token>` with the UUID & RADIUS API token of each organization, refer to the section Organization UUID & RADIUS API Token for finding these values.

If you are deploying a captive portal setup and can use the RADIUS User Token method, you can get away with having only one freeradius site for all the organizations and can simply copy the configuration shown below.

```
# /etc/freeradius/sites-enabled/default
# Remove `#` symbol from the line to uncomment it

server default {
    # if you are not using Radius Token authentication method, please uncomment
    # and set the values for <org_uuid> & <org_radius_api_token>
    # api_token_header = "Authorization: Bearer <org_uuid> <org_radius_api_token>"

    authorize {
        # if you are not using Radius Token authentication method, please uncomment the foll
        # update control { &REST-HTTP-Header += "${...api_token_header}" }
        rest
    }

    # this section can be left empty
    authenticate {}

    post-auth {
        # if you are not using Radius Token authentication method, please uncomment the foll
        # update control { &REST-HTTP-Header += "${...api_token_header}" }
        rest

        Post-Auth-Type REJECT {
            # if you are not using Radius Token authentication method, please uncomment the
            # update control { &REST-HTTP-Header += "${....api_token_header}" }
            rest
        }
    }

    accounting {
        # if you are not using Radius Token authentication method, please uncomment the foll
        # update control { &REST-HTTP-Header += "${...api_token_header}" }
        rest
    }
}
```

Please also ensure that `acct_unique` is present in the `pre-accounting` section:

```
preacct {
    # ...
    acct_unique
    # ...
}
```

Restart freeradius to make the configuration effective

Restart freeradius to load the new configuration:

```
service freeradius restart
# alternatively if you are using systemd
systemctl restart freeradius
```

In case of errors you can run freeradius in debug mode by running `freeradius -X` in order to find out the reason of the failure.

A common problem, especially during development and testing, is that the openwisp-radius application may not be running, in that case you can find out how to run the django development server in the Developer Installation Instructions section.

Also make sure that this server runs on the port specified in `/etc/freeradius/mods-enabled/rest`.

You may also want to take a look at the Freeradius documentation for further information that is freeradius specific.

Reconfigure the development environment using PostgreSQL

You'll have to reconfigure the development environment as well before being able to use openwisp-radius for managing the freeradius databases.

If you have installed for development, create a file `tests/local_settings.py` and add the following code to configure the database:

```
# openwisp-radius/tests/local_settings.py
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql_psycopg2",
        "NAME": "<db_name>",
        "USER": "<db_user>",
        "PASSWORD": "<db_password>",
        "HOST": "127.0.0.1",
        "PORT": "5432",
    },
}
```

Make sure the database by the name <db_name> is created and also the role <db_user> with <db_password> as password.

Using Radius Checks for Authorization Information

Traditionally, when using an SQL backend with Freeradius, user authorization information such as User-Name and "known good" password can be stored using the *radcheck* table provided by Freeradius' default SQL schema.

OpenWISP RADIUS instead uses the FreeRADIUS rlm_rest module in order to take advantage of the built in user management and authentication capabilities of Django (for more information about these topics see Configure the REST module and User authentication in Django).

When migrating from existing FreeRADIUS deployments or in cases where it is preferred to use the FreeRADIUS *radcheck* table for storing user credentials it is possible to utilize rlm_sql in parallel with (or instead of) rlm_rest for authorization.

Note

Bypassing the REST API of openwisp-radius means that you will have to manually create the radius check entries for each user you want to authenticate with FreeRADIUS.

Configuration

To configure support for accessing user credentials with Radius Checks ensure the `authorize` section of your site as follows contains the `sql` module:

```
# /etc/freeradius/sites-available/default

authorize {
    # ...
    sql  # <-- the sql module
    # ...
}
```

Debugging & Troubleshooting

In this section we will explain how to debug your freeradius instance.

Start freeradius in debug mode

When debugging we suggest you to open up a dedicated terminal window to run freeradius in debug mode:

```
# we need to stop the main freeradius process first
service freeradius stop
# alternatively if you are using systemd
systemctl stop freeradius
# launch freeradius in debug mode
freeradius -X
```

Testing authentication and authorization

You can do this with `radtest`:

```
# radtest <username> <password> <host> 10 <secret>
radtest admin admin localhost 10 testing123
```

A successful authentication will return similar output:

```
Sent Access-Request Id 215 from 0.0.0.0:34869 to 127.0.0.1:1812 length 75
    User-Name = "admin"
    User-Password = "admin"
    NAS-IP-Address = 127.0.0.1
    NAS-Port = 10
    Message-Authenticator = 0x00
    Cleartext-Password = "admin"
Received Access-Accept Id 215 from 127.0.0.1:1812 to 0.0.0.0:0 length 20
```

While an unsuccessful one will look like the following:

```
Sent Access-Request Id 85 from 0.0.0.0:51665 to 127.0.0.1:1812 length 73
    User-Name = "foo"
    User-Password = "bar"
    NAS-IP-Address = 127.0.0.1
    NAS-Port = 10
    Message-Authenticator = 0x00
    Cleartext-Password = "bar"
Received Access-Reject Id 85 from 127.0.0.1:1812 to 0.0.0.0:0 length 20
(0) -: Expected Access-Accept got Access-Reject
```

Alternatively, you can use `radclient` which allows more complex tests; in the following example we show how to test an authentication request which includes `Called-Station-ID` and `Calling-Station-ID`:

```
user="foo"
pass="bar"
called="00-11-22-33-44-55:localhost"
calling="00:11:22:33:44:55"
request="User-Name=$user,User-Password=$pass,Called-Station-ID=$called,Calling-Station-ID=$c
echo $request | radclient localhost auth testing123
```

Testing accounting

You can do this with `radclient`, but first of all you will have to create a text file like the following one:

```
# /tmp/accounting.txt

Acct-Session-Id = "35000006"
User-Name = "jim"
NAS-IP-Address = 172.16.64.91
NAS-Port = 1
NAS-Port-Type = Async
Acct-Status-Type = Interim-Update
Acct-Authentic = RADIUS
Service-Type = Login-User
Login-Service = Telnet
Login-IP-Host = 172.16.64.25
Acct-Delay-Time = 0
Acct-Session-Time = 261
Acct-Input-Octets = 9900909
Acct-Output-Octets = 10101010101
Called-Station-Id = 00-27-22-F3-FA-F1:hostname
Calling-Station-Id = 5c:7d:c1:72:a7:3b
```

Then you can call `radclient`:

```
radclient -f /tmp/accounting.txt -x 127.0.0.1 acct testing123
```

You should get the following output:

```
Sent Accounting-Request Id 83 from 0.0.0.0:51698 to 127.0.0.1:1813 length 154
    Acct-Session-Id = "35000006"
    User-Name = "jim"
    NAS-IP-Address = 172.16.64.91
    NAS-Port = 1
    NAS-Port-Type = Async
    Acct-Status-Type = Interim-Update
    Acct-Authentic = RADIUS
    Service-Type = Login-User
    Login-Service = Telnet
    Login-IP-Host = 172.16.64.25
    Acct-Delay-Time = 0
    Acct-Session-Time = 261
    Acct-Input-Octets = 9900909
    Acct-Output-Octets = 1511075509
    Called-Station-Id = "00-27-22-F3-FA-F1:hostname"
    Calling-Station-Id = "5c:7d:c1:72:a7:3b"
Received Accounting-Response Id 83 from 127.0.0.1:1813 to 0.0.0.0:0 length 20
```

Customizing your configuration

You can further customize your freeradius configuration and exploit the many features of freeradius but you will need to test how your configuration plays with *openwisp-radius*.

Freeradius Setup for WPA Enterprise (EAP-TTLS-PAP) authentication

This guide explains how to install and configure freeradius 3 in order to make it work with OpenWISP RADIUS for WPA Enterprise EAP-TTLS-PAP authentication.

The setup will allow users to authenticate via WiFi WPA Enterprise networks using their personal username and password of their django user accounts. Users can either be created manually via the admin interface, generated, imported from CSV, or can self register through a web page which makes use of the registration REST API (like OpenWISP WiFi Login Pages).

Prerequisites

Execute the steps explained in the following sections of the freeradius guide for captive portal authentication:

- How to install freeradius 3
- Enable the configured modules
- Configure the REST module

Then proceed with the rest of the document.

Freeradius configuration

Configure the sites

Main sites

In this scenario it is necessary to set up one FreeRADIUS site for each organization you want to support, each FreeRADIUS instance will therefore need two dedicated ports, one for authentication and one for accounting and a related inner tunnel configuration.

Let's create the site for an hypothetical organization called org-A.

Don't forget to substitute the occurrences of `<org_uuid>` and `<org_radius_api_token>` with the UUID & Radius API token of each organization, refer to the section Organization UUID & RADIUS API Token for finding these values.

```
# /etc/freeradius/sites-enabled/org_a

server org_a {
    listen {
        type = auth
        ipaddr = *
        # ensure each org has its own port
        port = 1812
        # adjust these as needed
        limit {
          max_connections = 16
          lifetime = 0
          idle_timeout = 30
        }
    }

    listen {
        ipaddr = *
        # ensure each org has its own port
        port = 1813
        type = acct
        limit {}
    }
```

```
    # IPv6 configuration skipped for brevity
    # consult the freeradius default configuration if you need
    # to add the IPv6 configuration

    # Substitute the following variables with
    # the organization UUID and RADIUS API Token
    api_token_header = "Authorization: Bearer <org_uuid> <org_radius_api_token>"

    authorize {
        eap-org_a {
           ok = return
        }

        update control { &REST-HTTP-Header += "${...api_token_header}" }
        rest
    }

    authenticate {
        Auth-Type eap-org_a {
            eap-org_a
        }
    }

    post-auth {
        update control { &REST-HTTP-Header += "${...api_token_header}" }
        rest

        Post-Auth-Type REJECT {
            update control { &REST-HTTP-Header += "${....api_token_header}" }
            rest
        }
    }

    accounting {
        update control { &REST-HTTP-Header += "${...api_token_header}" }
        rest
    }
}
```

Please also ensure that `acct_unique` is present in the `pre-accounting` section:

```
preacct {
    # ...
    acct_unique
    # ...
}
```

Inner tunnels

You will need to set up one inner tunnel for each organization too.

Following the example for a hypothetical organization named org-A:

```
# /etc/freeradius/sites-enabled/inner-tunnel

server inner-tunnel_org_a {
    listen {
        ipaddr = 127.0.0.1
        # each org will need a dedicated port for their inner tunnel
        port = 18120
        type = auth
```

```
    }

    api_token_header = "Authorization: Bearer <org_uuid> <org_radius_api_token>"

    authorize {
        filter_username
        update control { &REST-HTTP-Header += "${...api_token_header}" }
        rest

        eap-org_a {
            ok = return
        }

        expiration
        logintime

        pap
    }

    authenticate {
        Auth-Type PAP {
            pap
        }

        Auth-Type CHAP {
            chap
        }

        Auth-Type MS-CHAP {
            mschap
        }
        eap-org_a
    }

    session {}

    post-auth {
    }

    pre-proxy {}
    post-proxy {
        eap-org_a
    }
}
```

Configure the EAP modules

> **Note**
>
> Keep in mind these are basic sample configurations, once you get it working feel free to tweak it to make it more secure and fully featured.

You will need to set up one EAP module instance for each organization too.

Following the example for a hypothetical organization named org-A:

```
eap eap-org_a {
    default_eap_type = ttls
    timer_expire = 60
    ignore_unknown_eap_types = no
    cisco_accounting_username_bug = no
    max_sessions = ${max_requests}

    tls-config tls-common {
        # make sure to have a valid SSL certificate for production usage
        private_key_password = whatever
        private_key_file = /etc/ssl/private/ssl-cert-snakeoil.key
        certificate_file = /etc/ssl/certs/ssl-cert-snakeoil.pem
        ca_file = /etc/ssl/certs/ca-certificates.crt
        dh_file = ${certdir}/dh
        ca_path = ${cadir}
        cipher_list = "DEFAULT"
        cipher_server_preference = no
        ecdh_curve = "prime256v1"

        cache {
            enable = no
        }

        ocsp {
            enable = no
            override_cert_url = yes
            url = "http://127.0.0.1/ocsp/"
        }
    }

    ttls {
        tls = tls-common
        default_eap_type = pap
        copy_request_to_tunnel = yes
        use_tunneled_reply = yes
        virtual_server = "inner-tunnel_org_a"
    }
}
```

Repeating the steps for more organizations

Let's say you don't have only the hypothetical org-A in your system but more organizations, in that case you simply have to repeat the steps explained in the previous sections, substituting the occurrences of org-A with the names of the other organizations.

So if you have an organization named ACME Systems, copy the files and substitute the occurrences `org_a` with `acme_systems`.

Final steps

Once the configurations are ready, you should restart freeradius and then test/troubleshoot/debug your setup.

Implementing other EAP scenarios

Implementing other setups like EAP-TLS requires additional development effort.

OpenWISP Controller already supports x509 certificates, so it would be a matter of integrating the django-x509 module into OpenWISP RADIUS and then implement mechanisms for the users to securely download their certificates.

If you're interested in this feature, let us know via the support channels.

This module is also available in docker-openwisp although its usage is not recommended for production usage yet, unless the reader is willing to invest effort in adapting the docker images and configurations to overcome any roadblocks encountered.

WiFi Login Pages

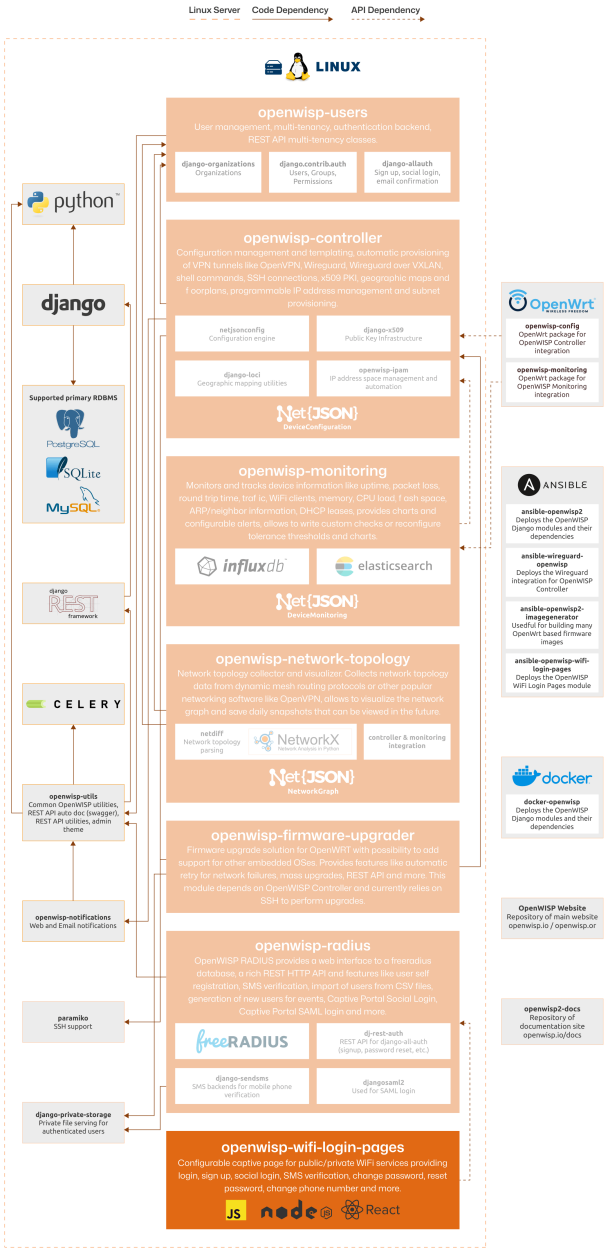
Seealso

Source code: github.com/openwisp/openwisp-wifi-login-pages.

OpenWISP WiFi login pages provides unified and consistent user experience for public/private WiFi services. This app replaces the classic captive/login page of a WiFi service by integrating the OpenWISP Radius API.

Refer to WiFi Login Pages: Features for a complete overview of features.

The following diagram illustrates the role of the WiFi Login Pages module within the OpenWISP architecture.



OpenWISP Architecture: highlighted wifi login pages module

Important

For an enhanced viewing experience, open the image above in a new browser tab.

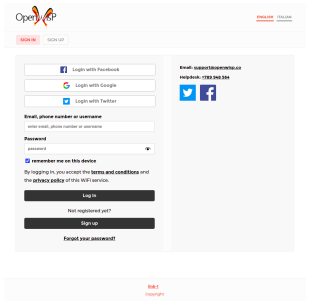
Refer to Architecture, Modules, Technologies for more information.

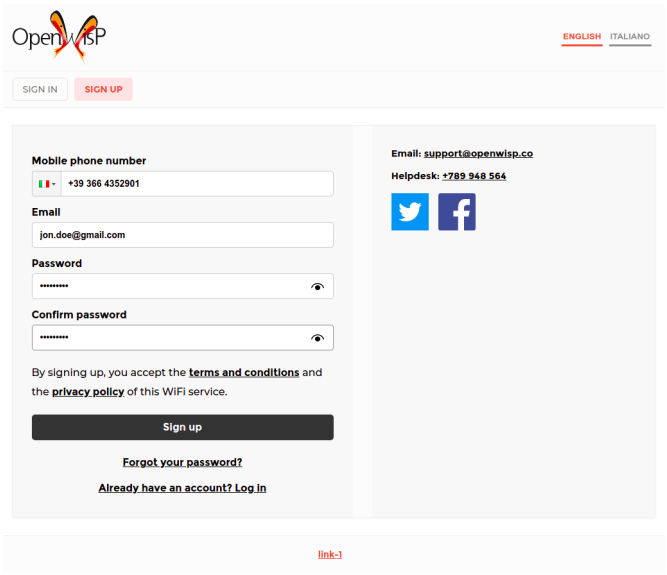
WiFi Login Pages: Features

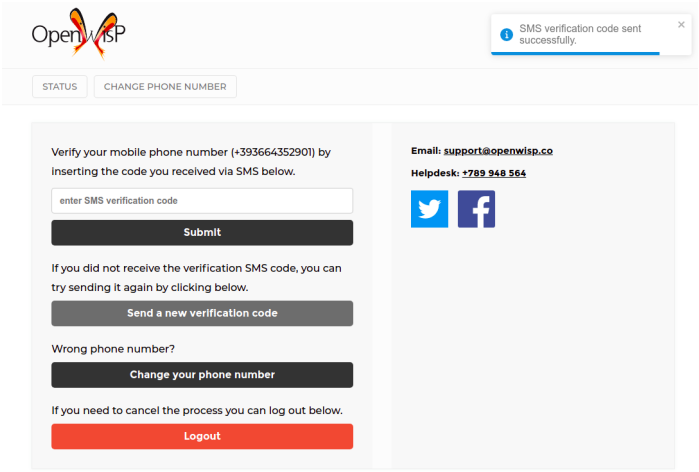
OpenWISP WiFi login pages offers the following features:

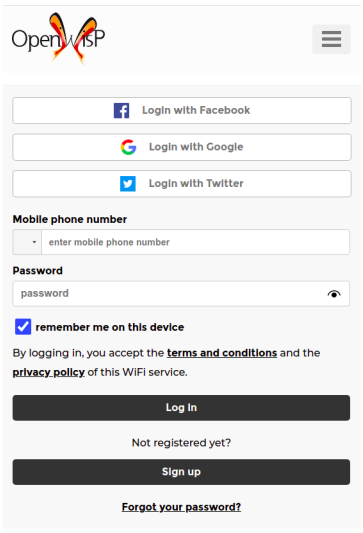
- Mobile first design (responsive UI)
- Sign up
- Optional support for mobile phone verification: verify phone number by inserting token sent via SMS, resend the SMS token
- Login to the WiFi service (by getting a radius user token from OpenWISP Radius and sending a POST to the captive portal login URL behind the scenes)
- Session status information
- Logout from the WiFi service (by sending a POST to the captive portal logout URL behind the scenes)
- Change password
- Reset password (password forgot)
- Support for Social Login and SAML
- Optional social login buttons (Facebook, Google, X/Twitter)
- Contact box showing the support email and/or phone number, as well as additional links specified via configuration
- Navigation menu (header and footer) with the possibility of specifying if links should be shown to every user or only authenticated or unauthenticated users
- Support for multiple organizations with the possibility of customizing the theme via CSS for each organization
- Support for multiple languages
- Possibility to change any text used in the pages
- Configurable Terms of Services and Privacy Policy for each organization
- Possibility of automatically logging in users who signed in previously (if the captive portal browser of their operating system supports cookies)
- Support for credit/debit card verification and paid subscription plans

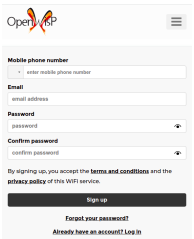
Screenshots




ENGLISH ITALIANO
SIGN IN
SIGN UP
Mobile phone number
+39 366 4352901
Email
jon.doe@gmail.com
Password
Confirm password
By signing up, you accept the terms and conditions and the privacy policy of this WiFi service.
Sign up
Forgot your password?
Already have an account? Log in
Email: support@openwisp.co
Helpdesk: +789 948 564
link-1


SMS verification code sent successfully.
STATUS
CHANGE PHONE NUMBER
Verify your mobile phone number (+393664352901) by inserting the code you received via SMS below.
enter SMS verification code
Submit
If you did not receive the verification SMS code, you can try sending it again by clicking below.
Send a new verification code
Wrong phone number?
Change your phone number
If you need to cancel the process you can log out below.
Logout
Email: support@openwisp.co
Helpdesk: +789 948 564


Login with Facebook
Login with Google
Login with Twitter
Mobile phone number
enter mobile phone number
Password
password
remember me on this device
By logging in, you accept the terms and conditions and the privacy policy of this WiFi service.
Log In
Not registered yet?
Sign up
Forgot your password?



Setup

Important

It is recommended to use the ansible-openwisp-wifi-login-pages for deploying OpenWISP WiFi Login Pages for production usage.

Add Organization configuration

Before users can login and sign up, you need to create the configuration of the captive page for the related OpenWISP organization. You can get the organization `uuid`, `slug` and `radius_secret` from the organization's admin in OpenWISP. After this, execute the following command:

```
yarn add-org
```

This command will present a series of interactive questions which make it easier for users to configure the application for their use case. It will prompt you to fill properties listed in the following table:

Property	Description
name	Required. Name of the organization.
slug	Required. Slug of the organization.
uuid	Required. UUID of the organization.
secret_key	Required. Token from organization radius settings.
captive portal login URL	Required. Captive portal login action URL
captive portal logout URL	Required. Captive portal logout action URL
openwisp radius URL	Required. URL to openwisp-radius.

Once all the questions are answered, the script will create a new directory, e.g.:

```
/organizations/{orgSlug}/
/organizations/{orgSlug}/client_assets/
/organizations/{orgSlug}/server_assets/
/organizations/{orgSlug}/{orgSlug}.yml
```

The `client_assets` directory shall contain static files like CSS, images, etc.. The `server_assets` directory is used for loading the content of Terms of Service and Privacy Policy. You can copy the desired files to these directories.

Note

The configuration of new organizations is generated from the template present in `/internals/generators/config.yml.hbs`.

The default configuration is stored at `/internals/config/default.yml`. If the configuration file of a specific organization misses a piece of configuration, then the default configuration is used to generate a complete configuration.

Use the following commands to start the project:

```
yarn setup
yarn start
```

If you need to change these values or any other settings later, you can edit the YAML file generated in the `/organizations` directory and rebuild the project.

Removing Sections of Configuration

To remove a specific section of the configuration, the `null` keyword can be used, this way the specific section flagged as `null` will be removed during the build process.

For example, to remove social login links:

```yaml
login_form:
  social_login:
    links: null
```

Note

Do not delete or edit default configuration (`/internals/config/default.yml`) as it is required to build and compile organization configurations.

Variants of the Same Configuration

In some cases it may be needed to have different variants of the same design but with different logos, or slightly different colors, wording and so on, but all these variants would be tied to the same service.

In this case it's possible to create new YAML configuration files (e.g.: `variant1.yml`, `variant2.yml`) in the directory `/organizations/{orgSlug}/`, and specify only the configuration keys which differ from the parent configuration.

Example variant of the default organization:

```yaml
---
name: "Variant1"
client:
  components:
    header:
      logo:
        url: "variant1-logo.svg"
        alternate_text: "variant1"
```

The configuration above has very little differences with the parent configuration: the name and logo are different, the rest is inherited from the parent organization.

Following example, the contents above should be placed in `/organizations/default/variant1.yml` and once the server is started again this new variant will be visible at `http://localhost:8080/default-variant1`.

It's possible to create multiple variants of different organizations, by making sure `default` is replaced with the actual organization `slug` that is being used.

And of course it's possible to customize more than just the name and logo, the example above has been kept short for brevity.

> **Note**
>
> If a variant defines a configuration option which contains an array/list of objects (e.g.: menu links), the array/list defined in the variant always overwrites fully what is defined in the parent configuration file.

Variant with Different Organization Slug / UUID / Secret

In some cases, different organizations may share an identical configuration, with very minor differences. Variants can be used also in these cases to minimize maintenance efforts.

The important thing to keep in mind is that the organization `slug`, `uuid`, `secret_key` need to be reset in the configuration file:

Example:

```
---
name: "<organization_name>"
slug: "<organization_slug>"
server:
  uuid: "<organization_uuid>"
  secret_key: "<organization_secret_key>"
client:
  css:
    - "index.css"
    - "<org-css-if-needed>"
  components:
    header:
      logo:
        url: "org-logo.svg"
        alternate_text: "..."
```

Support for Old Browsers

Polyfills are used to support old browsers on different platforms. It is recommended to add **cdnjs.cloudflare.com** to the allowed hostnames (walled garden) of the captive portal, otherwise the application will not be able to load in old browsers.

Configuring Sentry for Proxy Server

You can enable sentry logging for the proxy server by adding `sentry-env.json` in the root folder. The `sentry-env.json` file should contain configuration as following:

```
{
  ...
  "sentryTransportLogger": {
    // These options are passed to sentry SDK. Read more about available
    // options at https://github.com/aandrewww/winston-transport-sentry-node#sentry-common-o
    "sentry": {
      "dsn": "https://examplePublicKey@o0.ingest.sentry.io/0"
    },
    // Following options are related to Winston's SentryTransport. You can read
    // more at https://github.com/aandrewww/winston-transport-sentry-node#transport-related-
    "level": "warn",
    "levelsMap": {
```

```
        "silly": "debug",
        "verbose": "debug",
        "info": "info",
        "debug": "debug",
        "warn": "warning",
        "error": "error"
      }
    }
    ...
}
```

You can take reference from sentry-env.sample.json

Supporting Realms (RADIUS Proxy)

To enable support for realms, set `radius_realms` to `true` as in the example below:

```
---
name: "default name"
slug: "default"

settings:
  radius_realms: true
```

When support for `radius_realms` is `true` and the username inserted in the username field by the user includes an `@` sign, the login page will submit the credentials directly to the URL specified in `captive_portal_login_form`, hence bypassing this app altogether.

Keep in mind that in this use case, since users are basically authenticating against databases stored in other sources foreign to OpenWISP but trusted by the RADIUS configuration, the wifi-login-pages app stops making any sense, because users are registered elsewhere, do not have a local account on OpenWISP, therefore won't be able to authenticate nor change their personal details via the OpenWISP RADIUS API and this app.

Allowing Users to Manage Account from the Internet

The authentication flow might hang if a user tries to access their account from the public internet (without connecting to the WiFi service). It occurs because the **OpenWISP WiFi Login Page** waits for a response from the captive portal, which is usually inaccessible from the public internet. If your infrastructure has such a configuration then, follow the below instructions to avoid hanging of authentication flow.

Create a small web application which can serve the endpoints entered in `captive_portal_login_form.action` and `captive_portal_logout_form.action` of organization configuration.

The web application should serve the following HTML on those endpoints:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      window.parent.postMessage(
        {type: "internet-mode"},
        "https://wifi-login-pages.example.com/",
      );
    </script>
  </body>
</html>
```

> **Note**
>
> Replace `https://wifi-login-pages.example.com/` with `origin` of your **OpenWISP WiFi Login Pages** service.

Assign a dedicated DNS name to be used by both systems: the captive portal and the web application which simulates it. Then configure your captive portal to resolve this DNS name to its IP, while the public DNS resolution should point to the mock app just created. This way captive portal login and logout requests will not hang, allowing users to view/modify their account data also from the public internet.

Translations

Translations are loaded at runtime from the JSON files that were compiled during the build process according to the available languages defined and taking into account any customization of the translations.

Defining Available Languages

If there is more than one language in `i18n/` directory then update the organization configuration file by adding the support for that language like this:

```
default_language: "en"
languages:
  - text: "English"
    slug: "en"
  - text: "Italian"
    slug: "it"
```

Add Translations

Translation file with content headers can be created by running:

```
yarn translations-add {language_code} i18n/{file_name}.po
```

Here `file_name` can be `{orgSlug}_{language_code}.custom.po`, `{language_code}.custom.po\` or `{language_code}.po`.

The files created with the command above are mostly empty because when adding custom translations it is not needed to extract all the message identifiers from the code.

If instead you are adding support to a new language or updating the translations after having changed the code, you will need to extract the message identifiers, see update-translations for more information.

Update Translations

To extract or update translations in the `.po` file, use the following command:

```
yarn translations-update <path-to-po-file>
```

This will extract all the translations tags from the code and update `.po` file passed as argument.

Customizing Translations for a Specific Language

Create a translation file with name `{language_code}.custom.po` by running: `yarn translations-add <language-code> i18n/{language_code}.custom.po`

Now to override the translation placeholders (`msgid`) add the `msgstr` in the newly generated file for that specific `msgid`:

```
msgid ""
msgstr ""
"Content-Type: text/plain; charset=UTF-8\n"
"Plural-Forms: nplurals = 2; plural = (n != 1);\n"
"Language: en\n"
"MIME-Version: 1.0\n"
"Content-Transfer-Encoding: 8bit\n"

msgid "FORGOT_PASSWORD"
msgstr "Forgot password? Reset password"
```

During the build process customized language files will override all the *msgid* defined in the default language files.

Note

The custom files need not be duplicates of the default file i.e. translations can be defined for custom strings (i.e. *msgid* and *msgstr*).

Customizing Translations for a Specific Organization and Language

Create a translation file with name `{orgSlug}_{language_code}.custom.po` by running: `yarn translations-add <language-code> i18n/{orgSlug}_{language_code}.custom.po`

To override the translation placeholders (`msgid`) add the `msgstr` in the newly generated file for that specific `msgid`:

```
msgid ""
msgstr ""
"Content-Type: text/plain; charset=UTF-8\n"
"Plural-Forms: nplurals = 2; plural = (n != 1);\n"
"Language: en\n"
"MIME-Version: 1.0\n"
"Content-Transfer-Encoding: 8bit\n"

msgid "PHONE_LBL"
msgstr "mobile phone number (verification needed)"
```

During the build process custom organization language file will be used to create a JSON translation file used by that specific organization.

Note

Do not remove the content headers from the `.po` files as it is needed during the build process.

Handling Captive Portal / RADIUS Errors

This app can handle errors that may encountered during the authentication process (e.g.: maximum available daily/monthly time or bandwidth have been consumed).

To use this feature, you will have to update the error page of your captive portal to use postMessage for forwarding any error message to **OpenWISP WiFi Login Pages**.

Here is an example of authentication error page for pfSense:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      window.parent.postMessage(
        {type: "authError", message: "$PORTAL_MESSAGE$"},
        "https://wifi-login-pages.example.com/",
      );
    </script>
  </body>
</html>
```

Note

Replace `https://wifi-login-pages.example.com/` with `origin` of your **OpenWISP WiFi Login Pages** service.

With the right configuration, the error messages coming from freeradius or the captive portal will be visible to users on **OpenWISP WiFi Login Pages**.

Loading Extra JavaScript Files

It is possible to load extra javascript files, which may be needed for different reasons like error monitoring (Sentry), analytics (Matomo, Google analytics), etc.

It's possible to accomplish this in two ways which are explained below.

1. Loading Extra JavaScript Files for Whole Application (All Organizations)

Place the javascript files in `organizations/js` directory and it will be injected in HTML during the webpack build process for all the organizations.

These scripts are loaded before all the other Javascript code is loaded. This is done on purpose to ensure that any error monitoring code is loaded before everything else.

This feature should be used only for critical custom Javascript code.

2. Loading Extra JavaScript Files for a Specific Organization

Add the names of the extra javascript files in organization configuration. Example:

```
client:
  js:
    - "matomo-script.js"
    - "google-analytics.js"
```

Make sure that all these extra javascript files are be present in the `organizations/<org-slug>/client_assets` directory.

These scripts are loaded only after the rest of the page has finished loading.

This feature can be used to load non-critical custom Javascript code.

Settings

The main settings available in the organization YAML file are explained below.

Captive Portal Settings

`captive_portal_login_form`

This configuration section allows you to configure the hidden HTML form that submits the username, password, and any other required parameters to the captive portal to authenticate the user, after the credentials have been first verified via the OpenWISP REST API.

Let's take the following configuration sample for reference:

```
captive_portal_login_form:
  method: post
  action: https://captiveportal.wifiservice.com:8080/login/
  fields:
    username: username_field
    password: password_field
  additional_fields:
    - name: field1
      value: value1
    - name: field2
      value: value2
```

The example above will result in a HTML form like the following:

```
<form method="post" action="https://captiveportal.wifiservice.com:8080/login/">
  <input type="text" name="username_field" />
  <input type="password" name="password_field" />
  <input type="hidden" name="field1" value="value1" />
  <input type="hidden" name="field2" value="value2" />
</form>
```

You can adjust any parameter based on the expectations of the captive portal: most captive portal programs expect `POST` requests, although some may also accept `GET`. The input names for `username` and `password` may vary and will likely require customization.

For instance, PfSense expects `auth_user` and `auth_pass`, while Coova-Chilli expects `username` and `password`.

The `additional_fields` section allows you to specify any additional fields required by the captive portal. For instance, with PfSense, you need to include an extra field called `zone`, because PfSense allows defining multiple “Captive Portal Zones” with different configurations.

If you don't require any additional fields, simply set this section to an empty array `[]`, e.g.:

```
additional_fields: []
```

captive_portal_logout_form

This configuration section allows you to configure captive portal logout mechanism that allows users to close their browsing session.

Let's take the following configuration sample for reference:

```
captive_portal_logout_form:
  method: post
  action: https://captiveportal.wifiservice.com:8080/logout/
  fields:
    id: logout_id
  additional_fields:
    - name: field1
      value: value1
    - name: field2
      value: value2
```

The example above will result in a HTML form like the following:

```
<form method="post" action="https://captiveportal.wifiservice.com:8080/logout/">
  <input type="text" name="logout_id" value="{{ session_id }}" />
  <input type="hidden" name="field1" value="value1" />
  <input type="hidden" name="field2" value="value2" />
</form>
```

In the example above, `{{ session_id }}` represents the ID of the RADIUS session. This value is provided by WiFi Login Pages and retrieved via the OpenWISP RADIUS REST API. Some captive portals, like PfSense, require this information to complete the logout process successfully.

You can adjust any other parameter based on the expectations of the captive portal: most captive portal programs expect `POST` requests, although some may also accept `GET`.

```
additional_fields: []
```

Menu Items

By default, menu items are visible to any user, but it's possible to configure some items to be visible only to authenticated users, unauthenticated users, verified users, unverified users or users registered with specific registration methods by specifying the `authenticated`, `verified`, `methods_only` and `methods_excluded` properties.

- `authenticated: true` means visible only to authenticated users.
- `authenticated: false` means visible only to unauthenticated users.
- `verified: true` means visible to authenticated and verified users.
- `verified: false` means visible to only authenticated and unverified users.
- `methods_only: ["mobile_phone"]` means visible only to users registered with mobile phone verification.
- `methods_excluded: ["saml", "social_login"]` means not visible to users which sign in using SAML and social login.
- unspecified: link will be visible to any user (default behavior)

Let us consider the following configuration for the header, footer and contact components:

```
components:
  header:
    links:
      - text:
```

```
          en: "about"
        url: "/about"
      - text:
          en: "sign up"
        url: "/default/registration"
        authenticated: false
      - text:
          en: "change password"
        url: "/change-password"
        authenticated: true
        # if organization supports any verification method
        verified: true
        methods_excluded:
          - saml
          - social_login
      # if organization supports mobile verification
      - text:
          en: "change phone number"
        url: "/mobile/change-phone-number"
        authenticated: true
        methods_only:
          - mobile_phone
  footer:
    links:
      - text:
          en: "about"
        url: "/about"
      - text:
          en: "status"
        url: "/status"
        authenticated: true
  contact_page:
    social_links:
      - text:
          en: "support"
        url: "/support"
      - text:
          en: "twitter"
        url: "https://twitter.com/openwisp"
        authenticated: true
```

With the configuration above:

- `support` (from Contact) and `about` (from Header and Footer) links will be visible to any user.
- `sign up` (from Header) link will be visible to only unauthenticated users.
- the link to `twitter` (from Contact) and `change password` (from Header) links will be visible to only authenticated users
- change password will not be visible to users which sign in with social login or single sign-on (SAML)
- change mobile phone number will only be visible to users which have signed up with mobile phone verification

Notes:

- `methods_only` and `methods_excluded` only make sense for links which are visible to authenticated users
- using both `methods_excluded` and `methods_only` on the same link does not make sense

User Fields in Registration Form

The `setting` attribute of the fields `first_name`, `last_name`, `location` and `birth_date` can be used to indicate whether the fields shall be disabled (the default setting), allowed but not required or required.

The `setting` option can take any of the following values:

- `disabled`: (**the default value**) fields with this setting won't be shown.
- `allowed`: fields with this setting are shown but not required.
- `mandatory`: fields with this setting are shown and required.

Keep in mind that this configuration must mirror the configuration of openwisp-radius (OPENWISP_RADIUS_OPTIONAL_REGISTRATION_FIELDS).

Username Field in Login Form

The username field in the login form is automatically set to either a phone number input or an email text input depending on whether `mobile_phone_verification` is enabled or not.

However, it is possible to force the use of a standard text field if needed, for example, we may need to configure the username field to accept any value so that the OpenWISP Users Authentication Backend can then figure out if the value passed is a phone number, an email or a username:

```
login_form:
  input_fields:
    username:
      auto_switch_phone_input: false
      type: "text"
      pattern: null
```

Configuring Social Login

In order to enable users to log via third-party services like Google and Facebook, the Social Login feature of OpenWISP Radius must be configured and enabled.

Custom CSS Files

It's possible to specify multiple CSS files if needed.

```
client:
  css:
    - "index.css"
    - "custom.css"
```

Adding multiple CSS files can be useful when working with variants.

Custom HTML

It is possible to inject custom HTML in different languages in several parts of the application if needed.

Second Logo

```
header:
  logo:
    url: "logo1.png"
    alternate_text: "logo1"
  second_logo:
    url: "logo2.png"
    alternate_text: "logo2"
```

Sticky Message

```
header:
  sticky_html:
    en: >
      <p class="announcement">
        This site will go in schedule maintenance
        <b>tonight (10pm - 11pm)</b>
      </p>
```

Login Page

```
login_form:
  intro_html:
    en: >
      <div class="pre">
        Shown before the main content in the login page.
      </div>
  pre_html:
    en: >
      <div class="intro">
        Shown at the beginning of the login content box.
      </div>
  help_html:
    en: >
      <div class="intro">
        Shown above the login form, after social login buttons.
        Can be used to write custom help labels.
      </div>
  after_html:
    en: >
      <div class="intro">
        Shown at the end of the login content box.
      </div>
```

Contact Box

```
contact_page:
  pre_html:
    en: >
      <div class="contact">
        Shown at the beginning of the contact box.
      </div>
  after_html:
    en: >
      <div class="contact">
        Shown at the end of the contact box.
      </div>
```

Footer

```
footer:
  after_html:
    en: >
      <div class="contact">
        Shown at the bottom of the footer.
        Can be used to display copyright information, links to cookie policy, etc.
      </div>
```

Configuring SAML Login & Logout

To enable SAML login, the SAML feature of OpenWISP RADIUS must be enabled.

The only additional configuration needed is `saml_logout_url`, which is needed to perform SAML logout.

```
status_page:
  # other conf
  saml_logout_url: "https://openwisp.myservice.org/radius/saml2/logout/"
```

TOS & Privacy Policy

The terms of services and privacy policy pages are generated from markdown files which are specified in the YAML configuration.

The markdown files specified in the YAML configuration should be placed in: `/organizations/{orgSlug}/server_assets/`.

Configuring Logging

There are certain environment variables used to configure server logging. The details of environment variables to configure logging are mentioned below:

Environment Variable	Detail
LOG_LEVEL	(optional) This can be used to set the level of logging. The available values are `error`, `warn`, `info`, `http`, `verbose`, `debug` and `silly`. By default log level is set to `warn` for production.
ALL_LOG_FILE	(optional) To configure the path of the log file for all logs. The default path is `logs/all.log`
ERROR_LOG_FILE	(optional) To configure the path of the log file for error logs. The default path is `logs/error.log`
WARN_LOG_FILE	(optional) To configure the path of the log file for warn logs. The default path is `logs/warn.log`
INFO_LOG_FILE	(optional) To configure the path of the log file for info logs. The default path is `logs/info.log`
HTTP_LOG_FILE	(optional) To configure the path of the log file for http logs. The default path is `logs/http.log`
DEBUG_LOG_FILE	(optional) To configure the path of the log file for http logs. The default path is `logs/debug.log`

Mocking Captive Portal Login and Logout

During the development stage, the captive portal login and logout operations can be mocked by using the OpenWISP RADIUS captive portal mock views.

These URLs from OpenWISP RADIUS will be used by default in the development environment. The captive portal login and logout URLs and their parameters can be changed by editing the YAML configuration file of the respective organization.

Sign Up with Payment Flow

This application supports sign up with payment flows, either a one time payment, a free debit/credit card transaction for identity verification purposes or a subscription with periodic payments.

In order to work, this feature needs the premium **OpenWISP Subscriptions** module (get in touch with commercial support for more information).

Once the module mentioned above is installed and configured, in order to enable this feature, just create a new organization with the `yarn run add-org` command and answer `yes` to the following question:

Developer Docs

Note

This page is for developers who want to customize or extend OpenWISP WiFi Login Pages, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- OpenWISP WiFi Login Pages User Docs
- Deploy OpenWISP WiFi Login Pages for production

Developer Installation Instructions

Note

This page is for developers who want to customize or extend OpenWISP WiFi Login Pages, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- OpenWISP WiFi Login Pages User Docs
- Deploy OpenWISP WiFi Login Pages for production

Dependencies

- NodeJs >= 20.9.0
- NPM - Node package manager >= 10.1.0
- yarn - Yarn package manager >= 1.19.1

Prerequisites

OpenWISP RADIUS

OpenWISP WiFi Login Pages is a frontend for OpenWISP RADIUS. In order to use it, this app needs a running instance of OpenWISP RADIUS and an organization correctly configured, you can obtain this by following these steps:

- Follow the instructions to install OpenWISP RADIUS for development.
- After successfully starting the OpenWISP RADIUS server, open a browser and visit: `http://localhost:8000/admin/`, then sign in with the credentials of the `superuser` we created during the installation of `openwisp-radius`.
- Visit the change page of the organization you want to add to this module and note down the following parameters: `name`, `slug`, `uuid` and `token` (from the Organization RADIUS Settings).

Installing for Development

Fork and clone the forked repository:

```
git clone https://github.com/<your_fork>/openwisp-wifi-login-pages.git
```

Navigate into the cloned repository:

```
cd openwisp-wifi-login-pages
```

Install the dependencies:

```
yarn
```

Launch development server:

```
yarn start
```

You can access the application at http://localhost:8080/default/login/

Run tests with:

```
yarn test # headless tests
```

Running Automated Browser Tests

Prerequisites for running browser tests:

1. Gecko driver needs to be installed.
2. Having running instances of openwisp-radius and openwisp-wifi-login-pages is required.
3. `OPENWIPS_RADIUS_PATH` environment variable is needed to setup/tear down the database needed to run the browser tests. This can be set using the following command:

   ```
   export OPENWISP_RADIUS_PATH=<PATH_TO_OPENWISP_RADIUS_DIRECTORY>
   ```

4. If a virtual environment is used to run openwisp-radius then this needs to be activated before running browser tests.
5. Configuration file of `mobile` organization is needed before running `yarn start`. `mobile` organization can be created by running:

   ```
   node browser-test/create-mobile-configuration.js
   ```

6. In the test environment of openwisp-radius, the `default` organization must be present.

After doing all the prerequisites, you need to make sure OpenWISP RADIUS is running:

```
cd $OPENWISP_RADIUS_PATH
# enable python virtual environment if needed
./manage.py runserver
```

Then, in another terminal, from the root directory of this repository, you need to build this app and serve it:

```
yarn build-dev
yarn start
```

Then, in another terminal, from the root directory of this repository, you can finally run the browser based tests:

```
export OPENWISP_RADIUS_PATH=<PATH_TO_OPENWISP_RADIUS_DIRECTORY>
# enable python virtual environment if needed
yarn browser-test
```

Usage

Yarn Commands

List of yarn commands:

```
$ yarn start          # Run the app (runs both, client and server)
$ yarn setup          # Discover Organization configs and generate config.json and asset dire
$ yarn add-org        # Add new Organization configuration
$ yarn build          # Build the app
$ yarn server         # Run server
$ yarn client         # Run client
$ yarn coveralls      # Run coveralls
$ yarn coverage       # Run tests and generate coverage files
$ yarn lint           # Run ESLint
$ yarn lint:fix       # Run ESLint with automatically fix problems option
$ yarn format         # Run formatters to format the code
$ yarn test           # Run tests
$ yarn browser-test   # Run browser based selenium tests
$ yarn -- -u          # Update Jest Snapshots
```

Using Custom Ports

To start the client and/or server on a port of your liking, you must set environment variables before starting.

To run the client on port 4000 and the server on port 5000, use the following command:

```
$ CLIENT=4000 SERVER=5000 yarn start
```

You can also run the client and server commands separately:

```
$ SERVER=5000 yarn server
```

```
$ CLIENT=4000 SERVER=5000 yarn client
```

Note that you need to tell the client the server's port (unless you're using the default server port, which is 3030) so the client knows where he can find the server.

Running webpack-bundle-analyzer

This tool helps to keep the size of the JS files produced by the app in check.

Run it with:

```
yarn stats
```

Other useful resources:

- Settings

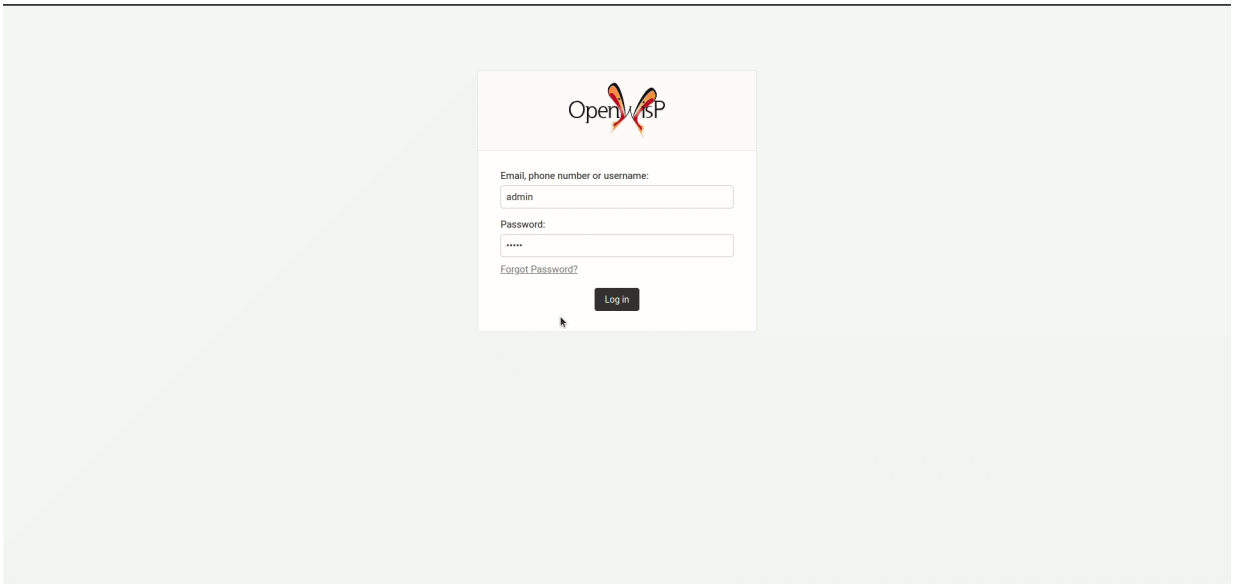
Note

For a demonstration of how this module is used, please refer to the following demo tutorial: WiFi Hotspot, Captive Portal (Public WiFi), Social Login.

IPAM

Seealso

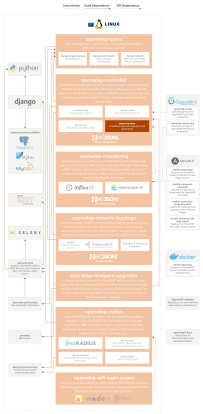
Source code: github.com/openwisp/openwisp-ipam.



OpenWISP IPAM provides IP Address Management (IPAM) features, refer to IPAM: Features for a complete overview. As a core dependency of the OpenWISP Controller, it facilitates the automatic provisioning of IP addresses for VPNs such as Wireguard and Zerotier, and allows to implement the Subnet Division Rules feature.

In addition to its integration with the OpenWISP ecosystem, OpenWISP IPAM can be used as a standalone Django app: developers proficient in Python and Django can leverage this module independently to enhance their projects, for more details on this subject please refer to the developer documentation.

The following diagram illustrates the role of the IPAM module within the OpenWISP architecture.



OpenWISP Architecture: highlighted IPAM module

Important

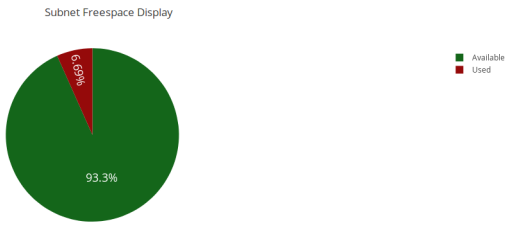
For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

IPAM: Features

OpenWISP IPAM provides the following capabilities:

- IPv4 and IPv6 IP address management
- IPv4 and IPv6 Subnet management
- CSV Import and Export of subnets and their IPs
- Automatic free space display for all subnets
- IP request module
- REST API for CRUD operations and main features
- Possibility to search for an IP or subnet
- Visual display for a specific subnet



Exporting and Importing Subnet

One can easily import and export *Subnet* data and it's Ip Addresses using *openwisp-ipam*. This works for both IPv4 and IPv6 types of networks.

Exporting

Data can be exported via the admin interface or by using a management command. The exported data is in *.csv* file format.

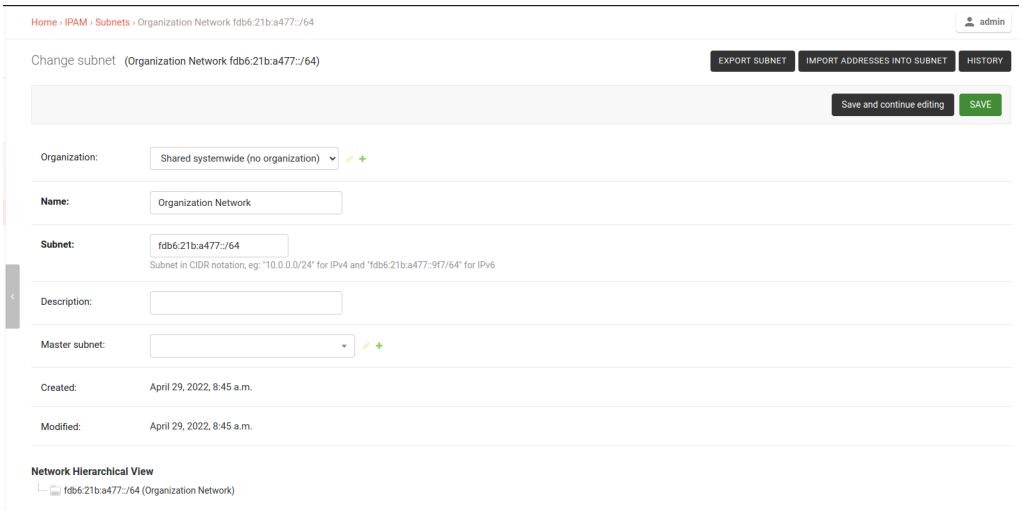
From Management Command

```
./manage.py export_subnet <subnet value>
```

This would export the subnet if it exists on the database.

From Admin Interface

Data can be exported from the admin interface by just clicking on the export button on the subnet's admin change view.



Importing

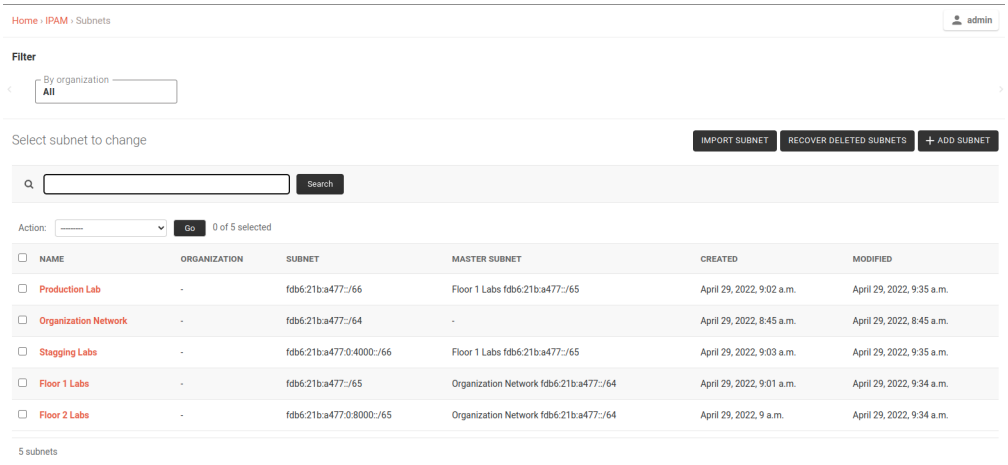
Data can be imported via the admin interface or by using a management command. The imported data file can be in *.csv* and *.xlsx* format. While importing data for ip addresses, the system checks if the subnet specified in the import file exists or not. If the subnet does not exists it will be created while importing data.

From Management Command

```
./manage.py import_subnet --file=<file path>
```

From Admin Interface

Data can be imported from the admin interface by just clicking on the import button on the subnet view.



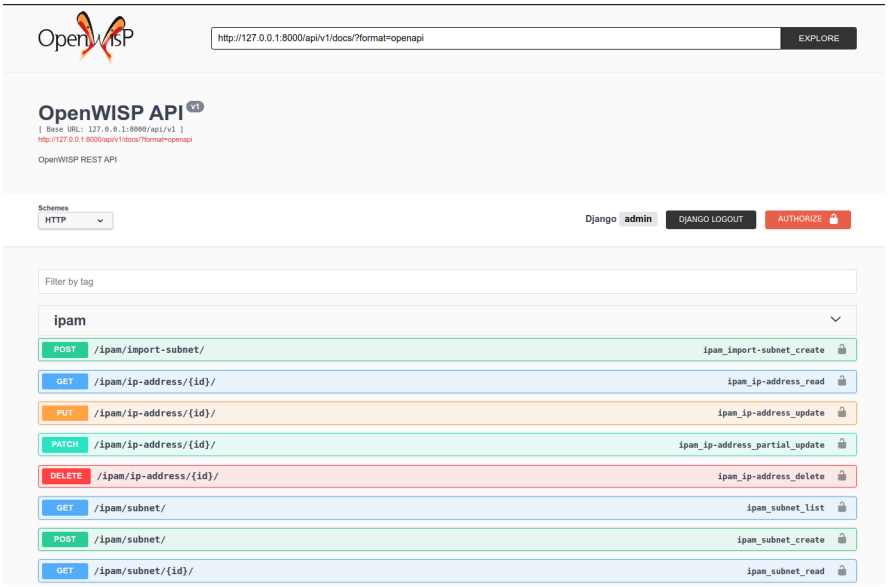
CSV File Format

Follow the following structure while creating *csv* file to import data.

```
Subnet Name
Subnet Value
Organization Slug

ip_address,description
<ip-address>,<optional-description>
<ip-address>,<optional-description>
<ip-address>,<optional-description>
```

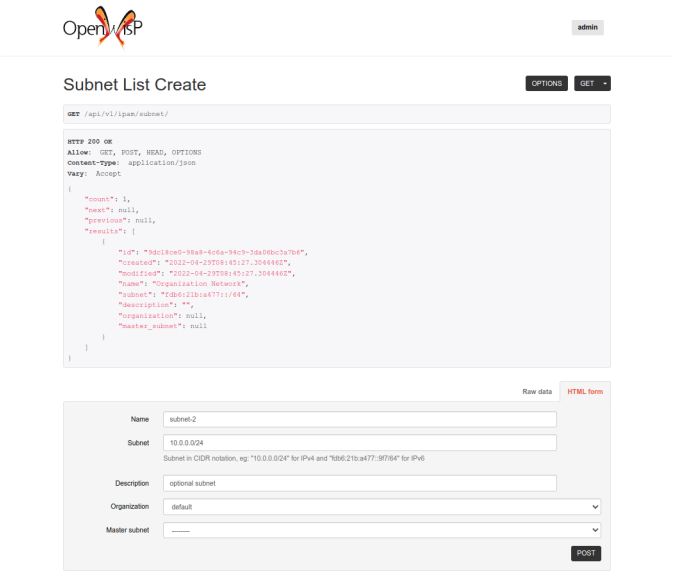
REST API

Live Documentation



A general live API documentation (following the OpenAPI specification) is available at `/api/v1/docs/`.

Browsable Web Interface



Additionally, opening any of the endpoints List of Endpoints directly in the browser will show the browsable API interface of Django-REST-Framework, which makes it even easier to find out the details of each endpoint.

Authentication

See openwisp-users: Authenticating with the User Token.

When browsing the API via the Live Documentation or the Browsable Web Interface, you can also use the session authentication by logging in the django admin.

API Throttling

To override the default API throttling settings, add the following to your `settings.py` file:

```
REST_FRAMEWORK = {
    "DEFAULT_THROTTLE_RATES": {
        "ipam": "100/hour",
    }
}
```

The rate descriptions used in `DEFAULT_THROTTLE_RATES` may include `second`, `minute`, `hour` or `day` as the throttle period.

Pagination

All *list* endpoints support the `page_size` parameter that allows paginating the results in conjunction with the `page` parameter.

```
GET /api/v1/<api endpoint url>/?page_size=10
GET /api/v1/<api endpoint url>/?page_size=10&page=2
```

List of Endpoints

Since the detailed explanation is contained in the Live Documentation and in the Browsable Web Interface of each endpoint, here we'll provide just a list of the available endpoints, for further information please open the URL of the endpoint in your browser.

Get Next Available IP

Fetch the next available IP address under a specific subnet.

GET

Returns the next available IP address under a subnet.

`/api/v1/ipam/subnet/<subnet_id>/get-next-available-ip/`

Request IP

A model method to create and fetch the next available IP address record under a subnet.

POST

Creates a record for next available IP address and returns JSON data of that record.

`POST /api/v1/ipam/subnet/<subnet_id>/request-ip/`

Param	Description
description	Optional description for the IP address

Response

```
{
    "ip_address": "ip_address",
    "subnet": "subnet_uuid",
    "description": "optional description"
}
```

Subnet IP Address List/Create

An api endpoint to retrieve or create IP addresses under a specific subnet.

GET

Returns the list of IP addresses under a particular subnet.

`/api/v1/ipam/subnet/<subnet_id>/ip-address/`

POST

Create a new `IP Address`.

`/api/v1/ipam/subnet/<subnet_id>/ip-address/`

Param	Description
ip_address	IPv6/IPv4 address value
subnet	Subnet UUID
description	Optional description for the IP address

Subnet List/Create

An api endpoint to create or retrieve the list of subnet instances.

GET

Returns the list of `Subnet` instances.

`/api/v1/ipam/subnet/`

POST

Create a new `Subnet`.

`/api/v1/ipam/subnet/`

Param	Description
subnet	Subnet value in CIDR format
master_subnet	Master Subnet UUID
description	Optional description for the IP address

Subnet Detail

An api endpoint for retrieving, updating or deleting a subnet instance.

GET

Get details of a `Subnet` instance

`/api/v1/ipam/subnet/<subnet-id>/`

DELETE

Delete a `Subnet` instance

`/api/v1/ipam/subnet/<subnet-id>/`

PUT

Update details of a `Subnet` instance.

`/api/v1/ipam/subnet/<subnet-id>/`

Param	Description
subnet	Subnet value in CIDR format
master_subnet	Master Subnet UUID
description	Optional description for the IP address

IP Address Detail

An api endpoint for retrieving, updating or deleting a IP address instance.

GET

Get details of an `IP address` instance.

`/api/v1/ipam/ip-address/<ip_address-id>/`

DELETE

Delete an `IP address` instance.

`/api/v1/ipam/ip-address/<ip_address-id>/`

PUT

Update details of an `IP address` instance.

`/api/v1/ipam/ip-address/<ip_address-id>/`

Param	Description
ip_address	IPv6/IPv4 value
subnet	Subnet UUID
description	Optional description for the IP address

Export Subnet

View to export subnet data.

POST

`/api/v1/ipam/subnet/<subnet-id>/export/`

Import Subnet

View to import subnet data.

POST

`/api/v1/ipam/import-subnet/`

Developer Docs

Note

This page is for developers who want to customize or extend OpenWISP IPAM, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP IPAM Documentation

Developer Installation Instructions

Note

This page is for developers who want to customize or extend OpenWISP IPAM, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP IPAM Documentation

Installing for Development

Install sqlite:

```
sudo apt-get install sqlite3 libsqlite3-dev openssl libssl-dev
```

Fork and clone the forked repository:

```
git clone git://github.com/<your_fork>/openwisp-ipam
```

Navigate into the cloned repository:

```
cd openwisp-ipam/
```

Setup and activate a virtual-environment (we'll be using virtualenv):

```
python -m virtualenv env
source env/bin/activate
```

Install development dependencies:

```
pip install -e .
pip install -r requirements-test.txt
```

Create database:

```
cd tests/
./manage.py migrate
./manage.py createsuperuser
```

Launch development server:

```
./manage.py runserver
```

You can access the admin interface at `http://127.0.0.1:8000/admin/`.

Run tests with:

```
# --parallel and --keepdb are optional but help to speed up the operation
./runtests.py --parallel --keepdb
```

Alternative Sources

Pypi

To install the latest Pypi:

```
pip install openwisp-ipam
```

Github

To install the latest development version tarball via HTTPs:

```
pip install https://github.com/openwisp/openwisp-ipam/tarball/master
```

Alternatively you can use the git protocol:

```
pip install -e git+git://github.com/openwisp/openwisp-ipam#egg=openwisp_ipam
```

Extending OpenWISP IPAM

Note

This page is for developers who want to customize or extend OpenWISP IPAM, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP IPAM Documentation

One of the core values of the OpenWISP project is Software Reusability, for this reason *openwisp-ipam* provides a set of base classes which can be imported, extended and reused to create derivative apps.

In order to implement your custom version of *openwisp-ipam*, you need to perform the steps described in this section.

When in doubt, the code in the test project and the sample app will serve you as source of truth: just replicate and adapt that code to get a basic derivative of *openwisp-ipam* working.

If you want to add new users fields, please follow the tutorial to extend the openwisp-users. As an example, we have extended *openwisp-users* to *sample_users* app and added a field `social_security_number` in the sample_users/models.py.

Important

If you plan on using a customized version of this module, we suggest to start with it since the beginning, because migrating your data from the default module to your extended version may be time consuming.

1. Initialize your Custom Module

The first thing you need to do is to create a new django app which will contain your custom version of *openwisp-ipam*.

A django app is nothing more than a python package (a directory of python scripts), in the following examples we'll call this django app `myipam`, but you can name it how you want:

```
django-admin startapp myipam
```

Keep in mind that the command mentioned above must be called from a directory which is available in your PYTHON_PATH so that you can then import the result into your project.

Now you need to add `myipam` to `INSTALLED_APPS` in your `settings.py`, ensuring also that `openwisp_ipam` has been removed:

```
INSTALLED_APPS = [
    # ... other apps ...
    "openwisp_utils.admin_theme",
    # all-auth
    "django.contrib.sites",
    "allauth",
    "allauth.account",
    "allauth.socialaccount",
    # openwisp2 modules
    "openwisp_users",
    # 'myipam',   <-- replace without your app-name here
    # admin
    "admin_auto_filters",
    "django.contrib.admin",
    # rest framework
    "rest_framework",
    # Other dependencies
    "reversion",
]
```

For more information about how to work with django projects and django apps, please refer to the django documentation.

2. Install `openwisp-ipam`

Install (and add to the requirements of your project) the `openwisp-ipam` python package:

```
pip install openwisp-ipam
```

3. Add `EXTENDED_APPS`

Add the following to your `settings.py`:

```
EXTENDED_APPS = ("openwisp_ipam",)
```

4. Add `openwisp_utils.staticfiles.DependencyFinder`

Add `openwisp_utils.staticfiles.DependencyFinder` to `STATICFILES_FINDERS` in your `settings.py`:

```
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
    "openwisp_utils.staticfiles.DependencyFinder",
]
```

5. Add `openwisp_utils.loaders.DependencyLoader`

Add `openwisp_utils.loaders.DependencyLoader` to `TEMPLATES` in your `settings.py`, but ensure it comes before `django.template.loaders.app_directories.Loader`:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "OPTIONS": {
            "loaders": [
                "django.template.loaders.filesystem.Loader",
                "openwisp_utils.loaders.DependencyLoader",
                "django.template.loaders.app_directories.Loader",
            ],
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    }
]
```

6. Inherit the AppConfig Class

Please refer to the following files in the sample app of the test project:

- sample_ipam/__init__.py.
- sample_ipam/apps.py.

You have to replicate and adapt that code in your project.

For more information regarding the concept of `AppConfig` please refer to the "Applications" section in the django documentation.

7. Create your Custom Models

For the purpose of showing an example, we added a simple "details" field to the models of the sample app in the test project.

You can add fields in a similar way in your `models.py` file.

Note

If you have questions about using, extending, or developing models, refer to the "Models" section of the Django documentation.

8. Add Swapper Configurations

Once you have created the models, add the following to your `settings.py`:

```
# Setting models for swapper module
OPENWISP_IPAM_IPADDRESS_MODEL = "myipam.IpAddress"
OPENWISP_IPAM_SUBNET_MODEL = "myipam.Subnet"
```

Substitute `myipam` with the name you chose in step 1.

9. Create Database Migrations

Create and apply database migrations:

```
./manage.py makemigrations
./manage.py migrate
```

For more information, refer to the "Migrations" section in the django documentation.

10. Create the Admin

Refer to the admin.py file of the sample app.

To introduce changes to the admin, you can do it in two main ways which are described below.

Note

For more information regarding how the django admin works, or how it can be customized, please refer to "The django admin site" section in the django documentation.

1. Monkey Patching

If the changes you need to add are relatively small, you can resort to monkey patching.

For example:

```
from openwisp_ipam.admin import IpAddressAdmin, SubnetAdmin

SubnetAdmin.app_label = "sample_ipam"
```

2. Inheriting Admin Classes

If you need to introduce significant changes and/or you don't want to resort to monkey patching, you can proceed as follows:

```
from django.contrib import admin
from openwisp_ipam.admin import (
    IpAddressAdmin as BaseIpAddressAdmin,
    SubnetAdmin as BaseSubnetAdmin,
```

```
)
from swapper import load_model

IpAddress = load_model("openwisp_ipam", "IpAddress")
Subnet = load_model("openwisp_ipam", "Subnet")

admin.site.unregister(IpAddress)
admin.site.unregister(Subnet)


@admin.register(IpAddress)
class IpAddressAdmin(BaseIpAddressAdmin):
    # add your changes here
    pass


@admin.register(Subnet)
class SubnetAdmin(BaseSubnetAdmin):
    app_label = "myipam"
    # add your changes here
```

Substitute `myipam` with the name you chose in step 1.

11. Create Root URL Configuration

```
from .sample_ipam import views as api_views
from openwisp_ipam.urls import get_urls

urlpatterns = [
    # ... other urls in your project ...
    # openwisp-ipam urls
    # path('', include(get_urls(api_views))) <-- Use only when changing API views (dicussed
    path("", include("openwisp_ipam.urls")),
]
```

For more information about URL configuration in django, please refer to the "URL dispatcher" section in the django documentation.

12. Import the Automated Tests

When developing a custom application based on this module, it's a good idea to import and run the base tests too, so that you can be sure the changes you're introducing are not breaking some of the existing features of *openwisp-ipam*.

In case you need to add breaking changes, you can overwrite the tests defined in the base classes to test your own behavior.

See the tests of the sample app to find out how to do this.

You can then run tests with:

```
# the --parallel flag is optional
./manage.py test --parallel myipam
```

Substitute `myipam` with the name you chose in step 1.

For more information about automated tests in django, please refer to "Testing in Django".

Other Base Classes That Can be Inherited and Extended

The following steps are not required and are intended for more advanced customization.

1. Extending the API Views

The API view classes can be extended into other django applications as well. Note that it is not required for extending openwisp-ipam to your app and this change is required only if you plan to make changes to the API views.

Create a view file as done in views.py.

For more information about django views, please refer to the views section in the django documentation.

Other useful resources:

- REST API

Notifications

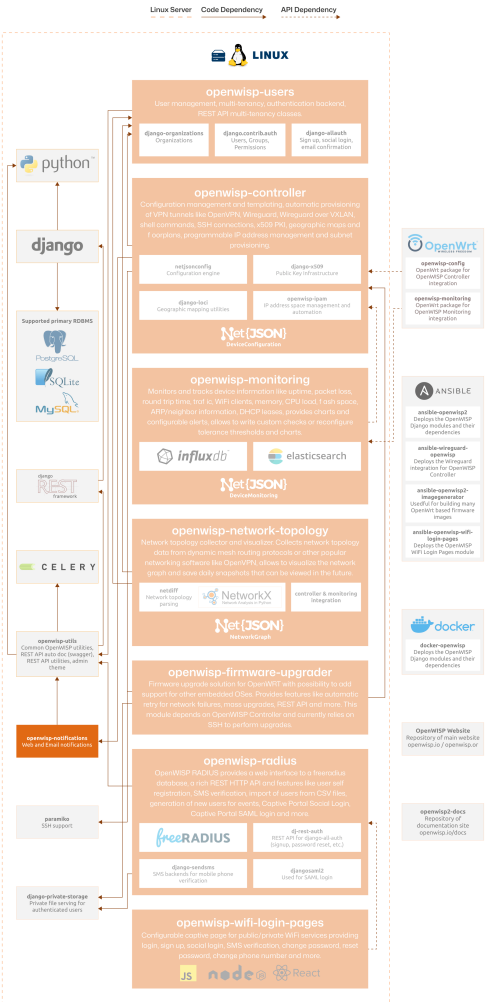
Seealso

Source code: github.com/openwisp/openwisp-notifications.

OpenWISP Notifications is a versatile system designed to deliver email and web notifications. Its primary function is to enable other OpenWISP modules to alert users about significant events occurring within their network. By seamlessly integrating with various OpenWISP components, it ensures users are promptly informed about critical updates and changes. This enhances the overall user experience by keeping network administrators aware and responsive to important developments.

For a comprehensive overview of features, please refer to the Notifications: Features page.

The following diagram illustrates the role of the Notifications module within the OpenWISP architecture.



OpenWISP Architecture: highlighted notifications module

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

Notifications: Features

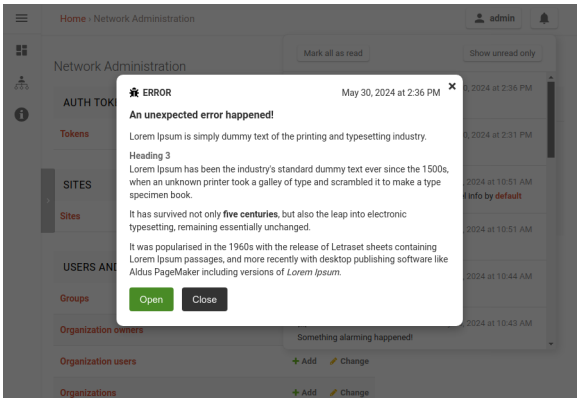
OpenWISP Notifications offers a robust set of features to keep users informed about significant events in their network. These features include:

- Sending Notifications
- Web Notifications
- Email Notifications
- Notification Types
- User notification preferences
- Silencing notifications for specific objects temporarily or permanently
- Automatic cleanup of old notifications
- Configurable host for API endpoints

Notification Types

OpenWISP Notifications allows defining notification types for recurring events. Think of a notification type as a template for notifications.

`generic_message`



This module includes a notification type called `generic_message`.

This notification type is designed to deliver custom messages in the user interface for infrequent events or errors that occur during background operations and cannot be communicated easily to the user in other ways.

These messages may require longer explanations and are therefore displayed in a dialog overlay, as shown in the screenshot above. This notification type does not send emails.

The following code example demonstrates how to send a notification of this type:

```
from openwisp_notifications.signals import notify

notify.send(
    type="generic_message",
    level="error",
    message="An unexpected error happened!",
    sender=User.objects.first(),
    target=User.objects.last(),
    description="""Lorem Ipsum is simply dummy text
of the printing and typesetting industry.

### Heading 3

Lorem Ipsum has been the industry's standard dummy text ever since the 1500s,
when an unknown printer took a galley of type and scrambled it to make a
type specimen book.

It has survived not only **five centuries**, but also the leap into
electronic typesetting, remaining essentially unchanged.

It was popularised in the 1960s with the release of Letraset sheets
containing Lorem Ipsum passages, and more recently with desktop publishing
software like Aldus PageMaker including versions of *Lorem Ipsum*.""",
)
```

Properties of Notification Types

The following properties can be configured for each notification type:

Property	Description
`level`	Sets `level` attribute of the notification.
`verb`	Sets `verb` attribute of the notification.
`verbose_name`	Sets display name of notification type.
`message`	Sets `message` attribute of the notification.
`email_subject`	Sets subject of the email notification.
`message_template`	Path to file having template for message of the notification.
`email_notification`	Sets preference for email notifications. Defaults to `True`.
`web_notification`	Sets preference for web notifications. Defaults to `True`.
`actor_link`	Overrides the default URL used for the `actor` object. You can pass a static URL or a dotted path to a callable which returns the object URL.
`action_object_link`	Overrides the default URL used for the `action` object. You can pass a static URL or a dotted path to a callable which returns the object URL.
`target_link`	Overrides the default URL used for the `target` object. You can pass a static URL or a dotted path to a callable which returns the object URL.

Note

It is recommended that a notification type configuration for recurring events contains either the `message` or `message_template` properties. If both are present, `message` is given preference over `message_template`.

If you don't plan on using `message` or `message_template`, it may be better to use the existing `generic_message` type. However, it's advised to do so only if the event being notified is infrequent.

The callable for `actor_link`, `action_object_link` and `target_link` should have the following signature:

```
def related_object_link_callable(notification, field, absolute_url=True):
    """
    notification: the notification object for which the URL will be created
    field: the related object field, any one of "actor", "action_object" or
           "target" field of the notification object
    absolute_url: boolean to flag if absolute URL should be returned
    """
    return "https://custom.domain.com/custom/url/"
```

Defining `message_template`

You can either extend default message template or write your own markdown formatted message template from scratch. An example to extend default message template is shown below.

```
# In templates/your_notifications/your_message_template.md
{% extends 'openwisp_notifications/default_message.md' %}
{% block body %}
    [{{ notification.target }}]({{ notification.target_link }}) has malfunctioned.
{% endblock body %}
```

You can access all attributes of the notification using `notification` variables in your message template as shown above. Additional attributes `actor_link`, `action_link` and `target_link` are also available for providing hyperlinks to respective object.

Important

After writing code for registering or unregistering notification types, it is recommended to run database migrations to create notification settlings for these notification types.

Sending Notifications

The `notify` signal

Notifications can be created using the `notify` signal. Here's an example which uses the generic_message notification type to alert users of an account being deactivated:

```
from django.contrib.auth import get_user_model
from swapper import load_model

from openwisp_notifications.signals import notify
```

```
User = get_user_model()
admin = User.objects.get(username="admin")
deactivated_user = User.objects.get(username="johndoe", is_active=False)

notify.send(
    sender=admin,
    type="generic_message",
    level="info",
    target=deactivated_user,
    message="{notification.actor} has deactivated {notification.target}",
)
```

The above snippet will send notifications to all superusers and organization administrators of the target object's organization who have opted-in to receive notifications. If the target object is omitted or does not have an organization, it will only send notifications to superusers.

You can override the recipients of the notification by passing the `recipient` keyword argument. The `recipient` argument can be a:

- `Group` object
- A list or queryset of `User` objects
- A single `User` object

However, these users will only be notified if they have opted-in to receive notifications.

The complete syntax for `notify` is:

```
notify.send(
    actor,
    recipient,
    verb,
    action_object,
    target,
    level,
    description,
    **kwargs,
)
```

Since `openwisp-notifications` uses `django-notifications` under the hood, usage of the `notify signal` has been kept unaffected to maintain consistency with `django-notifications`. You can learn more about accepted parameters from django-notifications documentation.

The `notify` signal supports the following additional parameters:

Parameter	Description
`type`	Set values of other parameters based on registered notification types Defaults to `None` meaning you need to provide other arguments.
`email_subject`	Sets subject of email notification to be sent. Defaults to the notification message.
`url`	Adds a URL in the email text, e.g.: `For more information see <url>.` Defaults to `None`, meaning the above message would not be added to the email text.

Passing Extra Data to Notifications

If needed, additional data, not known beforehand, can be included in the notification message.

A perfect example for this case is an error notification, the error message will vary depending on what has happened, so we cannot know until the notification is generated.

Here's how to do it:

```
from openwisp_notifications.types import register_notification_type

register_notification_type(
    "error_type",
    {
        "verbose_name": "Error",
        "level": "error",
        "verb": "error",
        "message": "Error: {error}",
        "email_subject": "Error subject: {error}",
    },
)
```

Then in the application code:

```
from openwisp_notifications.signals import notify

try:
    operation_which_can_fail()
except Exception as error:
    notify.send(type="error_type", sender=sender, error=str(error))
```

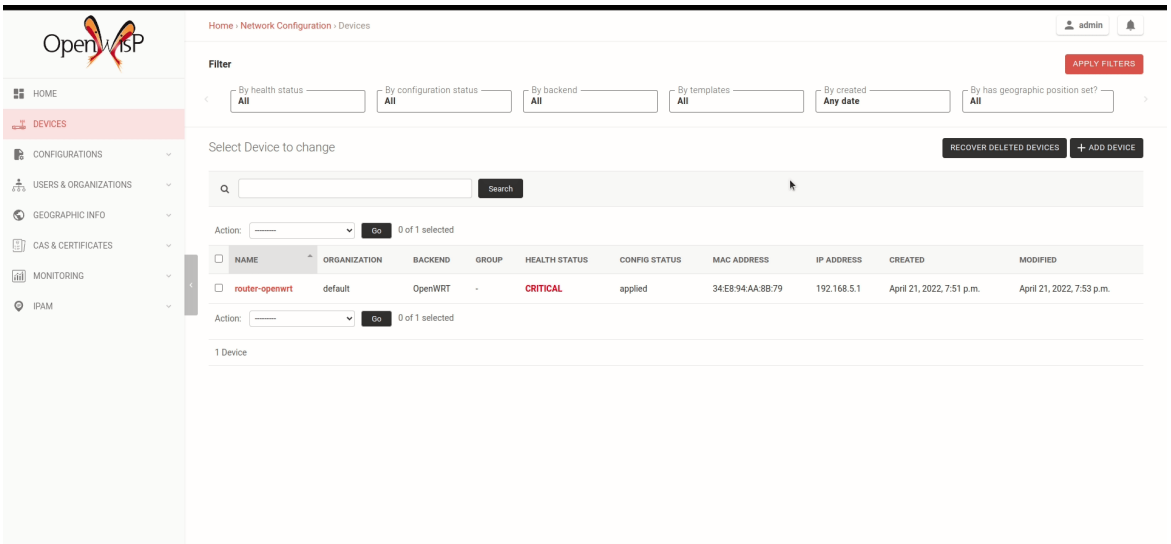
Since the `error_type` notification type defined the notification message, you don't need to pass the `message` argument in the notify signal. The message defined in the notification type will be used by the notification. The `error` argument is used to set the value of the `{error}` placeholder in the notification message.

Web & Email Notifications

Web Notifications

OpenWISP Notifications sends web notifications to recipients through Django's admin site. The following components facilitate browsing web notifications:

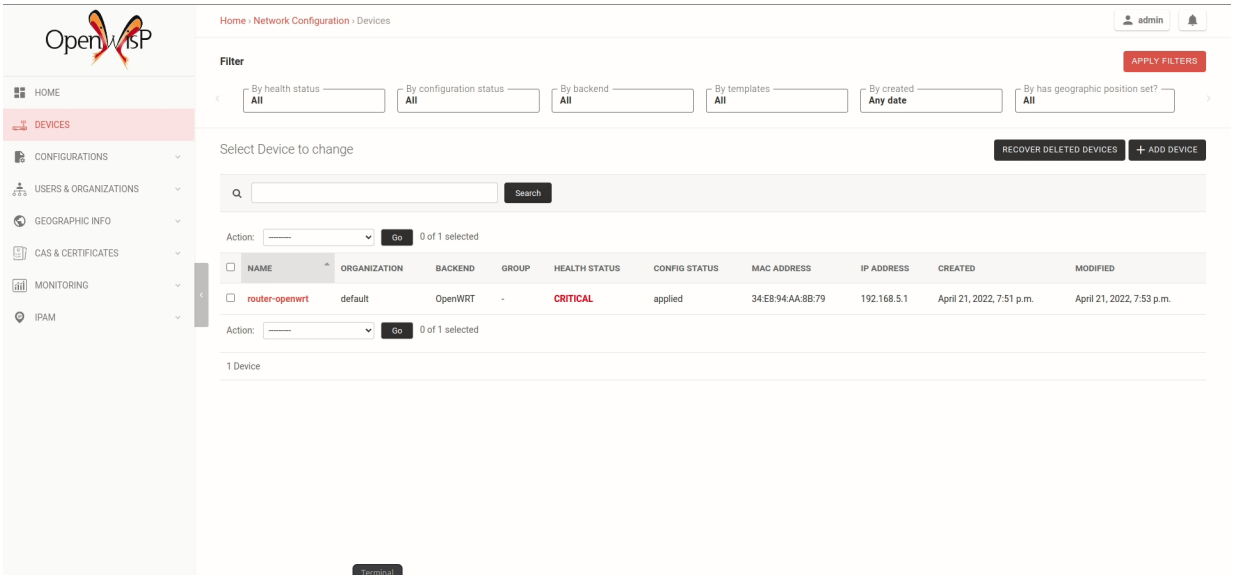
Notification Widget



A JavaScript widget has been added to make consuming notifications easy for users. The notification widget provides the following features:

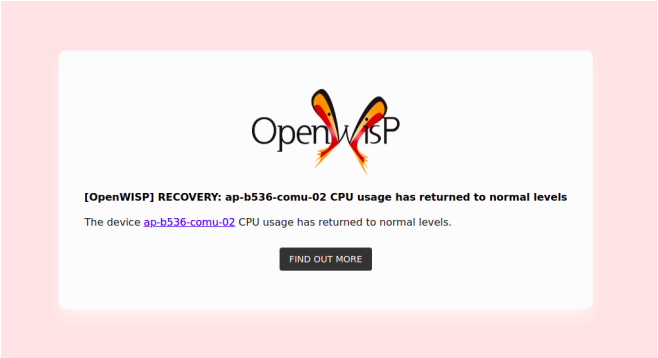
- User Interface to help users complete tasks quickly.
- Dynamically loads notifications with infinite scrolling to prevent unnecessary network requests.
- Option to filter unread notifications.
- Option to mark all notifications as read with a single click.

Notification Toasts



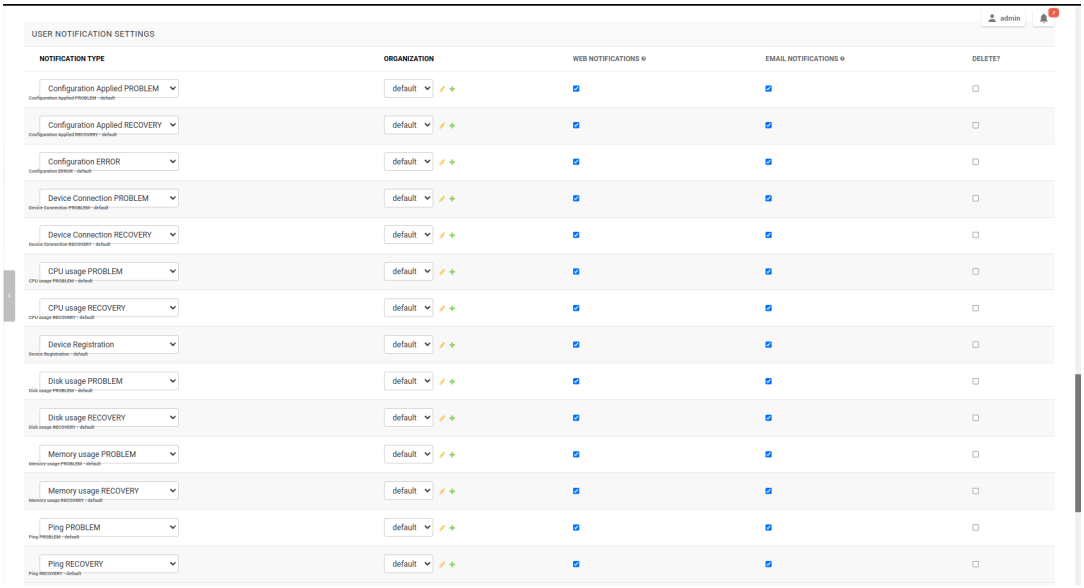
Notification toast delivers notifications in real-time, allowing users to read notifications without opening the notification widget. A notification bell sound is played each time a notification is displayed through the notification toast.

Email Notifications



Along with web notifications OpenWISP Notifications also sends email notifications leveraging the send_email feature of OpenWISP Utils.

Notification Preferences



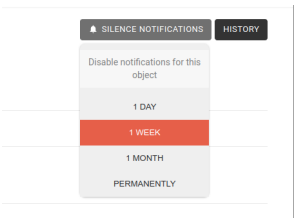
OpenWISP Notifications enables users to customize their notification preferences by selecting their preferred method of receiving updates—either through web notifications or email. These settings are organized by notification type and organization, allowing users to tailor their notification experience by opting to receive updates only from specific organizations or notification types.

Notification settings are automatically generated for all notification types and organizations for every user. Superusers have the ability to manage notification settings for all users, including adding or deleting them. Meanwhile, staff users can modify their preferred notification delivery methods, choosing between receiving notifications via web, email, or both. Additionally, users have the option to disable notifications entirely by turning off both web and email notification settings.

Note

If a user has not configured their preferences for email or web notifications for a specific notification type, the system will default to using the `email_notification` or `web_notification` option defined for that notification type.

Silencing Notifications for Specific Objects



OpenWISP Notifications allows users to silence all notifications generated by specific objects they are not interested in for a desired period of time or even permanently, while other users will keep receiving notifications normally.

Using the widget on an object's admin change form, a user can disable all notifications generated by that object for a day, week, month or permanently.

Note

This feature requires configuring "OPENWISP_NOTIFICATIONS_IGNORE_ENABLED_ADMIN" to enable the widget in the admin section of the required models.

Scheduled Deletion of Notifications

Important

If you have deployed OpenWISP using ansible-openwisp2 or docker-openwisp, then this feature has been already configured for you. Refer to the documentation of your deployment method to know the default value. This section is only for reference for users who wish to customize OpenWISP, or who have deployed OpenWISP in a different way.

OpenWISP Notifications provides a celery task to automatically delete notifications older than a preconfigured number of days. In order to run this task periodically, you will need to configure `CELERY_BEAT_SCHEDULE` in the Django project settings.

Note

If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

The celery task takes only one argument, i.e. number of days. You can provide any number of days in *args* key while configuring `CELERY_BEAT_SCHEDULE` setting.

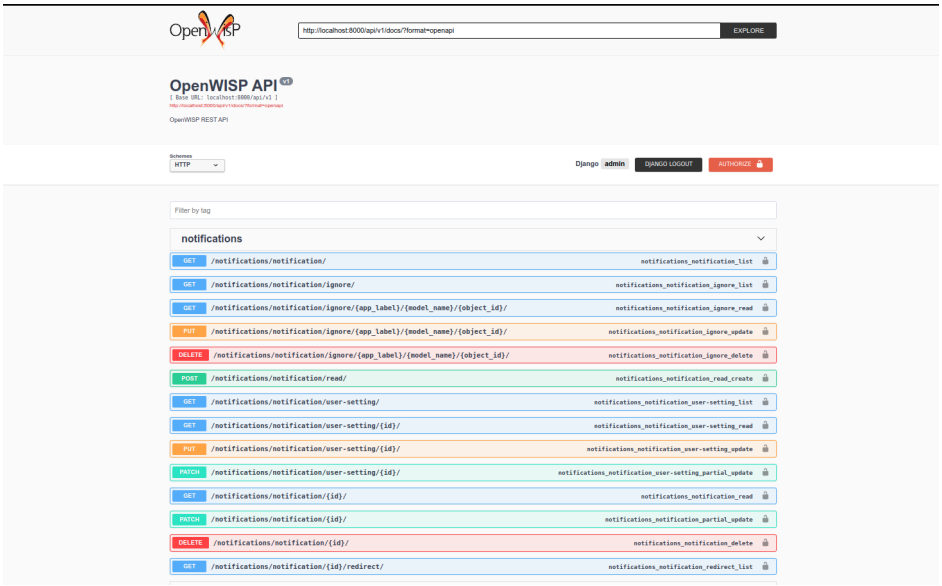
E.g., if you want notifications older than 10 days to get deleted automatically, then configure `CELERY_BEAT_SCHEDULE` as follows:

```
CELERY_BEAT_SCHEDULE.update(
    {
        "delete_old_notifications": {
            "task": "openwisp_notifications.tasks.delete_old_notifications",
            "schedule": timedelta(days=1),
            "args": (
                10,
            ),  # Here we have defined 10 instead of 90 as shown in setup instructions
        },
    }
)
```

Please refer to "Periodic Tasks" section of Celery's documentation to learn more.

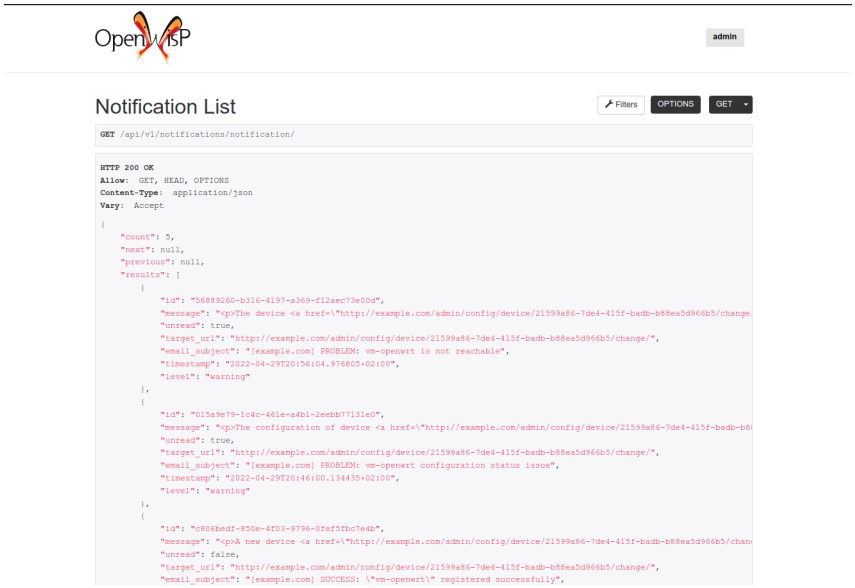
REST API

Live Documentation



A general live API documentation (following the OpenAPI specification) is available at `/api/v1/docs/`.

Browsable Web Interface



Additionally, opening any of the endpoints listed below directly in the browser will show the browsable API interface of Django-REST-Framework, which makes it even easier to find out the details of each endpoint.

Authentication

See openwisp-users: authenticating with the user token.

When browsing the API via the Live Documentation or the Browsable Web Interface, you can also use the session authentication by logging in the django admin.

Pagination

The *list* endpoint support the `page_size` parameter that allows paginating the results in conjunction with the `page` parameter.

```
GET /api/v1/notifications/notification/?page_size=10
GET /api/v1/notifications/notification/?page_size=10&page=2
```

List of Endpoints

Since the detailed explanation is contained in the Live Documentation and in the Browsable Web Interface of each point, here we'll provide just a list of the available endpoints, for further information please open the URL of the endpoint in your browser.

List User's Notifications

```
GET /api/v1/notifications/notification/
```

Available Filters

You can filter the list of notifications based on whether they are read or unread using the `unread` parameter.

To list read notifications:

```
GET /api/v1/notifications/notification/?unread=false
```

To list unread notifications:

```
GET /api/v1/notifications/notification/?unread=true
```

Mark All User's Notifications as Read

```
POST /api/v1/notifications/notification/read/
```

Get Notification Details

```
GET /api/v1/notifications/notification/{pk}/
```

Mark a Notification Read

```
PATCH /api/v1/notifications/notification/{pk}/
```

Delete a Notification

```
DELETE /api/v1/notifications/notification/{pk}/
```

List User's Notification Setting

```
GET /api/v1/notifications/notification/user-setting/
```

Available Filters

You can filter the list of user's notification setting based on their `organization_id`.

```
GET /api/v1/notifications/notification/user-setting/?organization={organization_id}
```

You can filter the list of user's notification setting based on their `organization_slug`.

```
GET /api/v1/notifications/notification/user-setting/?organization_slug={organization_slug}
```

You can filter the list of user's notification setting based on their `type`.

```
GET /api/v1/notifications/notification/user-setting/?type={type}
```

Get Notification Setting Details

```
GET /api/v1/notifications/notification/user-setting/{pk}/
```

Update Notification Setting Details

```
PATCH /api/v1/notifications/notification/user-setting/{pk}/
```

List User's Object Notification Setting

```
GET /api/v1/notifications/notification/ignore/
```

Get Object Notification Setting Details

```
GET /api/v1/notifications/notification/ignore/{app_label}/{model_name}/{object_id}/
```

Create Object Notification Setting

```
PUT /api/v1/notifications/notification/ignore/{app_label}/{model_name}/{object_id}/
```

Delete Object Notification Setting

```
DELETE /api/v1/notifications/notification/ignore/{app_label}/{model_name}/{object_id}/
```

Settings

> **Note**
>
> If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

`OPENWISP_NOTIFICATIONS_HOST`

type	`str`
default	Any domain defined in `ALLOWED_HOST`

This setting defines the domain at which API and Web Socket communicate for working of notification widget.

> **Note**
>
> You don't need to configure this setting if you don't host your API endpoints on a different sub-domain.

If your root domain is `example.com` and API and Web Socket are hosted at `api.example.com`, then configure setting as follows:

```
OPENWISP_NOTIFICATIONS_HOST = "https://api.example.com"
```

This feature requires you to allow CORS on your server. We use `django-cors-headers` module to easily setup CORS headers. Please refer django-core-headers' setup documentation.

Configure `django-cors-headers` settings as follows:

```
CORS_ALLOW_CREDENTIALS = True
CORS_ORIGIN_WHITELIST = ["https://www.example.com"]
```

Configure Django's settings as follows:

```
SESSION_COOKIE_DOMAIN = "example.com"
CSRF_COOKIE_DOMAIN = "example.com"
```

Please refer to Django's settings documentation for more information on `SESSION_COOKIE_DOMAIN` and `CSRF_COOKIE_DOMAIN` settings.

`OPENWISP_NOTIFICATIONS_SOUND`

type	`str`
default	notification_bell.mp3

This setting defines notification sound to be played when notification is received in real-time on admin site.

Provide a relative path (hosted on your web server) to audio file as show below.

```
OPENWISP_NOTIFICATIONS_SOUND = "your-appname/audio/notification.mp3"
```

`OPENWISP_NOTIFICATIONS_CACHE_TIMEOUT`

type	`int`
default	`172800` *(2 days, in seconds)*

It sets the number of seconds the notification contents should be stored in the cache. If you want cached notification content to never expire, then set it to `None`. Set it to `0` if you don't want to store notification contents in cache at all.

`OPENWISP_NOTIFICATIONS_IGNORE_ENABLED_ADMIN`

type	`list`
default	`[]`

This setting enables the widget which allows users to silence notifications for specific objects temporarily or permanently. in the change page of the specified `ModelAdmin` classes.

E.g., if you want to enable the widget for objects of `openwisp_users.models.User` model, then configure the setting as following:

```
OPENWISP_NOTIFICATIONS_IGNORE_ENABLED_ADMIN = [
    "openwisp_users.admin.UserAdmin"
]
```

`OPENWISP_NOTIFICATIONS_POPULATE_PREFERENCES_ON_MIGRATE`

type	`bool`
default	`True`

This setting allows to disable creating notification preferences on running migrations.

`OPENWISP_NOTIFICATIONS_NOTIFICATION_STORM_PREVENTION`

When the system starts creating a lot of notifications because of a general network outage (e.g.: a power outage, a global misconfiguration), the notification storm prevention mechanism avoids the constant displaying of new notification alerts as well as their sound, only the notification counter will continue updating periodically, although it won't emit any sound or create any other visual element until the notification storm is over.

This setting allows tweaking how this mechanism works.

The default configuration is as follows:

```
OPENWISP_NOTIFICATIONS_NOTIFICATION_STORM_PREVENTION = {
    # Time period for tracking burst of notifications (in seconds)
    "short_term_time_period": 10,
    # Number of notifications considered as a notification burst
    "short_term_notification_count": 6,
    # Time period for tracking notifications in long time interval (in seconds)
    "long_term_time_period": 180,
    # Number of notifications in long time interval to be considered as a notification storm
    "long_term_notification_count": 30,
    # Initial time for which notification updates should be skipped (in seconds)
    "initial_backoff": 1,
    # Time by which skipping of notification updates should be increased (in seconds)
    "backoff_increment": 1,
    # Maximum interval after which the notification widget should get updated (in seconds)
    "max_allowed_backoff": 15,
}
```

Management Commands

Note

This page is for developers who want to customize or extend OpenWISP Notifications, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Notifications User Docs

`populate_notification_preferences`

This command will populate notification preferences for all users for organizations they are member of.

> **Note**
>
> Before running this command make sure that the celery broker is running and **reachable** by celery workers.

Example usage:

```
# cd tests/
./manage.py populate_notification_preferences
```

`create_notification`

This command will create a dummy notification with `default` notification type for the members of `default` organization. This command is primarily provided for the sole purpose of testing notification in development only.

Example usage:

```
# cd tests/
./manage.py create_notification
```

Developer Docs

> **Note**
>
> This page is for developers who want to customize or extend OpenWISP Notifications, whether for bug fixes, new features, or contributions.
>
> For user guides and general information, please see:
>
> - General OpenWISP Quickstart
> - OpenWISP Notifications User Docs

Developer Installation Instructions

> **Note**
>
> This page is for developers who want to customize or extend OpenWISP Notifications, whether for bug fixes, new features, or contributions.
>
> For user guides and general information, please see:
>
> - General OpenWISP Quickstart
> - OpenWISP Notifications User Docs

Installing for Development

Install the system dependencies:

```
sudo apt install sqlite3 libsqlite3-dev openssl libssl-dev
```

Fork and clone the forked repository:

```
git clone git://github.com/<your_fork>/openwisp-notifications
```

Navigate into the cloned repository:

```
cd openwisp-notifications/
```

Launch Redis:

```
docker-compose up -d redis
```

Setup and activate a virtual-environment (we'll be using virtualenv):

```
python -m virtualenv env
source env/bin/activate
```

Make sure that your base python packages are up to date before moving to the next step:

```
pip install -U pip wheel setuptools
```

Install development dependencies:

```
pip install -e .
pip install -r requirements-test.txt
sudo npm install -g jshint stylelint
```

Create database:

```
cd tests/
./manage.py migrate
./manage.py createsuperuser
```

Launch celery worker (for background jobs):

```
celery -A openwisp2 worker -l info
```

Launch development server:

```
./manage.py runserver
```

You can access the admin interface at `http://127.0.0.1:8000/admin/`.

Run tests with:

```
# standard tests
./runtests.py

# If you running tests on PROD environment
./runtests.py --exclude skip_prod

# tests for the sample app
SAMPLE_APP=1 ./runtests.py
```

When running the last line of the previous example, the environment variable `SAMPLE_APP` activates the sample app in `/tests/openwisp2/` which is a simple django app that extends `openwisp-notifications` with the sole purpose of testing its extensibility, for more information regarding this concept, read the following section.

Run quality assurance tests with:

```
./run-qa-checks
```

Alternative Sources

Pypi

To install the latest Pypi:

```
pip install openwisp-notifications
```

Github

To install the latest development version tarball via HTTPs:

```
pip install https://github.com/openwisp/openwisp-notifications/tarball/master
```

Alternatively you can use the git protocol:

```
pip install -e git+git://github.com/openwisp/openwisp-notifications#egg=openwisp_notificatio
```

Code Utilities

Note

This page is for developers who want to customize or extend OpenWISP Notifications, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Notifications User Docs

Registering / Unregistering Notification Types

OpenWISP Notifications provides registering and unregistering notifications through utility functions `openwisp_notifications.types.register_notification_type` and `openwisp_notifications.types.unregister_notification_type`. Using these functions you can register or unregister notification types from your code.

Important

It is recommended that all notification types are registered or unregistered in `ready` method of your Django application's `AppConfig`.

`register_notification_type`

This function is used to register a new notification type from your code.

Syntax:

```
register_notification_type(type_name, type_config, models)
```

Parameter	Description
type_name	A `str` defining name of the notification type.
type_config	A `dict` defining configuration of the notification type.
models	An optional `list` of models that can be associated with the notification type.

An example usage has been shown below.

```
from openwisp_notifications.types import register_notification_type
from django.contrib.auth import get_user_model

User = get_user_model()

# Define configuration of your notification type
custom_type = {
    "level": "info",
    "verb": "added",
    "verbose_name": "device added",
    "message": "[{notification.target}]({notification.target_link}) was {notification.verb}
    "email_subject": "[{site.name}] A device has been added",
    "web_notification": True,
    "email_notification": True,
    # static URL for the actor object
    "actor": "https://openwisp.org/admin/config/device",
    # URL generation using callable for target object
    "target": "mymodule.target_object_link",
}

# Register your custom notification type
register_notification_type("custom_type", custom_type, models=[User])
```

It will raise `ImproperlyConfigured` exception if a notification type is already registered with same name(not to be confused with `verbose_name`).

Note

You can use `site` and `notification` variables while defining `message` and `email_subject` configuration of notification type. They refer to objects of `django.contrib.sites.models.Site` and `openwisp_notifications.models.Notification` respectively. This allows you to use any of their attributes in your configuration. Similarly to `message_template`, `message` property can also be formatted using markdown.

`unregister_notification_type`

This function is used to unregister a notification type from anywhere in your code.

Syntax:

```
unregister_notification_type(type_name)
```

Parameter	Description
type_name	A `str` defining name of the notification type.

An example usage is shown below.

```
from openwisp_notifications.types import unregister_notification_type

# Unregister previously registered notification type
unregister_notification_type("custom type")
```

It will raise `ImproperlyConfigured` exception if the concerned notification type is not registered.

Exceptions

`NotificationRenderException`

`openwisp_notifications.exceptions.NotificationRenderException`

Raised when notification properties(`email` or `message`) cannot be rendered from concerned *notification type*. It sub-classes `Exception` class.

It can be raised due to accessing non-existing keys like missing related objects in `email` or `message` setting of concerned *notification type*.

Notification Cache

In a typical OpenWISP installation, `actor`, `action_object` and `target` objects are same for a number of notifications. To optimize database queries, these objects are cached using Django's cache framework. The cached values are updated automatically to reflect actual data from database. You can control the duration of caching these objects using OPENWISP_NOTIFICATIONS_CACHE_TIMEOUT setting.

Cache Invalidation

The function `register_notification_cache_update` can be used to register a signal of a model which is being used as an `actor`, `action_object` and `target` objects. As these values are cached for the optimization purpose so their cached values are need to be changed when they are changed. You can register any signal you want which will delete the cached value. To register a signal you need to include following code in your `apps.py`.

```
from django.db.models.signals import post_save
from swapper import load_model


def ready(self):
    super().ready()

    # Include lines after this inside
    # ready function of you app config class
    from openwisp_notifications.handlers import (
        register_notification_cache_update,
    )

    model = load_model("app_name", "model_name")
    register_notification_cache_update(
        model,
        post_save,
        dispatch_uid="myapp_mymodel_notification_cache_invalidation",
    )
```

Important

You need to import `register_notification_cache_update` inside the `ready` function or you can define another function to register signals which will be called in `ready` and then it will be imported in this function. Also

`dispatch_uid` is unique identifier of a signal. You can pass any value you want but it needs to be unique. For more details read preventing duplicate signals section of Django documentation

Extending openwisp-notifications

Note

This page is for developers who want to customize or extend OpenWISP Notifications, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Notifications User Docs

One of the core values of the OpenWISP project is Software Reusability, for this reason OpenWISP Notifications provides a set of base classes which can be imported, extended and reused to create derivative apps.

In order to implement your custom version of *openwisp-notifications*, you need to perform the steps described in the rest of this section.

When in doubt, the code in test project and sample_notifications will guide you in the correct direction: just replicate and adapt that code to get a basic derivative of *openwisp-notifications* working.

Important

If you plan on using a customized version of this module, we suggest to start with it since the beginning, because migrating your data from the default module to your extended version may be time consuming.

1. Initialize your custom module

The first thing you need to do in order to extend *openwisp-notifications* is create a new django app which will contain your custom version of that *openwisp-notifications* app.

A django app is nothing more than a python package (a directory of python scripts), in the following examples we'll call this django app as `mynotifications` but you can name it how you want:

```
django-admin startapp mynotifications
```

Keep in mind that the command mentioned above must be called from a directory which is available in your PYTHON_PATH so that you can then import the result into your project.

Now you need to add `mynotifications` to `INSTALLED_APPS` in your `settings.py`, ensuring also that `openwisp_notifications` has been removed:

```
INSTALLED_APPS = [
    # ... other apps ...
    # 'openwisp_notifications',        <-- comment out or delete this line
    "mynotifications",
]
```

For more information about how to work with django projects and django apps, please refer to the django documentation.

2. Install `openwisp-notifications`

Install (and add to the requirement of your project) *openwisp-notifications*:

```
pip install -U https://github.com/openwisp/openwisp-notifications/tarball/master
```

3. Add `EXTENDED_APPS`

Add the following to your `settings.py`:

```
EXTENDED_APPS = ["openwisp_notifications"]
```

4. Add `openwisp_utils.staticfiles.DependencyFinder`

Add `openwisp_utils.staticfiles.DependencyFinder` to `STATICFILES_FINDERS` in your `settings.py`:

```
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
    "openwisp_utils.staticfiles.DependencyFinder",
]
```

5. Add `openwisp_utils.loaders.DependencyLoader`

Add `openwisp_utils.loaders.DependencyLoader` to `TEMPLATES` in your `settings.py`:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "OPTIONS": {
```

```
                "loaders": [
                    "django.template.loaders.filesystem.Loader",
                    "django.template.loaders.app_directories.Loader",
                    "openwisp_utils.loaders.DependencyLoader",
                ],
                "context_processors": [
                    "django.template.context_processors.debug",
                    "django.template.context_processors.request",
                    "django.contrib.auth.context_processors.auth",
                    "django.contrib.messages.context_processors.messages",
                ],
            },
        }
]
```

6. Inherit the AppConfig class

Please refer to the following files in the sample app of the test project:

- sample_notifications/__init__.py.
- sample_notifications/apps.py.

For more information regarding the concept of `AppConfig` please refer to the "Applications" section in the django documentation.

7. Create your custom models

For the purpose of showing an example, we added a simple "details" field to the models of the sample app in the test project.

You can add fields in a similar way in your `models.py` file.

> **Note**
>
> If you have questions about using, extending, or developing models, refer to the "Models" section of the Django documentation.

8. Add swapper configurations

Add the following to your `settings.py`:

```
# Setting models for swapper module
OPENWISP_NOTIFICATIONS_NOTIFICATION_MODEL = "mynotifications.Notification"
OPENWISP_NOTIFICATIONS_NOTIFICATIONSETTING_MODEL = (
    "mynotifications.NotificationSetting"
)
OPENWISP_NOTIFICATIONS_IGNOREOBJECTNOTIFICATION_MODEL = (
    "mynotifications.IgnoreObjectNotification"
)
```

9. Create database migrations

Create and apply database migrations:

```
./manage.py makemigrations
./manage.py migrate
```

For more information, refer to the "Migrations" section in the django documentation.

10. Create your custom admin

Refer to the admin.py file of the sample app.

To introduce changes to the admin, you can do it in two main ways which are described below.

> **Note**
>
> For more information regarding how the django admin works, or how it can be customized, please refer to "The django admin site" section in the django documentation.

1. Monkey patching

If the changes you need to add are relatively small, you can resort to monkey patching.

For example:

```
from openwisp_notifications.admin import NotificationSettingInline

NotificationSettingInline.list_display.insert(1, "my_custom_field")
NotificationSettingInline.ordering = ["-my_custom_field"]
```

2. Inheriting admin classes

If you need to introduce significant changes and/or you don't want to resort to monkey patching, you can proceed as follows:

```
from django.contrib import admin
from openwisp_notifications.admin import (
    NotificationSettingInline as BaseNotificationSettingInline,
)
from openwisp_notifications.swapper import load_model

NotificationSetting = load_model("NotificationSetting")

admin.site.unregister(NotificationSettingAdmin)
admin.site.unregister(NotificationSettingInline)


@admin.register(NotificationSetting)
class NotificationSettingInline(BaseNotificationSettingInline):
    # add your changes here
    pass
```

11. Create root URL configuration

Please refer to the urls.py file in the test project.

For more information about URL configuration in django, please refer to the "URL dispatcher" section in the django documentation.

12. Create root routing configuration

Please refer to the routing.py file in the test project.

For more information about URL configuration in django, please refer to the "Routing" section in the Channels documentation.

13. Create `celery.py`

Please refer to the celery.py file in the test project.

For more information about the usage of celery in django, please refer to the "First steps with Django" section in the celery documentation.

14. Import Celery Tasks

Add the following in your `settings.py` to import Celery tasks from `openwisp_notifications` app.

```
CELERY_IMPORTS = ("openwisp_notifications.tasks",)
```

15. Register Template Tags

If you need to use template tags, you will need to register them as shown in "templatetags/notification_tags.py" of sample_notifications.

For more information about template tags in django, please refer to the "Custom template tags and filters" section in the django documentation.

16. Register Notification Types

You can register notification types as shown in the section for registering notification types.

A reference for registering a notification type is also provided in sample_notifications/apps.py. The registered notification type of `sample_notifications` app is used for creating notifications when an object of `TestApp` model is created. You can use sample_notifications/models.py as reference for your implementation.

17. Import the automated tests

When developing a custom application based on this module, it's a good idea to import and run the base tests too, so that you can be sure the changes you're introducing are not breaking some of the existing feature of openwisp-notifications.

In case you need to add breaking changes, you can overwrite the tests defined in the base classes to test your own behavior.

See the tests of the sample_notifications to find out how to do this.

Note

Some tests will fail if `templatetags` and `admin/base.html` are not configured properly. See preceding sections to configure them properly.

Other base classes that can be inherited and extended

The following steps are not required and are intended for more advanced customization.

API views

The API view classes can be extended into other django applications as well. Note that it is not required for extending openwisp-notifications to your app and this change is required only if you plan to make changes to the API views.

Create a view file as done in sample_notifications/views.py

For more information regarding Django REST Framework API views, please refer to the "Generic views" section in the Django REST Framework documentation.

Web Socket Consumers

The Web Socket Consumer classes can be extended into other django applications as well. Note that it is not required for extending openwisp-notifications to your app and this change is required only if you plan to make changes to the consumers.

Create a consumer file as done in sample_notifications/consumers.py

For more information regarding Channels' Consumers, please refer to the "Consumers" section in the Channels documentation.

Other useful resources:

- REST API
- Settings

Utils

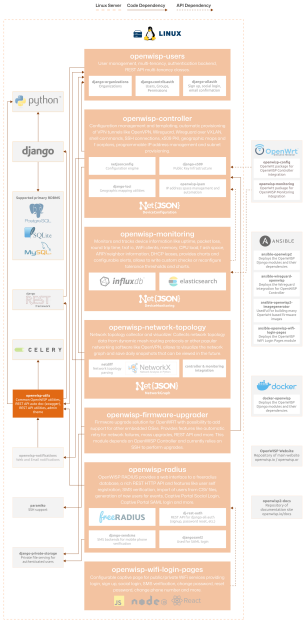
Seealso

Source code: github.com/openwisp/openwisp-utils.

The goal of OpenWISP Utils is to minimize duplication, ease maintenance, and enable the rapid development of new OpenWISP modules by leveraging battle-tested best practices.

This is achieved by providing code structures that are inherited, extended, and utilized across different modules in the OpenWISP ecosystem.

The following diagram illustrates the role of the Utils module within the OpenWISP architecture.



OpenWISP Architecture: highlighted utils module

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

Collection of Usage Metrics

The `openwisp-utils` module includes an optional sub-app `openwisp_utils.metric_collection`, which allows us to collect of the following information from OpenWISP instances:

- OpenWISP Version
- List of enabled OpenWISP modules and their version
- Operating System identifier, e.g.: Linux version, Kernel version, target platform (e.g. x86)
- Installation method, if available, e.g. ansible-openwisp2 or docker-openwisp

The data above is collected during the following events:

- **Install**: when OpenWISP is installed the first time
- **Upgrade**: when any OpenWISP module is upgraded
- **Heartbeat**: once every 24 hours

We collect data on OpenWISP usage to gauge user engagement, satisfaction, and upgrade patterns. This informs our development decisions, ensuring continuous improvement aligned with user needs.

To enhance our understanding and management of this data, we have integrated Clean Insights, a privacy-preserving analytics tool. Clean Insights allows us to responsibly gather and analyze usage metrics without compromising user privacy. It provides us with the means to make data-driven decisions while respecting our users' rights and trust.

We have taken great care to ensure no sensitive or personal data is being tracked.

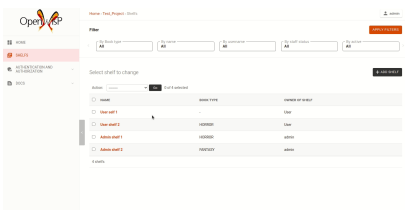
Opting Out from Metric Collection

You can opt-out from sharing this data any time from the "System Info" page. Alternatively, you can also remove the `openwisp_utils.metric_collection` app from `INSTALLED_APPS` in one of the following ways:

- If you are using the ansible-openwisp2 role, you can set the variable `openwisp2_usage_metric_collection` to `false` in your playbook.
- If you are using docker-openwisp, you can set set the environment variable `METRIC_COLLECTION` to `False` in the `.env` file.

However, **it would be very helpful to the project if you keep the colection of these metrics enabled**, because the feedback we get from this data is useful to guide the project in the right direction.

Admin Filters



The `admin_theme` sub app provides an improved UI for the *changelist* filter which occupies less space compared to the original implementation in django: filters are displayed horizontally on the top (instead of vertically on the side) and filter options are hidden in dropdown menus which are expanded once clicked.

Multiple filters can be applied at same time with the help of "apply filter" button. This button is only visible when total number of filters is greater than 4. When filters in use are less or equal to 4 the "apply filter" button is not visible and filters work like in the original django implementation (as soon as a filter option is selected the filter is applied and the page is reloaded).

Settings

> **Note**
>
> If you're unsure about what *"Django settings"* are, you can refer to How to Edit Django Settings in OpenWISP for guidance.

`OPENWISP_ADMIN_SITE_CLASS`

Default: `openwisp_utils.admin_theme.admin.OpenwispAdminSite`

If you need to use a customized admin site class, you can use this setting.

`OPENWISP_ADMIN_SITE_TITLE`

Default: `OpenWISP Admin`

Title value used in the `<title>` HTML tag of the admin site.

`OPENWISP_ADMIN_SITE_HEADER`

Default: `OpenWISP`

Heading text used in the main `<h1>` HTML tag (the logo) of the admin site.

`OPENWISP_ADMIN_INDEX_TITLE`

Default: `Network administration`

Title shown to users in the index page of the admin site.

`OPENWISP_ADMIN_DASHBOARD_ENABLED`

Default: `True`

When `True`, enables the OpenWISP Dashboard. Upon login, the user will be greeted with the dashboard instead of the default Django admin index page.

`OPENWISP_ADMIN_THEME_LINKS`

Default: `[]`

> **Note**
>
> This setting requires the admin_theme_settings context processor in order to work.

Allows to override the default CSS and favicon, as well as add extra `<link>` HTML elements if needed.

This setting overrides the default theme, you can reuse the default CSS or replace it entirely.

The following example shows how to keep using the default CSS, supply an additional CSS and replace the favicon.

Example usage:

```
OPENWISP_ADMIN_THEME_LINKS = [
    {
        "type": "text/css",
        "href": "/static/admin/css/openwisp.css",
        "rel": "stylesheet",
        "media": "all",
    },
    {
        "type": "text/css",
        "href": "/static/admin/css/custom-theme.css",
        "rel": "stylesheet",
        "media": "all",
    },
    {
        "type": "image/x-icon",
        "href": "/static/favicon.png",
        "rel": "icon",
    },
]
```

`OPENWISP_ADMIN_THEME_JS`

Default: `[]`

Allows to pass a list of strings representing URLs of custom JS files to load.

Example usage:

```
OPENWISP_ADMIN_THEME_JS = [
    "/static/custom-admin-theme.js",
]
```

`OPENWISP_ADMIN_SHOW_USERLINKS_BLOCK`

Default: `False`

When set to `True`, enables Django user links on the admin site.

i.e. (USER NAME/ VIEW SITE / CHANGE PASSWORD / LOG OUT).

These links are already shown in the main navigation menu and for this reason are hidden by default.

`OPENWISP_API_DOCS`

Default: `True`

Whether the OpenAPI documentation is enabled.

When enabled, you can view the available documentation using the Swagger endpoint at `/api/v1/docs/`.

You also need to add the following URL to your project `urls.py`:

```
urlpatterns += [
    url(r"^api/v1/", include("openwisp_utils.api.urls")),
]
```

OPENWISP_API_INFO

Default:

```
{
    "title": "OpenWISP API",
    "default_version": "v1",
    "description": "OpenWISP REST API",
}
```

Define OpenAPI general information. NOTE: This setting requires `OPENWISP_API_DOCS = True` to take effect.

For more information about optional parameters check the drf-yasg documentation.

OPENWISP_SLOW_TEST_THRESHOLD

Default: `[0.3, 1]` (seconds)

It can be used to change the thresholds used by TimeLoggingTestRunner to detect slow tests (0.3s by default) and highlight the slowest ones (1s by default) among them.

OPENWISP_STATICFILES_VERSIONED_EXCLUDE

Default: `['leaflet/*/*.png']`

Allows to pass a list of **Unix shell-style wildcards** for files to be excluded by CompressStaticFilesStorage.

By default Leaflet PNGs have been excluded to avoid bugs like openwisp/ansible-openwisp2#232.

Example usage:

```
OPENWISP_STATICFILES_VERSIONED_EXCLUDE = [
    "*png",
]
```

OPENWISP_HTML_EMAIL

type	`bool`
default	`True`

If `True`, an HTML themed version of the email can be sent using the send_email function.

OPENWISP_EMAIL_TEMPLATE

type	`str`
default	`openwisp_utils/email_template.html`

This setting allows to change the django template used for sending emails with the send_email function. It is recommended to extend the default email template as in the example below.

```
{% extends 'openwisp_utils/email_template.html' %}
{% block styles %}
{{ block.super }}
```

```
<style>
  .background {
    height: 100%;
    background: linear-gradient(to bottom, #8ccbbe 50%, #3797a4 50%);
    background-repeat: no-repeat;
    background-attachment: fixed;
    padding: 50px;
  }

  .mail-header {
    background-color: #3797a4;
    color: white;
  }
</style>
{% endblock styles %}
```

Similarly, you can customize the HTML of the template by overriding the `body` block. See email_template.html for reference implementation.

OPENWISP_EMAIL_LOGO

type	str
default	OpenWISP logo

This setting allows to change the logo which is displayed in HTML version of the email.

Note

Provide a URL which points to the logo on your own web server. Ensure that the URL provided is publicly accessible from the internet. Otherwise, the logo may not be displayed in the email. Please also note that SVG images do not get processed by some email clients like Gmail so it is recommended to use PNG images.

OPENWISP_CELERY_SOFT_TIME_LIMIT

type	int
default	30 (in seconds)

Sets the soft time limit for celery tasks using OpenwispCeleryTask.

OPENWISP_CELERY_HARD_TIME_LIMIT

type	int
default	120 (in seconds)

Sets the hard time limit for celery tasks using OpenwispCeleryTask.

OPENWISP_AUTOCOMPLETE_FILTER_VIEW

type	str
default	'openwisp_utils.admin_theme.views.AutocompleteJsonView'

Dotted path to the `AutocompleteJsonView` used by the `openwisp_utils.admin_theme.filters.AutocompleteFilter`.

Developer Docs

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

Developer Installation Instructions

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

Installing for Development

Install the system dependencies:

```
sudo apt-get install sqlite3 libsqlite3-dev

# For running E2E Selenium tests
sudo apt install chromium
```

Fork and clone the forked repository:

```
git clone git://github.com/<your_fork>/openwisp-utils
```

Navigate into the cloned repository:

```
cd openwisp-utils/
```

Setup and activate a virtual-environment (we'll be using virtualenv):

```
python -m virtualenv env
source env/bin/activate
```

Make sure that your base python packages are up to date before moving to the next step:

```
pip install -U pip wheel setuptools
```

Install development dependencies:

```
pip install -e .[qa,rest]
pip install -r requirements-test.txt
sudo npm install -g jshint stylelint
```

Set up the git *pre-push* hook to run tests and QA checks automatically right before the git push action, so that if anything fails the push operation will be aborted:

```
openwisp-pre-push-hook --install
```

Create database:

```
cd tests/
./manage.py migrate
./manage.py createsuperuser
```

Launch development server:

```
./manage.py runserver
```

You can access the admin interface at `http://127.0.0.1:8000/admin/`.

Run tests with:

```
./runtests.py --parallel
```

Run quality assurance tests with:

```
./run-qa-checks
```

Alternative Sources

Pypi

To install the latest Pypi:

```
pip install openwisp-utils
```

Github

To install the latest development version tarball via HTTPs:

```
pip install https://github.com/openwisp/openwisp-utils/tarball/master
```

Alternatively you can use the git protocol:

```
pip install -e git+git://github.com/openwisp/openwisp-utils#egg=openwisp_utils
```

OpenWISP Dashboard

Note

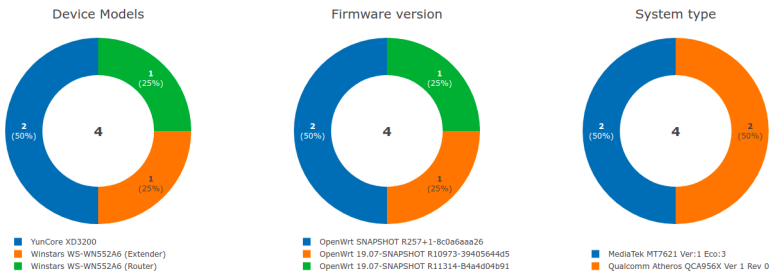
This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

The `admin_theme` sub app of this package provides an admin dashboard for OpenWISP which can be manipulated with the functions described in the next sections.

Example taken from the Controller Module:



`register_dashboard_template`

Allows including a specific django template in the OpenWISP dashboard.

It is designed to allow the inclusion of the geographic map shipped by OpenWISP Monitoring but can be used to include any custom element in the dashboard.

Note

It is possible to register templates to be loaded before or after charts using the `after_charts` keyword argument (see below).

Syntax:

```
register_dashboard_template(position, config)
```

Parameter	Description
`position`	(`int`) The position of the template.
`config`	(`dict`) The configuration of the template.
`extra_config`	**optional** (`dict`) Extra configuration you want to pass to custom template.

`after_charts`	**optional** (`bool`) Whether the template should be loaded after dashboard charts. Defaults to `False`, i.e. templates are loaded before dashboard charts by default.

Following properties can be configured for each template `config`:

Property	Description
`template`	(`str`) Path to pass to the template loader.
`css`	(`tuple`) List of CSS files to load in the HTML page.
`js`	(`tuple`) List of Javascript files to load in the HTML page.

Code example:

```
from openwisp_utils.admin_theme import register_dashboard_template

register_dashboard_template(
    position=0,
    config={
        "template": "admin/dashboard/device_map.html",
        "css": (
            "monitoring/css/device-map.css",
            "leaflet/leaflet.css",
            "monitoring/css/leaflet.fullscreen.css",
        ),
        "js": (
            "monitoring/js/device-map.js",
            "leaflet/leaflet.js",
            "leaflet/leaflet.extras.js",
            "monitoring/js/leaflet.fullscreen.min.js",
        ),
    },
    extra_config={
        "optional_variable": "any_valid_value",
    },
    after_charts=True,
)
```

It is recommended to register dashboard templates from the `ready` method of the AppConfig of the app where the templates are defined.

`unregister_dashboard_template`

This function can be used to remove a template from the dashboard.

Syntax:

```
unregister_dashboard_template(template_name)
```

Parameter	Description
`template_name`	(`str`) The name of the template to remove.

Code example:

```
from openwisp_utils.admin_theme import unregister_dashboard_template

unregister_dashboard_template("admin/dashboard/device_map.html")
```

An `ImproperlyConfigured` exception is raised the specified dashboard template is not registered.

`register_dashboard_chart`

Adds a chart to the OpenWISP dashboard.

At the moment only pie charts are supported.

The code works by defining the type of query which will be executed, and optionally, how the returned values have to be colored and labeled.

Syntax:

```
register_dashboard_chart(position, config)
```

Parameter	Description
`position`	(`int`) Position of the chart.
`config`	(`dict`) Configuration of chart.

Following properties can be configured for each chart `config`:

Property	Description
`query_params`	It is a required property in form of `dict`. Refer to the Dashboard Chart query_params table below for supported properties.
`colors`	An **optional** `dict` which can be used to define colors for each distinct value shown in the pie charts.
`labels`	An **optional** `dict` which can be used to define translatable strings for each distinct value shown in the pie charts. Can be used also to provide fallback human readable values for raw values stored in the database which would be otherwise hard to understand for the user.
`filters`	An **optional** `dict` which can be used when using `aggregate` and `annotate` in `query_params` to define the link that will be generated to filter results (pie charts are clickable and clicking on a portion of it will show the filtered results).
`main_filters`	An **optional** `dict` which can be used to add additional filtering on the target link.
`filtering`	An **optional** `str` which can be set to `'False'` (str) to disable filtering on target links. This is useful when clicking on any section of the chart should take user to the same URL.
`quick_link`	An **optional** `dict` which contains configuration for the quick link button rendered below the chart. Refer to the Dashboard chart quick_link table below for supported properties. **Note**: The chart legend is disabled if configuration for quick link button is provided.

Dashboard Chart `query_params`

Property	Description
`name`	(`str`) Chart title shown in the user interface.
`app_label`	(`str`) App label of the model that will be used to query the database.
`model`	(`str`) Name of the model that will be used to query the database.
`group_by`	(`str`) The property which will be used to group values.
`annotate`	Alternative to `group_by`, `dict` used for more complex queries.
`aggregate`	Alternative to `group_by`, `dict` used for more complex queries.
`filter`	`dict` used for filtering queryset.
`organization_field`	(`str`) If the model does not have a direct relation with the `Organization` model, then indirect relation can be specified using this property. E.g.: `device__organization_id`.

Dashboard chart `quick_link`

Property	Description
`url`	(`str`) URL for the anchor tag
`label`	(`str`) Label shown on the button
`title`	(`str`) Title attribute of the button element
`custom_css_classes`	(`list`) List of CSS classes that'll be applied on the button

Code example:

```
from openwisp_utils.admin_theme import register_dashboard_chart

register_dashboard_chart(
    position=1,
    config={
        "query_params": {
            "name": "Operator Project Distribution",
            "app_label": "test_project",
            "model": "operator",
            "group_by": "project__name",
        },
        "colors": {"Utils": "red", "User": "orange"},
        "quick_link": {
            "url": "/admin/test_project/operator",
            "label": "Open Operators list",
            "title": "View complete list of operators",
            "custom_css_classes": ["negative-top-20"],
        },
    },
)
```

For real world examples, look at the code of OpenWISP Controller and OpenWISP Monitoring.

An `ImproperlyConfigured` exception is raised if a dashboard element is already registered at same position.

It is recommended to register dashboard charts from the `ready` method of the AppConfig of the app where the models are defined. Checkout app.py of the test_project for reference.

`unregister_dashboard_chart`

This function can used to remove a chart from the dashboard.

Syntax:

```
unregister_dashboard_chart(chart_name)
```

Parameter	Description
`chart_name`	(`str`) The name of the chart to remove.

Code example:

```
from openwisp_utils.admin_theme import unregister_dashboard_chart

unregister_dashboard_chart("Operator Project Distribution")
```

An `ImproperlyConfigured` exception is raised the specified dashboard chart is not registered.

Main Navigation Menu

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

The `admin_theme` sub app of this package provides a navigation menu that can be manipulated with the functions described in the next sections.

Context Processor

For this feature to work, we must make sure that the context processor `openwisp_utils.admin_theme.context_processor.menu_groups` is enabled in `settings.py` as shown below.

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [],
        "OPTIONS": {
            "loaders": [
                # ... omitted ...
            ],
            "context_processors": [
                # ... other context processors ...
                "openwisp_utils.admin_theme.context_processor.menu_groups"  # <----- add thi
            ],
        },
    },
]
```

This context processor is enabled by default in any OpenWISP installer and in the test project of this module.

The `register_menu_group` function

Allows registering a new menu item or group at the specified position in the Main Navigation Menu.

Syntax:

```
register_menu_group(position, config)
```

Parameter	Description
`position`	(`int`) Position of the group or item.
`config`	(`dict`) Configuration of the group or item.

Code example:

```
from django.utils.translation import ugettext_lazy as _
from openwisp_utils.admin_theme.menu import register_menu_group

register_menu_group(
    position=1,
    config={
        "label": _("My Group"),
        "items": {
            1: {
                "label": _("Users List"),
                "model": "auth.User",
                "name": "changelist",
                "icon": "list-icon",
            },
            2: {
                "label": _("Add User"),
                "model": "auth.User",
                "name": "add",
                "icon": "add-icon",
            },
        },
        "icon": "user-group-icon",
    },
)
register_menu_group(
    position=2,
    config={
        "model": "test_project.Shelf",
        "name": "changelist",
        "label": _("View Shelf"),
        "icon": "shelf-icon",
    },
)
register_menu_group(
    position=3, config={"label": _("My Link"), "url": "https://link.com"}
)
```

An `ImproperlyConfigured` exception is raised if a menu element is already registered at the same position.

An `ImproperlyConfigured` exception is raised if the supplied configuration does not match with the different types of possible configurations available (different configurations will be discussed in the next section).

Note

It is recommended to use `register_menu_group` in the `ready` method of the `AppConfig`.

Important

`register_menu_items` is obsoleted by `register_menu_group` and will be removed in future versions. Links added using `register_menu_items` will be shown at the top of navigation menu and above any `register_menu_group` items.

Adding a Custom Link

To add a link that contains a custom URL the following syntax can be used.

Syntax:

```
register_menu_group(
    position=1,
    config={"label": "Link Label", "url": "link_url", "icon": "my-icon"},
)
```

Following is the description of the configuration:

Parameter	Description
`label`	(`str`) Display text for the link.
`url`	(`str`) URL for the link.
`icon`	An **optional** `str` CSS class name for the icon. No icon is displayed if not provided.

Adding a Model Link

To add a link that contains URL of add form or change list page of a model then following syntax can be used. Users will only be able to see links for models they have permission to either view or edit.

Syntax:

```
# add a link of list page
register_menu_group(
    position=1,
    config={
        "model": "my_project.MyModel",
        "name": "changelist",
        "label": "MyModel List",
        "icon": "my-model-list-class",
    },
)

# add a link of add page
register_menu_group(
    position=2,
    config={
        "model": "my_project.MyModel",
        "name": "add",
        "label": "MyModel Add Item",
        "icon": "my-model-add-class",
    },
)
```

Following is the description of the configuration:

Parameter	Description
`model`	(`str`) Model of the app for which you to add link.
`name`	(`str`) argument name, e.g.: *changelist* or *add*.
`label`	An **optional** `str` display text for the link. It is automatically generated if not provided.
`icon`	An **optional** `str` CSS class name for the icon. No icon is displayed if not provided.

Adding a Menu Group

To add a nested group of links in the menu the following syntax can be used. It creates a dropdown in the menu.

Syntax:

```
register_menu_group(
    position=1,
    config={
        "label": "My Group Label",
        "items": {
            1: {
                "label": "Link Label",
                "url": "link_url",
                "icon": "my-icon",
            },
            2: {
                "model": "my_project.MyModel",
                "name": "changelist",
                "label": "MyModel List",
                "icon": "my-model-list-class",
            },
        },
        "icon": "my-group-icon-class",
    },
)
```

Following is the description of the configuration:

Parameter	Description
`label`	(`str`) Display name for the link.
`items`	(`dict`) Items to be displayed in the dropdown. It can be a dict of custom links or model links with key as their position in the group.
`icon`	An **optional** `str` CSS class name for the icon. No icon is displayed if not provided.

The `register_menu_subitem` function

Allows adding an item to a registered group.

Syntax:

```
register_menu_subitem(group_position, item_position, config)
```

Parameter	Description
`group_position`	(`int`) Position of the group in which item should be added.
`item_position`	(`int`) Position at which item should be added in the group
`config`	(`dict`) Configuration of the item.

Code example:

```
from django.utils.translation import ugettext_lazy as _
from openwisp_utils.admin_theme.menu import register_menu_subitem

# To register a model link
register_menu_subitem(
    group_position=10,
    item_position=2,
    config={
        "label": _("Users List"),
```

```
        "model": "auth.User",
        "name": "changelist",
        "icon": "list-icon",
    },
)

# To register a custom link
register_menu_subitem(
    group_position=10,
    item_position=2,
    config={"label": _("My Link"), "url": "https://link.com"},
)
```

An `ImproperlyConfigured` exception is raised if the group is not already registered at `group_position`.

An `ImproperlyConfigured` exception is raised if the group already has an item registered at `item_position`.

It is only possible to register links to specific models or custom URL. An `ImproperlyConfigured` exception is raised if the configuration of group is provided in the function.

Important

It is recommended to use `register_menu_subitem` in the `ready` method of the `AppConfig`.

How to Use Custom Icons in the Menu

Create a CSS file and use the following syntax to provide the image for each icon used in the menu. The CSS class name should be the same as the `icon` parameter used in the configuration of a menu item or group. Also icon being used should be in `svg` format.

Example:

```
.icon-class-name {
    mask-image: url(imageurl);
    -webkit-mask-image: url(imageurl);
}
```

Follow the instructions in Supplying custom CSS and JS for the admin theme to know how to configure your OpenWISP instance to load custom CSS files.

Using the `admin_theme`

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

The admin theme requires Django >= 2.2..

Make sure `openwisp_utils.admin_theme` is listed in `INSTALLED_APPS` (`settings.py`):

```
INSTALLED_APPS = [
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "openwisp_utils.admin_theme",  # <----- add this
    # add when using autocomplete filter
    "admin_auto_filters",  # <----- add this
    "django.contrib.sites",
    # admin
    "django.contrib.admin",
]
```

Using `DependencyLoader` and `DependencyFinder`

Add the list of all packages extended to `EXTENDED_APPS` in `settings.py`.

For example, if you've extended `django_x509`:

```
EXTENDED_APPS = ["django_x509"]
```

`DependencyFinder`

This is a static finder which looks for static files in the `static` directory of the apps listed in `settings.EXTENDED_APPS`.

Add `openwisp_utils.staticfiles.DependencyFinder` to `STATICFILES_FINDERS` in `settings.py`.

```
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
    "openwisp_utils.staticfiles.DependencyFinder",  # <----- add this
]
```

`DependencyLoader`

This is a template loader which looks for templates in the `templates` directory of the apps listed in `settings.EXTENDED_APPS`.

Add `openwisp_utils.loaders.DependencyLoader` to template `loaders` in `settings.py` as shown below.

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [],
        "OPTIONS": {
            "loaders": [
                # ... other loaders ...
                "openwisp_utils.loaders.DependencyLoader",  # <----- add this
            ],
            "context_processors": [
```

```
                # ... omitted ...
            ],
        },
    },
]
```

Supplying Custom CSS and JS for the Admin Theme

Add `openwisp_utils.admin_theme.context_processor.admin_theme_settings` to template `context_processors` in `settings.py` as shown below. This will allow to set OPENWISP_ADMIN_THEME_LINKS and OPENWISP_ADMIN_THEME_JS settings to provide CSS and JS files to customize admin theme.

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [],
        "OPTIONS": {
            "loaders": [
                # ... omitted ...
            ],
            "context_processors": [
                # ... other context processors ...
                "openwisp_utils.admin_theme.context_processor.admin_theme_settings"  # <----
            ],
        },
    },
]
```

Note

You will have to deploy these static files on your own.

In order to make django able to find and load these files you may want to use the `STATICFILES_DIR` setting in `settings.py`.

You can learn more in the Django documentation.

Extend Admin Theme Programmatically

`openwisp_utils.admin_theme.theme.register_theme_link`

Allows adding items to OPENWISP_ADMIN_THEME_LINKS.

This function is meant to be used by third party apps or OpenWISP modules which aim to extend the core look and feel of the OpenWISP theme (e.g.: add new menu icons).

Syntax:

```
register_theme_link(links)
```

Parameter	Description
`links`	(`list`) List of *link* items to be added to OPENWISP_ADMIN_THEME_LINKS

`openwisp_utils.admin_theme.theme.unregister_theme_link`

Allows removing items from OPENWISP_ADMIN_THEME_LINKS.

This function is meant to be used by third party apps or OpenWISP modules which aim additional functionalities to UI of OpenWISP (e.g.: adding a support chat bot).

Syntax:

```
unregister_theme_link(links)
```

Parameter	Description
`links`	(`list`) List of *link* items to be removed from OPENWISP_ADMIN_THEME_LINKS

`openwisp_utils.admin_theme.theme.register_theme_js`

Allows adding items to OPENWISP_ADMIN_THEME_JS.

Syntax:

```
register_theme_js(js)
```

Parameter	Description
`js`	(`list`) List of relative path of *js* files to be added to OPENWISP_ADMIN_THEME_JS

`openwisp_utils.admin_theme.theme.unregister_theme_js`

Allows removing items from OPENWISP_ADMIN_THEME_JS.

Syntax:

```
unregister_theme_js(js)
```

Parameter	Description
`js`	(`list`) List of relative path of *js* files to be removed from OPENWISP_ADMIN_THEME_JS

Sending emails

`openwisp_utils.admin_theme.email.send_email`

This function allows sending email in both plain text and HTML version (using the template and logo that can be customized using OPENWISP_EMAIL_TEMPLATE and OPENWISP_EMAIL_LOGO respectively).

In case the HTML version if not needed it may be disabled by setting OPENWISP_HTML_EMAIL to `False`.

Syntax:

```
send_email(subject, body_text, body_html, recipients, **kwargs)
```

Parameter	Description
`subject`	(`str`) The subject of the email template.
`body_text`	(`str`) The body of the text message to be emailed.
`body_html`	(`str`) The body of the html template to be emailed.
`recipients`	(`list`) The list of recipients to send the mail to.

`extra_context`	**optional** (`dict`) Extra context which is passed to the template. The dictionary keys `call_to_action_text` and `call_to_action_url` can be passed to show a call to action button. Similarly, `footer` can be passed to add a footer.
`**kwargs`	Any additional keyword arguments (e.g. `attachments`, `headers`, etc.) are passed directly to the django.core.mail.EmailMultiAlternatives.

Important

Data passed in body should be validated and user supplied data should not be sent directly to the function.

Database Backends

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

`openwisp_utils.db.backends.spatialite`

This backend extends `django.contrib.gis.db.backends.spatialite` database backend to implement a workaround for handling issue with sqlite 3.36 and spatialite 5.

Quality Assurance Checks

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

This package contains some common QA checks that are used in the automated builds of different OpenWISP modules.

openwisp-qa-format

This shell script automatically formats Python and CSS code according to the OpenWISP coding style conventions.

It runs `isort` and `black` to format python code (these two dependencies are required and installed automatically when running `pip install openwisp-utils[qa]`).

The `stylelint` and `jshint` programs are used to perform style checks on CSS and JS code respectively, but they are optional: if `stylelint` and/or `jshint` are not installed, the check(s) will be skipped.

openwisp-qa-check

Shell script to run the following quality assurance checks:

- checkmigrations
- checkcommit
- checkendline
- checkpendingmigrations
- checkrst
- `flake8` - Python code linter
- `isort` - Sorts python imports alphabetically, and separated into sections
- `black` - Formats python code using a common standard
- `csslinter` - Formats and checks CSS code using stylelint common standard
- `jslinter` - Checks Javascript code using jshint common standard

If a check requires a flag, it can be passed forward in the same way.

Usage example:

```
openwisp-qa-check --migration-path <path> --message <commit-message>
```

Any unneeded checks can be skipped by passing `--skip-<check-name>`

Usage example:

```
openwisp-qa-check --skip-isort
```

For backward compatibility `csslinter` and `jslinter` are skipped by default. To run them during QA checks pass arguments as follows.

Usage example:

```
# To activate csslinter
openwisp-qa-check --csslinter

# To activate jslinter
openwisp-qa-check --jslinter
```

You can do multiple `checkmigrations` by passing the arguments with space-delimited string.

For example, this multiple `checkmigrations`:

```
checkmigrations --migrations-to-ignore 3 \
        --migration-path ./openwisp_users/migrations/ || exit 1
```

```
checkmigrations --migrations-to-ignore 2 \
        --migration-path ./tests/testapp/migrations/ || exit 1
```

Can be changed with:

```
openwisp-qa-check --migrations-to-ignore "3 2" \
        --migration-path "./openwisp_users/migrations/ ./tests/testapp/migrations/"
```

`checkmigrations`

Ensures the latest migrations created have a human readable name.

We want to avoid having many migrations named like `0003_auto_20150410_3242.py`.

This way we can reconstruct the evolution of our database schemas faster, with less efforts and hence less costs.

Usage example:

```
checkmigrations --migration-path ./django_freeradius/migrations/
```

`checkcommit`

Ensures the last commit message follows our commit message style guidelines.

We want to keep the commit log readable, consistent and easy to scan in order to make it easy to analyze the history of our modules, which is also a very important activity when performing maintenance.

Usage example:

```
checkcommit --message "$(git log --format=%B -n 1)"
```

If, for some reason, you wish to skip this QA check for a specific commit message you can add `#noqa` to the end of your commit message.

Usage example:

```
[qa] Improved #20

Simulation of a special unplanned case
#noqa
```

`checkendline`

Ensures that a blank line is kept at the end of each file.

`checkpendingmigrations`

Ensures there django migrations are up to date and no new migrations need to be created.

It accepts an optional `--migration-module` flag indicating the django app name that should be passed to `./manage.py makemigrations`, e.g.: `./manage.py makemigrations $MIGRATION_MODULE`.

`checkrst`

Checks the syntax of all ReStructuredText files to ensure they can be published on Pypi or using python-sphinx.

Custom Fields

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

This section describes custom fields defined in `openwisp_utils.fields` that can be used in Django models.

`openwisp_utils.fields.KeyField`

A model field which provides a random key or token, widely used across openwisp modules.

`openwisp_utils.fields.FallbackBooleanChoiceField`

This field extends Django's BooleanField and provides additional functionality for handling choices with a fallback value.

Note

- The field will return the **fallback value** whenever is set to `None`.
- Setting the same value as the **fallback value** will save `None` (NULL) in the database.

This field is particularly useful when you want to present a choice between enabled and disabled options.

```
from django.db import models
from openwisp_utils.fields import FallbackBooleanChoiceField
from myapp import settings as app_settings


class MyModel(models.Model):
    is_active = FallbackBooleanChoiceField(
        fallback=app_settings.IS_ACTIVE_FALLBACK,
    )
```

`openwisp_utils.fields.FallbackCharChoiceField`

This field extends Django's CharField and provides additional functionality for handling choices with a fallback value.

Note

- The field will return the **fallback value** whenever is set to `None`.
- Setting the same value as the **fallback value** will save `None` (NULL) in the database.

```
from django.db import models
from openwisp_utils.fields import FallbackCharChoiceField
from myapp import settings as app_settings


class MyModel(models.Model):
    is_first_name_required = FallbackCharChoiceField(
        max_length=32,
        choices=(
            ("disabled", _("Disabled")),
            ("allowed", _("Allowed")),
            ("mandatory", _("Mandatory")),
        ),
        fallback=app_settings.IS_FIRST_NAME_REQUIRED,
    )
```

`openwisp_utils.fields.FallbackCharField`

This field extends Django's CharField and provides additional functionality for handling text fields with a fallback value.

Note

- The field will return the **fallback value** whenever is set to `None`.
- Setting the same value as the **fallback value** will save `None` (NULL) in the database.

```
from django.db import models
from openwisp_utils.fields import FallbackCharField
from myapp import settings as app_settings


class MyModel(models.Model):
    greeting_text = FallbackCharField(
        max_length=200,
        fallback=app_settings.GREETING_TEXT,
    )
```

`openwisp_utils.fields.FallbackURLField`

This field extends Django's URLField and provides additional functionality for handling URL fields with a fallback value.

Note

- The field will return the **fallback value** whenever is set to `None`.

- Setting the same value as the **fallback value** will save `None` (NULL) in the database.

```
from django.db import models
from openwisp_utils.fields import FallbackURLField
from myapp import settings as app_settings


class MyModel(models.Model):
    password_reset_url = FallbackURLField(
        max_length=200,
        fallback=app_settings.DEFAULT_PASSWORD_RESET_URL,
    )
```

`openwisp_utils.fields.FallbackTextField`

This extends Django's TextField and provides additional functionality for handling text fields with a fallback value.

Note

- The field will return the **fallback value** whenever is set to `None`.
- Setting the same value as the **fallback value** will save `None` (NULL) in the database.

```
from django.db import models
from openwisp_utils.fields import FallbackTextField
from myapp import settings as app_settings


class MyModel(models.Model):
    extra_config = FallbackTextField(
        max_length=200,
        fallback=app_settings.EXTRA_CONFIG,
    )
```

`openwisp_utils.fields.FallbackPositiveIntegerField`

This extends Django's PositiveIntegerField and provides additional functionality for handling positive integer fields with a fallback value.

Note

- The field will return the **fallback value** whenever is set to `None`.
- Setting the same value as the **fallback value** will save `None` (NULL) in the database.

```
from django.db import models
from openwisp_utils.fields import FallbackPositiveIntegerField
from myapp import settings as app_settings


class MyModel(models.Model):
    count = FallbackPositiveIntegerField(
        fallback=app_settings.DEFAULT_COUNT,
    )
```

openwisp_utils.fields.FallbackDecimalField

This extends Django's DecimalField and provides additional functionality for handling decimal fields with a fallback value.

Note

- The field will return the **fallback value** whenever is set to `None`.
- Setting the same value as the **fallback value** will save `None` (NULL) in the database.

```
from django.db import models
from openwisp_utils.fields import FallbackDecimalField
from myapp import settings as app_settings


class MyModel(models.Model):
    price = FallbackDecimalField(
        max_digits=4,
        decimal_places=2,
        fallback=app_settings.DEFAULT_PRICE,
    )
```

Admin Utilities

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

`openwisp_utils.admin.TimeReadonlyAdminMixin`

Admin mixin which adds two read only fields `created` and `modified`.

This is an admin mixin for models inheriting `TimeStampedEditableModel` which adds the fields `created` and `modified` to the database.

`openwisp_utils.admin.ReadOnlyAdmin`

A read-only `ModelAdmin` base class.

Will include the `id` field by default, which can be excluded by supplying the `exclude` attribute, e.g.:

```
from openwisp_utils.admin import ReadOnlyAdmin


class PostAuthReadOnlyAdmin(ReadOnlyAdmin):
    exclude = ["id"]
```

`openwisp_utils.admin.AlwaysHasChangedMixin`

A mixin designed for inline items and model forms, ensures the item is created even if the default values are unchanged.

Without this, when creating new objects, inline items won't be saved unless users change the default values.

`openwisp_utils.admin.CopyableFieldsAdmin`

An admin class that allows to set admin fields to be read-only and makes it easy to copy the fields contents.

Useful for auto-generated fields such as UUIDs, secret keys, tokens, etc.

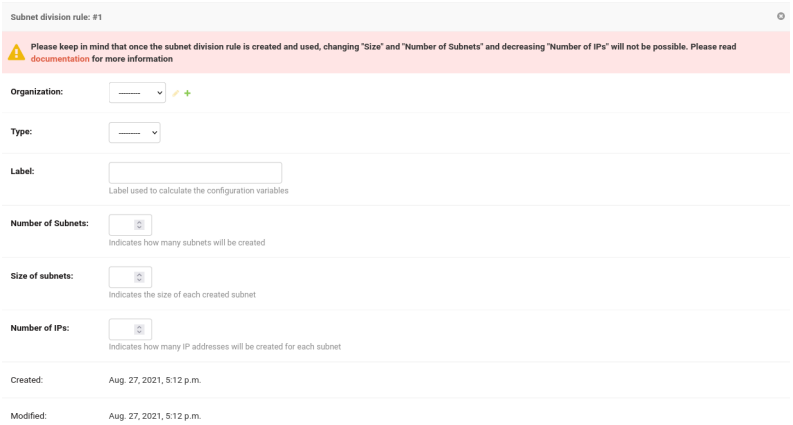
`openwisp_utils.admin.UUIDAdmin`

This class is a subclass of `CopyableFieldsAdmin` which sets `uuid` as the only copyable field. This class is kept for backward compatibility and convenience, since different models of various OpenWISP modules show `uuid` as the only copyable field.

`openwisp_utils.admin.ReceiveUrlAdmin`

An admin class that provides an URL as a read-only input field (to make it easy and quick to copy/paste).

`openwisp_utils.admin.HelpTextStackedInline`



A stacked inline admin class that displays a help text for entire inline object. Following is an example:

```
from openwisp_utils.admin import HelpTextStackedInline


class SubnetDivisionRuleInlineAdmin(
    MultitenantAdminMixin, TimeReadonlyAdminMixin, HelpTextStackedInline
):
    model = Model
    # It is required to set "help_text" attribute
    help_text = {
        # (required) Help text to display
        "text": _(
            "Please keep in mind that once the subnet division rule is created "
            'and used, changing "Size" and "Number of Subnets" and decreasing '
            '"Number of IPs" will not be possible.'
        ),
        # (optional) You can provide a link to documentation for user reference
        "documentation_url": (
            "https://github.com/openwisp/openwisp-utils"
        ),
        # (optional) Icon to be shown along with help text. By default it uses
        # "/static/admin/img/icon-alert.svg"
        "image_url": "/static/admin/img/icon-alert.svg",
    }
```

`openwisp_utils.admin_theme.filters.InputFilter`

The `admin_theme` sub app of this package provides an input filter that can be used in the *changelist* page to filter `UUIDField` or `CharField`.

Code example:

```
from django.contrib import admin
from openwisp_utils.admin_theme.filters import InputFilter
from my_app.models import MyModel


@admin.register(MyModel)
class MyModelAdmin(admin.ModelAdmin):
    list_filter = [
        ("my_field", InputFilter),
        "other_field",
        # ...
    ]
```

By default `InputFilter` use exact lookup to filter items which matches to the value being searched by the user. But this behavior can be changed by modifying `InputFilter` as following:

```
from django.contrib import admin
from openwisp_utils.admin_theme.filters import InputFilter
from my_app.models import MyModel


class MyInputFilter(InputFilter):
    lookup = "icontains"


@admin.register(MyModel)
class MyModelAdmin(admin.ModelAdmin):
    list_filter = [
        ("my_field", MyInputFilter),
```

```
        "other_field",
        # ...
    ]
```

To know about other lookups that can be used please check Django Lookup API Reference

`openwisp_utils.admin_theme.filters.SimpleInputFilter`

A stripped down version of `openwisp_utils.admin_theme.filters.InputFilter` that provides flexibility to customize filtering. It can be used to filter objects using indirectly related fields.

The derived filter class should define the `queryset` method as shown in following example:

```
from django.contrib import admin
from openwisp_utils.admin_theme.filters import SimpleInputFilter
from my_app.models import MyModel


class MyInputFilter(SimpleInputFilter):
    parameter_name = "shelf"
    title = _("Shelf")

    def queryset(self, request, queryset):
        if self.value() is not None:
            return queryset.filter(name__icontains=self.value())


@admin.register(MyModel)
class MyModelAdmin(admin.ModelAdmin):
    list_filter = [
        MyInputFilter,
        "other_field",
        # ...
    ]
```

`openwisp_utils.admin_theme.filters.AutocompleteFilter`

The `admin_theme` sub app of this package provides an auto complete filter that uses the *django-autocomplete* widget to load filter data asynchronously.

This filter can be helpful when the number of objects is too large to load all at once which may cause the slow loading of the page.

```
from django.contrib import admin
from openwisp_utils.admin_theme.filters import AutocompleteFilter
from my_app.models import MyModel, MyOtherModel


class MyAutoCompleteFilter(AutocompleteFilter):
    field_name = "field"
    parameter_name = "field_id"
    title = _("My Field")


@admin.register(MyModel)
class MyModelAdmin(admin.ModelAdmin):
    list_filter = [MyAutoCompleteFilter, ...]


@admin.register(MyOtherModel)
```

```
class MyOtherModelAdmin(admin.ModelAdmin):
    search_fields = ["id"]
```

To customize or know more about it, please refer to the django-admin-autocomplete-filter documentation.

Customizing the Submit Row in OpenWISP Admin

In the OpenWISP admin interface, the `submit_line.html` template controls the rendering of action buttons in the model form's submit row. OpenWISP Utils extends this template to allow the addition of custom buttons.

To add custom buttons, you can use the `additional_buttons` context variable. This variable should be a list of dictionaries, each representing a button with customizable properties such as type, class, value, title, URL, or even raw HTML content.

Here's an example of adding a custom button with both standard properties and raw HTML to the submit row in the `change_view` method:

```
from django.contrib import admin
from django.utils.safestring import mark_safe
from .models import MyModel


@admin.register(MyModel)
class MyModelAdmin(admin.ModelAdmin):
    def change_view(
        self, request, object_id, form_url="", extra_context=None
    ):
        extra_context = extra_context or {}
        extra_context["additional_buttons"] = [
            {
                "type": "button",
                "class": "btn btn-secondary",
                "value": "Custom Action",
                "title": "Perform a custom action",
                "url": "https://example.com",
            },
            {
                "raw_html": mark_safe(
                    '<button type="button" class="btn btn-warning" '
                    "onclick=\"alert('This is a raw HTML button!')\">"
                    "Raw HTML Button</button>"
                )
            },
        ]
        return super().change_view(
            request, object_id, form_url, extra_context
        )
```

In this example, two buttons are added to the submit row:

1. A standard button labeled "Custom Action" with a link to *https://example.com*.
2. A button rendered using raw HTML that triggers an alert when clicked, labeled "Raw HTML Button." The raw HTML is wrapped in *mark_safe* to ensure it is rendered correctly.

The *mark_safe* function is necessary to ensure that the raw HTML is rendered as HTML and not escaped as plain text.

Test Utilities

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

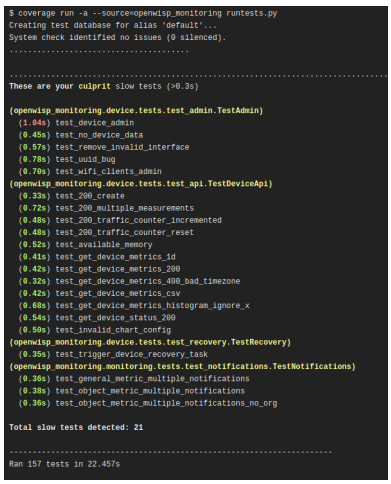
openwisp_utils.tests.catch_signal

This method can be used to mock a signal call in order to easily verify that the signal has been called.

Usage example as a context-manager:

```
from openwisp_utils.tests import catch_signal

with catch_signal(openwisp_signal) as handler:
    model_instance.trigger_signal()
    handler.assert_called_once_with(
        arg1="value1",
        arg2="value2",
        sender=ModelName,
        signal=openwisp_signal,
    )
```

openwisp_utils.tests.TimeLoggingTestRunner



This class extends the default test runner provided by Django and logs the time spent by each test, making it easier to spot slow tests by highlighting time taken by it in yellow (time shall be highlighted in red if it crosses the second threshold).

By default tests are considered slow if they take more than 0.3 seconds but you can control this with OPENWISP_SLOW_TEST_THRESHOLD.

In order to switch to this test runner you have set the following in your *settings.py*:

```
TEST_RUNNER = "openwisp_utils.tests.TimeLoggingTestRunner"
```

openwisp_utils.tests.capture_stdout

This decorator can be used to capture standard output produced by tests, either to silence it or to write assertions.

Example usage:

```
from openwisp_utils.tests import capture_stdout


@capture_stdout()
def test_something(self):
    function_generating_output()  # pseudo code


@capture_stdout()
def test_something_again(self, captured_ouput):
    # pseudo code
    function_generating_output()
    # now you can create assertions on the captured output
    self.assertIn("expected stdout", captured_ouput.getvalue())
    # if there are more than one assertions, clear the captured output first
    captured_error.truncate(0)
    captured_error.seek(0)
    # you can create new assertion now
    self.assertIn("another output", captured_ouput.getvalue())
```

Notes:

- If assertions need to be made on the captured output, an additional argument (in the example above is named `captured_output`) can be passed as an argument to the decorated test method, alternatively it can be omitted.
- A `StingIO` instance is used for capturing output by default but if needed it's possible to pass a custom `StringIO` instance to the decorator function.

openwisp_utils.tests.capture_stderr

Equivalent to `capture_stdout`, but for standard error.

Example usage:

```
from openwisp_utils.tests import capture_stderr


@capture_stderr()
def test_error(self):
    function_generating_error()  # pseudo code


@capture_stderr()
def test_error_again(self, captured_error):
    # pseudo code
    function_generating_error()
```

```
    # now you can create assertions on captured error
    self.assertIn("expected error", captured_error.getvalue())
    # if there are more than one assertions, clear the captured error first
    captured_error.truncate(0)
    captured_error.seek(0)
    # you can create new assertion now
    self.assertIn("another expected error", captured_error.getvalue())
```

`openwisp_utils.tests.capture_any_output`

Equivalent to `capture_stdout` and `capture_stderr`, but captures both types of output (standard output and standard error).

Example usage:

```
from openwisp_utils.tests import capture_any_output


@capture_any_output()
def test_something_out(self):
    function_generating_output()  # pseudo code


@capture_any_output()
def test_out_again(self, captured_output, captured_error):
    # pseudo code
    function_generating_output_and_errors()
    # now you can create assertions on captured error
    self.assertIn("expected stdout", captured_output.getvalue())
    self.assertIn("expected stderr", captured_error.getvalue())
```

`openwisp_utils.tests.AssertNumQueriesSubTestMixin`

This mixin overrides the assertNumQueries assertion from the django test case to run in a `subTest` so that the query check does not block the whole test if it fails.

Example usage:

```
from django.test import TestCase
from openwisp_utils.tests import AssertNumQueriesSubTestMixin


class MyTest(AssertNumQueriesSubTestMixin, TestCase):
    def my_test(self):
        with self.assertNumQueries(2):
            MyModel.objects.count()

        # the assertion above will fail but this line will be executed
        print("This will be printed anyway.")
```

`openwisp_utils.test_selenium_mixins.SeleniumTestMixin`

This mixin provides basic setup for Selenium tests with method to open URL and login and logout a user.

Other Utilities

Note

This documentation page is aimed at developers who want to customize, change or extend the code of OpenWISP Utils in order to modify its behavior (e.g.: for personal or commercial purposes or to fix a bug, implement a new feature or contribute to the project in general).

If you aren't a developer and you are looking for information on how to use OpenWISP, please refer to:

- General OpenWISP Quickstart
- OpenWISP Utils User Docs

Model Utilities

`openwisp_utils.base.UUIDModel`

Model class which provides a UUID4 primary key.

`openwisp_utils.base.TimeStampedEditableModel`

Model class inheriting `UUIDModel` which provides two additional fields:

- `created`
- `modified`

Which use respectively `AutoCreatedField`, `AutoLastModifiedField` from `model_utils.fields` (self-updating fields providing the creation date-time and the last modified date-time).

REST API Utilities

`openwisp_utils.api.serializers.ValidatedModelSerializer`

A model serializer which calls the model instance `full_clean()`.

`openwisp_utils.api.apps.ApiAppConfig`

If you're creating an OpenWISP module which provides a REST API built with Django REST Framework, chances is that you may need to define some default settings to control its throttling or other aspects.

Here's how to easily do it:

```
from django.conf import settings
from django.utils.translation import ugettext_lazy as _
from openwisp_utils.api.apps import ApiAppConfig


class MyModuleConfig(ApiAppConfig):
    name = "my_openwisp_module"
    label = "my_module"
    verbose_name = _("My OpenWISP Module")

    # assumes API is enabled by default
    API_ENABLED = getattr(
        settings, "MY_OPENWISP_MODULE_API_ENABLED", True
    )
    # set throttling rates for your module here
    REST_FRAMEWORK_SETTINGS = {
        "DEFAULT_THROTTLE_RATES": {"my_module": "400/hour"},
    }
```

Every openwisp module which has an API should use this class to configure its own default settings, which will be merged with the settings of the other modules.

Storage Utilities

`openwisp_utils.storage.CompressStaticFilesStorage`

A static storage backend for compression inheriting from django-compress-staticfiles's `CompressStaticFilesStorage` class.

Adds support for excluding file types using OPENWISP_STATICFILES_VERSIONED_EXCLUDE setting.

To use point `STATICFILES_STORAGE` to `openwisp_utils.storage.CompressStaticFilesStorage` in `settings.py`.

```
STATICFILES_STORAGE = "openwisp_utils.storage.CompressStaticFilesStorage"
```

Other Utilities

`openwisp_utils.utils.get_random_key`

Generates an random string of 32 characters.

`openwisp_utils.utils.deep_merge_dicts`

Returns a new `dict` which is the result of the merge of the two dictionaries, all elements are deep-copied to avoid modifying the original data structures.

Usage:

```
from openwisp_utils.utils import deep_merge_dicts

mergd_dict = deep_merge_dicts(dict1, dict2)
```

openwisp_utils.utils.default_or_test

If the program is being executed during automated tests the value supplied in the `test` argument will be returned, otherwise the one supplied in the `value` argument is returned.

```
from openwisp_utils.utils import default_or_test

THROTTLE_RATE = getattr(
    settings,
    "THROTTLE_RATE",
    default_or_test(value="20/day", test=None),
)
```

openwisp_utils.utils.print_color

default colors: `['white_bold', 'green_bold', 'yellow_bold', 'red_bold']`

If you want to print a string in `Red Bold`, you can do it as below.

```
from openwisp_utils.utils import print_color

print_color("This is the printed in Red Bold", color_name="red_bold")
```

You may also provide the `end` argument similar to built-in print method.

openwisp_utils.utils.SorrtedOrderedDict

Extends `collections.SortedDict` and implements logic to sort inserted items based on `key` value. Sorting is done at insert operation which incurs memory space overhead.

openwisp_utils.tasks.OpenwispCeleryTask

A custom celery task class that sets hard and soft time limits of celery tasks using OPENWISP_CELERY_HARD_TIME_LIMIT and OPENWISP_CELERY_SOFT_TIME_LIMIT settings respectively.

Usage:

```
from celery import shared_task

from openwisp_utils.tasks import OpenwispCeleryTask


@shared_task(base=OpenwispCeleryTask)
def your_celery_task():
    pass
```

Note: This task class should be used for regular background tasks but not for complex background tasks which can take a long time to execute (e.g.: firmware upgrades, network operations with retry mechanisms).

openwisp_utils.utils.retryable_request

A utility function for making HTTP requests with built-in retry logic. This function is useful for handling transient errors encountered during HTTP requests by automatically retrying failed requests with exponential backoff. It provides flexibility in configuring various retry parameters to suit different use cases.

Usage:

```
from openwisp_utils.utils import retryable_request

response = retryable_request(
    method="GET",
```

```
    url="https://openwisp.org",
    timeout=(4, 8),
    max_retries=3,
    backoff_factor=1,
    backoff_jitter=0.0,
    status_forcelist=(429, 500, 502, 503, 504),
    allowed_methods=(
        "HEAD",
        "GET",
        "PUT",
        "DELETE",
        "OPTIONS",
        "TRACE",
        "POST",
    ),
    retry_kwargs=None,
    headers={"Authorization": "Bearer token"},
)
```

Paramters:

- `method` (str): The HTTP method to be used for the request in lower case (e.g., 'get', 'post', etc.).
- `timeout` (tuple): A tuple containing two elements: connection timeout and read timeout in seconds (default: (4, 8)).
- `max_retries` (int): The maximum number of retry attempts in case of request failure (default: 3).
- `backoff_factor` (float): A factor by which the retry delay increases after each retry (default: 1).
- `backoff_jitter` (float): A jitter to apply to the backoff factor to prevent retry storms (default: 0.0).
- `status_forcelist` (tuple): A tuple of HTTP status codes for which retries should be attempted (default: (429, 500, 502, 503, 504)).
- `allowed_methods` (tuple): A tuple of HTTP methods that are allowed for the request (default: ('HEAD', 'GET', 'PUT', 'DELETE', 'OPTIONS', 'TRACE', 'POST')).
- `retry_kwargs` (dict): Additional keyword arguments to be passed to the retry mechanism (default: None).
- `**kwargs`: Additional keyword arguments to be passed to the underlying request method (e.g. 'headers', etc.).

This method will raise a `requests.exceptions.RetryError` if the request remains unsuccessful even after all retry attempts have been exhausted. This exception indicates that the operation could not be completed successfully despite the retry mechanism.

Re-usable GitHub Workflows

Replicate Commits to Version Branch

This re-usable workflow replicates commits from the `master` branch to a version branch. The version branch name is derived from the version of the Python package specified in the workflow.

Version branches are essential during development to ensure that each OpenWISP module depends on compatible versions of its OpenWISP dependencies. Without version branches, modules depending on the `master` branch of other modules may encounter errors, as the `master` branch could include future changes that are incompatible with previous versions. This makes it impossible to build a specific commit reliably after such changes.

To address this, we use version branches so that each module can depend on a compatible version of its dependencies during development. Managing these version branches manually is time-consuming, which is why this re-usable GitHub workflow automates the process of keeping version branches synchronized with the `master` branch.

You can invoke this workflow from another workflow using the following example:

```yaml
name: Replicate Commits to Version Branch

on:
  push:
    branches:
      - master

jobs:
  version-branch:
    uses: openwisp/openwisp-utils/.github/workflows/reusable-version-branch.yml@master
    with:
      # The name of the Python package (required)
      module_name: openwisp_utils
      # Whether to install the Python package. Defaults to false.
      install_package: true
```

Note

If the `master` branch is force-pushed, this workflow will fail due to conflicts. To resolve this, you must manually synchronize the version branch with the `master` branch. You can use the following commands to perform this synchronization:

```
VERSION=<enter-version-number> # e.g. 1.2
git fetch origin
git checkout $VERSION
git reset --hard origin/master
git push origin $VERSION --force-with-lease
```

Other useful resources:

- Settings

OpenWrt Agents

OpenWISP Config Agent

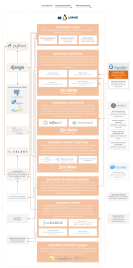
Seealso

Source code: github.com/openwisp/openwisp-config.

OpenWISP Config is an OpenWrt configuration agent that automates network management tasks. It interfaces with the OpenWISP Controller to streamline configuration deployment.

For a comprehensive overview of features, please refer to the OpenWISP Config: Features page.

The following diagram illustrates the role of the OpenWrt Config Agent in the OpenWISP architecture.



OpenWISP Architecture: highlighted OpenWISP Config Agent for OpenWrt

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

OpenWISP Config: Features

OpenWISP Config agent provides the following features:

- Fetches the latest configuration from the OpenWISP Controller, ensuring devices stay up-to-date.
- Combines centrally managed settings with local configurations, preserving local overrides.
- Performs rollback of previous configuration when the new configuration fails to apply.
- Simplifies onboarding by automatically registering devices with the controller using a shared secret.
- Supports OpenWrt hotplug events.

Quick Start Guide

To install the Config Agent on your OpenWrt system, follow these steps:

Download and install the latest build from downloads.openwisp.io. Copy the URL of the IPK file you want to download, then run the following commands on your OpenWrt device:

```
cd /tmp  # /tmp runs in memory
wget <URL-just-copied>
opkg update
opkg install ./<file-just-downloaded>
```

Replace `<URL-just-copied>` with the URL of the package from downloads.openwisp.io.

You can also install from the official OpenWrt packages:

```
opkg update
opkg install openwisp-config
```

Important

We recommend installing from our latest builds because the OpenWrt packages are not always up to date.

Once the config agent is installed, you need to configure it. Edit the config file located at `/etc/config/openwisp`.

You will see the default config file, as shown below.

```
# For more information about the config options please see the README
# or https://openwisp.io/docs/stable/openwrt-config-agent/user/settings.html

config controller 'http'
    #option url 'https://openwisp2.mynetwork.com'
    #option interval '120'
    #option verify_ssl '1'
    #option shared_secret ''
    #option consistent_key '1'
```

```
#option mac_interface 'eth0'
#option management_interface 'tun0'
#option merge_config '1'
#option test_config '1'
#option test_script '/usr/sbin/mytest'
#option hardware_id_script '/usr/sbin/read_hw_id'
#option hardware_id_key '1'
option uuid ''
option key ''
# curl options
#option connect_timeout '15'
#option max_time '30'
#option capath '/etc/ssl/certs'
#option cacert '/etc/ssl/certs/ca-certificates.crt'
# hooks
#option pre_reload_hook '/usr/sbin/my_pre_reload_hook'
#option post_reload_hook '/usr/sbin/my_post_reload_hook'
```

Uncomment and change the following fields:

- `url`: the hostname of your OpenWISP controller. For example, if you are hosting your OpenWISP server locally and set the IP Address to "192.168.56.2", the URL would be `https://192.168.56.2`.
- `verify_ssl`: set to `'0'` if your controller's SSL certificate is self-signed; in production, you need a valid SSL certificate to keep your instance secure.
- `shared_secret`: you can retrieve this from the OpenWISP admin panel, in the Organization settings. The list of organizations is available at `/admin/openwisp_users/organization/`.
- `management_interface`: this is the interface which OpenWISP uses to reach the device. Please refer to Setting Up the Management Network for more information.

Note

When testing or developing using the Django development server directly from your computer, make sure the server listens on all interfaces (`./manage.py runserver 0.0.0.0:8000`) and then just point OpenWISP Config to use your local IP address (e.g., `http://192.168.1.34:8000`).

Save the file and start openwisp-config:

```
/etc/init.d/openwisp-config restart
```

Your OpenWrt device should register itself to your OpenWISP controller. Check the devices page in the OpenWISP admin dashboard to make sure your device has registered successfully.

Seealso

- For troubleshooting and debugging, refer to Debugging.
- To learn more about the configuration options of the config agent, refer to Settings.
- For instructions on how to compile the package, refer to Compiling a Custom OpenWrt Image.
- Read about the complementary Monitoring Agent.

Settings

Configuration Options

UCI configuration options must go in `/etc/config/openwisp`.

- `url`: URL of controller, e.g.: `https://demo.openwisp.io`
- `interval`: time in seconds between checks for changes to the configuration, defaults to `120`
- `management_interval`: time in seconds between the management ip discovery attempts, defaults to `$interval/12`
- `registration_interval`: time in seconds between the registration attempts, defaults to `$interval/4`
- `verify_ssl`: whether SSL verification must be performed or not, defaults to `1`
- `shared_secret`: shared secret, needed for Automatic registration
- `consistent_key`: whether Consistent Key Generation is enabled or not, defaults to `1`
- `merge_config`: whether Merge Configuration is enabled or not, defaults to `1`
- `tags`: template tags to use during registration, multiple tags separated by space can be used, for more information see Template Tags
- `test_config`: whether a new configuration must be tested before being considered applied, defaults to `1`
- `test_retries`: maximum number of retries when doing the default configuration test, defaults to `3`
- `test_script`: custom test script, read more about this feature in Configuration Test
- `uuid`: unique identifier of the router configuration in the controller application
- `key`: key required to download the configuration
- `hardware_id_script`: custom script to read out a hardware id (e.g. a serial number), read more about this feature in Hardware ID
- `hardware_id_key`: whether to use the hardware id for key generation or not, defaults to `1`
- `bootup_delay`: maximum value in seconds of a random delay after boot, defaults to `10`, see Boot Up Delay
- `unmanaged`: list of config sections which won't be overwritten, see Unmanaged Configurations
- `capath`: value passed to curl `--capath` argument, by default is empty; see also curl capath argument
- `cacert`: value passed to curl `--cacert` argument, by default is empty; see also curl cacert argument
- `connect_timeout`: value passed to curl `--connect-timeout` argument, defaults to `15`; see curl connect-timeout argument
- `max_time`: value passed to curl `--max-time` argument, defaults to `30`; see curl max-time argument

- `mac_interface`: the interface from which the MAC address is taken when performing automatic registration, defaults to `eth0`
- `management_interface`: management interface name (both openwrt UCI names and linux interface names are supported), it's used to collect the management interface ip address and send this information to the OpenWISP server, for more information please read how to make sure OpenWISP can reach your devices
- `default_hostname`: if your firmware has a custom default hostname, you can use this configuration option so the agent can recognize it during registration and replicate the standard behavior (new device will be named after its mac address, to avoid having many new devices with the same name), the possible options are to either set this to the value of the default hostname used by your firmware, or set it to `*` to always force to register new devices using their mac address as their name (this last option is useful if you have a firmware which can work on different hardware models and each model has a different default hostname)
- `pre_reload_hook`: path to custom executable script, see pre-reload-hook
- `post_reload_hook`: path to custom executable script, see post-reload-hook
- `post_reload_delay`: delay in seconds to wait before the post-reload-hook and any configuration test, defaults to `5`
- `post_registration_hook`: path to custom executable script, see post-registration-hook
- `respawn_threshold`: time in seconds used as procd respawn threshold, defaults to `3600`
- `respawn_timeout`: time in seconds used as procd respawn timeout, defaults to `5`
- `respawn_retry`: number of procd respawn retries (use `0` for infinity), defaults to `5`
- `checksum_max_retries`: maximum number of retries for checksum requests which fail with 404, defaults to `5`, after these failures the agent will assume the device has been deleted from OpenWISP Controller and will exit; please keep in mind that due to `respawn_retry`, procd will try to respawn the agent after it exits, so the total number of attempts which will be tried has to be calculated as: `checksum_max_retries * respawn_retry`
- `checksum_retry_delay`: time in seconds between retries, defaults to `6`

Merge Configuration

By default the remote configuration is merged with the local one. This has several advantages:

- less boilerplate configuration stored in the remote controller
- local users can change local configurations without fear of losing their changes

It is possible to turn this feature off by setting `merge_config` to `0` in `/etc/config/openwisp`.

Details about the merging behavior:

- if a configuration option or list is present both in the remote configuration and in the local configuration, the remote configurations will overwrite the local ones
- configuration options that are present in the local configuration but are not present in the remote configuration will be retained
- configuration files that were present in the local configuration and are replaced by the remote configuration are backed up and eventually restored if the modifications are removed from the controller

Configuration Test

When a new configuration is downloaded, the agent will first backup the current running configuration, then it will try to apply the new one and perform a basic test, which consists in trying to contact the controller again;

If the test succeeds, the configuration is considered applied and the backup is deleted.

If the test fails, the backup is restored and the agent will log the failure via syslog (see Debugging for more information on auditing logs).

Disable Testing

To disable this feature, set the `test_config` option to `0`, then reload/restart *openwisp-config*.

Define Custom Tests

If the default test does not satisfy your needs, you can define your own tests in an **executable** script and indicate the path to this script in the `test_script` config option.

If the exit code of the executable script is higher than `0` the test will be considered failed.

Hardware ID

It is possible to use a unique hardware id for device identification, for example a serial number.

If `hardware_id_script` contains the path to an executable script, it will be used to read out the hardware id from the device. The hardware id will then be sent to the controller when the device is registered.

If the above configuration option is set then the hardware id will also be used for generating the device key, instead of the mac address. If you use a hardware id script but prefer to use the mac address for key generation then set `hardware_id_key` to `0`.

See also the related hardware ID settings in OpenWISP Controller.

Boot Up Delay

The option `bootup_delay` is used to delay the initialization of the agent for a random amount of seconds after the device boots.

The value specified in this option represents the maximum value of the range of possible random values, the minimum value being `0`.

The default value of this option is 10, meaning that the initialization of the agent will be delayed for a random number of seconds, this random number being comprised between `0` and `10`.

This feature is used to spread the load on the OpenWISP server when a large amount of devices boot at the same time after a blackout.

Large OpenWISP installations may want to increase this value.

Hooks

Warning

Hooks are deprecated in favor of Hotplug events.

Below are described the available hooks in *openwisp-config*.

`pre-reload-hook`

Defaults to `/etc/openwisp/pre-reload-hook`; the hook is not called if the path does not point to an executable script file.

This hook is called each time *openwisp-config* applies a configuration, but **before services are reloaded**, more precisely in these situations:

- after a new remote configuration is downloaded and applied

- after a configuration test failed (see Configuration Test) and a previous backup is restored

You can use this hook to perform custom actions before services are reloaded, e.g.: to perform auto-configuration with LibreMesh.

Example configuration:

```
config controller 'http'
        ...
        option pre_reload_hook '/usr/sbin/my-pre-reload-hook'
        ...
```

Complete example:

```
# set hook in configuration
uci set openwisp.http.pre_reload_hook='/usr/sbin/my-pre-reload-hook'
uci commit openwisp
# create hook script
cat <<EOF > /usr/sbin/my-pre-reload-hook
#!/bin/sh
# put your custom operations here
EOF
# make script executable
chmod +x /usr/sbin/my-pre-reload-hook
# reload openwisp-config by using procd's convenient utility
reload_config
```

`post-reload-hook`

Defaults to `/etc/openwisp/post-reload-hook`; the hook is not called if the path does not point to an executable script file.

Same as *pre_reload_hook* but with the difference that this hook is called after the configuration services have been reloaded.

`post-registration-hook`

Defaults to `/etc/openwisp/post-registration-hook`;

Path to an executable script that will be called after the registration is completed.

Unmanaged Configurations

In some cases it could be necessary to ensure that some configuration sections won't be overwritten by the controller.

These settings are called "unmanaged", in the sense that they are not managed remotely. In the default configuration of *openwisp-config* there are no unmanaged settings.

Example unmanaged settings:

```
config controller 'http'
        ...
        list unmanaged 'system.@led'
        list unmanaged 'network.loopback'
        list unmanaged 'network.@switch'
        list unmanaged 'network.@switch_vlan'
        ...
```

Note the lines with the @ sign; this syntax means any UCI section of the specified type will be unmanaged.

In the previous example, the loopback interface, all `led` settings, all `switch` and `switch_vlan` directives will never be overwritten by the remote configuration and will only be editable via SSH or via the web interface.

Automatic registration

When the agent starts, if both `uuid` and `key` are not defined, it will consider the router to be unregistered and it will attempt to perform an automatic registration.

The automatic registration is performed only if `shared_secret` is correctly set.

The device will choose as name one of its mac addresses, unless its hostname is not `OpenWrt`, in the latter case it will simply register itself with the current hostname.

When the registration is completed, the agent will automatically set `uuid` and `key` in `/etc/config/openwisp`.

To enable this feature by default on your firmware images, follow the procedure described in Compiling a Custom OpenWrt Image.

Consistent Key Generation

When using Automatic registration, this feature allows devices to keep the same configuration even if reset or reflashed.

The `key` is generated consistently with an operation like `md5sum(mac_address + shared_secret)`; this allows the controller application to recognize that an existing device is registering itself again.

The `mac_interface` configuration key specifies which interface is used to calculate the mac address, this setting defaults to `eth0`. If no `eth0` interface exists, the first non-loopback, non-bridge and non-tap interface is used. You won't need to change this setting often, but if you do, ensure you choose a physical interface which has constant mac address.

The "Consistent key generation" feature is enabled by default, but must be enabled also in the controller application in order to work.

Hotplug Events

The agent sends the following Hotplug events:

- After the registration is successfully completed: `post-registration`
- After the registration failed: `registration-failed`
- When the agent first starts after the booting process: `bootup`
- After any subsequent restart: `restart`
- After the configuration has been successfully applied: `config-applied`
- After the previous configuration has been restored: `config-restored`
- Before services are reloaded: `pre-reload`
- After services have been reloaded: `post-reload`
- After the agent has finished its check cycle, before going to sleep: `end-of-cycle`

If a hotplug event is sent by *openwisp-config* then all scripts existing in `/etc/hotplug.d/openwisp/` will be executed. In scripts the type of event is visible in the variable `$ACTION`. For example, a script to log the hotplug events, `/etc/hotplug.d/openwisp/01_log_events`, could look like this:

```
#!/bin/sh

logger "openwisp-config sent a hotplug event. Action: $ACTION"
```

It will create log entries like this:

```
Wed Jun 22 06:15:17 2022 user.notice root: openwisp-config sent a hotplug event. Action: reg
```

For more information on using these events refer to the Hotplug Events OpenWrt Documentation.

Compiling a Custom OpenWrt Image

If you are managing many devices and customizing your `openwisp-config` configuration by hand on each new device, you should switch to using a custom OpenWrt firmware image that includes `openwisp-config` and its precompiled configuration file, this strategy has a few important benefits:

- you can save yourself the effort of installing and configuring `openwisp-config` on each device
- you can enable Automatic registration by setting `shared_secret`, hence saving extra time and effort to register each device on the controller app
- if you happen to reset the firmware to initial settings, these precompiled settings will be restored as well

The following procedure illustrates how to compile a custom OpenWrt image with a precompiled minimal `/etc/config/openwisp` configuration file:

```
git clone https://github.com/openwrt/openwrt.git openwrt
cd openwrt
git checkout <openwrt-branch>

# include precompiled file
mkdir -p files/etc/config
cat <<EOF > files/etc/config/openwisp
config controller 'http'
    # change the values of the following 2 options
    option url 'https://demo.openwisp.io'
    option shared_secret 'nzXTd7qpXKPNdrWZDsYoMxbGpOrEVjeD'
EOF

# configure feeds
echo "src-git openwisp https://github.com/openwisp/openwisp-config.git" > feeds.conf
cat feeds.conf.default >> feeds.conf
./scripts/feeds update -a
./scripts/feeds install -a
# replace with your desired arch target
arch="ar71xx"
echo "CONFIG_TARGET_$arch=y" > .config
echo "CONFIG_PACKAGE_openwisp-config=y" >> .config
make defconfig
# compile with verbose output
make -j1 V=s
```

Automate Compilation for Different Organizations

If you are working with OpenWISP, there are chances you may be compiling several images for different organizations (clients or non-profit communities) and use cases (full featured, mesh, 4G, etc).

Doing this by hand without tracking your changes can lead you into a very disorganized and messy situation.

To alleviate this pain you can use ansible-openwisp2-imagegenerator.

Debugging

Debugging *openwisp-config* can be easily done by using the `logread` command:

```
logread
```

Use grep to filter out any other log message:

```
logread | grep openwisp
```

If you are in doubt openwisp-config is running at all, you can check with:

```
ps | grep openwisp
```

You should see something like:

```
3800 root      1200 S    {openwisp-config} /bin/sh /usr/sbin/openwisp-config --url https://d
```

You can inspect the version of openwisp-config currently installed with:

```
openwisp-config --version
```

Developer Documentation

Note

This page is for developers who want to customize or extend OpenWISP Config, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWISP Config User Docs

Compiling openwisp-config

The following procedure illustrates how to compile *openwisp-config* and its dependencies:

```
git clone https://github.com/openwrt/openwrt.git openwrt
cd openwrt
git checkout <openwrt-branch>

# configure feeds
echo "src-git openwisp https://github.com/openwisp/openwisp-config.git" > feeds.conf
cat feeds.conf.default >> feeds.conf
./scripts/feeds update -a
./scripts/feeds install -a
# any arch/target is fine because the package is architecture indipendent
arch="ar71xx"
echo "CONFIG_TARGET_$arch=y" > .config;
echo "CONFIG_PACKAGE_openwisp-config=y" >> .config
make defconfig
make tools/install
make toolchain/install
make package/openwisp-config/compile
```

Alternatively, you can configure your build interactively with `make menuconfig`, in this case you will need to select *openwisp-config* by going to `Administration > openwisp`:

```
git clone https://github.com/openwrt/openwrt.git openwrt
cd openwrt
git checkout <openwrt-branch>

# configure feeds
echo "src-git openwisp https://github.com/openwisp/openwisp-config.git" > feeds.conf
cat feeds.conf.default >> feeds.conf
./scripts/feeds update -a
./scripts/feeds install -a
```

```
make menuconfig
# go to Administration > openwisp and select the variant you need interactively
make -j1 V=s
```

Quality Assurance Checks

We use LuaFormatter and shfmt to format lua files and shell scripts respectively.

First of all, you will need install the lua packages mentioned above, then you can format all files with:

```
./qa-format
```

To run quality assurance checks you can use the `run-qa-checks` script:

```
# install openwisp-utils QA tools first
pip install openwisp-utils[qa]

# run QA checks before committing code
./run-qa-checks
```

Run tests

To run the unit tests, you must install the required dependencies first; to do this, you can take a look at the install-dev.sh script.

You can run all the unit tests by launching the dedicated script:

```
./runtests
```

Alternatively, you can run specific tests, e.g.:

```
cd openwisp-config/tests/
lua test_utils.lua -v
```

OpenWISP Monitoring Agent

Seealso

Source code: github.com/openwisp/openwrt-openwisp-monitoring.

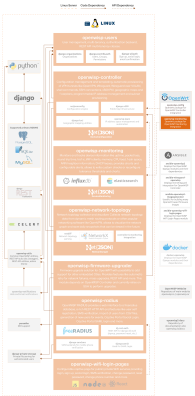
The OpenWISP Monitoring OpenWrt agent is responsible for collecting monitoring metrics from network devices and sending them to a central OpenWISP Monitoring Server via HTTPS, allowing to collect critical network metrics without the need of a VPN.

These metrics include:

- General system information, uptime
- Interface traffic
- WiFi client statistics
- CPU load averages
- Memory usage
- Storage space and usage
- Cellular Modem Status, Cellular Signal Quality/Strength

By collecting this data, administrators gain valuable insights into network health and performance, facilitating proactive troubleshooting of potential issues.

The following diagram illustrates the role of OpenWrt Monitoring Agent within the OpenWISP architecture.



OpenWISP Architecture: highlighted OpenWrt Monitoring Agent

Important

For an enhanced viewing experience, open the image above in a new browser tab.

Refer to Architecture, Modules, Technologies for more information.

Quick Start Guide

To install the Monitoring Agent on your OpenWrt system, follow these steps.

Download and install the latest builds of both *netjson-monitoring* and *openwisp-monitoring* from downloads.openwisp.io. Copy the URL of the IPK file you want to download, then run the following commands on your OpenWrt device:

```
cd /tmp  # /tmp runs in memory
opkg update
# Install netjson-monitoring first
wget <URL-just-copied>
opkg install ./<file-just-downloaded>
# Install openwisp-monitoring last
wget <URL-just-copied>
opkg install ./<file-just-downloaded>
```

Replace `<URL-just-copied>` with the URL of the respective package from downloads.openwisp.io.

Now you can start the agent:

```
/etc/init.d/openwisp-monitoring start
```

Seealso

- For troubleshooting and debugging, refer to Debugging.
- To learn more about the configuration options of the monitoring agent, refer to Settings.
- For instructions on how to compile the package, refer to Compiling the Monitoring Agent.
- Read about the complementary Config Agent.

Settings

Configuration Options

UCI configuration options should be placed in `/etc/config/openwisp-monitoring`.

- `monitored_interfaces`: Specifies the interfaces to be monitored. Defaults to `*`, meaning all interfaces.
- `interval`: Sets the interval in seconds for the agent to send data to the server. The default is `300` seconds.
- `verbose_mode`: Can be enabled by setting to `1` to assist in debugging. The default is `0` (disabled).
- `required_memory`: Minimum available memory required to temporarily store data. Defaults to `0.05` (5 percent).
- `max_retries`: Maximum number of retries if there is a failure in sending data to the server. The default is `5` retries.
- `bootup_delay`: Maximum value, in seconds, of a random delay after boot-up. Defaults to `10`. See Boot-Up Delay.

If the maximum retries are reached, the agent will attempt to send data in the next cycle.

Collecting vs. Sending

The monitoring agent uses two procd services: one for collecting data and another for sending it.

This setup allows for more flexible handling of data transmission failures. Data collected during network outages can be sent later, while new data continues to be collected. If there is a backlog of data to upload, the collection process will continue independently.

The monitoring agent operates in two modes: `send` and `collect`.

Collect Mode

When the OpenWISP monitoring agent operates in this mode, it is responsible for collecting and storing data.

The agent periodically checks if there is enough memory available. If sufficient memory is detected, data will be collected and saved in temporary storage with a timestamp (in UTC).

Once the data is stored, a signal is sent to the other agent to ensure the data is transmitted promptly.

Important

Ensure that the date and time on the device are correctly set. Incorrect timestamps can lead to inaccurate data in the time series database.

Send Mode

When operating in this mode, the OpenWISP monitoring agent handles data transmission.

The agent checks for available data files in temporary storage. If no data files are found, the agent will wait for the specified interval and check again. This process continues until data files are detected. If a signal is received from the other agent, the wait will be interrupted, and the agent will start sending data.

If the agent fails to send data, a randomized backoff (between 2 and 15 seconds) is used to retry until the `max_retries` limit is reached. If all attempts fail, the agent will try again in the next cycle.

Upon successful data transmission, the corresponding data file is deleted, and the agent checks for any remaining files.

SIGUSR1 signals are used to trigger immediate data transmission when new data is collected. The service will continue to attempt data transmission at regular intervals.

Boot-Up Delay

The `bootup_delay` option introduces a random delay during the agent's initialization after the device boots.

This option specifies the maximum value for the random delay, with a minimum value of `0`.

The default setting is 10, meaning the agent's initialization will be delayed by a random number of seconds, ranging from `0` to `10`.

This feature is designed to distribute the load on the OpenWISP server when a large number of devices boot simultaneously after a power outage.

Large OpenWISP installations may benefit from increasing this value.

Debugging

Debugging the *openwisp-monitoring package* can be easily done by using the `logread` command:

```
logread | grep openwisp-monitoring
```

In case of any issue, you can enable verbose_mode.

If you are in that doubt openwisp-monitoring is running at all or not, you can check with:

```
ps | grep openwisp-monitoring
```

You should see something like:

```
2712 root      1224 S    /bin/sh /usr/sbin/openwisp-monitoring --interval 300 --monitored_in
2713 root      1224 S    /bin/sh /usr/sbin/openwisp-monitoring --url https://demo.openwisp.i
```

You can inspect the version of openwisp-monitoring currently installed with:

```
openwisp-monitoring --version
```

Developer Documentation

Note

This page is for developers who want to customize or extend the OpenWrt package for OpenWISP Monitoring, whether for bug fixes, new features, or contributions.

For user guides and general information, please see:

- General OpenWISP Quickstart
- OpenWrt OpenWISP Monitoring Docs

Compiling the Monitoring Agent 460

Compiling the Monitoring Agent

This repository ships 2 OpenWrt packages:

- **netjson-monitoring**: provides NetJSON DeviceMonitoring output
- **openwisp-monitoring**: daemon which collects and sends NetJSON DeviceMonitoring data to OpenWISP Monitoring It depends on **netjson-monitoring** and openwisp-config

The following procedure illustrates how to compile *openwisp-monitoring*, *netjson-monitoring* and their dependencies:

```
git clone https://git.openwrt.org/openwrt/openwrt.git
cd openwrt
git checkout <openwrt-branch>

# configure feeds
echo "src-git openwisp_config https://github.com/openwisp/openwisp-config.git" > feeds.conf
echo "src-git openwisp_monitoring https://github.com/openwisp/openwrt-openwisp-monitoring.gi
cat feeds.conf.default >> feeds.conf
./scripts/feeds update -a
./scripts/feeds install -a
echo "CONFIG_PACKAGE_netjson-monitoring=y" >> .config
echo "CONFIG_PACKAGE_openwisp-monitoring=y" >> .config
make defconfig
make tools/install
make toolchain/install
make package/openwisp-monitoring/compile
```

The compiled packages will go in `bin/packages/*/openwisp`.

Alternatively, you can configure your build interactively with `make menuconfig`, in this case you will need to select the *openwisp-monitoring* and *netjson-monitoring* by going to `Administration > admin > openwisp`:

```
git clone https://git.openwrt.org/openwrt/openwrt.git
cd openwrt
git checkout <openwrt-branch>

# configure feeds
echo "src-git openwisp_config https://github.com/openwisp/openwisp-config.git" > feeds.conf
echo "src-git openwisp_monitoring https://github.com/openwisp/openwrt-openwisp-monitoring.gi
cat feeds.conf.default >> feeds.conf
./scripts/feeds update -a
./scripts/feeds install -a
make menuconfig
# go to Administration > admin > openwisp and select the packages you need interactively
make tools/install
make toolchain/install
make package/openwisp-monitoring/compile
```

Quality Assurance Checks

We use LuaFormatter and shfmt to format lua files and shell scripts respectively.

Once they are installed, you can format all files by:

```
./qa-format
```

Run quality assurance tests with:

```
#install openwisp-utils QA tools first
pip install openwisp-utils[qa]

#run QA checks before committing code
./run-qa-checks
```

Run tests

To run the unit tests, you must install the required dependencies first; to do this, you can take a look at the install-dev.sh script.

Install test requirements:

```
sudo ./install-dev.sh
```

You can run all unit tests by launching the dedicated script:

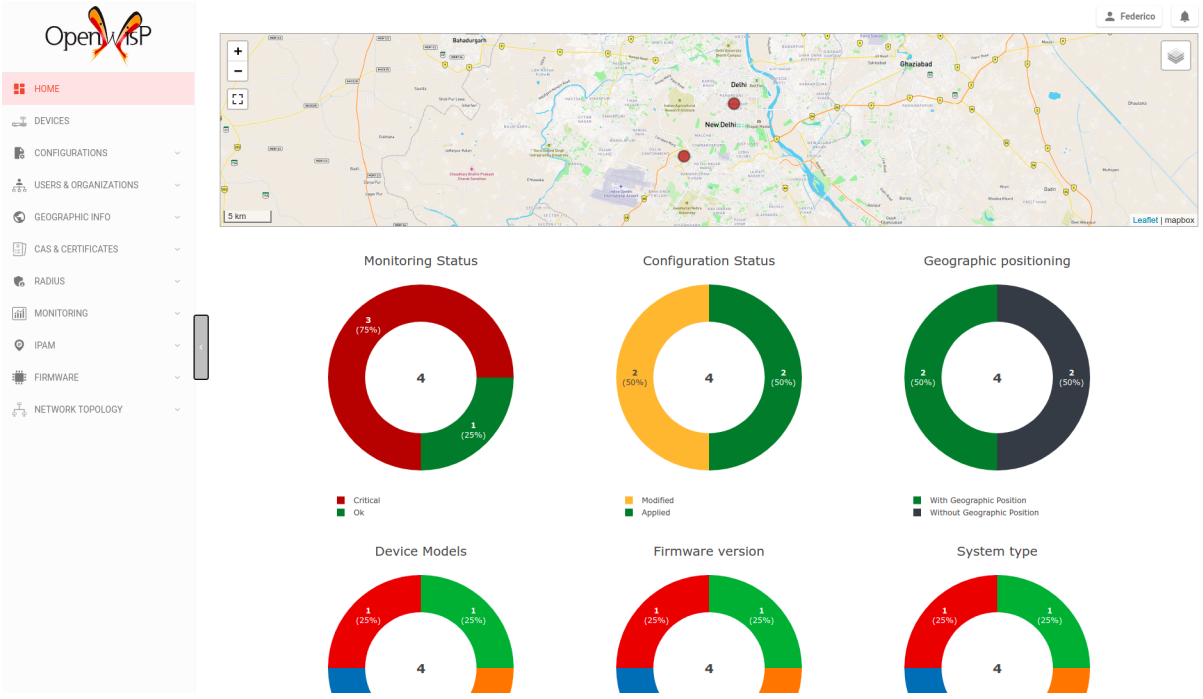
```
./runtests
```

Alternatively, you can run specific tests, e.g.:

```
cd openwrt-openwisp-monitoring/tests/
lua test_utils.lua -v
```

Tutorials

OpenWISP Demo



Screenshot of OpenWISP web UI dashboard.

Accessing the demo system

- **URL**: demo.openwisp.io
- **Username**: demo
- **Password**: tester123

The content of the demo organization is reset every day at 1:00 AM UTC, and the demo user's password is reset every minute.

To ensure the safety and integrity of our managed OpenWISP system, certain features are disabled for the demo user, including:

- Deleting existing devices
- Sending custom shell commands to devices
- Sending password change commands to devices
- Uploading new firmware builds
- Launching firmware upgrade operations
- Creating new users or modifying the details of the demo organization
- Changing the details of RADIUS groups

If you would like to test any of these features, we offer a free 30-day trial period. You can access the request form for the free trial by using the demo system, or by contacting our support.

Firmware instructions (flashing OpenWISP Firmware)

We offer an OpenWrt-based firmware that includes all the packages typically used in OpenWISP installations.

This firmware can help you quickly get started and test the core features of OpenWISP Cloud.

If you prefer to use your existing firmware, please refer to the the alternative firmware instructions.

1. Downloading the firmware

To download the OpenWISP firmware for your device, visit downloads.openwisp.io and select the appropriate target architecture and image.

At present, we are generating firmware only for `ath79`, but we plan to add support for more targets in the future.

If your device is not currently supported, please let us know through our support channels and/or follow our alternative firmware instructions below.

2. Flashing the firmware

You can Flash the firmware via web UI, or via other means available on OpenWrt.

Make sure not to keep settings: supply the *-n* command line option to *sysupgrade*. If you're using the OpenWrt web UI, there is a specific checkbox labeled "Keep settings and retain the current configuration" which appears just before confirming the upgrade and needs to be unchecked.

Alternative firmware instructions

If your device is missing from our list of available firmware images or if you have a custom firmware you do not want to lose, you can get the basic features working by downloading and installing the following packages on your device:

- openvpn (management tunnel, needed for active checks and push operations)
- openwisp-config
- openwisp-monitoring (and its dependency netjson-monitoring)

The easiest thing is to use the following commands:

```
opkg update
# install OpenVPN
opkg install openvpn-wolfssl
# install OpenWISP agents
opkg install openwisp-config
opkg install openwisp-monitoring
```

If you want to install more recent versions of the OpenWISP packages, you can download them onto your device from downloads.openwisp.io and then install them, e.g.:

```
opkg update
# install OpenVPN anyway
opkg install openvpn-wolfssl
cd /tmp

# WARNING: the URL may change overtime, so verify the right URL
# from downloads.openwisp.io

wget https://downloads.openwisp.io/openwisp-config/latest/openwisp-config_1.1.0-1_all.ipk
wget https://downloads.openwisp.io/openwisp-monitoring/latest/netjson-monitoring_0.2.1-1_all
wget https://downloads.openwisp.io/openwisp-monitoring/latest/openwisp-monitoring_0.2.1-1_al
opkg install openwisp-config_1.1.0a-1_all.ipk
opkg install netjson-monitoring_0.2.0a-1_all.ipk
opkg install openwisp-monitoring_0.2.0a-1_all.ipk
```

Note

If `wget` doesn't work (e.g.: SSL issues), you can use `curl`, or alternatively you can download the packages onto your machine and from there upload them to your device via `scp`.

Once the packages are installed, copy the following contents to `/etc/config/openwisp`:

```
config controller 'http'
    option url 'https://cloud.openwisp.io'
    # the following shared secret is for the demo organization
    option shared_secret 'nzXTd7qpXKPNdrWZDsYoMxbGpOrEVjeD'
    option management_interface 'tun0'
```

Once the configuration has been changed, you will need to restart the agent:

```
service openwisp_config restart
```

Connecting your device to OpenWISP

Once your device is flashed, connect an Ethernet cable from your LAN into one of the LAN ports.

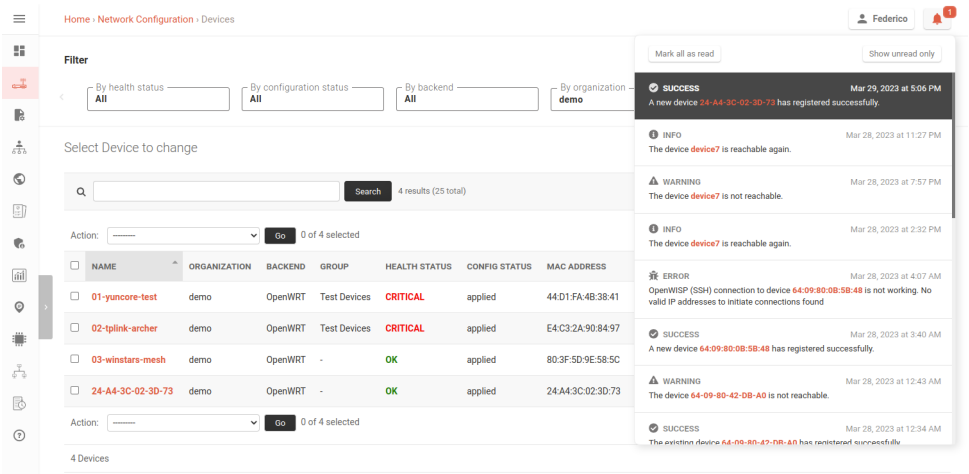
DHCP client mode

Assuming your LAN is equipped with a DHCP server (usually your main ISP router), after booting up, the device will be assigned an IP address from the LAN DHCP server. At this point, the device should be able to reach the internet and register to the OpenWISP demo system.

Static address mode

If your LAN does not have a DHCP server, you will need to configure a static IP address and gateway address for the LAN interface.

Registration



If the above steps have been completed correctly, and the device is connected to the internet, then it will automatically register and appear in the list of available devices for the demo organization. You will then be able to

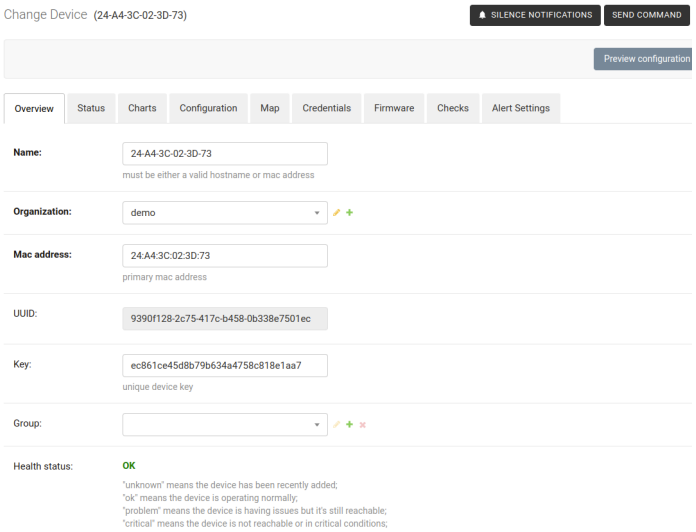
locate the device by its MAC address, as shown in the screenshot above, or by its name if you have changed it from "OpenWrt" to something else.

At this point, the device should have already downloaded and applied the configuration. After a few minutes the management tunnel will be set up and the device will start collecting monitoring information.

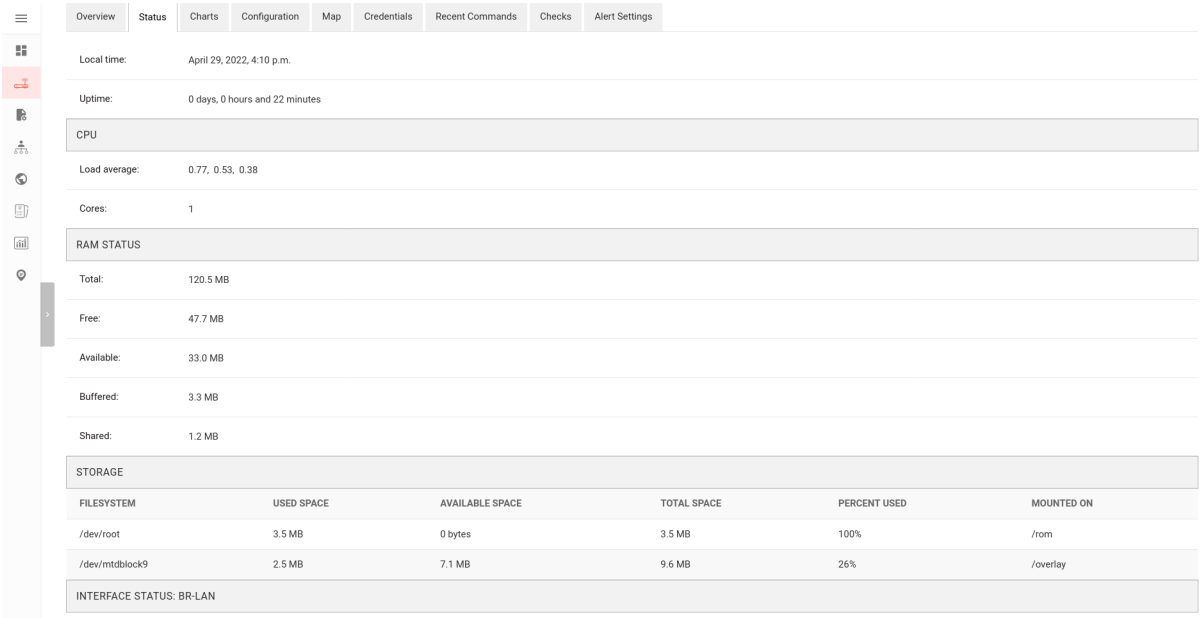
Monitoring charts and status

Once the OpenWISP Monitoring package has been installed and launched, it will start collecting metrics from your device. You will be able to see this information displayed in the UI, which will be similar to the screenshots shown below.

Health status



Device Status



INTERFACE STATUS: BR-LAN

ADDRESS / MASK	PROTOCOL
192.168.254.1 / 24	static
192.168.1.39 / 24	dhcp

INTERFACE STATUS: WLAN1

Type:	Wireless
Mode:	access point
SSID:	publicwifi-demo
Channel:	11
Frequency:	2.462 GHz
Transmission Power:	20 dBm
Signal:	-60 dBm
Noise:	-95 dBm
Associated clients:	2

ASSOCIATED CLIENT MAC ADDRESS	VENDOR	HT	VHT	WMM	WDS	WPS
b0:e1:7e:30:16:44	HUAWEI TECHNOLOGIES CO.,LTD	✓	✗	✓	✗	✗
20:f4:78:19:3b:38	Xiaomi Communications Co Ltd	✓	✗	✓	✗	✗

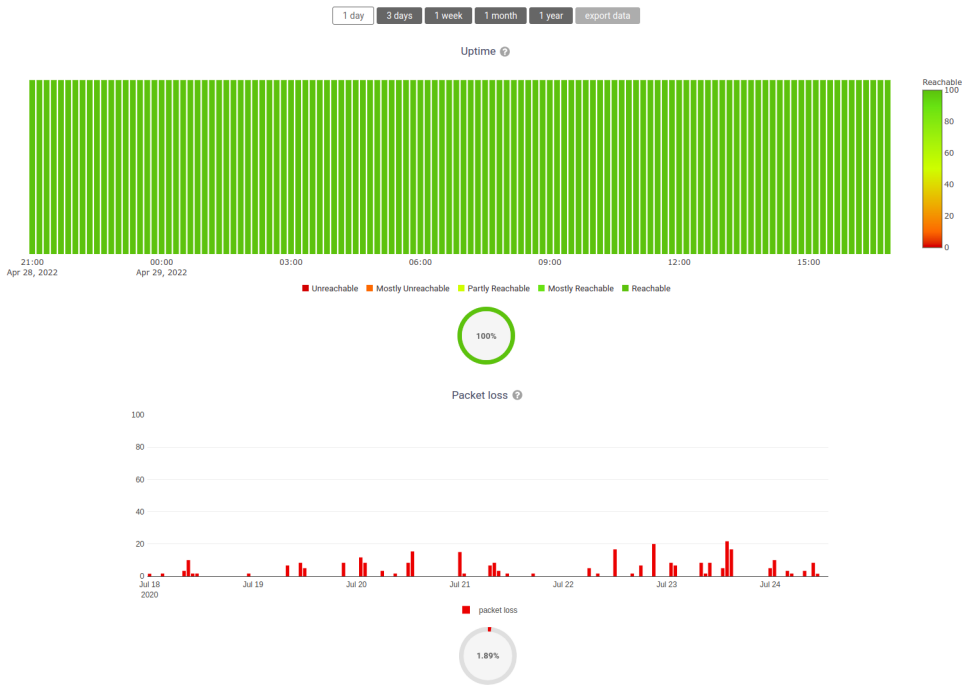
NEIGHBORS

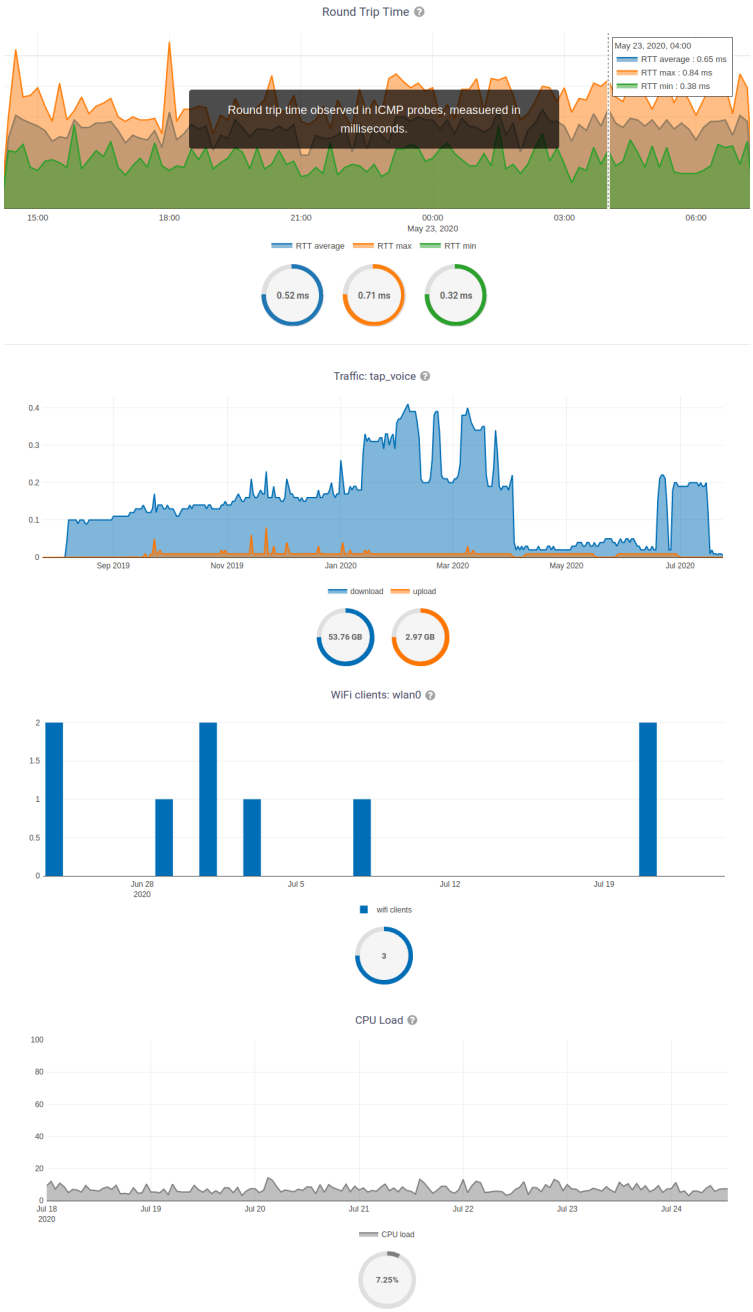
IP ADDRESS	MAC ADDRESS	VENDOR	INTERFACE	STATE
192.168.1.1	34:57:60:cb:7f:48	MitraStar Technology Corp.	br-lan	REACHABLE
192.168.1.42	b4:69:21:d2:a6:d1	Intel Corporate	br-lan	REACHABLE
fe80::b2e1:7eff:fe30:1644	b0:e1:7e:30:16:44	HUAWEI TECHNOLOGIES CO.,LTD	br-fvgwifi	STALE
fe80::22f4:78ff:fe19:3b38	20:f4:78:19:3b:38	Xiaomi Communications Co Ltd	br-fvgwifi	STALE

DHCP LEASES

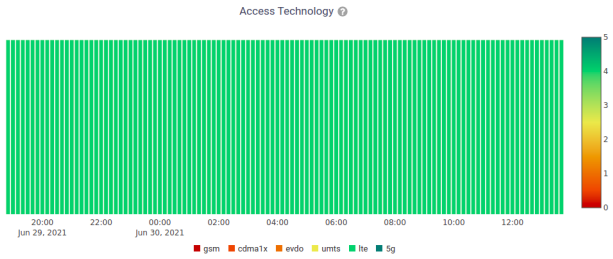
IP ADDRESS	MAC ADDRESS	VENDOR	CLIENT NAME	CLIENT ID	EXPIRY
10.0.0.240	b8:ae:ed:73:ee:51	Elitegroup Computer Systems Co.,Ltd.	DESKTOP-J7NT068	01:b8:ae:ed:73:ee:51	25 Jul 2020, 1:02 a.m.
10.0.0.207	00:04:f2:5d:0e:f3	Polycom	Polycom_0004f25d0ef3	*	24 Jul 2020, 10:33 p.m.

Charts





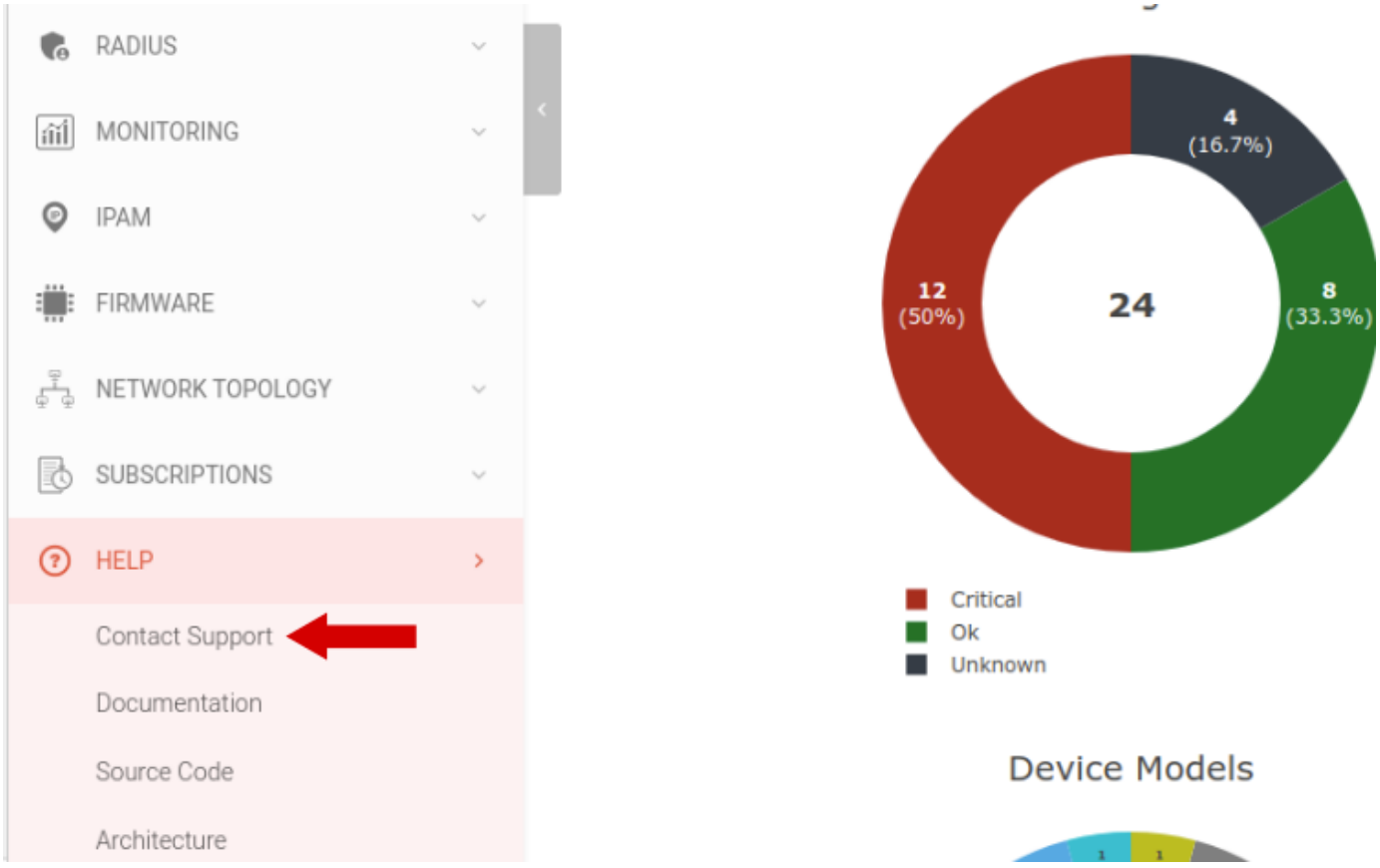
The following charts are displayed only for devices with mobile connections (e.g.: 3G, LTE).



Find out more information about the Monitoring module of OpenWISP.

Get help

If you need help or want to request a free 30-day trial of the full feature set, you can write to us via the support channels or just click on the tab *Contact support* as indicated in the screenshot below.



Seealso

- Open and/or WPA protected WiFi Access Point SSID
- WiFi Hotspot, Captive Portal (Public WiFi), Social Login
- How to Set Up a Wireless Mesh Network
- How to Set Up WPA Enterprise (EAP-TTLS-PAP) authentication

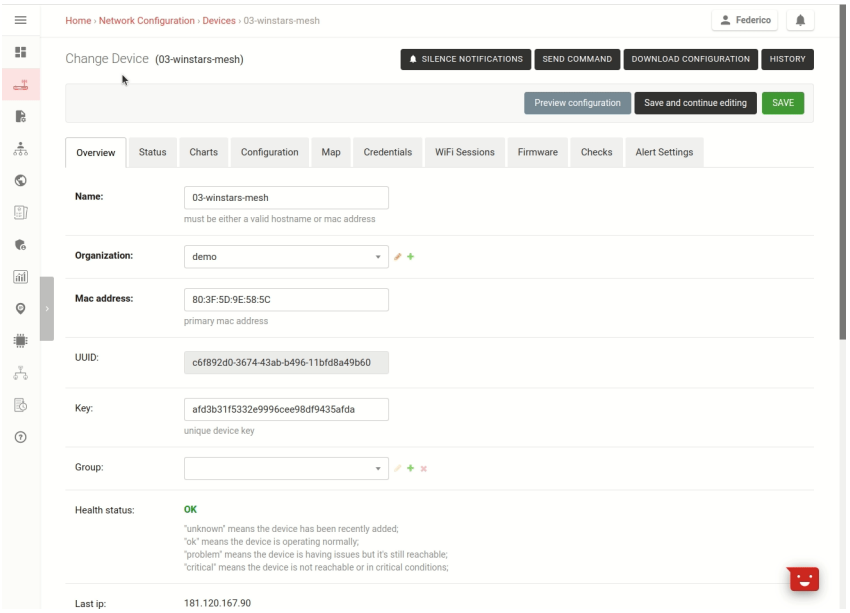
How to Set Up WiFi Access Point SSIDs

Introduction & Prerequisites

This tutorial shows different ways to set up a WiFi SSID in access point mode on your devices.

The requirement for this to work is that your device must be equipped with at least one radio and that it is named `radio0` in the OpenWrt configuration (this is the default).

Set Up an Open Access Point SSID on a Device



Open the device detail page of your device, then go to the configuration tab, then scroll down and click on "Configuration Menu", then select "Interfaces", then click on "Add new interface", select "Wireless interface", then add `wlan0` as interface name, `radio0` for the radio, then type any SSID you want, then in "Attached networks" click on "Add network" and type `lan`, this will bridge this WiFi interface to the LAN interface, now click on "Save and continue".

The screenshot below shows how the preview will look like.

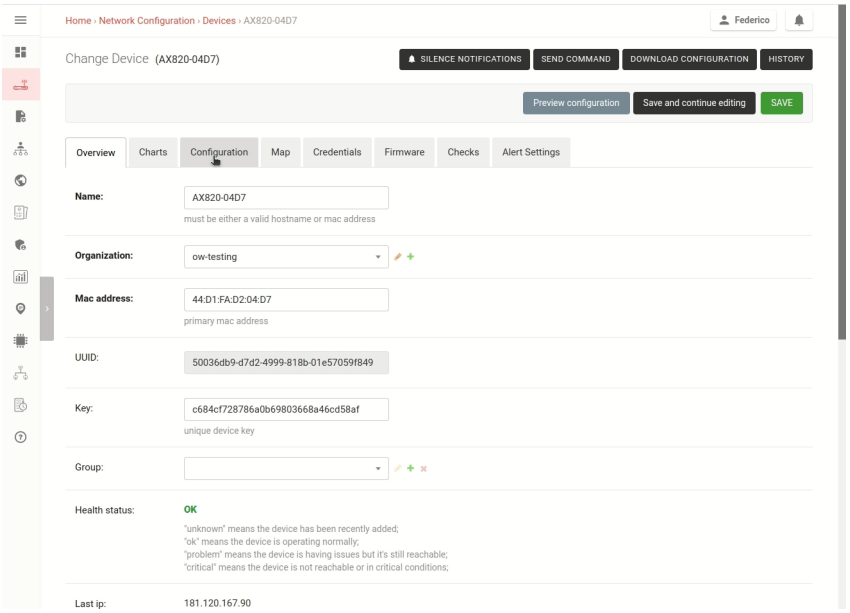
```
config wifi-iface 'wifi_wlan0'
        option device 'radio0'
        option disabled '0'
        option encryption 'none'
        option hidden '0'
        option ieee80211r '0'
        option ifname 'wlan0'
        option isolate '0'
        option macfilter 'disable'
        option mode 'ap'
        option network 'lan'
        option rsn_preauth '0'
        option ssid 'Test Open SSID'
        option wds '0'
        option wmm '1'
```

Once the configuration is applied on the device, the SSID will be broadcast.

```
root@03-wlnstars-mesh:~ # iwinfo
wlan0     ESSID: "Test Open SSID"
          Access Point: 80:3F:5D:9E:58:5E
          Mode: Master  Channel: 11 (2.462 GHz)
          Center Channel 1: 11 2: unknown
          Tx-Power: 26 dBm  Link Quality: 49/70
          Signal: -61 dBm  Noise: unknown
          Bit Rate: 117.0 MBit/s
          Encryption: none
          Type: nl80211  HW Mode(s): 802.11bgn
          Hardware: 14C3:7603 14C3:7603 [MediaTek MT7603E]
          TX power offset: none
          Frequency offset: none
          Supports VAPs: yes  PHY name: phy0
```

Once clients start to connect to this access point their information will be logged in the WiFi Sessions tab.

Set Up a WPA Encrypted Access Point SSID on a Device



Open the detail page of your device, then go to the configuration tab, then scroll down and click on "Configuration Menu", then select "Interfaces", then click on "Add new interface", select "Wireless interface", then add `wlan0` as interface name, `radio0` for the radio, then type any SSID you want, then in "Attached networks" click on "Add network" and type `lan`, this will bridge this WiFi interface to the LAN interface, now select the desired encryption, for example, WPA3/WPA2 Personal Mixed Mode, enter the password and finally click on "Save and continue".

The screenshot below shows how the preview will look like.

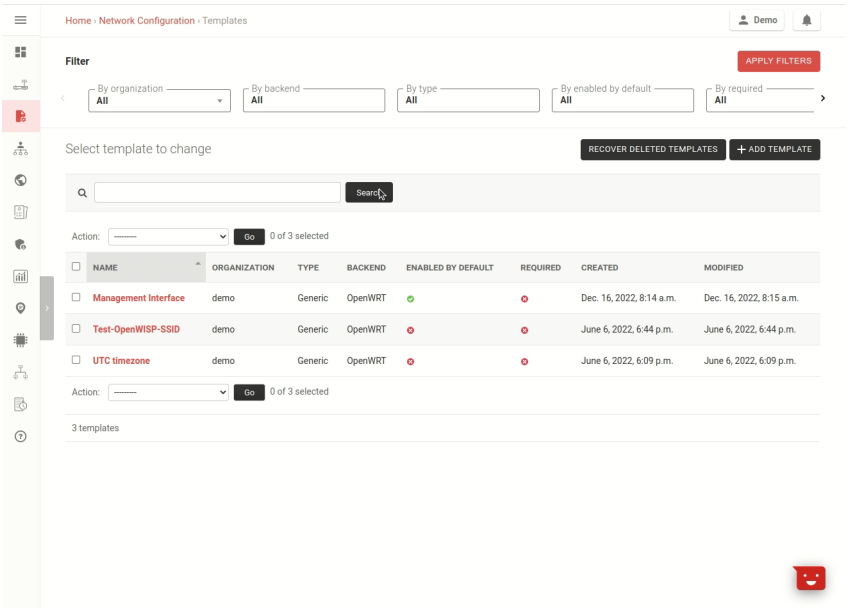
```
config wifi-iface 'wifi_wlan0'
        option device 'radio0'
        option disabled '0'
        option encryption 'sae-mixed+ccmp'
        option hidden '0'
        option ieee80211r '0'
        option ieee80211w '1'
        option ifname 'wlan0'
        option isolate '0'
        option key 'test12345678'
        option macfilter 'disable'
        option mode 'ap'
        option network 'lan'
        option rsn_preauth '0'
        option ssid 'Encrypted SSID'
        option wds '0'
        option wmm '1'
```

Once the configuration is applied on the device, the SSID will be broadcast.

```
wlan0     ESSID: "Encrypted SSID"
          Access Point: 80:3F:5D:9E:58:5E
          Mode: Master  Channel: 11 (2.462 GHz)
          Center Channel 1: 11 2: unknown
          Tx-Power: 26 dBm  Link Quality: 55/70
          Signal: -55 dBm  Noise: unknown
          Bit Rate: 144.4 MBit/s
          Encryption: WPA2 PSK (CCMP)
          Type: nl80211  HW Mode(s): 802.11bgn
          Hardware: 14C3:7603 14C3:7603 [MediaTek MT7603E]
          TX power offset: none
          Frequency offset: none
          Supports VAPs: yes  PHY name: phy0
```

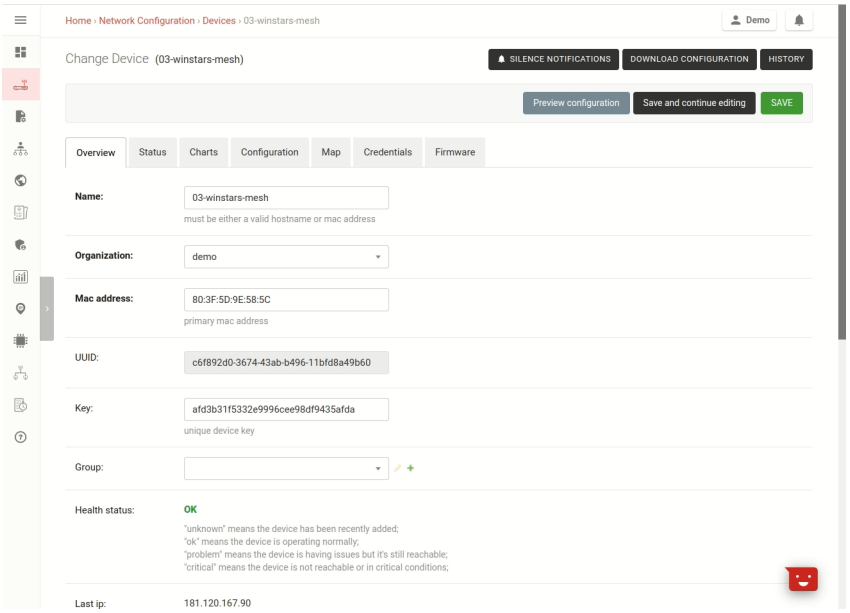
Once clients start to connect to this access point their information will be logged in the WiFi Sessions tab.

Set Up the Same SSID and Password on Multiple Devices



The procedure is very similar to the previous one, with the difference that we will be using a configuration template, then we will assign this template to the devices we want to have the SSID.

In this example we are defining two configuration variables: `wlan0_ssid` and `wlan0_password`, this allows us to change the SSID and password on a specific device if we need. Below you can find a demonstration of how to change these default template values from the device page in the "configuration variables" section.



The template can even be flagged as "Default" if we want this to be applied automatically when new devices register!

Hint

If you want to find out more about configuration templates and/or variables, consult the respective documentation sections:

- Configuration Templates
- Configuration Variables

Multiple SSIDs, multiple radios

Dual radio (2.4 GHz and 5 GHz) hardware is very common nowadays.

Multiple WiFi interfaces can be created for each available radio, as long as they have different names. The SSID can be the same, although this only makes sense for having the same SSID broadcast on different WiFi bands (e.g.: 2.4 GHz and 5 GHz).

In order to do this, just repeat the procedure shown in the previous sections, with the difference that instead of adding only one interface, you will have to add multiple wireless interfaces and define a different `name` and, if you want to deploy the SSID on different bands, a different value for the `radio` field, e.g. `radio0` and `radio1`.

Roaming (802.11r: Fast BSS Transition)

Fast transition enables WiFi clients to seamlessly roam between access points without interrupting media flows, such as video or phone calls, streaming, etc., caused by delays in re-authentication.

Enabling 802.11r on OpenWrt via OpenWISP can be easily done with the following steps:

1. Prepare a WiFi AP template as explained in the previous sections, ensuring that the SSID used on the access points remains consistent.
2. Check the "roaming" checkbox.
3. Check the "FT PSK generate local" checkbox.
4. Increase the default "reassociation deadline" to at least 2000.
5. Save the changes.

To verify whether WiFi clients are roaming between APs, launch the shell command `logread -f` on each AP. Then, move the WiFi client from one AP to another, making sure they are sufficiently distant.

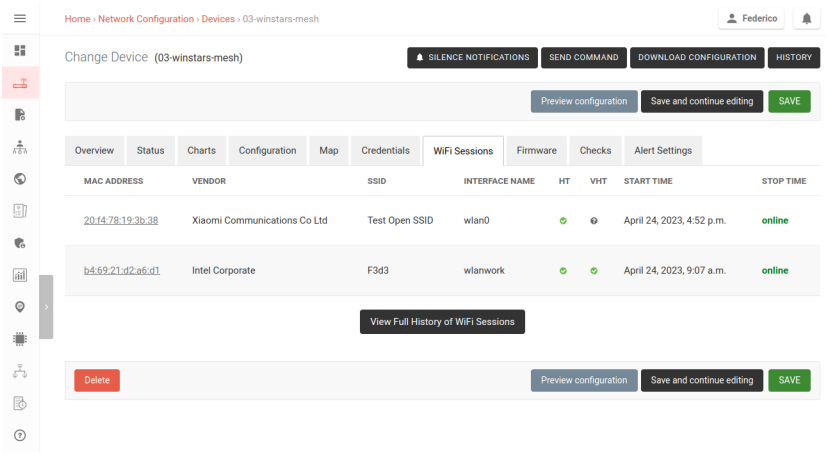
When the WiFi client successfully transitions from one AP to another, you should see log lines like:

```
WPA: FT authentication already completed - do not start 4-way handshake
```

You may wish to test the configuration and adjust the following options:

- Reassociation deadline: Increase it to avoid frequent timeouts on busy networks.
- *FT-over-DS*.

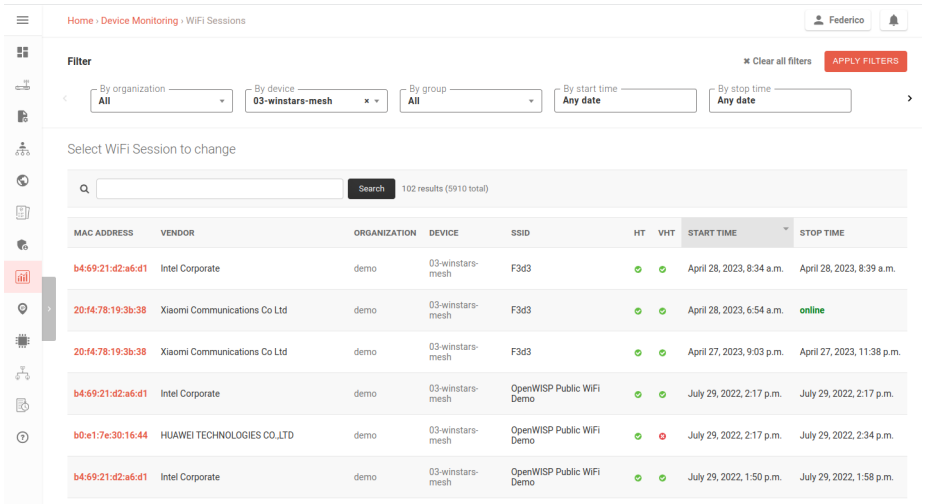
Monitoring WiFi Clients



Since OpenWISP 23, in the device page, whenever any WiFi client data is collected by the Monitoring module of OpenWISP, a "WiFi Sessions" tab will appear as in the screenshot above, showing WiFi clients connected right now.

The data is sent by default by devices every 5 minutes.

Clicking on "Full History of WiFi Sessions" will redirect to the full list of all clients which have connected to this access point, as shown below.



In this page it will be possible to use more filters and even perform a text search.

Seealso

- WiFi Hotspot, Captive Portal (Public WiFi), Social Login
- How to Set Up a Wireless Mesh Network
- How to Set Up WPA Enterprise (EAP-TTLS-PAP) authentication

WiFi Hotspot & Captive Portal

Introduction & Prerequisites

OpenWISP is widely used as an **open source software** solution for **WiFi Hotspot Management** in **Public WiFi** settings.

In this tutorial, we'll explain the technical details of the most common **WiFi Hotspot** deployments and how to test the most important functionalities of this use case on the OpenWISP Demo System.

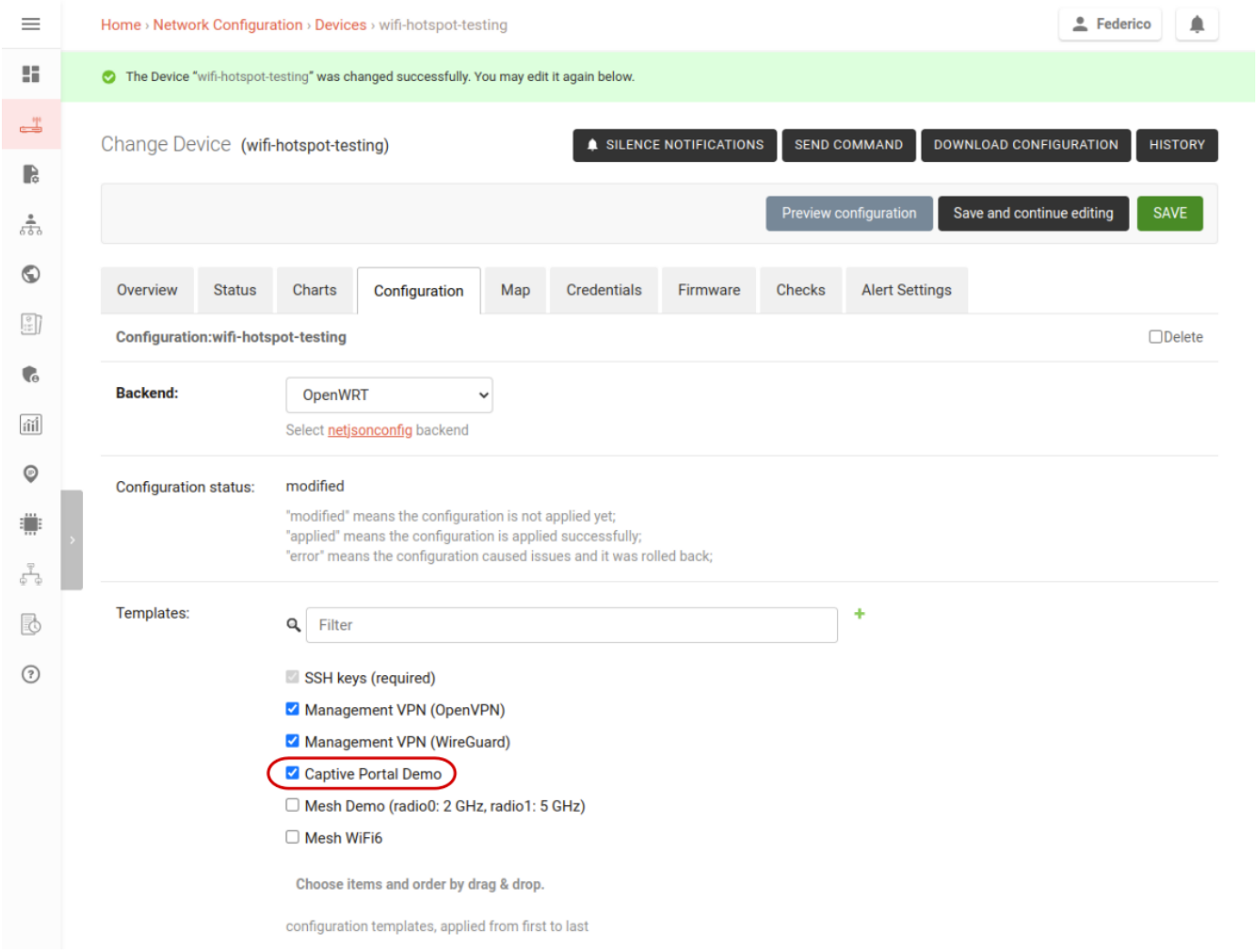
The **OpenWrt** firmware image for the OpenWISP Demo System includes a *captive portal* package called Coova-Chilli. This supports the RADIUS protocol, a standard security protocol used in Accounting, Authorization and Authentication (AAA), a way of authenticating, authorizing, and rate-limiting network usage supported by networking hardware and software.

Warning

Unfortunately, at the moment, installing Coova-Chilli from the OpenWrt packages will not work because the default configuration of the Coova-Chilli OpenWrt package does not enable the `chilli-redir` feature, nor has SSL support enabled, which will not allow the captive portal to redirect the user to the captive page and will not support HTTPs requests.

The OpenVPN package is also required and included in the firmware instructions for the OpenWISP Demo System, as it's needed to facilitate secure communication between the Coova-Chilli captive portal and FreeRADIUS over the Management VPN tunnel. This setup prevents the routing of unencrypted RADIUS packets through the public internet, ensuring security, privacy, and mitigating potential legal risks associated with exposing users' personal information to malicious actors.

Enable Captive Portal Template



Hint

If you don't know what a template is, please see Configuration Templates.

If you flashed the *OpenWrt* based firmware and registered your device as explained in the OpenWISP Demo Page, proceed to assign the captive portal template to your device:

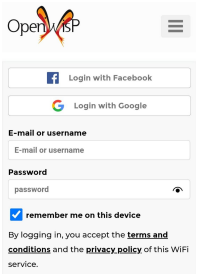
- Go to the device list.
- Open the device details.
- Click on the configuration tab.
- Select the "Captive Portal Demo" template.
- Hit "Save".

Then, make sure the *OpenVPN management tunnel* is working otherwise the captive portal software will not be able to talk to the demo FreeRADIUS server instance.

Shortly after the configuration is applied successfully, the Public WiFi SSID will be broadcast by the *access point*.

Accessing the Public WiFI Hotspot

Connect your laptop or phone to the SSID "OpenWISP Public WiFi Demo". If everything is working correctly, your operating system should open a browser window showing the captive page as shown in the screenshot above.

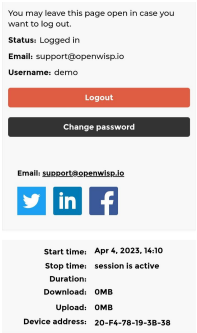


At this point, sign in using the same credentials you used to access the demo system (`demo`/`tester123`).

Note

Trying to surf the internet without authenticating will not work.

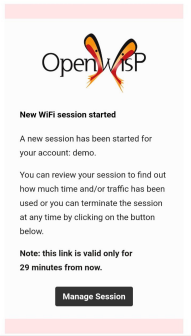
Once you've logged in, you'll see a status page as shown in the following screenshot:



This page communicates that the user can now use the internet provided by the hotspot, it also provides the following features:

- It shows a list of the user's sessions, including the start time, stop time, duration, traffic consumed (download and upload), and the MAC address of the device that accessed the WiFi service.
- It allows the account password and phone number (if SMS verification is enabled, which is not the case for the demo system) to be changed.
- It allows users to close their session and log out (more on why this is useful below).

On some mobile operating systems, the mini-browser automatically closes when switching windows, for example, when opening the real browser to surf the internet. This can be problematic if the user needs to use one of the features of the status page listed above.



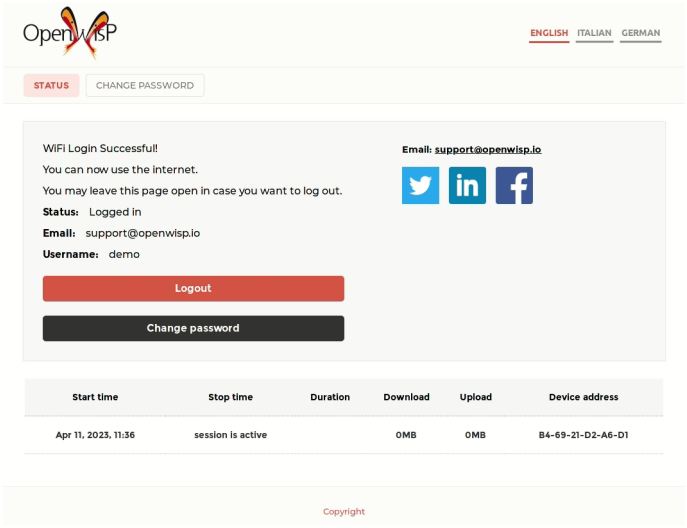
To resolve this, OpenWISP will send an email to the user with a magic link. This will allow the user access to the status page of WiFi Login Pages without entering their credentials again, as shown in the image above.

Note

For more technical information and implementation details about the magic links feature, refer to the related section: openwisp_users.api.authentication.SesameAuthentication.

If you are using the demo account, the email will be sent to the email address linked to the demo account. Therefore, if you want to try this feature, you will have to sign up for your own account or use the social login feature. Please see more information on this below.

Logging Out

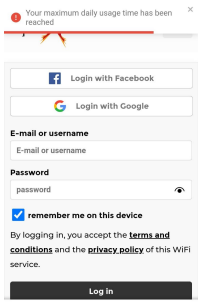


Most WiFi hotspot services have limitations in place that do not allow users to browse indefinitely.

Some services only allow surfing for a limited amount of time per day, while others limit the amount of data you can consume. Some services use a combination of both methods and when either the daily time or data limit is reached, the session is closed.

Therefore, users who plan to use the service again later on the same day, should log out to avoid consuming their daily time and/or data.

Session Limits



The default session limits in the **OpenWISP RADIUS** configuration are 300 MB of daily traffic or three hours of daily surfing.

Note

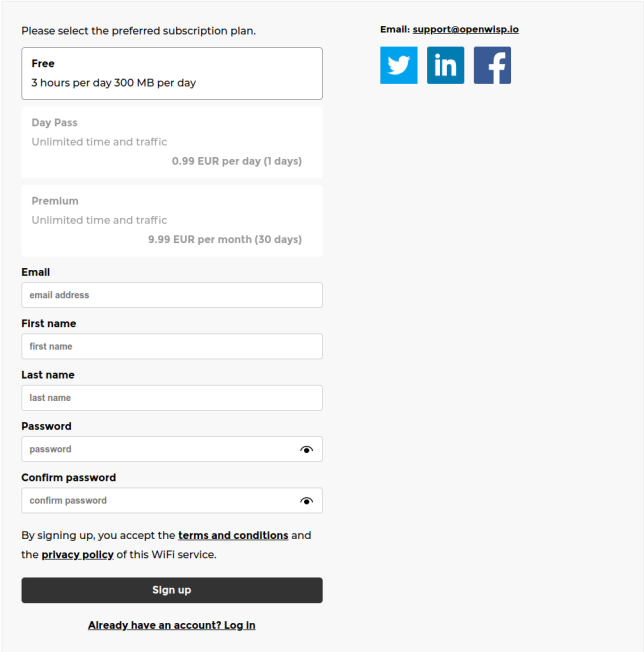
To find out more technical information about this topic please read: OpenWISP RADIUS - Enforcing session limits.

Automatic Captive Portal Login

The WiFi Login Pages application. allows users who have logged in previously, and who use a browser which supports cookies (not all mini-browsers that are used for captive portal logins do), to automatically log in without entering their credentials again.

The video below demonstrates this feature:

Sign Up

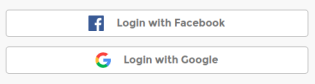


To sign up for the *WiFi hotspot demo*, select the free plan and enter dummy data (this data is deleted every day). However, it is recommended that you enter a real email address so that you can test features that require receiving emails, such as email confirmation, password reset, and the "WiFi session started" notification.

Note

The sign up process uses the OpenWISP RADIUS REST API under the hood.

Social Login



Another way to sign up for a free WiFi hotspot account is to use social login. Simply click on one of the social login buttons to initiate the process.

Please note that your personal data is stored for less than 24 hours, as the demo system is reset every day.

Note

For more technical information about social login, please read OpenWISP RADIUS - Social Login

Paid WiFi Hotspot Subscription Plans

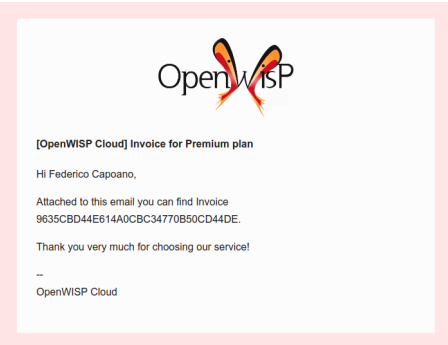
Testing the **WiFi hotspot paid subscription plans** is easy, the demo system is configured to use the Paypal Sandbox, a test version of Paypal with unlimited fake money, which allows users to test the feature at any time without incurring any costs.

Follow these steps to try the *paid WiFi subscription* feature:

- Sign up for one of the non-free plans.
- Enter your real email address and dummy personal information.
- Click "Proceed with the payment."
- Enter the following Paypal credentials: `support@openwisp.io` / `tester123` and click on "start session".
- Choose to pay with Paypal balance and click "Continue to Review Order."

After following the steps above you will be logged in to the WiFi service and redirected to the status page, from then on you can surf the web.

You should also receive a test invoice via email as in the screenshots below.



OpenWISP

Invoice ID #9635CBD44E614A0CBC34770B50CD44DE
Issued: 2023-04-14
Order date: 2023-04-14

Seller	**Buyer**
OpenWISP Test Test street, 123 123-3444 SolarCity EE - Estonia TAX ID 1222233334444555	Federico Capoano Test 0000 Test IT - Italy

Description	Unit price	Qty.	Subtotal	TAX	TAX Amount	**Total**
Premium plan	9,99 EUR	1	9,99 EUR	22,00 %	2,20 EUR	**12,19 EUR**

Payment: electronic payment

Payment status: paid on 2023-04-14

Seealso

- Open and/or WPA protected WiFi Access Point SSID
- How to Set Up a Wireless Mesh Network
- How to Set Up WPA Enterprise (EAP-TTLS-PAP) authentication

How to Set Up WPA Enterprise (EAP-TTLS-PAP) Authentication

Introduction & Prerequisites

In this tutorial, we will guide you on how to set up WPA Enterprise (EAP-TTLS-PAP) authentication for WiFi networks using OpenWISP. The RADIUS capabilities of OpenWISP provide integration with FreeRADIUS to allow users to authenticate with their Django user accounts. Users can either be created manually via the admin interface, generated with voucher-like codes, imported from CSV or can register autonomously via the REST API of OpenWISP RADIUS.

Enable OpenWISP RADIUS

Note

If you are following this tutorial on our Demo System, you can skip this step.

To use WPA2 Enterprise, the RADIUS module must be enabled first.

See Enabling the RADIUS Module in the Ansible OpenWISP role. In Docker OpenWISP, the RADIUS module is enabled by default.

VPN Tunnel

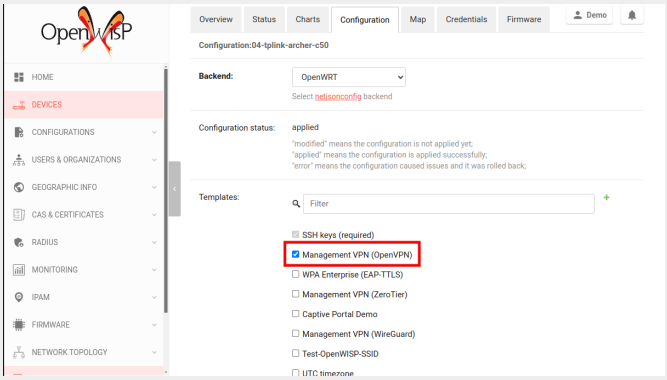
We recommend setting up a VPN tunnel to secure the communication between the RADIUS server and the NAS devices.

Routing unencrypted RADIUS traffic through the internet is not recommended for security. When security breaches in the RADIUS protocol are discovered (like the "Blast-RADIUS attack" in July 2024), your entire network would be at risk.

If you are using OpenWrt, you can use OpenWISP to automate the provisioning of OpenVPN tunnels on your OpenWrt devices. For more information, please refer to Automating OpenVPN Tunnels.

Note

If you are following this tutorial on our Demo System, the `Management VPN (OpenVPN)` template will be applied to your device by default. If not, you need to enable that template on your device. Otherwise, your device won't connect to the FreeRADIUS server.



Using radsec (RADIUS over TLS) is a good option, but it's not covered in this tutorial.

Firmware Requirements

To use WPA Enterprise authentication, your firmware needs to be equipped with a version of the `wpad` package that supports WPA Enterprise encryption.

Please refer to the OpenWrt WPA encryption documentation for more information.

In tutorial we use OpenVPN to tunnel RADIUS packets from NAS devices to FreeRADIUS, for this reason you must ensure that your OpenWrt device has the `openvpn` package installed.

Note

The **OpenWrt** firmware image provided for the OpenWISP Demo System includes `openvpn` and the full `wpad` package by default.

One Radio Available

At least one radio named `radio0` needs to be available and enabled for the successful execution of this tutorial.

For simplicity, we will focus on a single radio, but it's important to note that the WPA Enterprise functionality can be extended to multiple radios if necessary.

Alternatively, you have the option of using WPA Enterprise encryption on one radio while the other radios use different encryption methods. However, these additional scenarios are not explained in this tutorial and are left as an exercise for the reader.

Configuring FreeRADIUS for WPA Enterprise

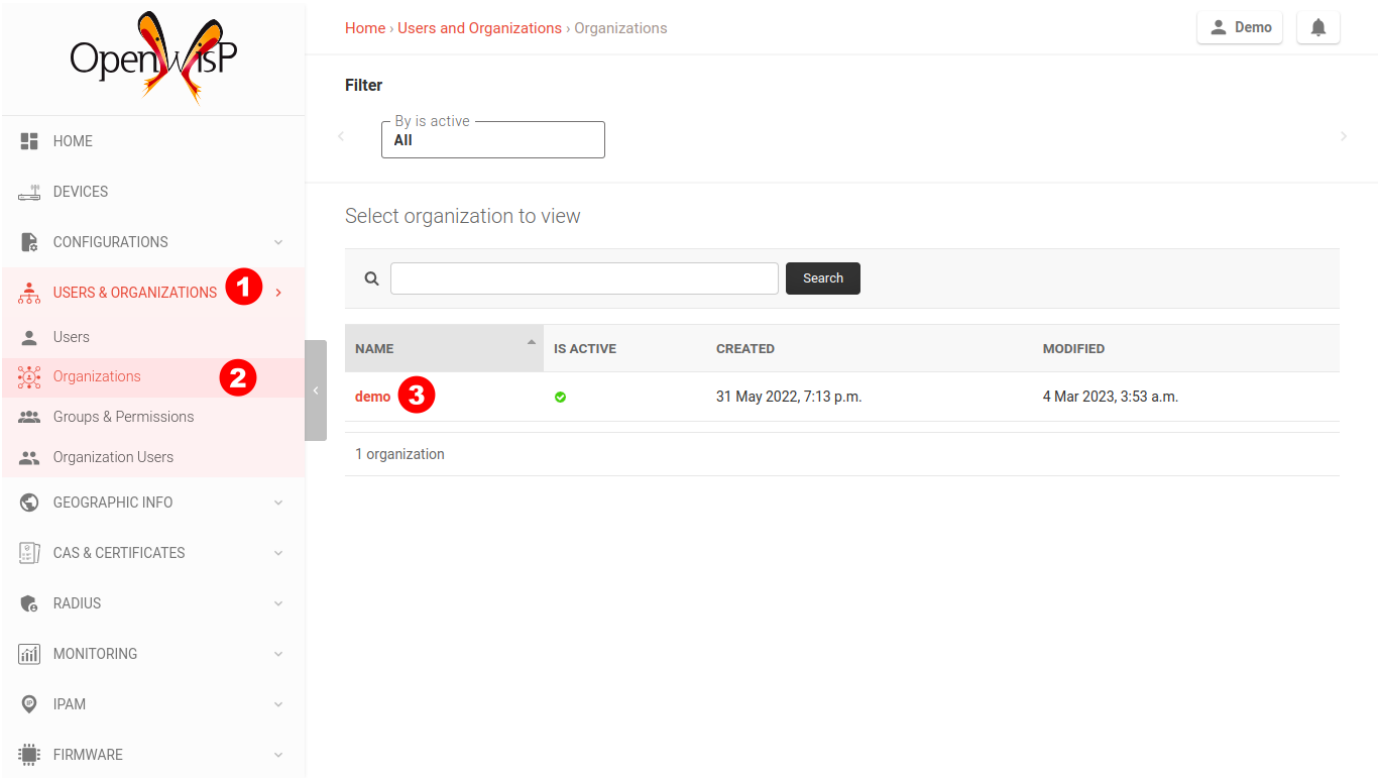
Note

If you are following this tutorial on our Demo System, you can skip this step.

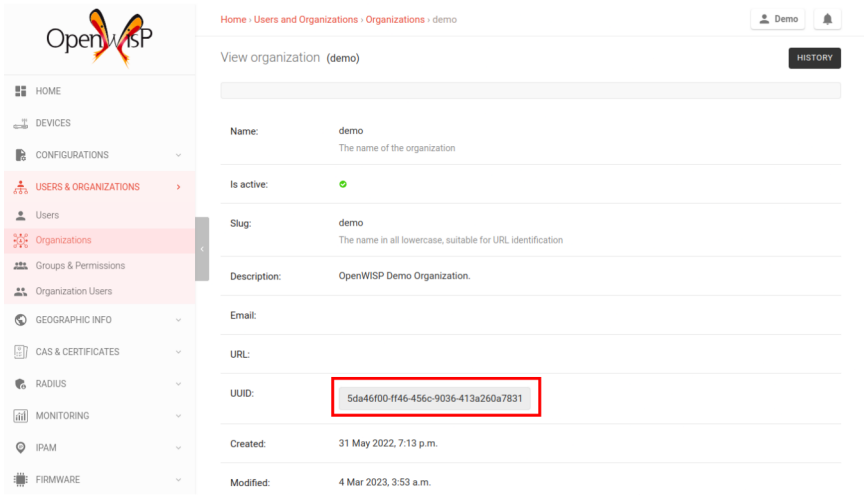
Before making changes to the FreeRADIUS configuration, we need to gather the following information:

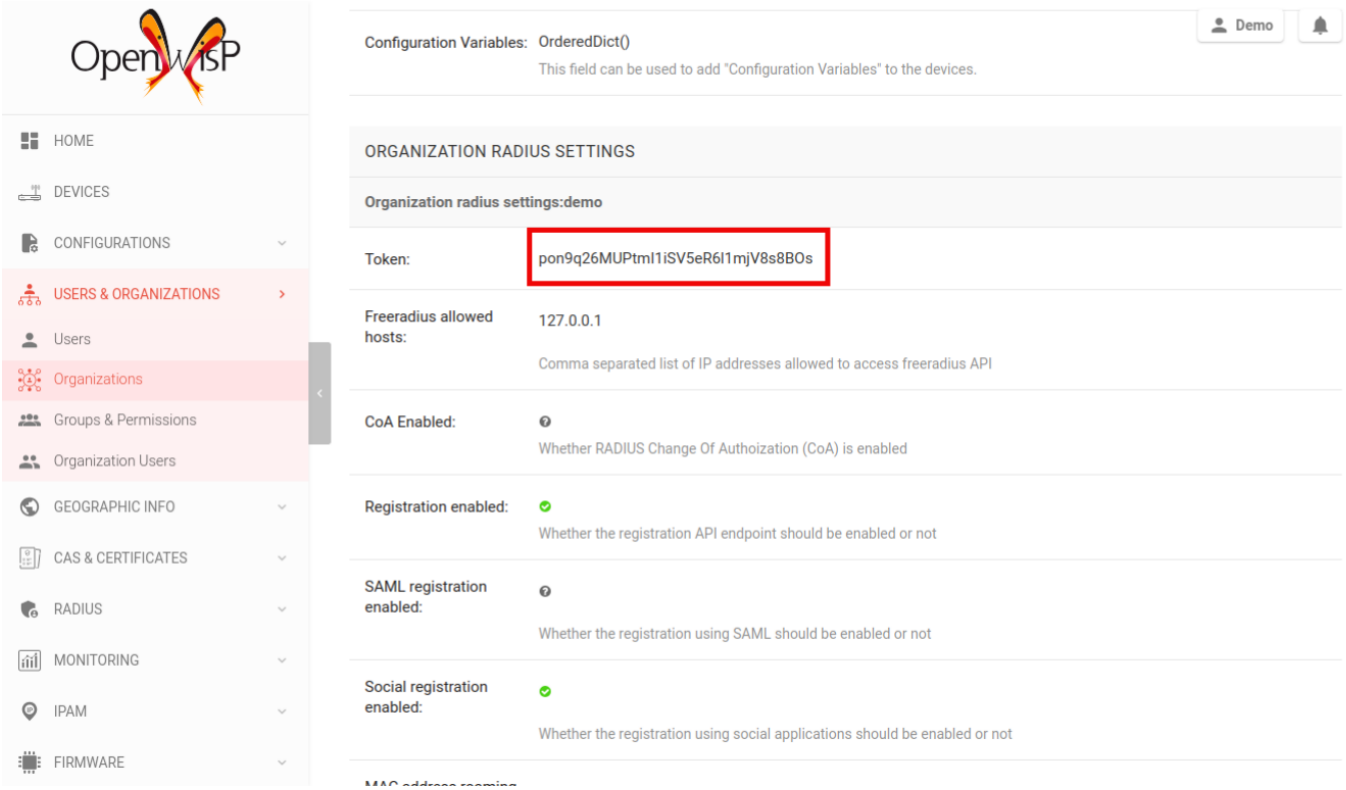
- Organization's UUID
- Organization's RADIUS token

From the OpenWISP navigation menu, go to `Users & Organizations` and then `Organizations`. From here, click on the desired organization.



From the organization's page, find the organization's UUID and RADIUS token.





This is a good point to decide whether to use self-signed certificates or public certificates issued by a trusted Certificate Authority (CA). Both options have their pros and cons, and the choice largely depends on your specific requirements and constraints.

Self-Signed Certificates

Pros:

- Generated locally without involving a third-party CA.
- Eliminates the need for external entities, reducing the risk of compromised trust.

Cons:

- Requires installation of the self-signed CA on all client devices.

Public Certificates

Pros:

- Issued by trusted CAs, thus works out of the box with most devices.

Cons:

- Higher risk of compromise.
- More cumbersome to set up.

We recommend using the Ansible OpenWISP2 role, which simplifies configuring FreeRADIUS to use WPA Enterprise. Please refer to the "Configuring FreeRADIUS for WPA Enterprise (EAP-TTLS-PAP)" section in the ansible-openwisp2 documentation for details.

If you prefer to configure the FreeRADIUS site manually, refer to the "Freeradius Setup for WPA Enterprise (EAP-TTLS-PAP) authentication" section of the OpenWISP RADIUS documentation.

Creating the Template

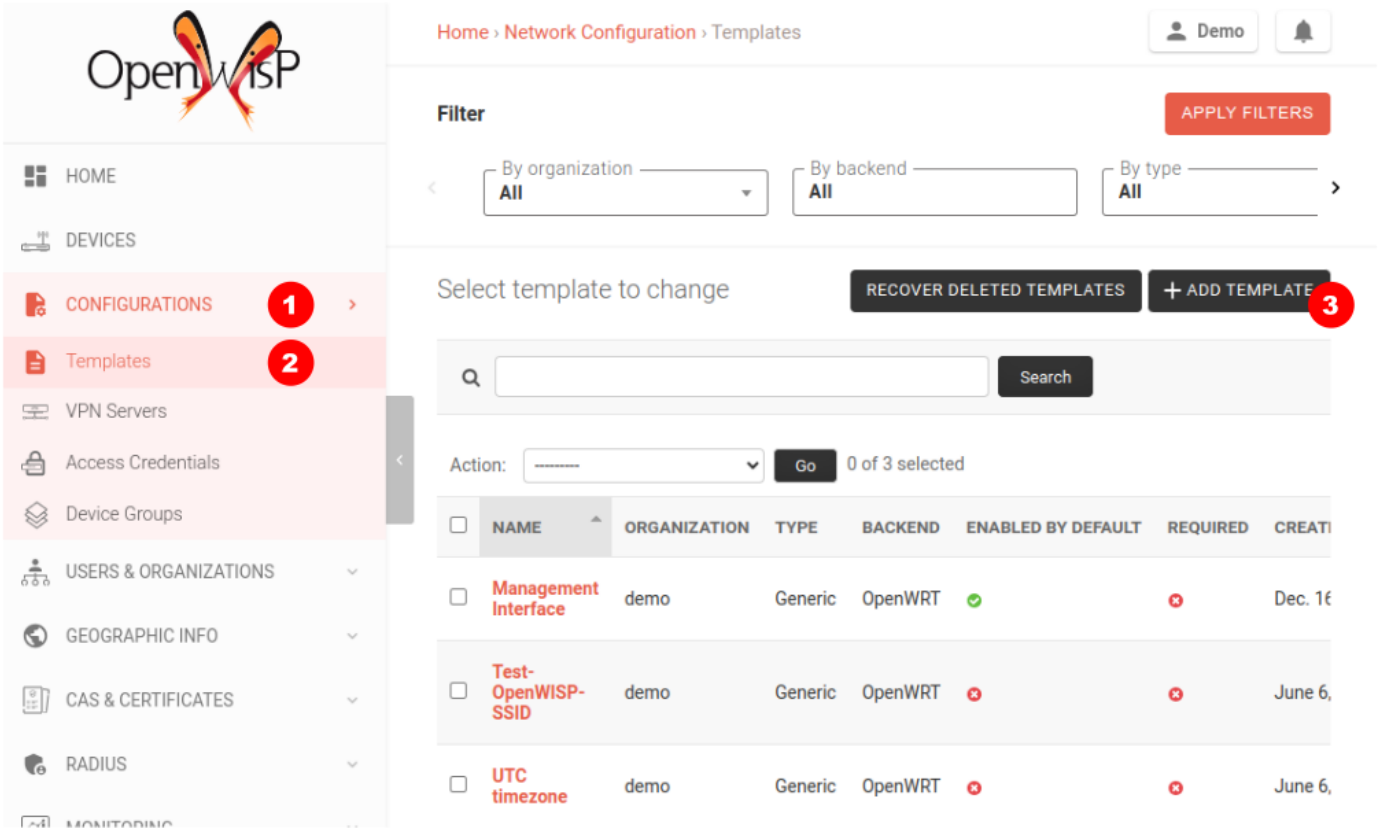
Note

This template is also available in our Demo System as WPA Enterprise (EAP-TTLS), **feel free to try it out!**

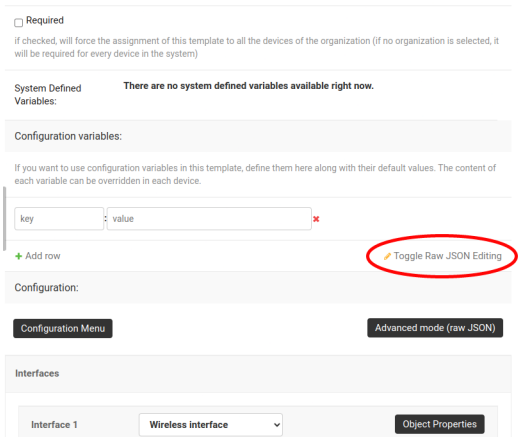
Hint

If you don't know what a template is, please see Configuration Templates.

From the OpenWISP navigation menu, go to `Configurations` and then `Templates`, from here click on `Add template`.

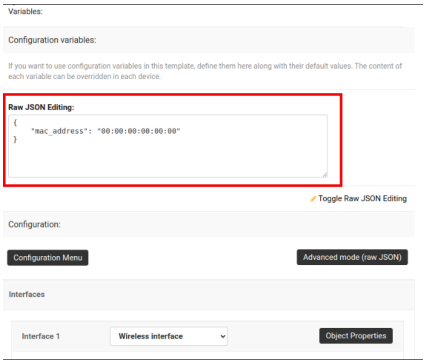


Fill in the name, organization, leave type set to "Generic", and backend set to "OpenWrt". Scroll down to the `Configuration variables` section, then click on "Toggle Raw JSON Editing".



Paste the following JSON in the `Raw JSON Editing` field.

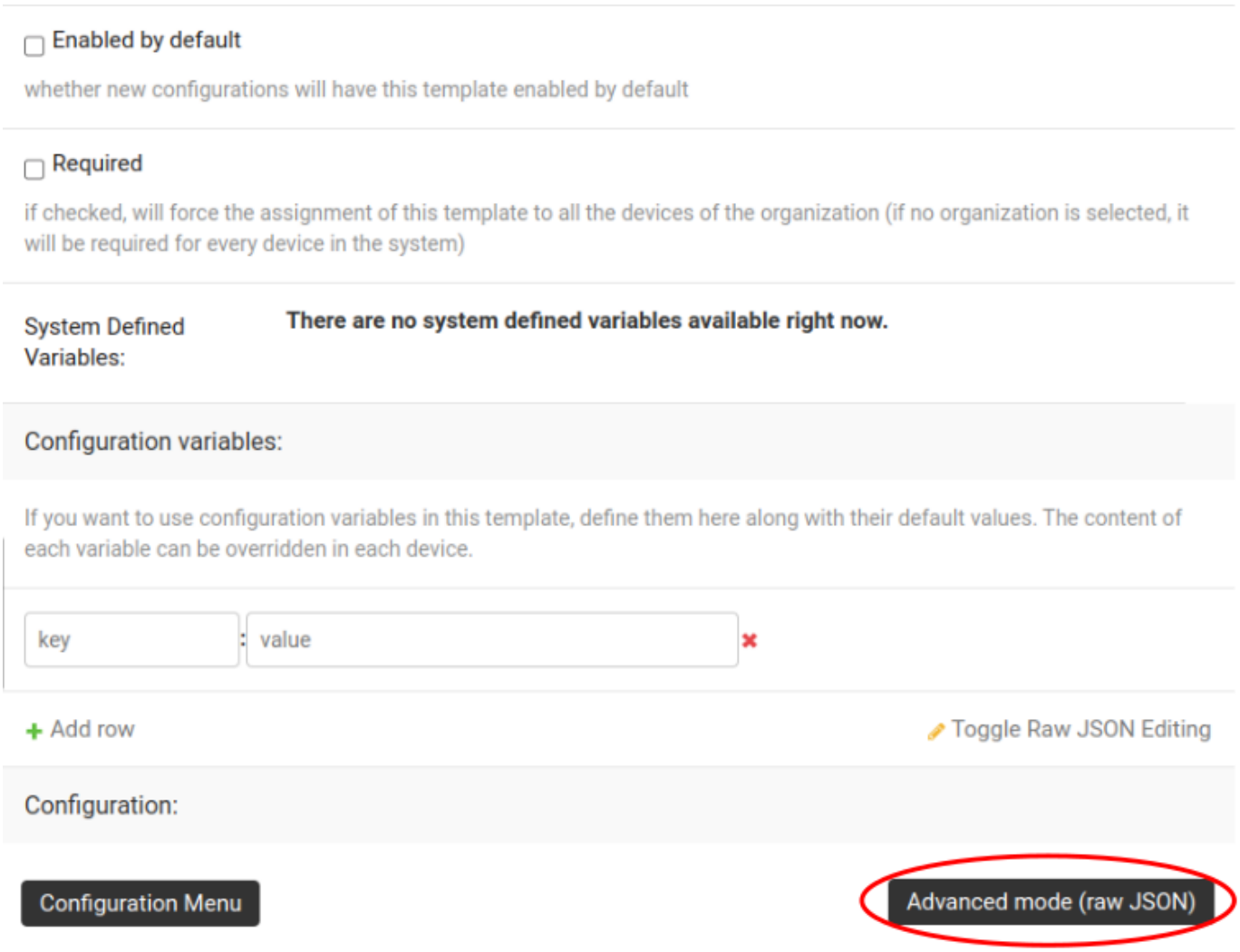
```
{
    "mac_address": "00:00:00:00:00:00"
}
```



Hint

For more information about variables, please refer to Configuration Variables.

Scroll down to the `Configuration` section, then click on "Advanced mode (raw JSON)".



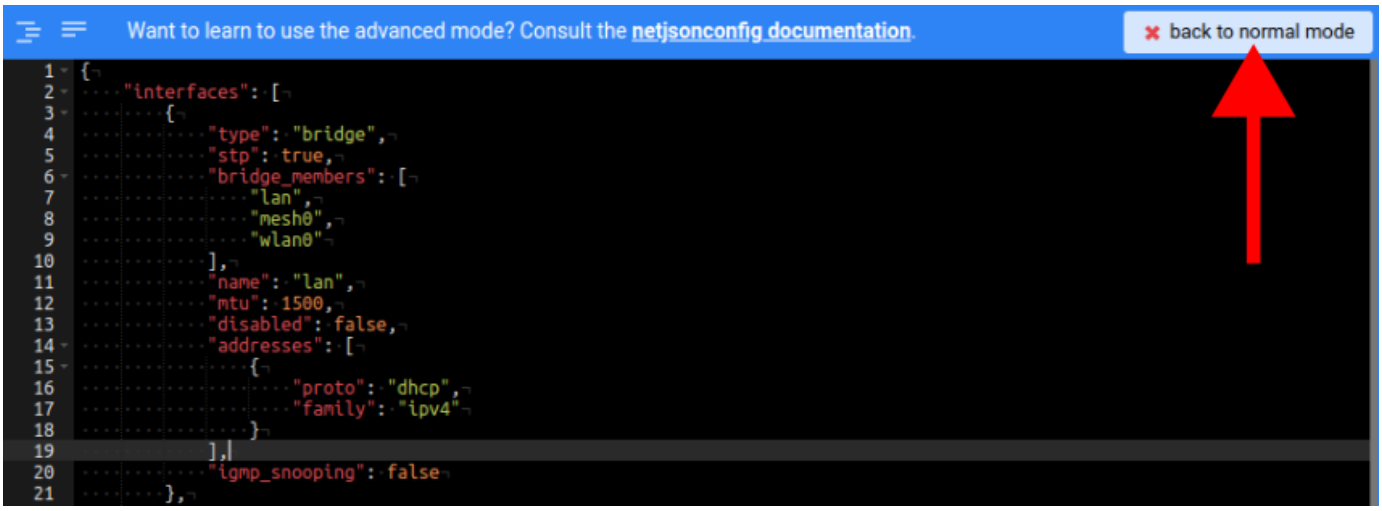
Before copying the following NetJSON to the advanced mode editor, you will need to update these fields to reflect your configuration:

- `key` - RADIUS secret should be the same as set in NAS
- `server` - RADIUS server authentication IP
- `port` - RADIUS server authentication port
- `acct_server` - RADIUS accounting server IP
- `acct_server_port` - RADIUS accounting server port

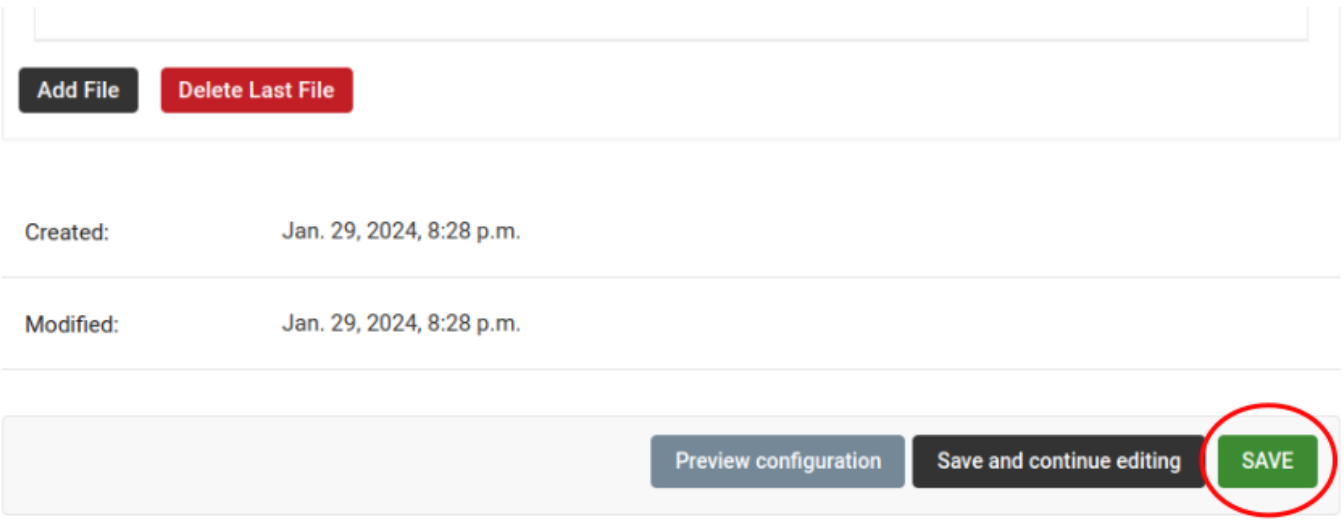
```
{
    "interfaces": [{
        "name": "wlan_eap",
        "type": "wireless",
        "mac": "{{mac_address}}",
        "mtu": 1500,
        "disabled": false,
        "network": "",
        "autostart": true,
        "addresses": [],
        "wireless": {
            "network": [
                "lan"
            ],
            "mode": "access_point",
```

```
                "radio": "radio0",
                "ssid": "WPA Enterprise 2 (EAP-PAP-TTLS)",
                "ack_distance": 0,
                "rts_threshold": 0,
                "frag_threshold": 0,
                "hidden": false,
                "wds": false,
                "wmm": true,
                "isolate": false,
                "ieee80211r": false,
                "reassociation_deadline": 1000,
                "ft_psk_generate_local": false,
                "ft_over_ds": true,
                "rsn_preauth": false,
                "macfilter": "disable",
                "maclist": [],
                "encryption": {
                    "protocol": "wpa2_enterprise",
                    "key": "testing123",
                    "disabled": false,
                    "cipher": "auto",
                    "ieee80211w": "0",
                    "server": "10.8.0.1",
                    "port": 1822,
                    "acct_server": "10.8.0.1",
                    "acct_server_port": 1823
                }
            }
        }],
        "files": [{
            "path": "/etc/openwisp/pre-reload-hook",
            "mode": "0700",
            "contents": "#!/bin/sh\n\n# Ensure radio0 is enabled \nuci set wireless.radio0.disab
        }]
    }
```

Then click on "back to normal mode" to close the advanced mode editor.



Now you can save the new template.



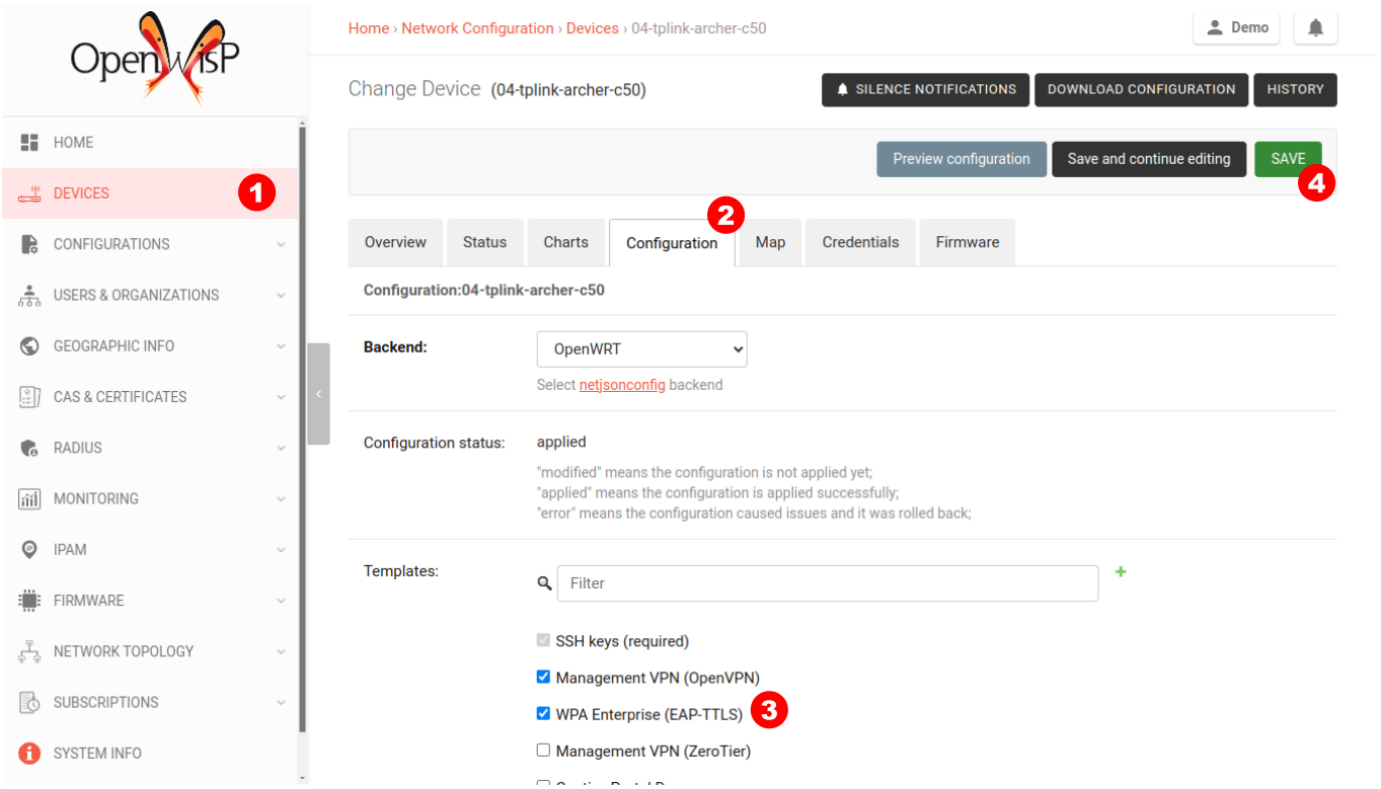
At this point, you're ready to assign the template to your devices. However, before doing so, you may want to read on to understand the different components of this template:

- The `wlan_eap` creates the wireless interface that supports WPA2 Enterprise encryption bound to `radio0`. This interface is attached to the `lan` interface, which is configured to provide internet access in the default OpenWrt configuration.
- A `pre-reload-hook` script is executed before OpenWrt reloads its services to ensure that `radio0` is enabled.
- The `mac_address` configuration variable is added to the template as a placeholder. When the template is applied to a device, the device's actual MAC address will automatically override the placeholder, ensuring that the wireless interface is created with the correct MAC address. This is necessary for tracing which device is being used in RADIUS accounting stats.

Enable the WPA Enterprise Template on the Devices

Now it is time to apply this template to the devices where you want to enable WPA Enterprise authentication on WiFi.

Click on `Devices` in the navigation menu, click on the device you want to assign the WPA Enterprise template to, then go to the `Configuration` tab, select the template just created, and then click on save.



Connecting to the WiFi with WPA2 Enterprise

For brevity, this section only includes an example of connecting a smartphone running Android 11 to the WiFi network. Similar steps can typically be followed on other devices. If unsure, consult your device's manual for guidance.

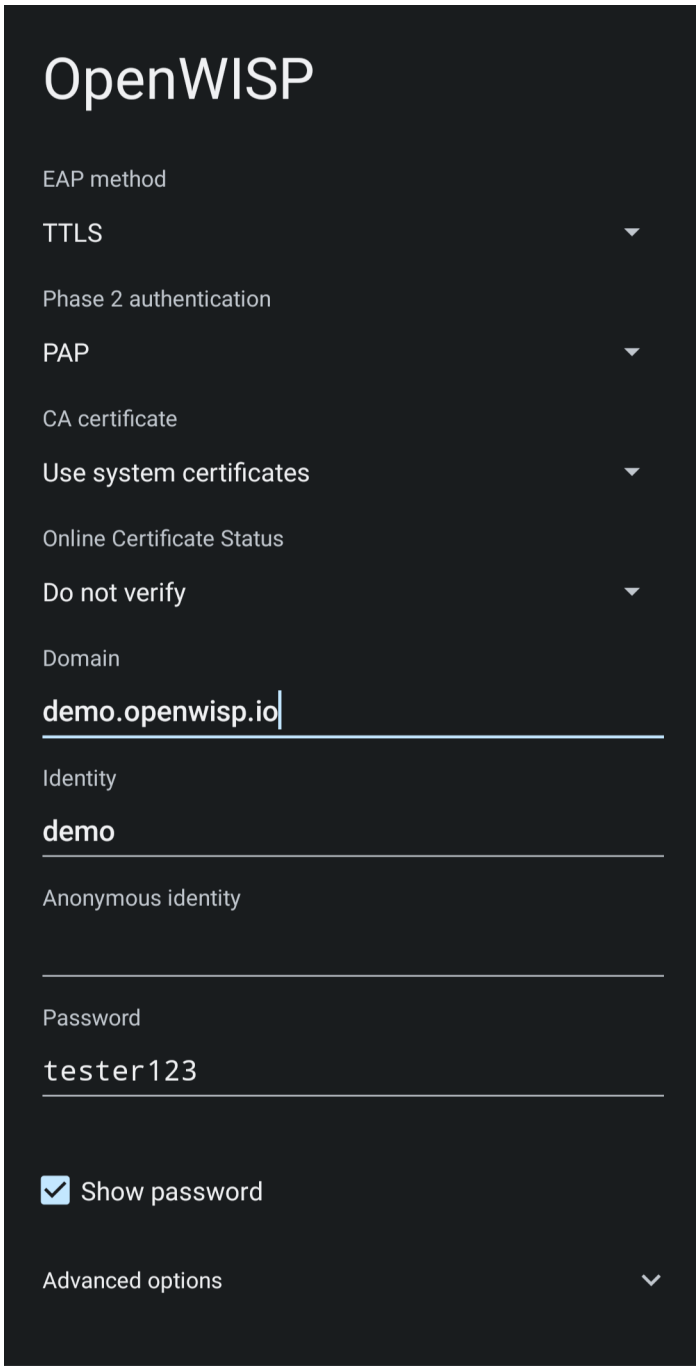
Find the "OpenWISP" SSID in the list of available WiFi networks on your mobile and click on it. Fill in the details as follows:

- **EAP method**: Set this to TTLS.
- **Phase 2 authentication**: Set this to PAP.
- **CA certificate: Select one of the options based on your**
  FreeRADIUS configuration.
- **Domain**: Enter the domain based on the server certificate used by FreeRADIUS.
- **Identity** and **Password**: Use the OpenWISP user's username for `Identity` and password for `Password`.

Note

If you are trying this feature on our OpenWISP Demo System, you can use the **demo** user to authenticate. You will need to update the following fields as mentioned:

- **CA certificate**: Set this to `Use system certificates`
- **Domain**: Set this to `demo.openwisp.io`
- **Identity** and **Password**: Use the demo user credentials.

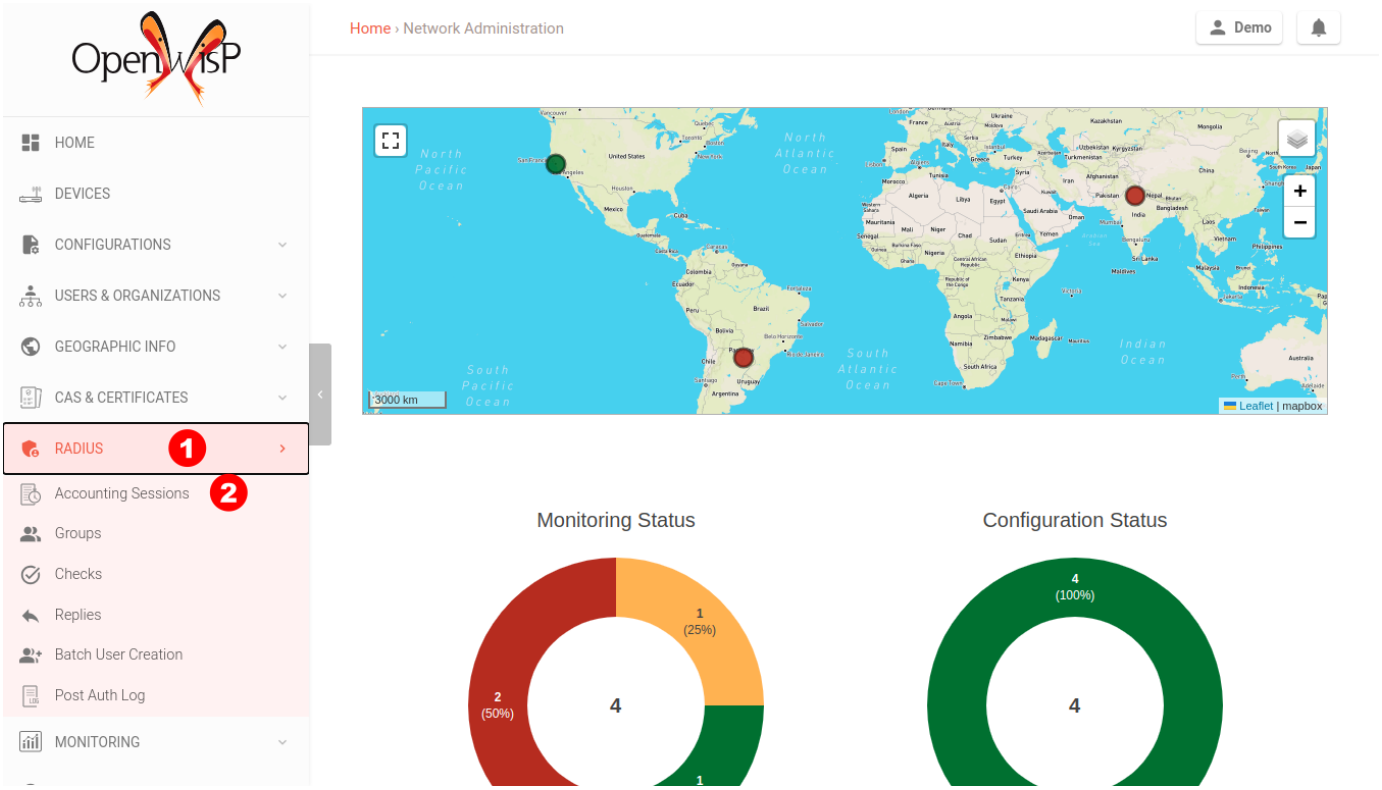


You can leave the **Advanced options** unchanged and click on **Connect** after filling in the details.

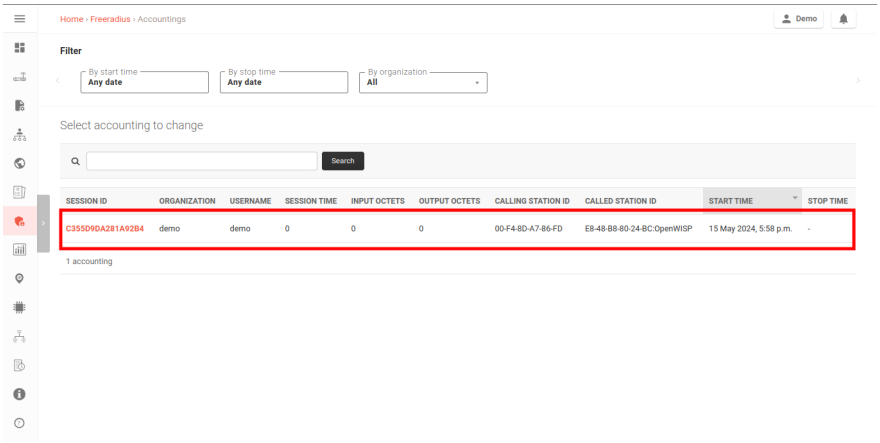
Verifying and Debugging

If everything worked as expected, your device should connect to the WiFi and allow you to browse the internet.

You can also verify the RADIUS session created on OpenWISP. From the OpenWISP navigation menu, go to `RADIUS` and then `Accounting Sessions`.



You should see a RADIUS accounting session for this device.

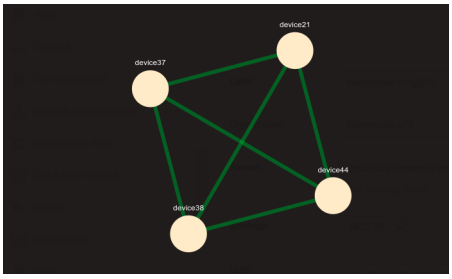


If your smartphone does not connect to the internet, you can troubleshoot the FreeRADIUS configuration by following the steps in the Debugging & Troubleshooting.

Seealso

- Open and/or WPA protected WiFi Access Point SSID
- WiFi Hotspot, Captive Portal (Public WiFi), Social Login
- How to Set Up a Wireless Mesh Network

How to Set Up a Wireless Mesh Network



Introduction & Prerequisites

What is a Mesh Network?

A **mesh network** is a **decentralized network architecture** where each node not only communicates with its immediate neighbors but also relays data for other mesh nodes, creating a peer-to-peer network.

The word "mesh" primarily describes the interconnected topology of the network, while **wireless mesh networks** specifically refer to mesh networks deployed using standard WiFi bands (2.4 GHz / 5 GHz) as the physical connection medium.

The advantages of this network architecture include:

- **Resilience**: Due to its interconnected topology, there's no single point of failure, so the dynamic routing mesh protocols used to route traffic are able to implement *self-healing* behavior, rerouting traffic along alternative paths when a link fails. This redundancy ensures continued operation and makes the network *resilient to temporary failures*.
- **Flexibility**: Deploying new nodes or relocating existing ones is straightforward due to consistent configurations across all nodes. This allows the network to scale without increasing configuration maintenance costs. Additionally, ad-hoc deployment is possible without extensive planning.

These benefits make *mesh networking technologies* particularly valuable for expanding WiFi coverage area in large spaces like offices, spacious houses, and rural areas, while controlling deployment and maintenance costs.

How to configure a wireless mesh network?

In this tutorial, we'll guide you through the *best practices for mesh network setup* using the mesh mode (also known as 802.11s) on OpenWrt through OpenWISP. Additionally, we'll provide valuable tips on monitoring and maintaining the mesh network, focusing on signal strength and network performance.

This tutorial focuses on using **open source solutions for mesh networking**.

Firmware Requirements

In order to use mesh mode with wireless encryption, your firmware needs to be equipped with a version of the `wpad` package which supports mesh encryption.

Please refer to the OpenWrt 802.11s documentation for more information.

Note

The **OpenWrt** firmware image provided for the OpenWISP Demo System includes the full `wpad` package by default.

General Assumptions

In this tutorial we make a few assumptions and choices which are explained below.

At Least 2 Devices

We assume you are already managing and monitoring at least two devices through your OpenWISP instance.

One Radio Available

We require at least one radio named `radio0` to be available and enabled for the successful execution of this tutorial.

For simplicity, we will focus on a single radio, but it's important to note that the mesh functionality can be extended to multiple radios if necessary. This can improve backhaul performance and reduce interference.

Alternatively, you have the option of running the mesh on one radio while the access points operate on another radio to avoid interference and increase the performance of the mesh network, mitigating issues like interference, optimizing for latency and throughput.

However, these additional scenarios are not explained in this tutorial and are left as an exercise for the reader.

Existing DHCP server on the LAN

WiFi in mesh mode (802.11s) operates at the layer 2 protocol, enabling us to bridge the mesh interface with the LAN interface, effectively creating a wireless extension of the LAN network.

This configuration assumes that the mesh devices will function as wireless extenders for an existing LAN, already equipped with a DHCP server.

Consequently, we will define a `br-lan` interface in DHCP client mode, with the spanning tree protocol enabled.

This helps prevent loops in case of accidental Ethernet cable connections to another mesh extender within the LAN.

Additionally, we will disable the default DHCP server on the LAN interface, which comes preconfigured in OpenWrt.

Creating the Template

Hint

If you don't know what a template is, please see Configuration Templates.

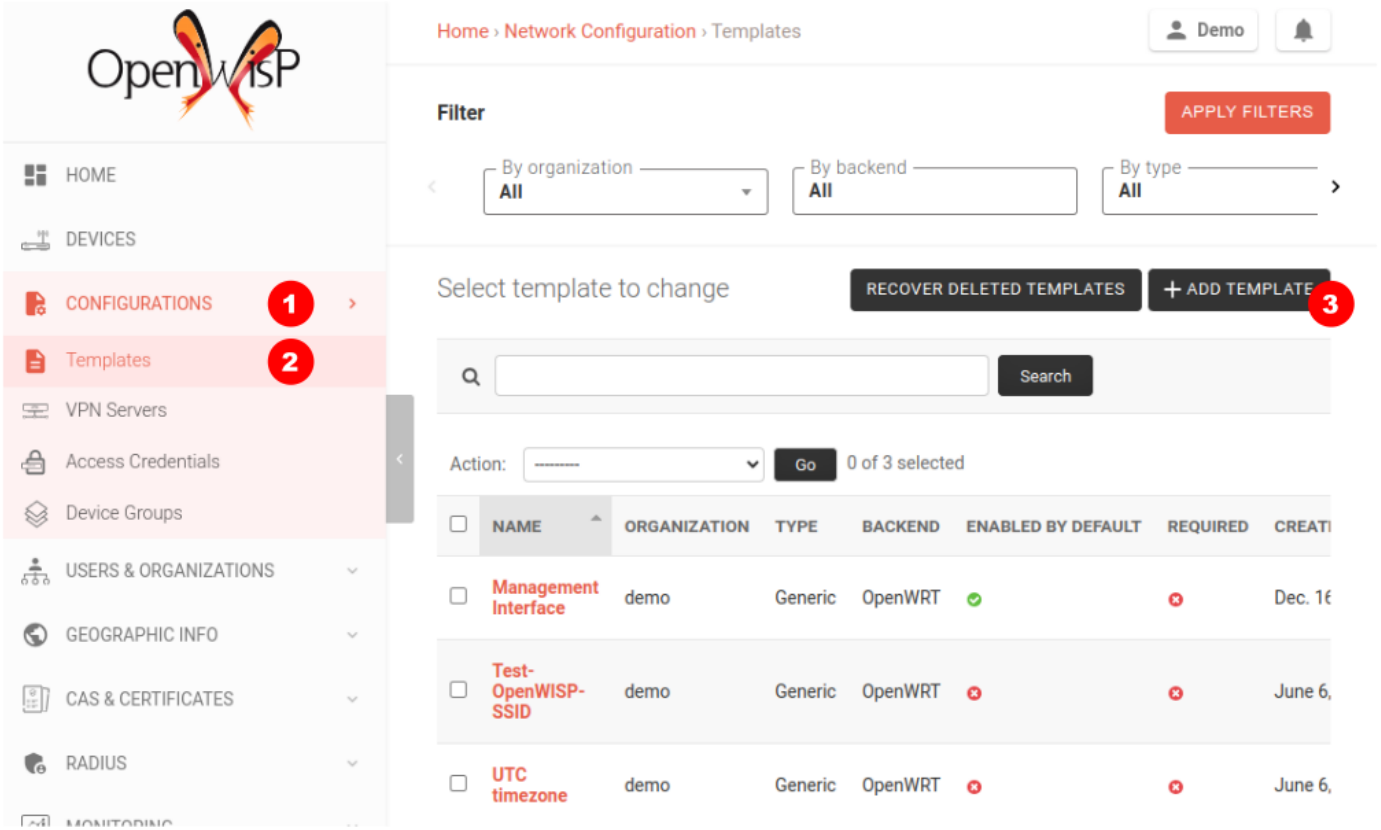
Note

This template is also available in our Demo System as Mesh Demo, **feel free to try it out!**

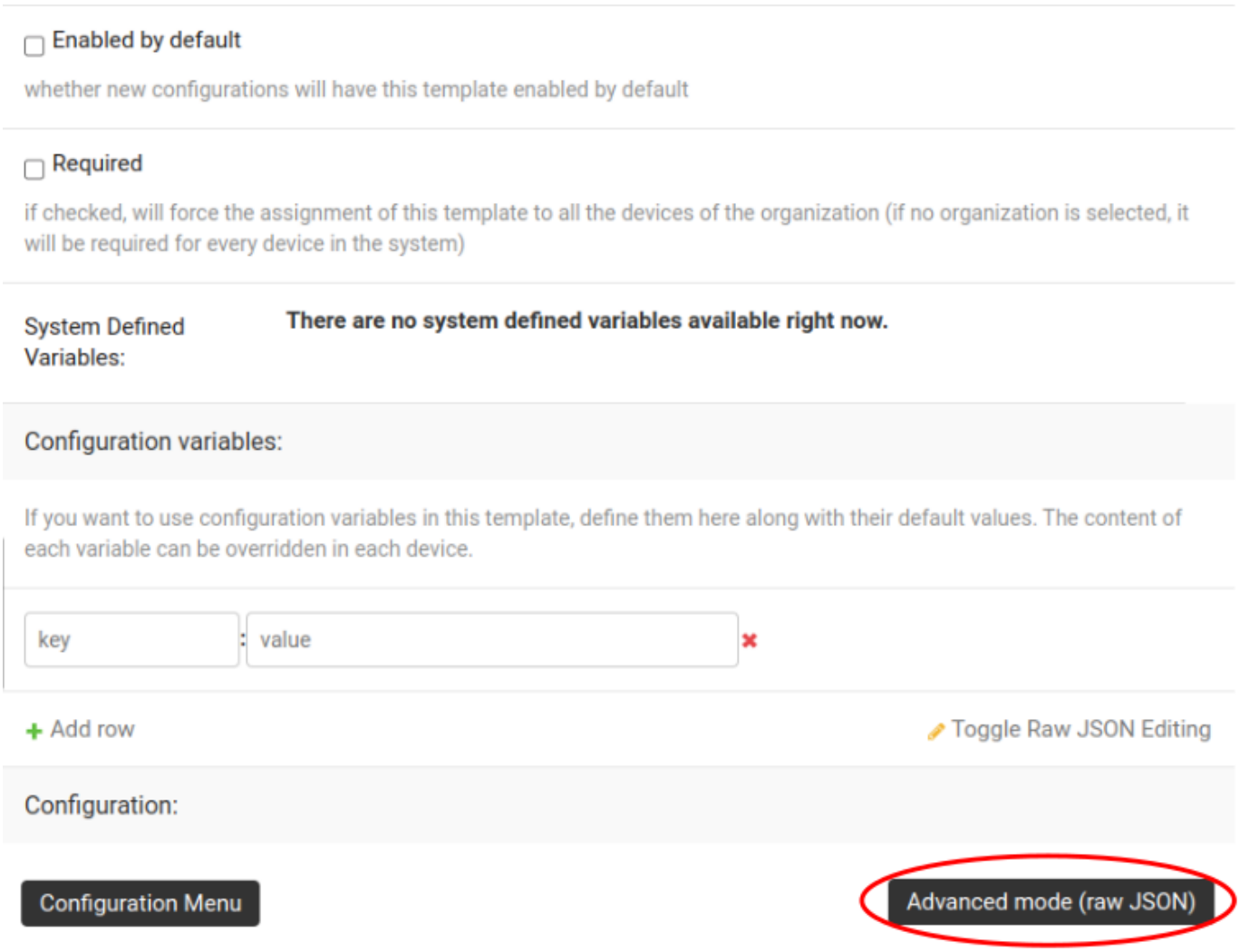
How to automate a mesh network?

In this section we'll explain how to automate the provisioning of new mesh nodes with a Mesh Configuration Template.

From the OpenWISP navigation menu, go to `Configurations` and then `Templates`, from here click on the `Add template`.



Fill in name, organization, leave type set to "Generic", backend set to "OpenWrt", scroll down to the Configuration section, then click on "Advanced mode (raw JSON)".



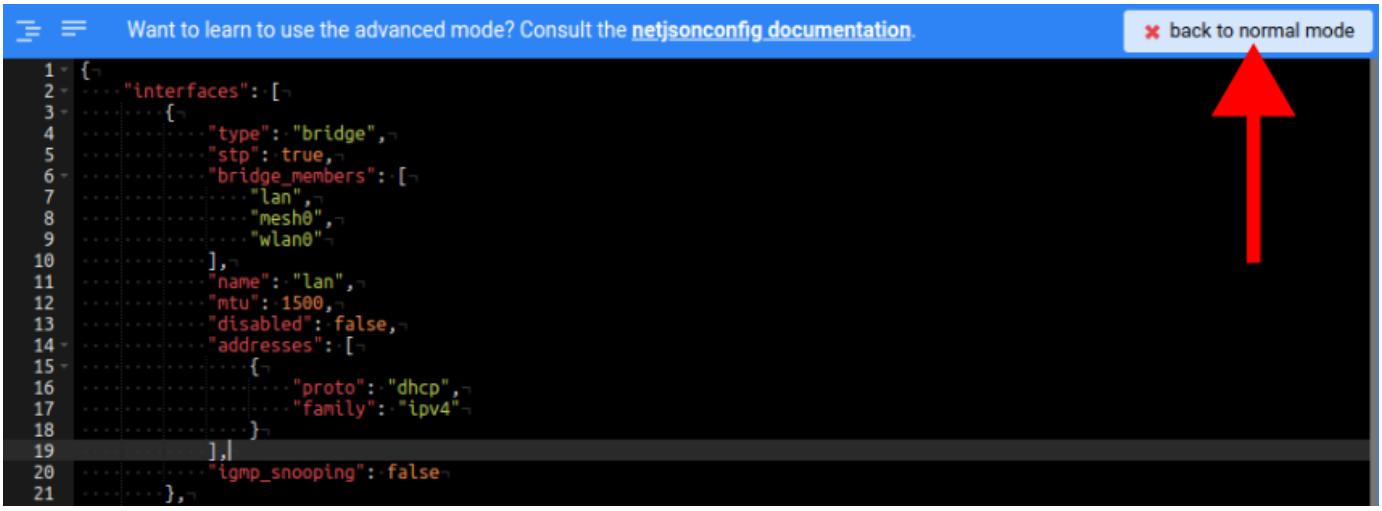
Once the advanced mode editor is open you can paste the following NetJSON:

```
{
    "interfaces": [
        {
            "name": "lan",
            "type": "bridge",
            "mtu": 1500,
            "disabled": false,
            "stp": true,
            "igmp_snooping": false,
            "bridge_members": [
                "lan",
                "mesh0",
                "wlan0"
            ],
            "addresses": [
                {
                    "proto": "dhcp",
                    "family": "ipv4"
                }
            ]
        },
        {
            "type": "wireless",
```

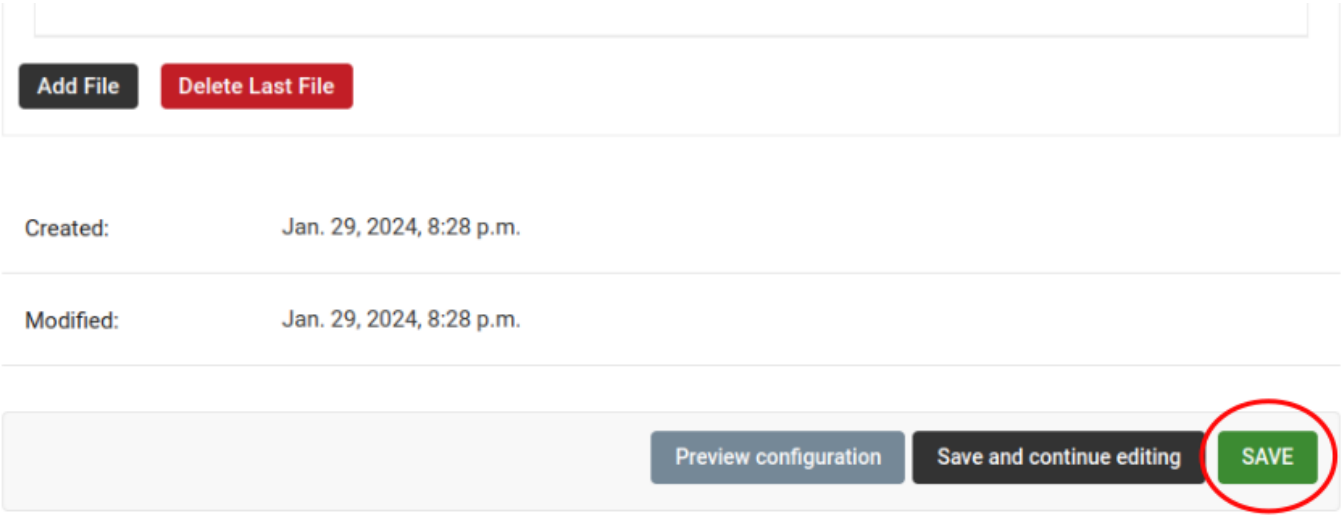
```
            "name": "mesh0",
            "mtu": 1500,
            "disabled": false,
            "wireless": {
                "mode": "802.11s",
                "radio": "radio0",
                "ack_distance": 0,
                "rts_threshold": 0,
                "frag_threshold": 0,
                "mesh_id": "mesh0",
                "encryption": {
                    "protocol": "wpa2_personal",
                    "key": "0penW1SP0987654321",
                    "disabled": false,
                    "cipher": "auto",
                    "ieee80211w": "0"
                },
                "network": [
                    "lan"
                ]
            }
        },
        {
            "type": "wireless",
            "name": "wlan0",
            "mtu": 1500,
            "disabled": false,
            "wireless": {
                "mode": "access_point",
                "radio": "radio0",
                "ssid": "Mesh AP",
                "hidden": false,
                "wds": false,
                "wmm": true,
                "isolate": false,
                "ieee80211r": true,
                "reassociation_deadline": 1000,
                "ft_psk_generate_local": false,
                "ft_over_ds": true,
                "rsn_preauth": false,
                "macfilter": "disable",
                "maclist": [],
                "encryption": {
                    "protocol": "wpa2_personal_mixed",
                    "key": "meshApTesting1234",
                    "disabled": false,
                    "cipher": "ccmp",
                    "ieee80211w": "1"
                },
                "network": [
                    "lan"
                ]
            }
        }
    ],
    "files": [
        {
            "path": "/etc/openwisp/pre-reload-hook",
            "mode": "0700",
            "contents": "#!/bin/sh\n\n# delete any br-lan definition to avoid conflicts\nuci
```

```
        }
    ]
}
```

Then click on "back to normal mode" to close the advanced mode editor.



Now you can save the new template.



At this point you're ready to assign the template to your devices, but before doing so you may want to read on to understand the different components of this template:

- The `br-lan` defines a bridge with the following members: `lan`, `mesh0` and `wlan0`.
- The `mesh0` provides the encrypted wireless mesh interface bound to `radio0`.
- The `wlan0` interface provides WiFi access to the mesh network for clients not equipped with 802.11s.
- A `pre-reload-hook` script which is executed before OpenWrt reloads its services to make the configuration changes effective.

Why we use a `pre-reload-hook` script

In the template shared above, we utilize a pre-reload-hook script to execute the following configuration changes:

- Ensure that `radio0` is enabled, set on a specific channel and country code to allow communication between mesh nodes. You can customize the channel and country code according to your preferences. However, make these changes before deploying your mesh nodes and disconnecting them from the Ethernet network, as modifying the channel or country code on an active mesh network will disrupt it.

- Disable the default DHCP server preconfigured in OpenWrt on the `br-lan` interface to prevent interference with the existing DHCP server in the LAN.
- Increase the `test_retries` option of the openwisp-config agent to 8. This enhancement enhances the agent's resilience to temporary failures in reaching the OpenWISP server after applying configuration changes. Mesh configuration changes trigger a reload of the WiFi stack, which may take a few minutes to become effective. During this period, we want to avoid the agent to mistakenly consider the connection as lost, to prevent it from flagging the upgrade as failed and rollback to the previous configuration.

We could have redefined the entire configuration for `radio0`, the LAN DHCP server and openwisp-config, but doing so would have posed some issues:

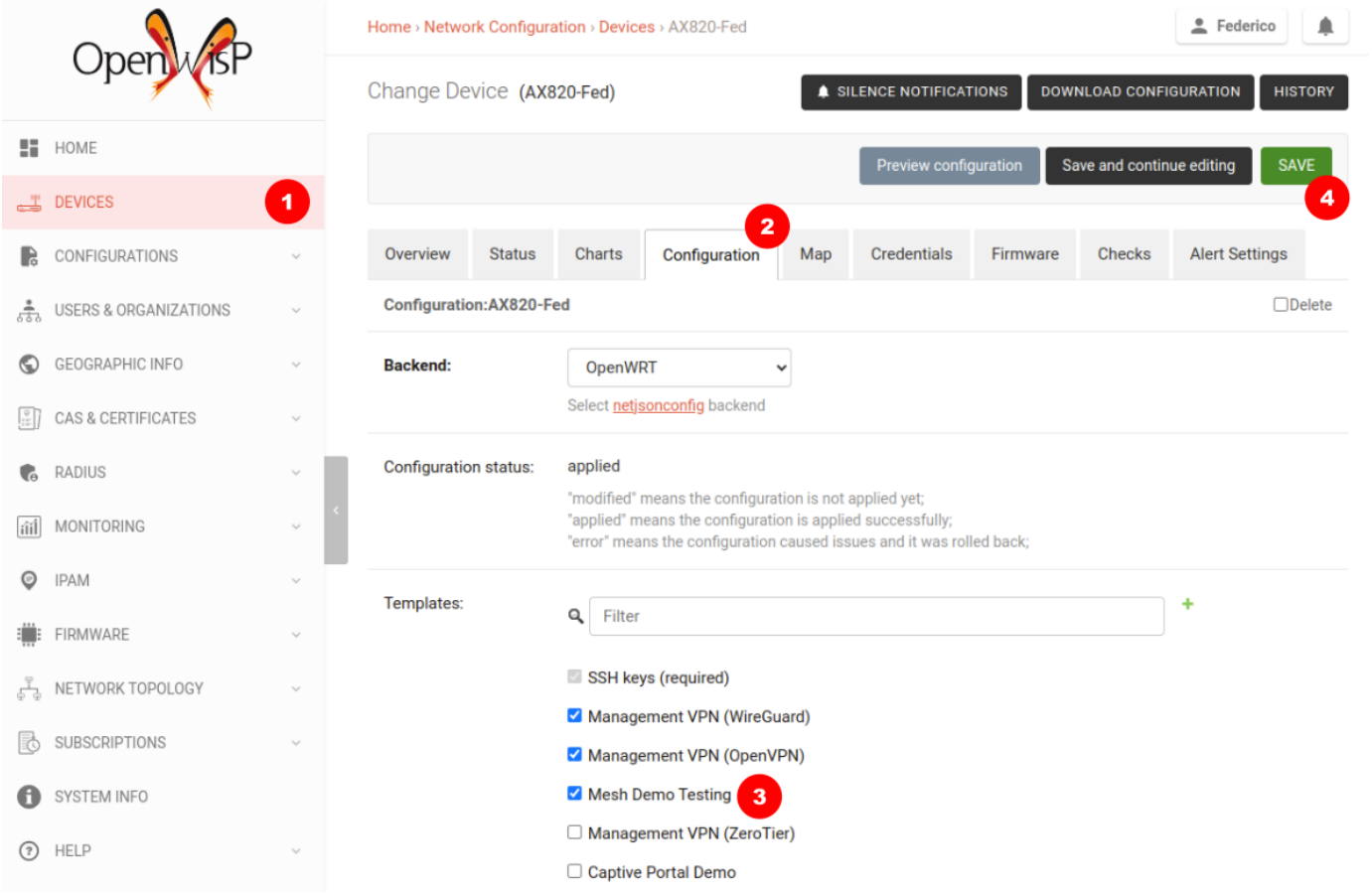
- There's no guarantee that the same radio settings will work uniformly on every hardware supported by OpenWrt. By altering only the necessary settings, we ensure the same template can be applied across a broad spectrum of devices, making the tutorial easy for a wide range of users.
- Creating a template that includes all possible settings would result in verbosity, making it challenging for readers to digest.

Once you have successfully set this up, feel free to modify the template configuration and tailor any part to suit your requirements.

Enable the Mesh Template on the Devices

Now is time to apply this *mesh template* to the nodes that we want to make part of the mesh.

Click on "devices" in the navigation menu, click on the device you want to assign the mesh template to, then go to the "Configuration" tab, select the template just created, then click on save.



Verifying and Debugging

Once the configuration is applied to the device, if you access your device via SSH you can double check that everything worked fine by comparing the output you get from the command outputs shown below.

Check the bridge with `brctl show`:

```
bridge name bridge id               STP enabled     interfaces
br-lan              7fff.44d1fad204c5       yes             lan
                                                            wlan0
                                                            mesh0
```

Check the WiFi interfaces with `iwinfo`:

```
mesh0     ESSID: "mesh0"
          Access Point: 44:D1:FA:D2:00:01
          Mode: Mesh Point  Channel: 1 (2.412 GHz)  HT Mode: HT20
          Center Channel 1: 1 2: unknown
          Tx-Power: 20 dBm  Link Quality: 68/70
          Signal: -42 dBm  Noise: -87 dBm
          Bit Rate: 1.0 MBit/s
          Encryption: WPA3 SAE (CCMP)
          Type: nl80211  HW Mode(s): 802.11ax/b/g/n
          Hardware: 14C3:7915 14C3:7915 [MediaTek MT7915E]
          TX power offset: none
          Frequency offset: none
          Supports VAPs: yes  PHY name: phy0

wlan0     ESSID: "Mesh AP"
          Access Point: 44:D1:FA:D2:00:01
          Mode: Master  Channel: 1 (2.412 GHz)  HT Mode: HE20
          Center Channel 1: 1 2: unknown
          Tx-Power: 20 dBm  Link Quality: unknown/70
          Signal: unknown  Noise: -85 dBm
          Bit Rate: unknown
          Encryption: mixed WPA2/WPA3 PSK/SAE (CCMP)
          Type: nl80211  HW Mode(s): 802.11ax/b/g/n
          Hardware: 14C3:7915 14C3:7915 [MediaTek MT7915E]
          TX power offset: none
          Frequency offset: none
          Supports VAPs: yes  PHY name: phy0
```

Once you have assigned the template to at least two devices which are close to each other, you can verify whether they have formed a mesh with `iw mesh0 station dump`, which should return the number of connected mesh nodes (called stations):

```
Station 44:d1:fa:d2:04:d6 (on mesh0)
    inactive time:  10 ms
    rx bytes:       9050195
    rx packets:     80356
    tx bytes:       1169064
    tx packets:     7196
    tx retries:     0
    tx failed:      0
    rx drop misc:   200
    signal:         -42 [-43, -49] dBm
    signal avg:     -42 [-43, -49] dBm
    Toffset:        287058701286 us
    tx bitrate:     243.7 MBit/s HE-MCS 10 HE-NSS 2 HE-GI 1 HE-DCM 0
    tx duration:    32732793 us
    rx bitrate:     258.0 MBit/s HE-MCS 10 HE-NSS 2 HE-GI 0 HE-DCM 0
    rx duration:    3451735 us
    airtime weight: 256
    mesh llid:      0
    mesh plid:      0
```

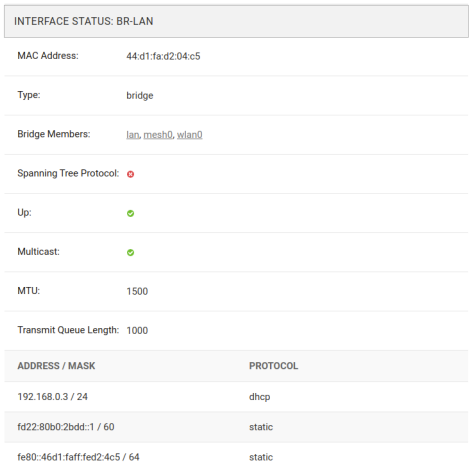
```
mesh plink:     ESTAB
mesh airtime link metric: 48
mesh connected to gate: yes
mesh connected to auth server:  no
mesh local PS mode:     ACTIVE
mesh peer PS mode:      ACTIVE
mesh non-peer PS mode:  ACTIVE
authorized:     yes
authenticated:  yes
associated:     yes
preamble:       long
WMM/WME:        yes
MFP:            yes
TDLS peer:      no
DTIM period:    2
beacon interval:100
connected time: 3511 seconds
associated at [boottime]:       272718.754s
associated at:  1706572676925 ms
current time:   1706576187500 ms
```

If you didn't get the expected results we recommend looking at the `logread` output and look for any critical error shown in the log output, this should help you to fix it.

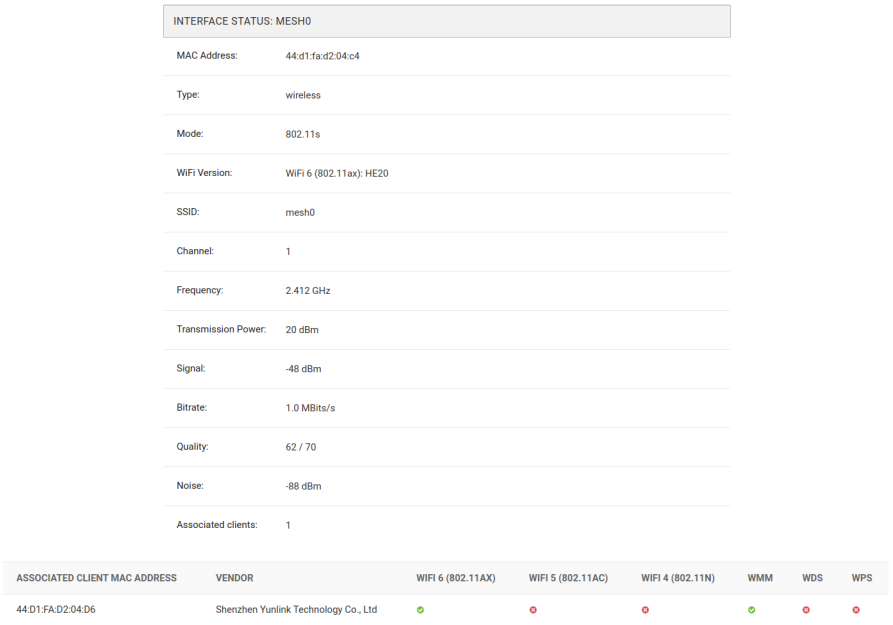
Monitoring the Mesh Nodes

If everything has worked out successfully and you have the OpenWISP monitoring agent running correctly on your device, you should start seeing monitoring information about the mesh network in the status tab of the device page.

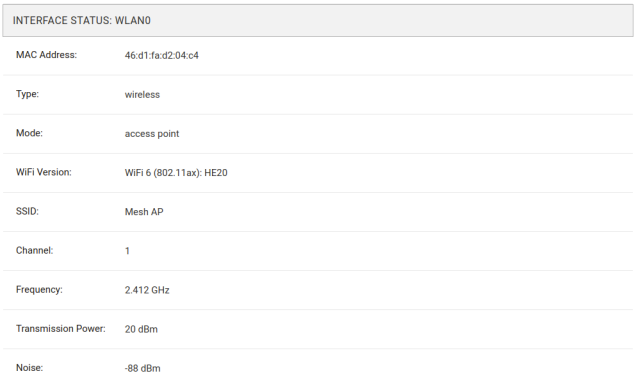
Bridge interface:



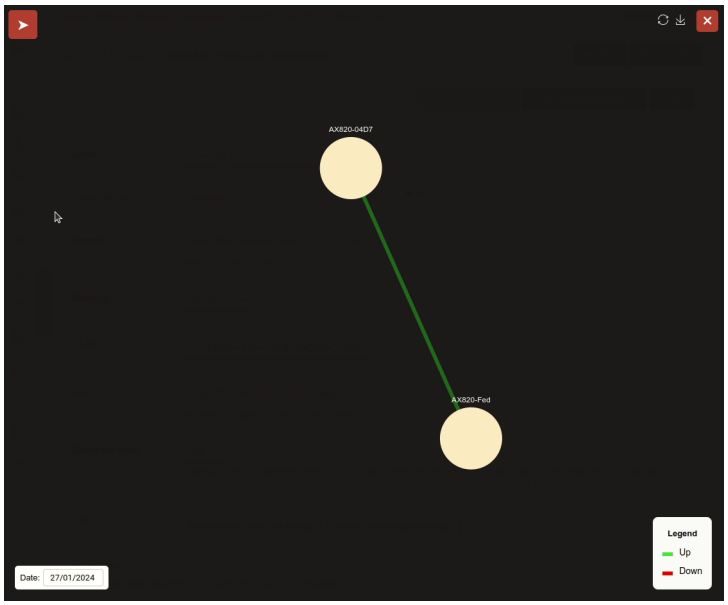
Mesh0 interface:



Wlan0 interface:



Mesh Topology Collection and Visualization

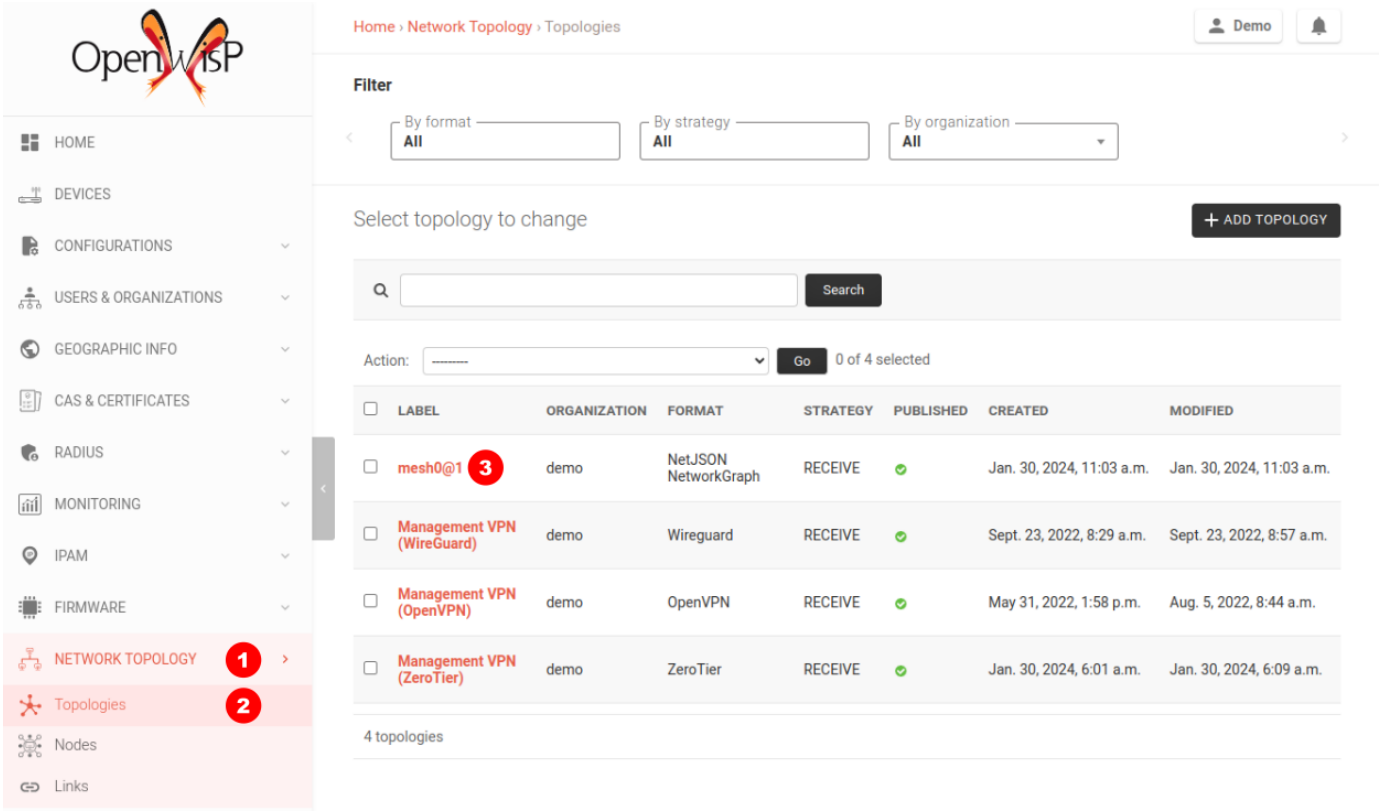


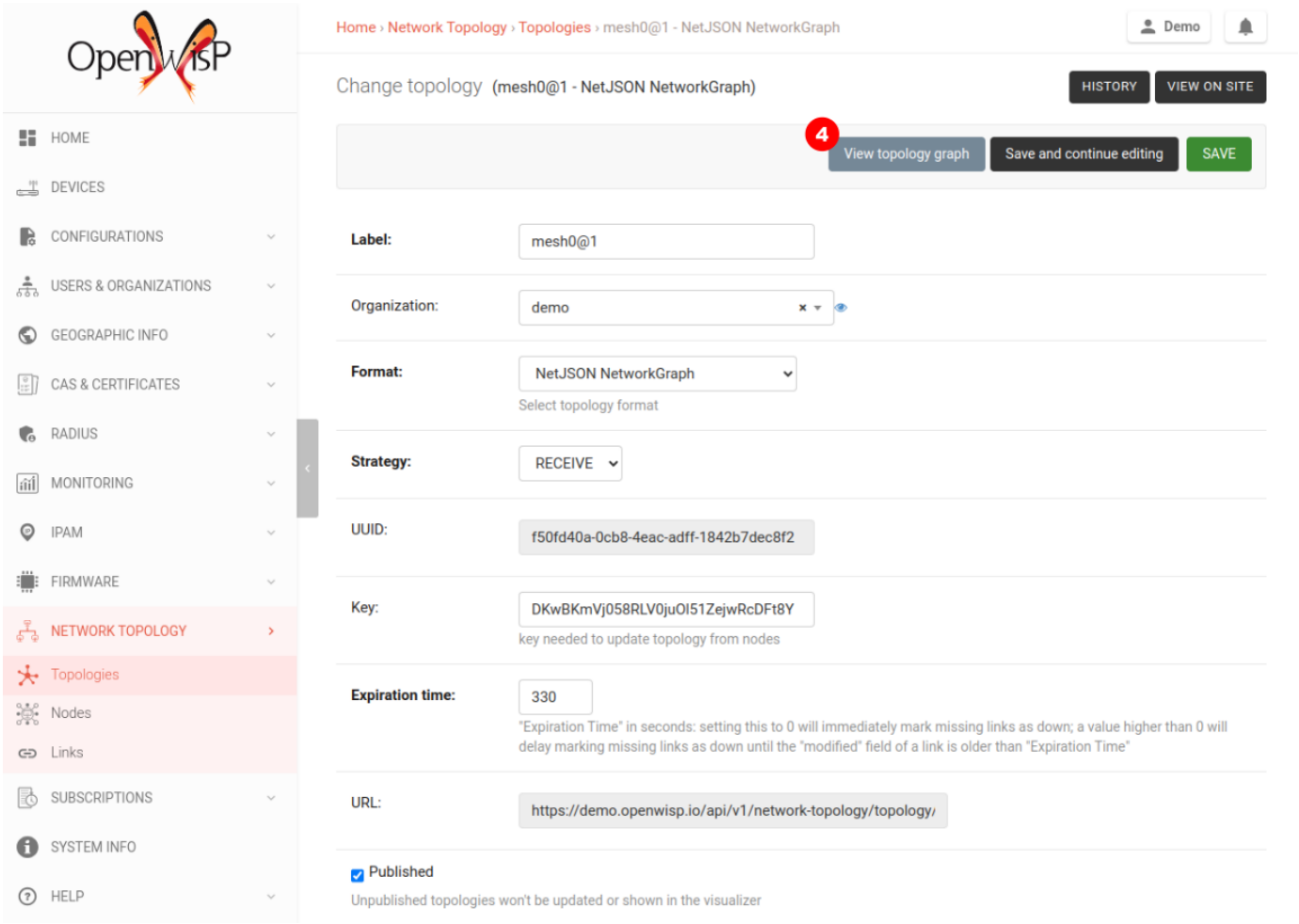
In June 2023, we introduced a new feature to the Network Topology module of OpenWISP, enabling the automatic collection of *mesh network topology* data from for visualization purposes.

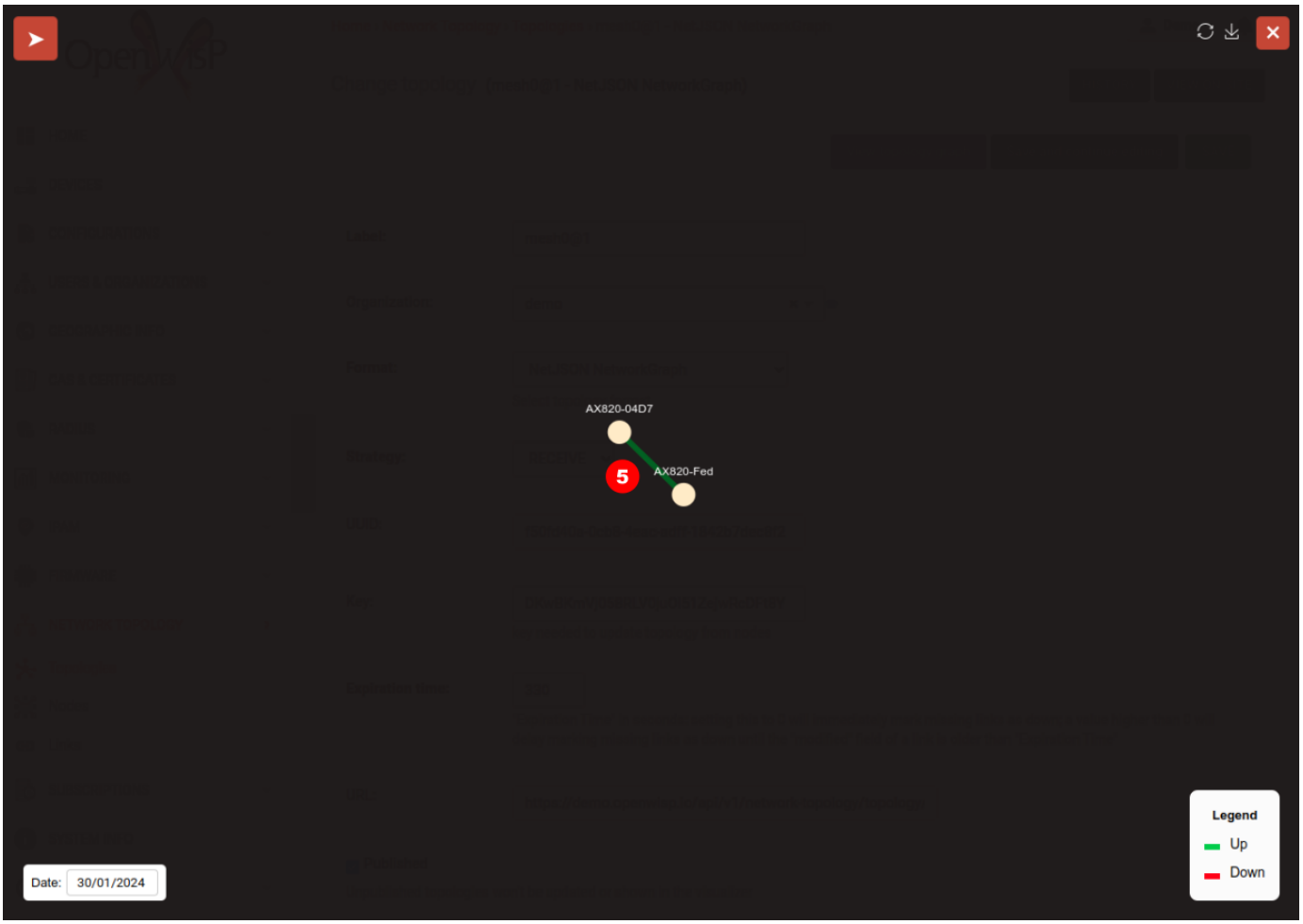
Setting up this feature is beyond the scope of this tutorial, but we provide pointers to demonstrate its usefulness and guide you in finding the information needed to set it up:

- Relevant Network Topology documentation
- Github pull request: [feature] WiFi Mesh integration

If you have been playing with our **Demo System**, you can try this feature there! You only have to register at least 2 devices to the Demo System, enable the Mesh Demo template on your devices and wait a few minutes until the data is collected and shown in the **Network Topology List** as shown below.




OpenWISP
HOME
DEVICES
CONFIGURATIONS
USERS & ORGANIZATIONS
GEOGRAPHIC INFO
CAS & CERTIFICATES
RADIUS
MONITORING
IPAM
FIRMWARE
NETWORK TOPOLOGY
Topologies
Nodes
Links
SUBSCRIPTIONS
SYSTEM INFO
HELP
Home › Network Topology › Topologies › mesh0@1 - NetJSON NetworkGraph
Demo
Change topology (mesh0@1 - NetJSON NetworkGraph)
HISTORY
VIEW ON SITE
4
View topology graph
Save and continue editing
SAVE
Label:
mesh0@1
Organization:
demo
Format:
NetJSON NetworkGraph
Select topology format
Strategy:
RECEIVE
UUID:
f50fd40a-0cb8-4eac-adff-1842b7dec8f2
Key:
DKwBKmVj058RLV0juOI51ZejwRcDFt8Y
key needed to update topology from nodes
Expiration time:
330
"Expiration Time" in seconds: setting this to 0 will immediately mark missing links as down; a value higher than 0 will delay marking missing links as down until the "modified" field of a link is older than "Expiration Time"
URL:
https://demo.openwisp.io/api/v1/network-topology/topology/
Published
Unpublished topologies won't be updated or shown in the visualizer



Changing the Default 802.11s Routing Protocol

Switching the mesh routing protocol can be beneficial for optimizing the most efficient path between two nodes and reducing the number of hops, but it is essential to configure it correctly to achieve optimal performance.

Using a mesh routing protocol other than the default protocol shipped in the 802.11s implementation is out of scope of this tutorial but can be done.

You will need to turn off mesh forwarding and configure the routing daemon of your choice.

Seealso

- Open and/or WPA protected WiFi Access Point SSID
- WiFi Hotspot, Captive Portal (Public WiFi), Social Login
- How to Set Up WPA Enterprise (EAP-TTLS-PAP) authentication

Community Resources

Help us to grow

You don't need necessarily to be a programmer in order to help out.

An apparently insignificant action can have a very positive impact on the project and in this page we'll explain why it's in your interest to help the project grow.

Table of Contents:

Are you using OpenWISP for your organization?

If you are using OpenWISP for your company or no profit organization, it's in your best interest to help the project to grow, because the more we grow as a community, the more contributors we'll attract which in turn will help us to improve the software, its documentation and keep alive the support channels.

Even small and apparently meaningless actions can make a big difference if performed by a sufficient number of people.

Note

If you need commercial support for your business, see the paragraph about Commercial support and funding development.

How to help

1. Open new discussion threads

The Github Discussions Forum and the Mailing List are excellent places to ask questions or share information regarding OpenWISP.

Every question and its replies are archived and indexed by search engines, creating a repository of solved problems that people can find over time.

For this reason, **using these channels for support questions should be preferred over the chats**.

Warning

Please be mindful that **over 700 people read these channels** and **discussions are indexed forever**. For these reasons, you should:

- Keep the focus of the discussion technical.
- Avoid irrelevant comments.

- Be mindful about what you write.
- Keep the tone calm and constructive.
- Be respectful to the volunteers who reply in their free time.
- Avoid generating noise.

When subscribing to the mailing list, we suggest choosing one of these options:

- Receive all emails by creating a filter in your mailbox that moves the messages to a dedicated folder.
- Receive a periodic summary (abridged or digest).

2. Send feedback

When you use OpenWISP, you may find ideas about improvements, new features or you may incur in bugs.

It's very helpful to us if you send us your feedback in some way. The preferred way to send feedback is to use the mailing list, but you can send feedback in any way you want.

If you have found a bug we will likely ask you to open a bug report in a specific github repository, if you can follow up with this activity it will be very helpful to us.

3. Stars on github

Unfortunately, when evaluating a project, a disproportionate amount of people look at the github stars as a method of evaluation on how popular a project is and if they don't see many stars they discard the idea of using it.

OpenWISP is composed of many modules and for that reason we don't have a single super popular github repository with thousands of stars, but when new users and developers look at our github organization page they may not get this at first glance and they will start looking for the numbers of stars.

Yes, we know it sounds silly, but since it doesn't cost you anything, it would be really useful if you could **take a look at our projects on github** and **star the ones you find most interesting**.

4. Documentation

If you find anything in this documentation that you think may be improved, please edit the document on github and send us a pull request, alternatively you can file a bug report or write to the support channels.

5. Social media

If you are using OpenWISP, it's very useful to let the world know about it by sharing a public post on social media using the *#openwisp* hashtag.

We also have a twitter account and a facebook page you can follow to help us share news about our community.

If more people talk about OpenWISP on social media, we increase the chance that those who have the will and technical skills to contribute will hear about its existence.

6. Blogging

Write a blog post about how you are using OpenWISP!

It would be great if you could explain the reasons for which you chose OpenWISP, the traits you like about it and the traits you don't like about it.

This is **VERY** helpful not only for the core developers but also for potential readers that may find your blog post and read about your use case: maybe they have the same use case and they want to know if OpenWISP is a good fit for them.

A concise, straight to the point blog post with some images and screenshots will go a long way in attracting new people into the community.

7. Conferences & Meetups

If you like to share your knowledge at conferences and meetups, you may cite OpenWISP in one of your presentations or lightning talks, you may also show some of its features, if relevant.

8. Participate

By participating actively in the support channels you can also help us a lot: the welcoming level of an open source community is a key factor in attracting a good numbers of contributors.

9. Contribute technically

Are you skilled in one of the following areas?

- technical writing
- python
- networking
- graphic/web design
- frontend development
- OpenWrt
- Freeradius
- linux
- devops

If yes, you can help us greatly. Find out more about this subject in How to contribute to OpenWISP.

10. Commercial support and funding development

Please refer to Commercial Support.

Press

In this page we aim to collect the following:

- presentations, blog posts and academic publications in which OpenWISP is either the main subject or it's mentioned
- logos and other design files

Presentations

OpenWISP: a Hackable Network Management System for the 21st Century

Presented by *Federico Capoano* at the IETF Meeting 103 Bangkok:

- slides

django-freeradius at PyCon Italia 2018

Presented by *Fiorella De Luca* at PyCon Italy 2018:

- video
- abstract

OpenWISP 2: the modular configuration manager for OpenWrt

Presented by *Federico Capoano* at OpenWrt Summit 2017 in Prague:

- video
- slides

Applying the Unix Philosophy to Django projects

Presented by *ederico Capoano* at PyCon Italy 2017:

- video
- slides

Opening Proprietary Networks with OpenWISP

Lightning talk by *Federico Capoano* at DjangoCon Europe 2017:

- slides

OpenWISP2 a self hosted solution to control OpenWrt/LEDE devices

Talk by *Federico Capoano* at FOSDEM 2017 in Brussels:

- video
- abstract

Do you really need to fork OpenWrt?

Presented at OpenWrt Summit 2015 in Dublin:

- video

OpenWISP GARR Conference 2011

Interview for GARR Conference presented by *Davide Guerri* (in Italian):

- video

OpenWISP e Progetti WiFi Nazionali

Interview for GARRTV by *Davide Guerri* (in Italian):

- video

Blog Posts

- How Bottom-up Broadband will overcome the 'last mile' problem
- netjsonconfig: convert NetJSON to OpenWrt UCI
- Automate OpenWrt/LEDE firmware generation with Ansible
- django-x509: a reusable django app for PKI management

- Network Topology Visualizer: django-netjsongraph
- Marco and Alessia for an increasingly open network (in Italian)
- Fly with Uniurb and OpenWISP to the Google Summer of Code 2018 (in Italian)
- Uniurb at the Google Summer of Code with OpenWISP2 and Marco (in Italian)
- Post by the Metropolitan City of Rome (in Italian)

Google Summer of Code Blog Posts

2023 Contributors

- ZeroTier Tunnels Support for OpenWISP Controller by *Aryaman* (*Aryamanz29*).

2022 Contributors

- Iperf3 Check for OpenWISP Monitoring by *Aryaman* (*Aryamanz29*).
- Improve netjsongraph.js for its new release by *Vaishnav Nair* (*totallynotvaishnav*).

2021 Students

- OpenWISP REST API by *Manish Kumar Shah* (*manishshah120*).
- OpenWrt OpenWISP Monitoring by *Kapil Bansal* (*devkapilbansal*).
- OpenWISP WiFi Login Pages by *Sankalp* (*codesankalp*).
- Modern UI/UX by *Nitesh Sinha* (*nitehsinha17*).
- Revamp Netengine and add its SNMP capability to OpenWISP Monitoring by *Purhan Kaushik* (*purhan*).

2020 Students

- Introducing OpenWISP Monitoring: Project report by *Hardik Jain* (*nepython*).
- Merge django reusable-apps by *Ajay Tripathi* (*atb00ker*).
- OpenWISP Notifications Module by *Gagan Deep* (*pandafy*).

2019 Students

- Dockerization of OpenWISP by *Ajay Tripathi* (*atb00ker*).
- Project Report: NetJSONGraph.js Library of OpenWISP by *KuTuGu*.

2018 Students

- OpenWISP IPAM: IP Address Management tool for OpenWISP2 by *Anurag Sharma*.

2017 Students

- Adding AirOS support to netjsonconfig by *Edoardo Putti*.
- Building a Javascript Based Configuration UI for OpenWISP by *Nkhoh Gaston Che*.

- OpenWISP 2 Network Topology by *Rohith A. S. R. K*.
- Google Summer of Code 2017 Django-freeradius by *Fiorella De Luca*.
- Raspbian backend for OpenWISP 2 by *Ritwick DSouza*.

Research and publications

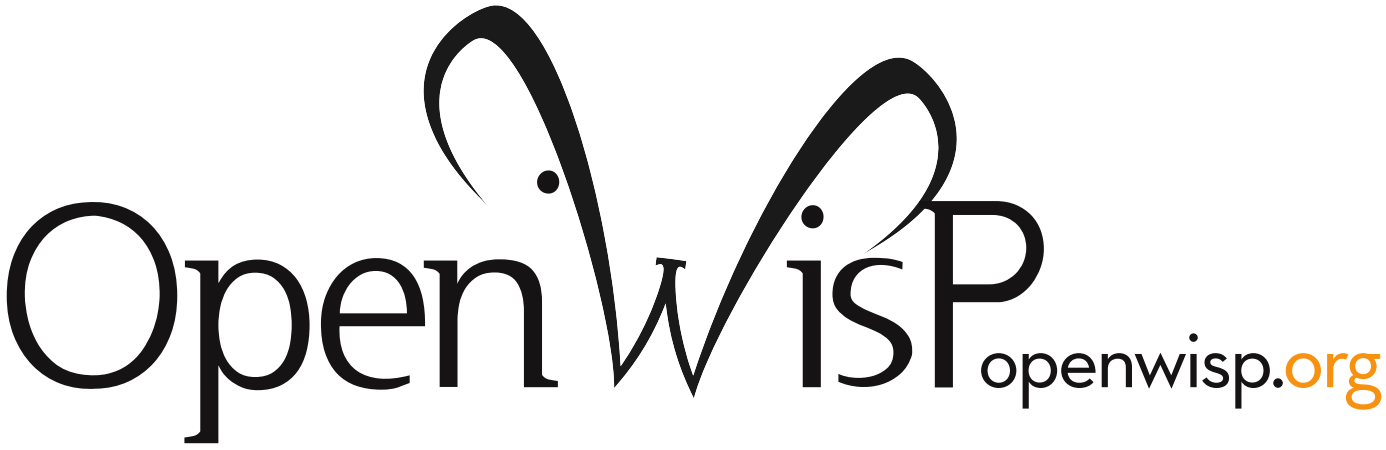
- **A Comprehensive Study on OpenWISP for Evolving Infrastructure Needs**
- **Monitoring Community Networks: Report on Experimentations on Community Networks**
- **Network Infrastructure as Commons**
- **Bottom-up Broadband Initiatives in the Commons for Europe Project**
- **Free Europe WiFi by Justel Pizarro** (in Spanish)
- **Bottom-up Broadband: Free Software Philosophy Applied to Networking Initiatives**
- **Study of community organizations and the creation of a collaborative environment for the initiative "Bottom up Broadband"** (in Catalan)
- **Control and management of WiFi networks** (in Slovenian)
- **IEEE publication**: ProvinciaWiFi: A 1000 hotspot free, public, open source WiFi network
- **OpenWISP, an original open source solution for the diffusion of wifi services** (in Italian)

Logos and Graphic material

OpenWISP Logo (Black Foreground)

OpenWISP Logo (White Foreground)

OpenWISP Logo (Black Foreground, with openwisp.org)



Code of Conduct

1. Purpose

OpenWISP aims to be a welcoming organization for contributors with the most varied and diverse backgrounds possible. We are devoted towards providing a friendly, safe and welcoming environment for all, regardless of gender, sexual orientation, ability, ethnicity, socioeconomic status, and religion.

This code of conduct outlines our expectations for all those who participate in our community, as well as the consequences for unacceptable behavior.

We invite all those who participate in OpenWISP to help us create safe and positive experiences for everyone.

2. Open Source Citizenship

An additional purpose of this Code of Conduct is to boost open source citizenship by encouraging participants to recognize and strengthen the relationships between our actions and their effects on our community.

Communities mirror the societies in which they exist and positive action is essential to prevent the many forms of inequality and abuses of power that exist in society.

If you see someone who is making an extra effort to ensure our community is welcoming, friendly, and encourages all participants to contribute to the fullest extent, we want to know.

3. Expected Behavior

The following behaviors are expected and requested of all community members:

- Participate in an authentic and active way. In doing so, you contribute to the health and longevity of this community.
- Exercise consideration and respect in your speech and actions.
- Attempt collaboration before conflict.
- Refrain from demeaning, discriminatory, or harassing behavior and speech.
- Be mindful of your surroundings and of your fellow participants. Alert community leaders if you notice a dangerous situation, someone in distress, or violations of this Code of Conduct, even if they seem inconsequential.
- Remember that community event venues may be shared with members of the public; please be respectful to all patrons of these locations.

4. Unacceptable Behavior

The following behaviors are considered harassment and are unacceptable within our community:

- Violence, threats of violence or violent language directed against another person.
- Sexist, racist, homophobic, transphobic, ableist or otherwise discriminatory jokes and language.
- Posting or displaying sexually explicit or violent material.
- Posting or threatening to post other people’s personally identifying information ("doxing").
- Personal insults, particularly those related to gender, sexual orientation, race, religion, or disability.
- Inappropriate photography or recording.
- Inappropriate physical contact. You should have someone’s consent before touching them.
- Unwelcome sexual attention. This includes, sexual comments or jokes; inappropriate touching, groping, and unwelcome sexual advances.
- Deliberate intimidation, stalking or following (online or in person).
- Advocating for, or encouraging, any of the above behavior.
- Sustained disruption of community events, including talks and presentations.

5. Consequences of Unacceptable Behavior

We do not tolerate harassment of the participants in any form. Unacceptable behavior from any community member, including sponsors and those with decision-making authority, will not be tolerated.

Anyone asked to stop unacceptable behavior is expected to comply immediately.

If a community member engages in unacceptable behavior, the community organizers may take any action they deem appropriate, up to and including a temporary ban or permanent expulsion from the community without warning (and without refund in the case of a paid event).

6. Reporting Guidelines

If you are being harassed, noticed that someone else is being harassed, or have any other concerns, please contact community organizers immediately.

Additionally, community organizers are available to aid community members to engage with local law enforcement or to otherwise help those experiencing unacceptable behavior feel safe. In the situation of in-person events, organizers will also provide escorts as desired by the person experiencing distress.

7. Addressing Grievances

If you feel you have been falsely or unfairly accused of violating this Code of Conduct, you should get in touch with the OpenWISP community managers by sending a short explanation of your grievance.

Your grievance will be handled in accordance with our existing governing policies.

8. Scope

All community participants (contributors, paid or otherwise; sponsors; and other guests) must abide by this Code of Conduct in all forms of communications within the community such as venues, online and in-person as well as in all one-on-one communications pertaining to community business.

This code of conduct and its related procedures also applies to unacceptable behavior occurring outside the scope of community activities when such behavior has the potential to adversely affect the safety and well-being of community members.

9. Contact info

E-mail:

10. License and attribution

This Code of Conduct is distributed under a Creative Commons Attribution-ShareAlike License.

Portions of text derived from the Django Under The Hood.

Developer Resources

Welcome to the Developer Resources section! If you're a developer eager to contribute to OpenWISP, you've come to the right place. This section provides a wealth of information to help you get started, contribute effectively, and make the most out of your development experience with OpenWISP.

Contributing guidelines

We are glad and thankful that you want to contribute to OpenWISP.

Important

Please read these guidelines carefully, it will help to save precious time for everyone involved.

Table of Contents:

Introduce yourself

It won't hurt to join our main communication channel and introduce yourself, although to coordinate with one another on technical matters we use the development channel. Use these two channels share feedback, share your OpenWISP derivative work, ask questions or announce your intentions.

Look for open issues

Check out these two kanban boards:

- OpenWISP Contributor's Board: lists issues that are suited to newcomers.
- OpenWISP Priorities for next releases, lists issues that are more urgently needed by the community and is frequently used and reviewed by more seasoned contributors.

If there's anything you don't understand regarding the board or a specific github issue, don't hesitate to ask questions in our general chat.

You don't need to wait for the issue to be assigned to you. Just check if there is anyone else actively working on it (e.g.: an open pull request with recent activity). If nobody else is actively working on it, **just announce your intention to work on it by leaving a comment in the issue**.

Priorities for the next release

When we are close to releasing a new major version of OpenWISP, we will encourage all contributors to focus on the **To Do** column of the OpenWISP Priorities for next releases board and filter the issues according to their expertise:

- **Newcomer**: filter by Good first issue or Hacktoberfest.
- **Expert**: filter by Important.

Setup

Once you have chosen an issue to work on, read the documentation section of the module you want to contribute to, follow the setup instructions, each module has its own specific developer installation instructions which we highly advise to read carefully.

Important

For a complete list of the OpenWISP modules, refer to Architecture, Modules, Technologies.

How to commit your changes properly

Our main development branch is master, it's our central development branch.

You should open a pull request on github. The pull request will be merged only once the CI build completes successfully (automated tests, code coverage check, QA checks, etc.) and after project maintainers have reviewed and tested it.

You can run QA checks locally by running `./run-qa-checks` in the top level directory of the repository you're working on. Every OpenWISP module should have this script (if a module doesn't have it, please open an issue on github).

1. Branch naming guidelines

Create a new branch for your patch, use a self-descriptive name, e.g.:

```
git pull origin master
# if there's an issue your patch addresses
git checkout -b issues/48-issue-title-shortened

# if there is no issue for your branch, (we suggest creating one anyway)
# use a descriptive name
git checkout -b autoregistration
```

2. Commit message style guidelines

Please follow our commit message style conventions.

If the issue is present on Github, use following commit style:

```
[module/file/feature] Short description #<issue-number>

Long description here.
Fixes #<issue-number>
```

Here's a real world commit message example from one of our modules:

```
[admin] Fixed VPN context in preview #57

Fortunately it was just a frontend JS issue.
The preview instance was getting the UUID of the Device
object instead of the Config object, and that prevented
the system from finding the associated VPN and fill the
context VPN keys correctly.

Fixes #57
```

Moreover, keep in mind the following guidelines:

- commits should be descriptive in nature, the message should explain the nature of the change
- make sure to follow the code style used in the module you are contributing to
- before committing and pushing the changes, test the code both manually and automatically with the automated test suite if applicable
- after pushing your branch code, make a pull-request of that corresponding change of yours which should contain a descriptive message and mention the issue number as suggested in the example above
- make sure to send one pull request for each feature. Whenever changes are requested during reviews, please send new commits (do not amend previous commits), if multiple commits are present in a single pull request, they will be squashed in a single commit by the maintainers before merging
- in case of big features in which multiple related features/changes needs to be implemented, multiple commits (one commit per feature) in a single PR are acceptable.

3. Pull-Request guidelines

After pushing your changes to your fork, prepare a new Pull Request (from now on we will shorten it often to just *PR*):

- from your forked repository of the project select your branch and click "New Pull Request"
- check the changes tab and review the changes again to ensure everything is correct

- write a concise description of the PR, if an issue exists for
- after submitting your PR, check back again whether your PR has passed our required tests and style checks
- if the tests fail for some reason, try to fix them and if you get stuck seek our help on our communication channels
- if the tests pass, maintainers will review the PR and may ask you to improve details or changes, please be patient: creating a good quality open source project takes a bit of sweat and effort; ensure to follow up with this type of operations
- once everything is fine with us we'll merge your PR

4. Avoiding unnecessary changes

Keep your contribution focused and change the least amount of lines of code as possible needed to reach the goal you're working on.

Avoid changes unrelated to the feature/fix/change you're working on.

Avoid changes related to white-space (spaces, tabs, blank lines) by setting your editor as follows:

- always add a blank line at the end of the file
- clear empty lines containing only spaces or tabs
- show white space (this will help you to spot unnecessary white space)

Coding Style Conventions

1. Python code conventions

OpenWISP follows PEP 8 -- Style Guide for Python Code and several other style conventions which can be enforced by using the following tools:

- `openwisp-qa-format`: this command is shipped in openwisp-utils, a dependency used in every OpenWISP python module, it formats the Python code according to the OpenWISP style conventions, it's based on popular tools like: isort and black (**please do not run black directly** but always call `openwisp-qa-format`)
- `./run-qa-checks`: it's a script present in the top level directory of each OpenWISP module and performs all the QA checks that are specific to each module. It mainly calls the `openwisp-qa-check` command, which performs several common QA checks used across all OpenWISP modules to ensure consistency (including flake8), for more info consult the documentation of openwisp-qa-check.

Keep in mind that the QA checks defined in the `run-qa-checks` script are also executed in the CI builds, which will fail if any QA check fails.

To fix QA check failures, run `openwisp-qa-format` and apply manual fixes if needed until `./run-qa-checks` runs without errors.

Note

If you want to learn more about our usage of python and django, we suggest reading Useful Python & Django Tools for OpenWISP Development.

2. Javascript code conventions

- OpenWISP follows standard JavaScript coding style conventions that are generally accepted or the ones that are specified in .jshintrc files; find out more about JSHint here

- please follow this JavaScript Style Guide and Coding Conventions link for proper explanation and wonderful examples

3. OpenWrt related conventions

OpenWISP follows the standard OpenWrt coding style conventions of OpenWrt:

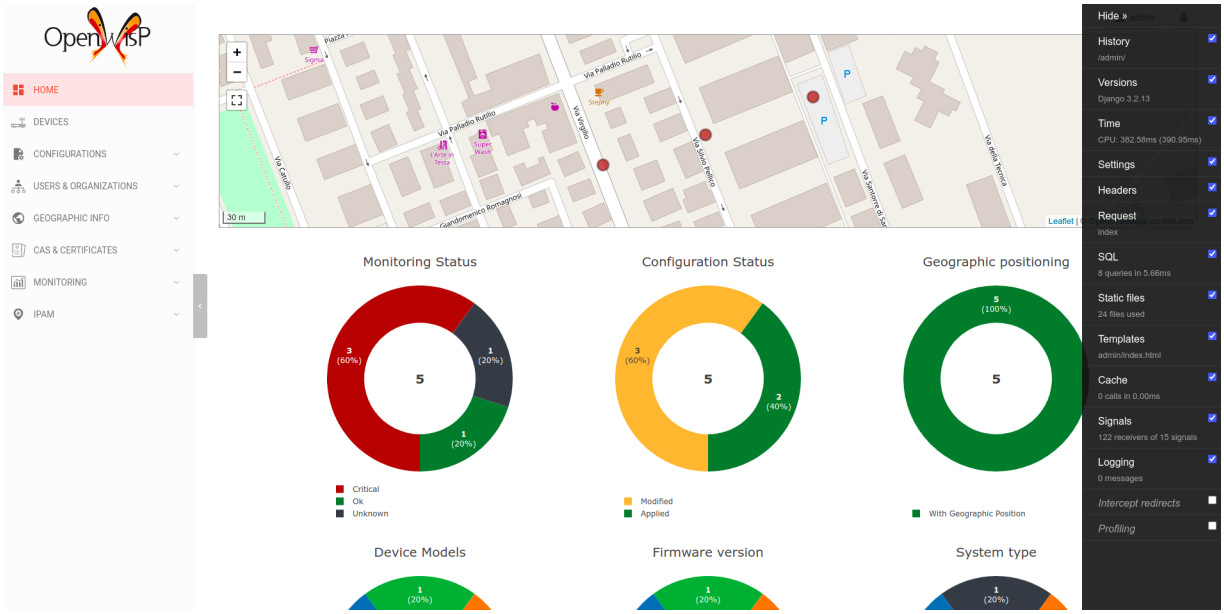
- Working with Patches
- Naming patches
- Adding new files.

Thank You

If you follow these guidelines closely your contribution will have a very positive impact on the OpenWISP project.

Thanks a lot for your patience.

Useful Python & Django Tools for OpenWISP Development



In this page we aim to help users and contributors who want to work on the internal code of OpenWISP in the following ways:

1. By explaining **why OpenWISP uses Python and Django** as its main technologies for the backend application
2. By introducing some Python tools and Django extensions which are **extremely useful during development and debugging**.

Table of Contents:

Why Python?

Note

The first version of OpenWISP was written in Ruby.

OpenWISP 2 was rewritten in Python because Ruby developers were becoming scarce, which led to stagnation. The widespread use of Python in the networking world also played a significant role in this decision.

Python is an interpreted, high-level programming language designed for general-purpose programming, emphasizing productivity, fast prototyping, and high readability.

Python is widely used today, with major organizations like Google, Mozilla, and Dropbox extensively employing it in their systems.

Here are the main reasons why OpenWISP is written in Python:

- It is widely used in the networking and configuration management world. Famous libraries such as networkx, ansible, salt, paramiko, and fabric are written in Python. This allows our users to work with a familiar programming language.
- Finding developers who know Python is not a hard task, which helps the community grow and contributes to the improvement of the OpenWISP software ecosystem over time.
- Python allows great flexibility and extensibility, making OpenWISP hackable and highly customizable. This aligns with our emphasis on software reusability, which is one of the core values of our project.

Resources for learning Python:

- LearnPython.org.
- SoloLearn (a detailed beginner course).

Why Django?

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.

In OpenWISP we chose Django mainly for these reasons:

- It has a rich ecosystem and pluggable apps that allow us to accomplish a lot very quickly.
- It has been battle-tested over many years by a large number of users and high-profile companies.
- Security vulnerabilities are usually privately disclosed to the developers and quickly fixed.
- Being popular, it's easy to find Python developers with experience in Django who can quickly start contributing to OpenWISP.
- Django projects are easily customizable by editing a `settings.py` file. This allows OpenWISP to design its modules so they can be imported into larger, more complex, and customized applications, enabling the creation of tailored network management solutions. **This makes OpenWISP similar to a framework**: users can use the default installation, but if they need a more tailored solution, they can use it as a base, avoiding the need to redevelop a lot of code from scratch.

Resources for learning Django:

- Official Basic Django Tutorial
- DjangoGirls Tutorial (excellent for absolute beginners!)

PS: If you are wondering why the second tutorial mentions the word "Girls," we suggest taking a look at djangogirls.org.

Why Django REST Framework?

Django REST framework is a powerful and flexible toolkit for building Web APIs, used and trusted by internationally recognized companies including Mozilla, Red Hat, Heroku, and Eventbrite.

Here are some reasons why OpenWISP uses Django REST framework:

- Simplicity, flexibility, quality, and extensive test coverage of the source code.
- Powerful serialization engine compatible with both ORM and non-ORM data sources.
- Clean, simple views for resources, using Django's class-based views.
- Efficient HTTP response handling and content type negotiation using HTTP Accept headers.
- Easy publishing of metadata along with querysets.

Resources for learning Django REST Framework:

- Django REST Framework Official Tutorial

Useful Development Tools

IPython and ipdb

IPython (Interactive Python) is a command shell for interactive computing in multiple programming languages, originally developed for Python. It offers introspection, rich media, shell syntax, tab completion, and history.

It provides:

- A powerful interactive shell with syntax highlighting
- A browser-based notebook interface with support for code, text, mathematical expressions, inline plots, and other media
- Support for interactive data visualization and use of GUI toolkits
- Flexible, embeddable interpreters to load into one's own projects
- Tools for parallel computing

More details, including installation and updates, can be found on the official website.

As for ipdb, it allows the use of the `ipython` shell when using the Python debugger (`pdb`).

Try adding this line in a Django project (or an OpenWISP module), for example in a `settings.py` file:

```
import ipdb

ipdb.set_trace()
```

Now load the Django development server and have fun while learning how to debug Python code!

Django Extensions

Django Extensions is a collection of extensions for the Django framework. These include management commands, additional database fields, admin extensions, and much more. We will focus on three of them for now: `shell_plus`, `runserver_plus`, and `show_urls`.

Django Extensions can be installed with:

```
pip install django-extensions
```

shell_plus: Django shell which automatically imports the project settings and the django models defined in the settings.

runserver_plus: the typical `runserver` with the Werkzeug debugger baked in.

`show_urls`: displays the registered URLs of a Django project.

Django Debug Toolbar

The Django Debug Toolbar is a configurable set of panels that display various debug information about the current HTTP request/response and, when clicked, provide more details about the panel's content.

It can be installed with:

```
pip install django-debug-toolbar
```

More information can be found in the django-debug-toolbar documentation.

Using these Tools in OpenWISP

These tools can be added to an OpenWISP development environment to significantly improve the efficiency and experience of development. Here's a guide on how to use them in OpenWISP Controller.

In the `tests/` folder, `local_settings.example.py` should be copied and renamed to `local_settings.py` for customization. This technique can be used in other OpenWISP development environments too.

```
cd tests/
cp local_settings_example.py local_settings.py
```

Follow the installation steps for OpenWISP Controller. Run the command `pipenv install --dev`, then run `pipenv run ./manage.py migrate` and `pipenv run ./manage.py createsuperuser`. Ensure `SPATIALITE_LIBRARY_PATH` is specified in the `local_settings.py` file.

To start the development server with more debugging information, run:

```
python manage.py runserver_plus
```

For an interactive shell, use `ipython` alongside `shell_plus` by running:

```
./manage.py shell_plus --ipython
```

To debug the code, use `ipdb`. For example:

```
ipdb mymodule.py
```

This command will provide a list of lines where errors have been found or lines that can be further optimized.

To use `django-debug-toolbar` for displaying information about processes occurring on the website, some configuration is required. Add the following lines to your `local_settings.py`:

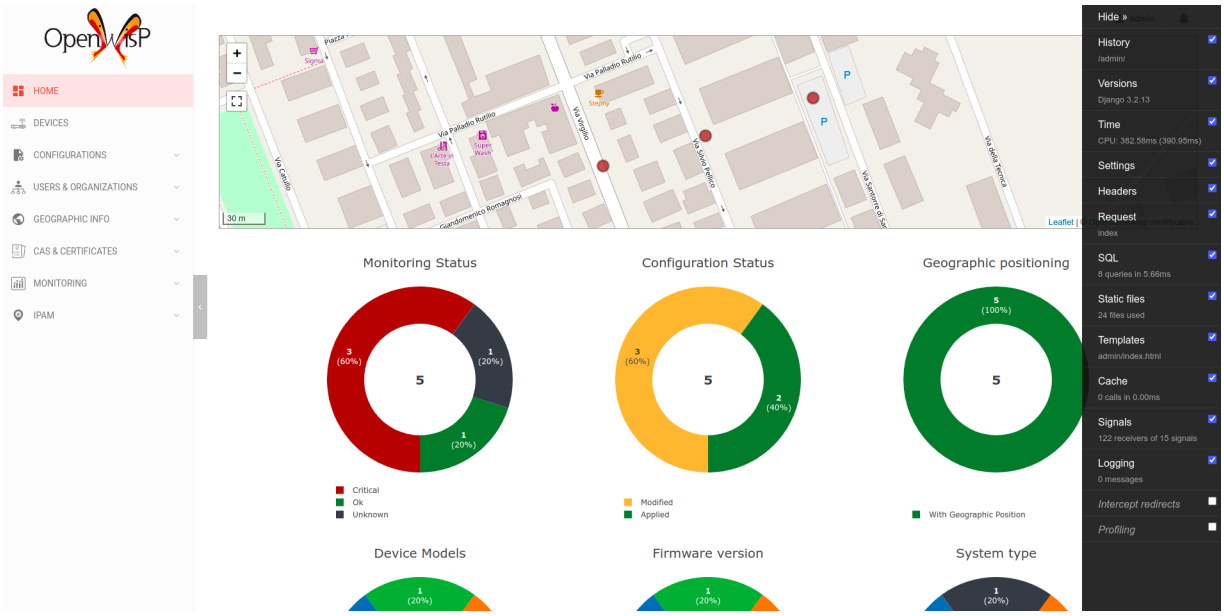
```
from django.conf import settings

settings.INSTALLED_APPS += ["debug_toolbar", "django_extensions"]
settings.MIDDLEWARE += ["debug_toolbar.middleware.DebugToolbarMiddleware"]
INTERNAL_IPS = ["127.0.0.1"]
```

This ensures that the Django Debug Toolbar is displayed. Note that *django_extensions* is already included in `settings.py`.

Finally, add the Debug Toolbar's URL to the URLconf of `openwisp-controller` as shown in the installation tutorial, though this should already be present in the last lines of `urls.py`:

```
from django.conf import settings

if settings.DEBUG and "debug_toolbar" in settings.INSTALLED_APPS:
    import debug_toolbar

    urlpatterns.append(url(r"^__debug__/", include(debug_toolbar.urls)))
```

When you open `http://127.0.0.1:8000` in the browser and log in with the credentials created earlier, you should see something like this:



Now that you know the basics, you can experiment and apply these techniques to other OpenWISP modules.

Google Summer of Code

Google Summer of Code

Note

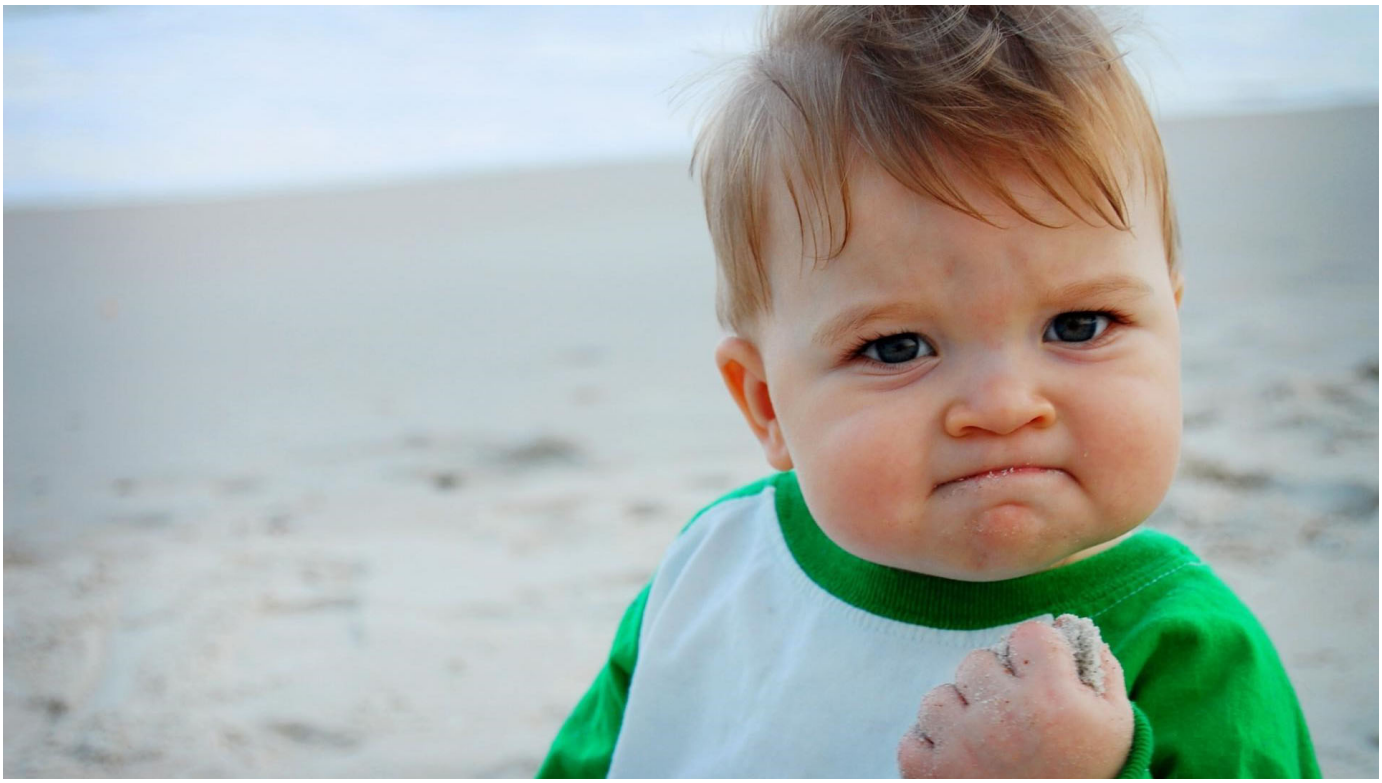
OpenWISP is a mentoring organization for the Google Summer of Code 2024.

If you are reading this page you are probably considering OpenWISP as a possible mentoring organization for the Google Summer of Code, that's great!

If you are looking for a **friendly community** where **your contribution will have a very tangible positive effect from the first day of your participation** and where **you can grow your tech skills at 360°**, then **CONGRATULATIONS!** OpenWISP is the right organization for you.

Table of Contents:

How to run a successful Google Summer of Code



First of all: PLEASE, PLEASE, read all the information contained in this page (including links!) because this will save everybody involved a lot of time. We would rather spend our time coding than repeating the same stuff over and over.

Have you read the Student manual yet? If not, please do **because it's a MUST if you want to be successful**!

Communication with the rest of the community is vital for a successful Google Summer of Code, please join our communication channels, join our mailing list (we have a dedicated mailing list for GSoC, receive all emails please, and filter them in your mail box so they are moved to an "OpenWISP" folder), present yourself in our general chat, tell us who you are, what your values are, what is attracting to OpenWISP and don't be cold like a robot! Stay human :-).

Traits we look for in applicants

We participate in GSoC because we believe it's a great opportunity for us to give back to Open Source by helping newcomers to get trained and thrive in this industry, but we also do it because we want to grow the pool of maintainers of our project so we can help a greater number of users to use OpenWISP successfully.

Contributors who also become maintainers and start working professionally with OpenWISP are rare, but over time we found out the traits that are good leading indicators for contributors who are likely to become core members of our project, **here are the traits we look for in GSoC applicants which give a higher chance of getting selected**:

- **Genuinely interested in networking**: we look for people who are genuinely attracted in the topics we cover because we believe they are the ones who most likely will benefit from a long term contribution to our project.
- **Participate actively**: they become active participants of the community, not just by submitting pull requests, but also by helping new users or reviewing patches of other less experienced contributors.
- **Put effort in understanding**: they put effort in understanding the problem they need to solve and the outcomes that is expected from them, which means actively researching the problem, expand the project idea with more details, create a prototype, note down a list of questions regarding points that are not clear.
- **Value the time of mentors**: they read carefully the description of issues and put effort in understanding what they have to do, when something is not clear they do not hesitate to explain the problem carefully via email or on github.

- **Parallelize tasks when waiting for a reply**: while they wait for mentors to review or answer their questions, they start tackling other issues for which they have enough information to get started, in order to avoid staying idle.
- **Value quality**: they ensure their work is of the highest quality and doesn't break existing features of the system thanks to thorough testing before flagging a patch as ready to be merged.

How to become an OpenWISP star

Here's a few quick tricks you can use to become a star in our community:

- read the founding values and goals of OpenWISP, are you on our side?
- study and follow closely the contributing guidelines
- be patient in the interaction with your mentors, we are all volunteers, we are taking our time to mentor you from our free time which we usually spend family and loved ones
- we know our documentation is incomplete and fragmented, we are working hard to fix it; if you find a passage that is not clear or you have an idea about how to improve it, **please let us know!**
- start using OpenWISP 2: install it, run it, play with it; understand its structure
- start contributing (e.g.: fix easy bugs, write documentation, improve tests); look for open issues in our most used repositories on github.com/openwisp (ask in our support channels before starting to code please! we have many legacy repositories that are not under active development anymore)
- if we ask you to open an issue in one of our github repository, please take at least 5 minutes of time to write a proper bug report
- watch the OpenWISP 2 presentation at the recent OpenWrt Summit 2017 and read the slides of this more technical OpenWISP 2 talk
- try using OpenWISP in real use case scenarios (find out if there's a free wifi community near your area), spend time reading its code, ask questions

- try to participate in the community, if a fellow member is in need of help and you know how to help him, please do so, we will reward you

Time to start hacking

If you are not familiar with the following concepts yet, take the time to read these resources, it will help you to speed up your raise to the top!

Programming languages and frameworks:

- Python (book)
- Django (official documentation)
- Lua (video tutorial)
- **Shell**
  (video tutorial)
- Javascript (tutorial)

Networking concepts:

- Introduction to networking terminology

Configuration management:

- Introduction to configuration management
- Writing Ansible playbooks
- Creating Ansible roles from scratch

Project ideas

- Project Ideas 2024

Application Template

Please make sure to include the information requested below in your GSoC application.

1. Your Details

- Full name
- Date of birth
- Country/Region
- Email
- GitHub/GitLab profile
- Phone number
- What's your availability in UTC times?

2. Tell Us About Yourself

- What is your background?
- Have you ever contributed to open-source software projects? If yes, how?
- Please list the links to your OpenWISP contributions and/or notable contributions to other Open Source & Free Software projects.
- Do you have any experience with OpenWrt?
- Do you have a router at home on which you can flash OpenWrt to test OpenWISP?
- What's your motivation for working on OpenWISP during the Google Summer of Code?

3. Your GSoC Project

- Project Title
- Possible Mentor
- Measurable Outcomes
- Project Details:

  How are you going to implement the solution?

  What technologies do you want to use?

  Make sure to include code samples.

  Linking to a repository containing a prototype and an explicative README, which includes screenshots or GIF recordings demonstrating how the prototype works, is a great way to demonstrate your technical understanding and boost your chances.
- Project Schedule: Can you provide a rough estimate? When can you begin to work?
- Availability: How many hours per week can you spend working on this? What other obligations do you have this summer?

4. After GSoC

- Are you interested in continuing to collaborate with OpenWISP after the GSoC ends?
- Will you help maintain your implementation for a while?
- If we get new business opportunities to build new features, are you interested in occasional freelance paid work?

  It's not enough to reply "YES," please explain what your motivation is (e.g., gaining experience, tech challenges).

GSoC Project Ideas 2024

Tip

Do you want to apply with us?

We have a page that describes how to increase your chances of success. **Please read it carefully.**

Read our Google Summer of Code guidelines.

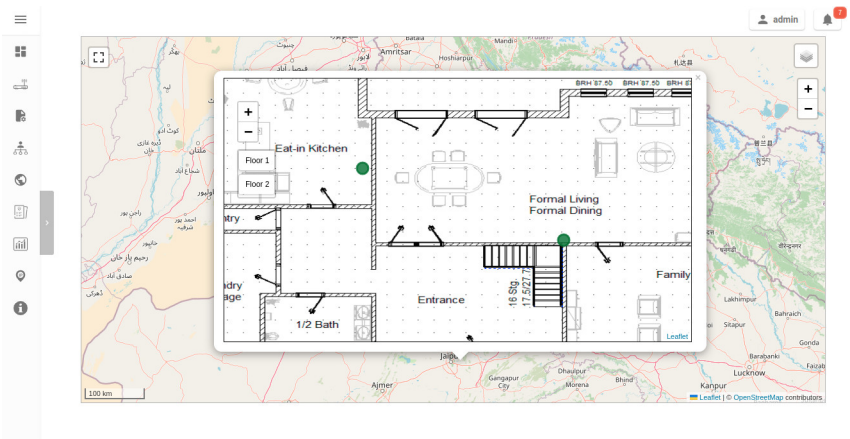
Table of Contents:

General suggestions and warnings

- **Project ideas describe the goals we want to achieve but may miss details that have to be defined during the project**: we expect applicants to do their own research, propose solutions and be ready to deal with uncertainty and solve challenges that will come up during the project
- **Code and prototypes are preferred over detailed documents and unreliable estimates**: rather than using your time to write a very long application document, we suggest to invest in writing a prototype (which means the code may be thrown out entirely) which will help you understand the challenges of the project you want to work on; your application should refer to the prototype or other Github contributions you made to OpenWISP that show you have the capability to succeed in the project idea you are applying for.
- **Applicants who have either shown to have or have shown to be fast learners for the required hard and soft skills by contributing to OpenWISP have a lot more chances of being accepted**: in order to get started contributing refer to the OpenWISP Contributing Guidelines
- **Get trained in the projects you want to apply for**: once applicants have completed some basic training by contributing to OpenWISP we highly suggest to start working on some aspects of the project they are interested in applying: all projects listed this year are improvements of existing modules so these modules already have a list of open issues which can be solved as part of your advanced training. It will also be possible to complete some of the tasks listed in the project idea right now before GSoC starts. We will list some easy tasks in the project idea for this purpose.

Project Ideas

Improve OpenWISP General Map: Indoor, Mobile, Linkable URLs



Important

Languages and technologies used: **Python**, **Django**, **JavaScript**, **Leaflet**, **netjsongraph.js**.

Mentors: *Federico Capoano*, *Gagan Deep*.

Project size: 350 hours.

Difficulty rate: medium.

This GSoC project aims to enhance the user experience of the general map within OpenWISP, a feature introduced in the last stable version.

By developing a dedicated map page, facilitating precise device tracking, and seamlessly integrating indoor floor plans, the project endeavors to significantly improve the usability and functionality of the mapping interface, ensuring a more intuitive and effective user experience.

Prerequisites to work on this project

Applicants must demonstrate a solid understanding of Python, Django, Leaflet library, JavaScript, OpenWISP Controller, OpenWISP Monitoring. and netjsongraph.js.

Expected outcomes

- Add a dedicated map page: Introduce a dedicated page to display all network devices on a map. This view will offer the same functionality as the map in the dashboard, with the sole difference being that this page focuses on rendering only the map. It will be used for linking specific points on the map within the rest of the OpenWISP UI.
- Allow tracking mobile coordinates: OpenWISP Controller provides a way for devices to update their co-ordinates, we want to make the map able to update in real time as devices send their updated coordinates.
- Integrate indoor floor plan functionality in the map: The netjsongraph.js library allows to render indoor maps, we want to make use of this feature to display the indoor location of devices and we want this feature to be accessible from the general map. When zooming in on a device which is flagged as indoor and has floor plans saved in the database, users should see an option to switch to the indoor view. This view would show the floor plan of the indoor location and any device located on the floor plan, it shall also account for the following use cases:
    - An indoor location can have multiple floors. The view should be allow users to navigate between different floors.
    - There can be multiple devices on the same floor. The view should show all the devices on a floor. This will require developing an API endpoint which returns location of devices on the floor plan
- Make map actions bookmarkable: Update the URL when clicking on a node/link to view its details. Visiting this URL should automatically focus on the specified node/link and display its details, if available. This functionality should also accommodate geo-maps using coordinates. Clicking on a node/link to view it's details should update the the page's URL. When visiting this URL, the map should automatically focus the said node/link. It shall also open the node's/link's details if they are available. This should work on geographic maps, indoor maps and logical maps.
- Add button to general map from device detail: Implement a button on the device detail page to allow users to navigate from the device detail to the general map and inspect the device's location on the map. The map should focus on the specific device in question. This feature should also be available for indoor maps, providing a button in the floor plan section to open the general map with the indoor view focused.

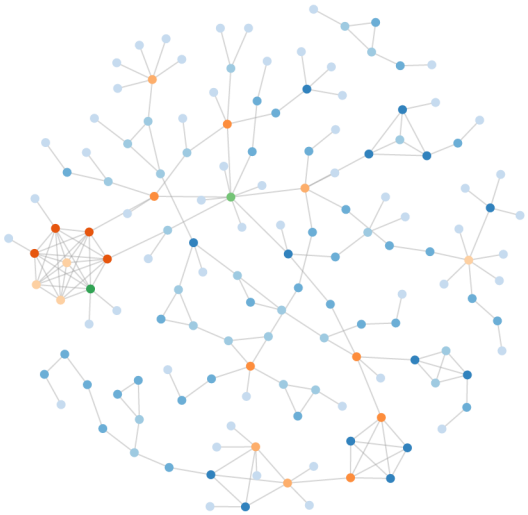
Throughout the code changes, it is imperative to maintain stable test coverage and keep the README documentation up to date.

Note

The "expected outcomes" mentioned above include links to corresponding GitHub issues. However, these issues may not cover all aspects of the project and are primarily intended to gather technical details. Applicants are encouraged to seek clarification, propose solutions and open more issues if needed.

Applicants are also expected to deepen their understanding of the UI changes required by preparing *wireframes* or *mockups*, which must be included in their application. Demonstrating a willingness and enthusiasm to learn about UI/UX development is crucial for the success of this project.

Improve netjsongraph.js resiliency and visualization



Important

Languages and technologies used: **Javascript**, **NodeJS**, **HTML**, **CSS**

Mentors: *Federico Capoano* (more mentors TBA).

Project size: 175 hours.

Difficulty rate: medium.

The goal of this project is to improve the latest version of the netjsongraph.js visualization library to improve resiliency and functionality.

Prerequisites to work on this project

The contributor should have a proven track record and experience with Javascript, React JS, NodeJS, HTML and CSS.

Familiarity with OpenWISP Network Topology and OpenWISP Monitoring is a plus.

Expected outcomes

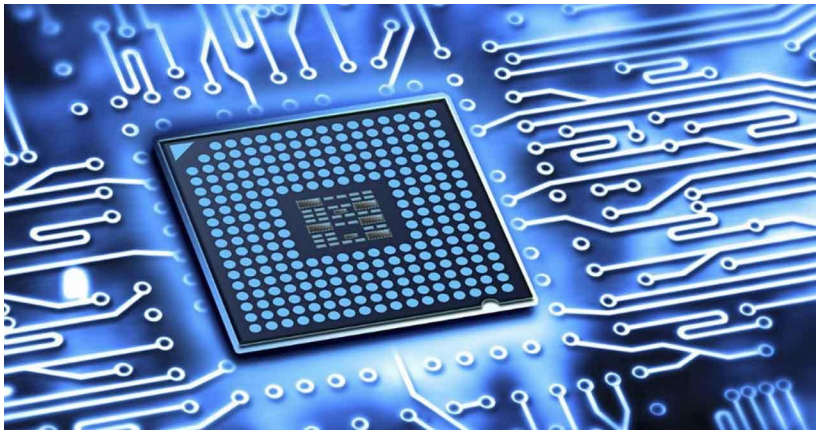
The applicant must open pull requests for the following issues which must be merged by the final closing date of the program:

- Allow showing node names on geo map on high zoom levels: The node names should be shown by default on high zoom levels.
- Map should respect zoom levels of tile providers: We shall limit the map zoom levels based on the tile provider. We can make the supported zoom levels configurable and provide sensible defaults.
- Prevent overlapping of clusters: The clusters of different categories with the same location are overlapped. Instead, we should find a way to prevent this behavior.
- Add resiliency for invalid data: The library should not crash if invalid data is provided, e.g. different nodes with same ID. Instead, it should handle such cases gracefully and log the errors.
- Display additional data (connected clients) on nodes: It shall be possible to show connected clients on nodes. This feature needs to be flexible, such that it can be used to show different kinds of data.
- Show node labels only after hitting a certain zoom level: At present, the node labels become cluttered and unreadable when zoomed out excessively. To enhance readability, we need to add a feature in the library that allows configuring the zoom level at which node labels should start appearing.

Each issue contains the details which the applicant needs to know in order to complete the project successfully.

At each step of code changing the test coverage must be maintained stable and the documentation in the README must be kept up to date.

Improve UX and Flexibility of the Firmware Upgrader Module



Important

Languages and technologies used: **Python**, **Django**, **OpenWrt**.

Mentors: *Federico Capoano* (more mentors TBA).

Project size: 175 hours.

Difficulty rate: easy/medium.

The goal of this project is to improve the Firmware Upgrader module to make its mass upgrade operation feature more versatile and to improve the user experience by showing progress in real time.

Prerequisites to work on this project

The applicant must demonstrate good understanding of Python, Django, Javascript and OpenWISP Controller.

They must demonstrate also a basic understanding of OpenWISP Firmware Upgrader, OpenWrt and UI development.

Prior experience with OpenWrt is not extremely required but welcome.

Expected outcomes

The applicant must open pull-requests for the following issues which must be merged by the final closing date of the program:

- [feature] REST API is missing endpoints for DeviceFirmware
- [feature:UI] Show upgrade progress in real time in the UI
- [feature] Allow to perform mass upgrade of devices by their group
- [feature] Allow to perform mass upgrade of devices by their location

Each issue contains the details which the applicant needs to know in order to complete the project successfully.

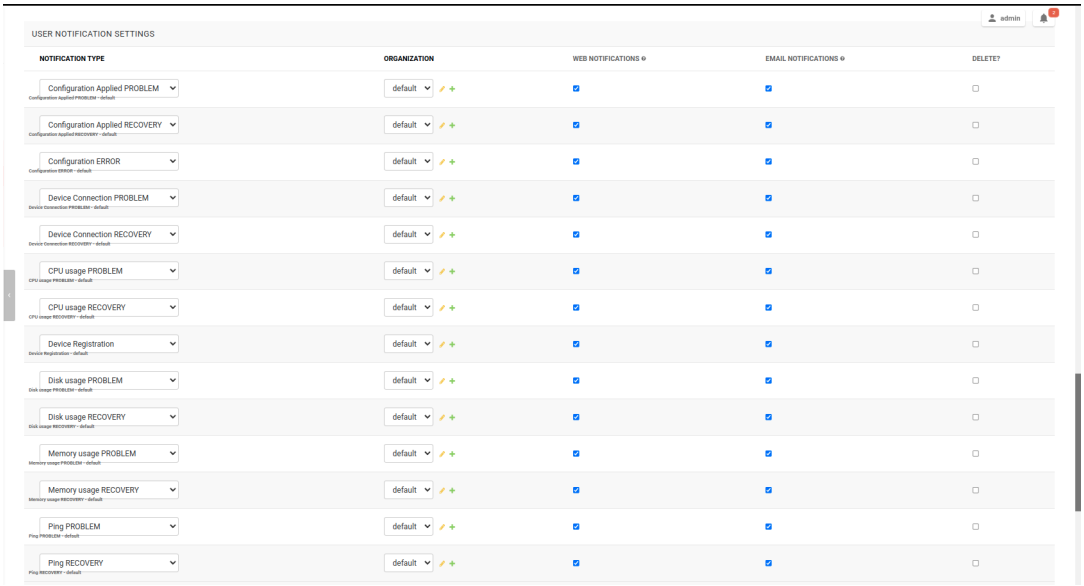
At each step of code changing the test coverage must be maintained stable and the documentation in the README must be kept up to date.

Training Issues

The applicant may warm up in the application phase by working on the following issues:

- [bug] FileNotFoundError when trying to delete an image which links a non existing file
- [change] Improve endpoints to download firmware images
- [feature] Allow management of UpgradeOperation objects in the admin

Improve UX of the Notifications Module



Important

Languages and technologies used: **Python**, **Django**, **JavaScript**, **HTML**, **CSS**

Mentors: *Gagan Deep* (pandafy) (more mentors TBA).

Project size: 175 hours.

Difficulty rate: medium.

The goal of this project is to improve the user experience for managing of the notification module in regards to managing notification preferences and batching of email notifications.

Prerequisites to work on this project

The applicant must demonstrate good understanding of OpenWISP Notifications, it's integration in OpenWISP Controller and OpenWISP Monitoring.

The applicant must demonstrate at least basic UI/UX development skills and eagerness to learn more about this subject.

Expected outcomes

The applicant must open pull-requests for the following issues which must be merged by the final closing date of the program:

- [feature] Batch email notifications to prevent email flooding: this issue has priority because when this happens it causes most users to want to disable email notifications.
- [feature] Allow to disable notifications for all organizations or keep everything disabled except notifications for specific organizations.
- [feature] Add REST API to manage notification preferences of other users.
- [feature] Add a dedicated view for managing notification preferences.
- [feature] Add link to manage notification preferences to email notifications.

Each issue contains the details which the applicant needs to know in order to complete the project successfully.

At each step of code changing the test coverage must be maintained stable and the documentation in the README must be kept up to date.

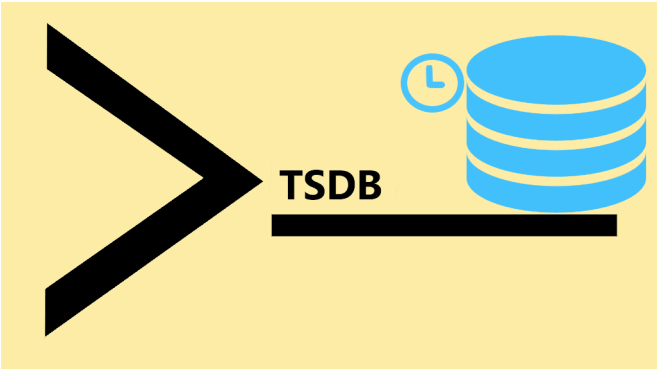
Applicants are expected to gain more understanding of the UI changes requested with the help of *wireframes* which must be included in the application; experience in wireframing is considered an important factor, alternatively mentors will guide applicants in learning more about the subject. Willingness and eagerness to learn more about this subject, as well as UI/UX development are paramount.

Training Issues

The applicant may warm up in the application phase by working on the following issues:

- [feature] Add dedicated notification type for internal errors
- [change] Allow relative paths

Add more timeseries database clients to OpenWISP Monitoring



Important

Languages and technologies used: **Python**, **Django**, **InfluxDB**, **Elasticsearch**.

Mentors: *Federico Capoano*, *Gagan Deep* (more mentors TBA).

Project size: 175 hours.

Difficulty rate: medium.

The goal of this project is to add more Time Series DB options to OpenWISP while keeping good maintainability.

Prerequisites to work on this project

The applicant must demonstrate good understanding of OpenWISP Monitoring, and demonstrate basic knowledge of NetJSON format, **InfluxDB** and **Elasticsearch**.

Expected outcomes

- Complete the support to Elasticsearch. Support to Elasticsearch was added in 2020 but was not completed.
  - The old pull request has to be updated on the current code base
  - The merge conflicts have to be resolved
  - All the tests must pass, new tests for new charts and metrics added to *InfluxDB* must be added (see [feature] Chart mobile (LTE/5G/UMTS/GSM) signal strength #270)
  - The usage shall be documented, we must make sure there's at least one dedicated CI build for **Elasticsearch**
  - We must allow to install and use **Elasticsearch** instead of **InfluxDB** from ansible-openwisp2 and docker-openwisp
  - The requests to Elasticsearch shall be optimized as described in [timeseries] Optimize elasticsearch #168.
- Add support for InfluxDB 2.0 as a new timeseries backend, this way we can support both `InfluxDB <= 1.8` and `InfluxDB >= 2.0`.
  - All the automated tests for **InfluxDB 1.8** must be replicated and must pass
  - The usage and setup shall be documented
  - We must make sure there's at least one dedicated CI build for Elasticsearch
  - We must allow choosing between **InfluxDB 1.8** and **InfluxDB 2.0** from ansible-openwisp2 and docker-openwisp.