nodewatcher Documentation

Release 3.0b0

wlan slovenija

May 07, 2018

Contents

nodewatcher is one of the projects of wlan slovenija open wireless network. Its main goal is the development of an open source network planning, deployment, monitoring and maintanance platform with emphasis on community.

CHAPTER 1

Contents

Setting Up the Development Environment

In order to make nodewatcher development as simple as possible, a Docker development environment can be built from the provided Dockerfile. To help with preparing the environment and running common tasks, Docker Compose configuration is also provided. This section describes how to use both to work on nodewatcher. It is assumed that Docker and *Docker Compose* are already installed, so in order to install them, please follow the guides on their respective pages.

Running the Development Instance

In order to ensure that you have the latest versions of all dependent Docker images, you have to first run the following command in the top-level nodewatcher directory:

```
$ docker-compose pull
```

Then, to build the development instance, run:

```
$ docker-compose build
```

You should re-run these two commands when performing `git pull` if you think that the build dependencies or development environment configuration might have changed to ensure that your Docker images are synced. Finally, to run the development server simply execute the following command:

```
$ docker-compose up
```

Note: Default container configuration stores the database under `/tmp/nodewatcher-db`, so it will be removed on host machine restarts. You may change this location by editing `docker-compose.yml`.

The following containers will be created and started when you run the above commands:

- `db` contains the PostgreSQL 9.5 database server with installed extension PostGIS 2.1.
- `influxdb` contains the InfluxDB time-series database server.
- `redis` contains the Redis server.
- `builder*` contain firmware builders that you can use for development (see *Running a Firmware Builder*).
- `generator` contains the Celery workers for the firmware image generator. These workers connect to the `builder` via SSH in order to build firmware images.
- `monitorq` contains the Celery workers for handling monitoring push requests.

- `web` contains the nodewatcher frontend (Django development server), running on port `8000` by default.

Initializing the Database

Note: This and all further sections assume that the development environment has been started via `docker-compose up` and is running (in the background or in another terminal). If the environment is not running, some of the following commands will fail.

Starting the `db` container as above will create the database for you. If you need to reinitalize the database at any later time, you need to stop the application (by running `docker-compose stop`) and then remove the database directory, which is `/tmp/nodewatcher-db` by default. Restarting the application will then re-create the database.

Then, to populate the database with nodewatcher schema call `migrate`:

```
$ docker-compose run web python manage.py migrate
```

This will initialize the database schema.

Note: When running any Django management command, do not forget to run it via the container as otherwise the settings will be wrong. You can do it by prefixing commands with `docker-compose run web` like this:

```
$ docker-compose run web python manage.py <command...>
```

Compiling Stylesheets

For default nodewatcher stylesheets we are using Compass, which means that SCSS files have to be compiled into CSS files before CSS files can be served to users. Compass is built on top of Sass, which is an an extension of CSS3 and adds nested rules, variables, mixins, selector inheritance, and more. Sass generates well formatted CSS and makes our stylesheets easier to organize and maintain. It also makes stylesheets easier to extend and reuse so it is easier to build on default nodewatcher stylesheets and add small changes you might want to add for your installation.

To compile SCSS files into CSS files run:

```
$ docker-compose run web python manage.py collectstatic -l
```

Initial Setup

From Scratch

If you are installing the nodewatcher from scratch, you should probably now create a Django admin account. You can do this by opening nodewatcher initial setup page at http://localhost:8000/setup/, and follow instructions there.

Afterwards, you can login with this account into the nodewatcher at http://localhost:8000/account/login/ or into the nodewatcher's admin interface at http://localhost:8000/admin/.

Note: Depending on your Docker installation the web interface might not be available at `localhost` but at some other address.

From nodewatcher v2 Dump

If you have a JSON data export from nodewatcher version 2 available and would like to migrate to version 3, the procedure is as follows (after you have already performed the database initialization above). Let us assume that the dump is stored in a file called `dump.json` (note that the dump file must be located inside the top-level directory as commands are executed inside the container which only sees what is under the toplevel nodewatcher directory). The dump can be imported by running:

```
$ docker-compose run web python manage.py import_nw2 dump.json
```

Now the database is ready for use with nodewatcher 3.

Running a Firmware Builder

In order to develop firmware generator related functions, a firmware builder is provided with the development installation. In order to use it, you have to configure it via the nodewatcher admin interface. First, you have to create a default build channel and add a builder (or multiple builders if you want support for multiple architectures).

The following information should be used when adding a builder:

- **Host:**
  - builderar71xx
  - builderlantiq
  - builderar71xx_lede
- **Private key:**

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAoaOhSCNIim6VPg6SakvQEbZ+I2l2QLnPOkJGgnNBQimmkIdk
KH4M07ImzyApLMl38USTOa5RMMAH+kcHhq7ajOPaWRr5oUtH4aAqJhjQtqgDQ5AD
5bwIbNiT6f4xXh+8A1VEK/g9TaHOHWjm3XQu660bTWtHIfZwH2AkyXMUhaevVXFI
o/SF+zuutMOAi9ny/Mmvz+N39iGVanBXnz3mOf08nXhPBjGUKOS/u0SjEfa4WeCW
PQkG0IApIZfSENjnOOnUw6FLcXueehFqd4KgUb3eAl3DJQ0t43dTr1uRxySyIEOu
rrkvAvSOVW83KcaVfqnzsRHllxkYXdFfR+v9pwIDAQABAoIBAQCasF0GC3Y9vQyo
wgtPHNS4TtyyiRg5Y1k1mP2flCts5S1ZfajVs6QU6JaJfG7LRNe7lvQKrI9Yxz4b
P+Ss+SOA2AI7ajxiJwhYng4YPYFofKv6+ZUxQ90QRchwk+qs+FPXIL/IEJ6ib+ow
bRcb8jeBJj5Nvg/qKc0tybYK8E5AhS7FF6JfCtRff6IWch0vFDHoml7J4VS1dFYt
N/HcXPMM+Semf50LzyOvF1Yc/BWFpzmKG14qsGgJ/GBEw5UfI/oJKVBG95T+Lvk3
1zeDQGMYiOSBbaem/u5rR3erkRiGroYN6qbAWSKd9ZNtXyyRlyBSO/iJkNYsFMeq
hnaw8DfZAoGBANYFtHjvl1LtTVjpS/oa5b1ik/MkcE/qiAdve8zzYrlQclDkhFeT
Eqq0geSQrWJ28+xfxVndRjO1DykJ8ye45myQTXqQS592qFs21WMOJxWw+phT+CQ1
VMV0mlOpT/n1FisoTFQ4cv66zT3IY8ZH6PtNt3L0U4UzDbTJi+JBoXtlAoGBAMFX
tib24wIObtpoqRD0+ZOtnPg9t4wE6vteFkGVSxXy7w32DxuQFW6lzJI9z4yVU37I
VaTQ+hFECUmXRkGWgLziNMcIpQ6Q5KD0ZhfurrzCfF5tYQIZPbbpN3qy6xs/frnb
gG0hts+aNQga6Oj3f/fxNuacNPioq5am+BtOnXIbAoGAD9usW6mlFMfwiz3+GzIT
A81iGQoCKGnAWoywJ6eBESoczlGgXLzRDUUCuuTddAZMXJ9cCCSVJw+rZ+cM1Uym
BjVLCGHYuKAaKWgOt6A81Saf6tmN8WDiPx88sCZDfsniMqBxx2vHWYiN3J4UhoSd
hsFjbmkcJyp5QYQNkV47kOECgYEAnou8tWsTcKZBRR06NsuMtgtSg0ao80s9HnBj
M9inQBJ88ifq76FR0fBoNyw0vIXfeEHz6TntNqdiLlS8qiAu5bVhri1qnO04bry7
07hI1kVuE0kCmeP09b99XULHBQsmcmaLg/J3pPpBrqnSgOgkqj/F04oY7ifyvZGi
N1JaTi0CgYEAsH1m5atSGjScUMiVTiWjnYB2E00cBB6a84UfS359+LvkJdDHRptt
IjAnJaI31jpR2GSIQ9ck5SVNRKn8TO7hGMncSq6/CCJTwdAI9pzED4typVs341Wo
BZ9HO5E5TUQTXTKkKR4kPT2wyfsjCBEJl76RIt7WyJnEbj1fIcn+OZo=
-----END RSA PRIVATE KEY-----
```

Warning: This public/private key pair should only be used for development. For production deployments you should generate new key pairs and configure them appropriately (the public key can be configured by setting the `BUILDER_PUBLIC_KEY` environmental variable on the builder Docker container).
In order to generate a new RSA key pair, you may use:

```
$ ssh-keygen -f builder.key -C "builder@host"
```

This will generate a `builder.key` (private key) and `builder.key.pub` (public key).

Running the Monitoring System

In order to enable data collection from nodes, the monitoring system needs to be running. It is important that the nodewatcher instance is able to connect to the nodes directly by their IP addresses. This can usually be achieved by establishing a VPN tunnel to some server that is connected to the mesh network.

Then, there are two configuration options that need to be set in `settings.py`:

- `OLSRD_MONITOR_HOST` should point to an IP address where an `olsrd` instance is responding to HTTP requests about the routing state using the `txtinfo` plugin. In the default configuration, this will be used by the `modules.routing.olsr` module to enumerate visible nodes and obtain topology information.
- `MEASUREMENT_SOURCE_NODE` should be set to an UUID of a node that is performing the RTT measurements (this means that such a node must first be created using nodewatcher). This option is planned to be removed from `settings.py` and moved into the administration interface.

After the above settings are configured, one may run the monitoring system by issuing:

```
$ docker-compose run web python manage.py monitord
```

There are some additional options which might be useful during development:

- `--run=<run>` will only execute one run instead of all runs configured using `MONITOR_RUNS` setting.
- `--cycles=<cycles>` will only perform a fixed amount of cycles before terminating. By default, the monitor process will run indefinitely.
- `--process-only-node=<node-uuid>` may be used to only perform monitoring processing on a single node, identified by its UUID.

Note: The monitoring system may use a lot of CPU and memory resources when there are a lot of nodes to process.

Registry ORM Extensions

The concept of modularity brings new challenges to the *nodewatcher* platform. Foremost, a question poses itself: how to enable modules to add their own configuration options to registered nodes. The platform has to provide interfaces for:

- Persistent storage of configuration data, where schema is defined by module developer.
- Ability to automatically generate forms from the schema, so that the developer does not need to define the forms manually.
- Validation of all data at entry time in broader context of node configuration: one configuration value could have dependencies on other modules which require suitable validation at value change. Furthermore, to satisfy the

needs of the firmware generator, we wanted to assure that user gets feedback about possible invalid configuration for particular hardware immediately, so that she can correct the error directly, and not just after firmware generator fails at some later stage.

- Specification of context-sensitive default configuration values. For example: *node in project X has in the case that hardware supports multiple SSID on interface of ad-hoc type by default SSID "mesh.wlan-si.net", otherwise, in the case hardware supports only one SSID, the default should be "open.wlan-si.net"*.

To cover all the requirements listed above we developed a component named registry. It is a thin layer above existing Django ORM which allows defining schemata, automatic generation of corresponding configuration forms, setup of default configuration and validation of provided schema. Furthermore, hierarchical relations between elements are supported, too (eg. one radio can have multiple interfaces which can in turn have multiple addresses) which is a challenge especially when generating suitable user interface and rules evaluation.

Extending and adding functionalities is possible at multiple points so that, for example, developer can change how forms behave (which can dynamically adapt to values of other fields). It is possible at successful validation of configuration to execute various actions (eg. automatic IP addresses allocation based on selected requests, or validation configuration for particular hardware in the case of the firmware generation).

nodewatcher 3.0 uses this component for storing logs about nodes and for extracting configuration for all other modules (firmware generator, network monitoring, etc.).

Registry Items

Registry items are standard Django models that TODO.

Registration Points in nodewatcher

Core nodewatcher platform currently provides the following two registration points:

- `node.config` attached to the `Node` model. This registration point holds all per-node configuration models that the modules provide (see *Node Configuration Schema*).
- `node.monitoring` attached to the `Node` model. This registration point holds all per-node monitoring models that the modules provide (see *Node Monitoring Schema*).

Queryset Extensions

Registration points extend the object manager of the class their are attached to. Additional methods are provided so that you can directly query the registry hierarchy. Since multiple registration points can be attached to the same model a registration point must first be selected before making any queries. This can be done by using the `regpoint` method as follows:

```
Node.objects.regpoint('config')
```

Since instances of `Node` themselves don't have any fields besides the UUID you have to join them with registry models in order to obtain the required data. In order to make this easier, a `registry_fields` method is provided on the query set:

```
Node.objects.regpoint('config').registry_fields(
    name='core.general__name',
    type='core.type__type',
    router_id='core.routerid__router_id',
    project='core.project__project__name',
).regpoint('monitoring').registry_fields(
```

```
    last_seen='core.general__last_seen',
)
```

The resulting `Node` (actually the results will be instances of a special proxy class called `NodeRegistryProxy`, because we cannot modify fields in the existing model class) instances will have additional fields called `name`, `type`, `router_id`, `project` and `last_seen` that will have the values of their respective registry models. Values for these fields will be obtained by performing joins with tables of the registry item models that provide these fields. Note that multiple models may be registered under the same registry identifier (for example `core.general` in the above case actually has `GeneralConfig` and `CgmGeneralConfig` registered). In such cases all models will be traversed and the one providing field `name` will be selected for joining (again, in the above case this would correspond to `GeneralConfig`).

Similarly, you can use registry fields in filter expressions using `registry_filter` method:

```
Node.objects.regpoint('config').registry_filter(
    core_general__name='test-4',
)
```

The same rule as above applies for model resolution.

Firmware Image Generation

TODO.

Configuration Generating Modules

TODO.

Builders

Builders generate firmware images. They are packaged as Docker images, which you run, and then you can connect to them over SSH and issue commands to build a new firmware image. Builders are platform and architecture specific. Their main purpose is to bundle the whole toolchain needed to generate firmware images into isolated, shareable, and reproducible containers.

Nodewatcher uses builders to generate images automatically. You have to register builders you want to use through nodewatcher's admin interface.

Build Version

Each builder version corresponds to the the source code version which was used to create the builder. The version of a particular builder is automatically fetched from its Docker container when the builder is registered with nodewatcher.

Build Channel

Build channels allow you to group multiple builders together. For example, you could have two groups `stable` and `experimental`. You would put builders for stable release of the source code for your firmware into `stable`. You can put builders with new experimental features or builders with unstable code still being developed into `experimental` channel.

Then, for each node maintainers can choose which build channel to use for generation of its firmware image. This allows for example for some nodes to always use stable firmware versions, while maintainers who like to help testing development versions can use experimental channel for their nodes.

Platforms

Nodewatcher platform descriptors are defined in `nodewatcher/modules/platforms` and are Python modules. Platform descriptors define way that firmware images are generated. Currently supported platforms are OpenWrt and LEDE. Suport for new platforms can be done by extending existing descriptors if your platform is based on existing platforms or make a new one that suits your platform.

Device Descriptors

Nodewatcher device descriptors are defined in `nodewatcher/modules/devices` and are normal Python modules. The best way to define a new descriptor is to extend an already existing one. This can be done if a similar device is already supported (has a descriptor) and you just need to modify some minor things. In this case your new descriptor may simply extend an existing one using Python class inheritance and omit some of the attributes. In any case identifier and name must be defined (and be unique).

The following things need to be determined for a device descriptor:

- `identifier` is an all-lowercase unique device identifier, which should consist of a manufacturer prefix, a model identifier and a version.
- `name` is a human-readable device name consisting of a model identifier and a version. It should not contain a manufacturer name.
- `manufacturer` is a human-readable manufacturer name.
- `url` is an URL containing more information about the device.
- `architecture` identifies the OpenWrt architecture that the device needs (eg. `ar71xx`).
- `antennas` is a list of antenna descriptors, which describe physical antennas attached to the device by default.
- `radios` is a list of radio descriptors, which describe radios present on the device. Each radio specifies the protocols that it supports, antennas that it has attached and features that it supports. Each radio is identified by its logical identifier, which is platform-independent (eg. `wifi0`).
- `switches` contains a list of switch descriptors, which describe ethernet switches present on the device. Each switch is identified by its logical identifier, which is platform-independent (eg. `switch0`). Each switch can also define multiple VLAN presets that can be used when configuring the switch. If the switch supports custom configuration, it should have the `configurable` attribute set to `True`.
- `ports` contains a list of ethernet ports, which describe physical ethernet ports. Each port is identified by its logical identifier, which is platform-independent (eg. `wan0`, `lan0`, etc.).
- `port_map` contains the mapping of logical port names to platform-specific identifiers (one mapping per platform, eg. OpenWrt). It maps all platform-independent identifiers (eg. `wifi0`, `switch0`, `wan0`, `lan0`) to identifiers used on a specific platform (eg. `radio0`, `switch0`, `eth1`, `eth0`). For switch configurations a special `SwitchPortMap` instance may be used to define VLAN interface naming.
- `drivers` defines the drivers used by the radios on each platform. For example, on OpenWrt this may be `mac80211`.
- `profiles` define the platform-specific device profiles that should be used when building an image and paths to the resulting firmware files. Note that a single profile may be used for multiple devices. For example, on OpenWrt a profile `TLWR741` generates a firmware file `openwrt-ar71xx-generic-tl-wr741nd-v1-squashfs-factory.bin` for the TP-Link

WR741NDv1 device and this must be configured here. The firmware filenames can also contain filename patterns containing `?` and `*` to match different filenames at once.

The best way to determine the values for `radios`, `switches`, `ports`, `port_map` and `drivers` is to boot a stock version of OpenWrt on the device and check the default configuration inside `/etc/config/network` and `/etc/config/wireless`.

OpenWrt profiles may be listed by running `make info` on the generated image builder.

Because LEDE platform is very similar to OpenWrt, if OpenWrt profile, for a device you are interested in, already exists, you can often base the LEDE device profile on OpenWrt's device profile for that particular device.

Resources

TODO.

Mixins

TODO.

Node Configuration Schema

Per-node configuration schema in nodewatcher is built from various Django models and mixins, using light extensions provided by the *Registry ORM Extensions*. This documentation specifies, for each registry item identifier (see *Registry Items*) the models that provide parts of the final schema.

Each node is defined as an instance of `nodewatcher.core.models.Node` and represents a network-connected device that may be managed by nodewatcher. The model instance itself only provides a universally unique identifier (UUID) and has no other attributes. All configuration attributes are added by various models through the use of the registry.

core.general

The general schema contains a basic node name configuration.

In case firmware generation support is enabled (by loading the `nodewatcher.core.generator.cgm` module), an extended set of options becomes available. These options configure the specific target device and firmware version that should be generated when requested.

core.type

The `nodewatcher.modules.administration.types` module provides a schema extension that enables types for each node to be configured. Currently, the following node types are registered by default:

- server,
- wireless,
- test,
- mobile,
- dead,

- unknown.

Additional types may be registered by other modules.

core.project

The `nodewatcher.modules.administration.projects` module provides a schema extension that enables projects for each node to be configured. Projects provide logical or geographical structuring of node deployments.

Each project may have multiple SSID configurations attached.

For each node a project may then be configured.

core.description

The `nodewatcher.modules.administration.description` module provides a schema extension that enables unstructured notes and an URL to be added to any node.

core.location

The `nodewatcher.modules.administration.location` module provides a schema extension that provides geographical positioning of a node.

core.routerid

To identify nodes within the routing protocols a configuration schema is provided to configure the router identifier of a node. Each node may have multiple router identifiers. The following router identifier families are registered by default:

- ipv4,
- ipv6.

Additional families may be registered by other modules (for example MAC addresses for L2 routing protocols).

For IP based router identifiers, there exist two specializations. The first enables static IP based router ID configuration.

The second enables allocation of router identifiers from IP pools.

core.authentication

There are multiple options provided for configuring user authentication on the nodes. By default, the `nodewatcher.core.generator.cgm` module provides a base class for all authentication mechanisms.

A password configuration option is also provided by default.

Public key authentication is provided by `nodewatcher.modules.authentication.public_key` module. It extends the schema with the public key authentication method. Multiple authentication methods may be configured for each node.

Public keys should be provided in a SSH-compatible encoding format, for example:

```
ssh-rsa AAAAB3NzaC1yc2EAAAADA...Oipsw25uxIvkDKjAxzQ== user@host
```

core.roles

The module `nodewatcher.modules.administration.roles` adds support for specifying roles that a node may have. Several default roles are provided.

The following roles are provided by default:

- `system` (the node has an important system function, required for network operation),
- `border-router` (the node is a border router, enabling access to external networks),
- `vpn-server` (the node provides a VPN server for other nodes),
- `redundancy-required` (the node requires multiple redundant links).

core.switch

When a device supports switch configurations, the switch may be configured using this registry item. A concrete implementation is provided with the core module.

All switches must define a default preset under the identifier `default` in their device descriptor.

core.switch.vlan

Each switch can contain multiple VLAN configurations, which define how individual switch ports are grouped into VLANs.

core.interfaces

In order to configure network interfaces, several interface types are implemented. All interface configurations extend a base class.

An abstract mixin is provided for configuring interfaces which may be used for routing purposes by the registered routing protocols. In case an interface should support routing, it should include the mixin among its bases.

The following interface types are currently implemented by `nodewatcher.core.generator.cgm`:

- ethernet,
- wifi radio,
- wifi virtual interface,
- mobile,
- vpn,
- bridge.

Wireless interfaces are split into two configuration classes. The first class instances represent physical wireless radios.

Then, for each wireless radio, multiple wireless virtual interfaces may be configured. Each virtual interface will cause a new wireless network to be created. By default, the following wireless modes are registered:

- mesh,
- ap,
- sta.

Additional modes may be registered by other modules.

Modules may provide support for VPN tunnel interfaces. The `nodewatcher.modules.vpn.tunneldigger` module provides support for configuring Tunneldigger-based tunnels.

Configuration classes for mobile interfaces, like 3G, are also provided.

Bridges consisting of multiple other devices may also be configured.

core.interfaces.network

For each of the above physical interfaces, various network configurations may be set. Note that not all network configurations may be used on all types of interfaces. A base class is provided for all network configurations.

An abstract mixin is provided for configuring networks which may be announce via a dynamic routing protocol. In case a network should support such announcement, it should include the mixin among its bases.

Another abstract mixin is provided for configuring networks which may be leased to clients via DHCP or other protocols. In case a network should support leases, it should include the mixin among its bases.

The simplest is a static IP network configuration.

Resources may also be configured from various pools (for available fields, see *Resources*).

Interfaces may also be configured to obtain addresses via DHCP.

Ethernet interfaces may also be configured via PPPoE.

For bridge slaves, network configuration specifies the master bridge interface.

core.interfaces.limits

Various limits (like QoS) may also be configured for each interface.

Currently, a throughput limit may be configured.

core.servers.dns

Multiple DNS servers may also be configured.

Node Monitoring Schema

Per-node monitoring schema in nodewatcher is built from various Django models and mixins, using light extensions provided by the *Registry ORM Extensions*. This documentation specifies, for each registry item identifier (see *Registry Items*) the models that provide parts of the final schema.

Each node is defined as an instance of `nodewatcher.core.models.Node` and represents a network-connected device that may be managed by nodewatcher. The model instance itself only provides a universally unique identifier (UUID) and has no other attributes. All monitoring attributes are added by various models through the use of the registry.

core.general

core.interfaces

core.interfaces.network

system.status

system.resources.general

system.resources.network

network.routing.topology

network.routing.announces

network.measurement.rtt

network.clients

Translation

Making Strings Translatable

To make your strings translatable edit them as follows:

- in Django templates: `{% trans "Some string" %}`
- in Python files: `_("some string")`
- in JavaScript files: `gettext("some string")`

Creation of Translation Files

Warning: It is important to run following commands from `nodewatcher` subdirectory of the repository!

Add your wanted language(s) in `settings.py` under `LANGUAGES` and make sure your Django applications you want translated have a directory called `locale` (if not, you should create one).

When all string are marked for translation and settings are prepared run command:

```
../manage.py makemessages -l <language code>
../manage.py makemessages -l <language code> -d djangojs
```

Note: You should replace `<language code>` with the language code you are creating translation files for.

Example for Slovenian language:

```
../manage.py makemessages -l sl
../manage.py makemessages -l sl -d djangojs
```

This will create file `django.po` under `locale/sl/LC_MESSAGES/`

Note: All other Directories are automatically created.

Translation

Open file `django.po` with text editor or with special translation tool (Poedit for example) and translate strings. Original string is named `msgid "Some string"` and under it there is `msgstr` which contains an empty string where you should write your translation.

Compiling Translation Files

After you are done translating run the command:

```
../manage.py compilemessages
```

and a new file `django.mo` will be created.

Translated strings should now be available in *nodewatcher*.

Troubleshooting

You should really read documentation on translation in Django, but here are some troubleshooting notes:

- you need `gettext` installed (see instructions for Windows)
- sometimes it is necessary to restart development HTTP server to get new compiled translations to work
- yes, you have to compile `.po` files into `.mo` files after you edit `.po` files
- verify changes to `.po` files and make sure that not whole files are changed just because your `gettext` changed `/` into `\`, manually clean/revert such changes

CHAPTER 2

Paper

The paper presenting the project was published in journal Computer Networks, in the special issue on community networks, DOI 10.1016/j.comnet.2015.09.021.

Available at:

- http://www.sciencedirect.com/science/article/pii/S1389128615003400
- https://www.sharelatex.com/github/repos/wlanslovenija/nodewatcher-paper/builds/latest/output.pdf
- http://arxiv.org/abs/1601.02372

CHAPTER 3

Source Code, Issue Tracker and Mailing List

For development *wlan slovenija* open wireless network development Trac is used, so you can see existing open tickets or open a new one there. Source code is available on GitHub. If you have any questions or if you want to discuss the project, use nodewatcher mailing list.

CHAPTER 4

Indices and Tables

- *genindex*
- *search*