WAM-Server Documentation

Hendrik Huyskens

Aug 13, 2020

Contents:

1 Introduction 1

2 Helpers 11

3 API Changes 15

4 Contribute 17

5 Continuous integration 19

6 Edit documentation 21

7 Indices and tables 23

CHAPTER 1

Introduction

WAM-Server can be set up either manually (setting up database, environment and wam by hand) or via docker. Both methods will be presented in the following.

Contents

- *Introduction*
  - *Idea*
  - *Installation via Docker*
    - *Prerequisites*
    - *Setup*
  - *Installation from Scratch*
    - *Prerequisites*
    - *PostgreSQL Setup*
    - *PostGIS Setup*
    - *Message Broker*
  - *Getting Started*
    - *Setup on Linux:*
    - *Setup on Windows:*
    - *Configuration file*
  - *Deploying server without apps*
    - *Example app*
  - *Deploying server with custom apps*

- *Celery*
  - *Setup*
  - *Tasks*

1.1 Idea

WebAppMap-Server [WAM] provides a basic and expandable Django infrastructure to easily add applications.

1.2 Installation via Docker

1.2.1 Prerequisites

- git is installed.
- Docker and Docker-Compose are installed.

1.2.2 Setup

Following folder structure is recommended and used in the following:

```
wam_docker
+-- docker (Contains docker config and WAM code basis)
|   +-- docker-compose.yml
|   +-- WAM (WAM-Codebasis; WAM-apps are integrated here later)
+-- config (Config for WAM and apps)
|   +-- config.cfg
```

Note: Changes to this structure have to be adopted in config file and docker config (docker-compose.yml)

Setup of folder structure and code basis:

```
mkdir wam_docker
cd wam_docker
mkdir docker
mkdir config
cd docker
git clone https://github.com/rl-institut/WAM.git
cp WAM/docker-compose.yml .
cp WAM/.config/config.cfg ../config/
```

Next, config files have to be adopted (*docker-compose.yml* und *config.cfg*). Finally, following command builds image and starts new container:

```
sudo docker-compose up -d --build
```

WAM-Server should now be available under 127.0.0.1:5000 !

1.3 Installation from Scratch

1.3.1 Prerequisites

- PostgreSQL library should be installed.
- A PostgreSQL database should be created (see *PostgreSQL Setup*).
- *PostGIS* library should be installed (see *PostGIS Setup*).
- if *Setup* shall be used, a *Message Broker* must be used.

1.3.2 PostgreSQL Setup

This section describes the installtion of PostgreSQL on Linux and Windows.

Linux

The following instructions are for Ubuntu and inspired from here.

First, create a user name (here, *wam_admin* is used for the *USER* field of the config file configuration)

```
sudo -u postgres createuser --superuser wam_admin
```

Then enter in psql shell

```
sudo -u postgres psql
```

There, change the password for the user *wam_admin*

```
postgres=# \password wam_admin
```

Enter the same password you will use under the *PASSWORD* field in the config file (configuration) and exit the shell with *\q*

Then, create the database you will use under the *NAME* field in the config file (configuration)

```
sudo -u postgres createdb -O wam_admin wam_database
```

Whenever you want to use the database you should run

```
sudo service postgresql start
```

This can be stopped using the command

```
sudo service postgresql stop
```

Windows

1. Download and install latest PostgreSQL for Windows.
2. At the end of or after the install of PostgreSQL for Windows use *Stack Builder* (will be installed with PostgresSQL) to install *Spatial Extensions -> PostGIS bundle*

In the Windows command line (cmd.exe) or Powershell:

3. Set the path environment variable for PostgreSQL (to be able to use PostgreSQL via the command line), e.g.:

```
SETX PATH "%PATH%;C:\Program Files\PostgreSQL\11\bin"
```

4. Login to “psql” as user “postgres”:

```
psql -U postgres
```

5. In psql create superuser “wam_admin”:

```
CREATE ROLE wam_admin WITH LOGIN SUPERUSER INHERIT CREATEDB CREATEROLE REPLICATION;
```

6. Set a password for the superuser wam_admin:

```
\password wam_admin
```

7. Quit psql:

```
\q
```

8. Create database:

```
createdb -U wam_admin wam_database
```

How to start, stop and restart PostgreSQL on Windows:

1. Start:

```
pg_ctl -D "<drive letter>:\path\to\PostgreSQL\<version>\data" start
```

2. Stop:

```
pg_ctl -D "<drive letter>:\path\to\PostgreSQL\<version>\data" stop
```

3. Restart:

```
pg_ctl -D "<drive letter>:\path\to\PostgreSQL\<version>\data" restart
```

1.3.3 PostGIS Setup

This section describes the installtion of PostGIS on Linux and Windows.

Linux (Ubuntu)

```
sudo apt-get install binutils libproj-dev gdal-bin
```

```
sudo apt-get install postgis postgresql-10-postgis-2.4
```

For other systems see https://postgis.net/.

Activate postgis extension (execute as SQL query) to make it work:

```
CREATE EXTENSION postgis;
```

Windows

If not already installed, use *Stack Builder* (will be installed with PostgresSQL) to install *Spatial Extensions -> PostGIS bundle*

1. Login to psql as wam_admin in wam_database:

```
psql -U wam_admin wam_database
```

2. Create postgis extension:

```
CREATE EXTENSION postgis;
```

3. Quit psql:

```
\q
```

1.3.4 Message Broker

On our WAM-Server the message broker RabbitMQ is running in a docker container (see RabbitMQDocker). Developers allowed to use our service can connect to it by setting up a ssh-tunnel:

```
ssh -fNL 5672:localhost:5672 wam_user@wam.rl-institut.de
```

Other users have to setup their own message broker

1.4 Getting Started

1.4.1 Setup on Linux:

Clone repository from GitHub via:

```
git clone https://github.com/rl-institut/WAM.git
```

Setup conda environment with required packages via:

```
conda env create -f environment.yml
```

Afterwards, applications can be "plugged-in" by simply cloning application into the root directory and adding application name to environment variable *WAM_APPS* (see environment). Requirements and configuration of an application can be found at *Deploying server with custom apps*

Linux: Environment Variables

WAM-Server needs at least the following environment variables:

- WAM_CONFIG_PATH: full path to the configuration file (see *Configuration file* for the file content)
- WAM_APPS: Apps which shall be loaded within *INSTALLED_APPS*. Additionally, individual app settings are loaded (see *Deploying server with custom apps*).

On Ubuntu, one can add them to the `bashrc` file, so that they are loaded automatically :

```
nano ~/.bashrc
```

then add the following two lines

```
export WAM_CONFIG_PATH=<path to your WAM repo>/.config/config.cfg
export WAM_APPS=<name of wam app 1>,<name of wam app 2>
```

1.4.2 Setup on Windows:

Prerequisites:

- Git Bash (included in Git for Windows)
- Install the *small* version of Conda (Miniconda), the big version Anaconda is not needed (reverse install settings, include Anaconda to PATH but don't install as default): Link.

1. Open Git Bash and create a project folder

```
mkdir project_folder_name
cd project_folder_name
```

2. Clone repository from GitHub via:

```
git clone https://github.com/rl-institut/WAM.git
```

3. Setup conda environment with required packages via:

```
conda env create -f environment.yml
```

Afterwards, applications can be "plugged-in" by simply cloning application into the root directory and adding application name to environment variable *WAM_APPS* (see environment). Requirements and configuration of an application can be found at *Deploying server with custom apps*

4. Install package dependencies of your WAM app(s):

- Example app: stemp_abw

```
conda install -c conda-forge gdal
conda activate django
conda install shapely
pip install -r requirements.txt
pip install -r stemp_abw/requirements.txt
```

Windows: Environment Variables

WAM-Server needs at least the following environment variables:

- WAM_CONFIG_PATH: full path to the configuration file (see *Configuration file* for the file content)
- WAM_APPS: Apps which shall be loaded within *INSTALLED_APPS*. Additionally, individual app settings are loaded (see *Deploying server with custom apps*).

Add them to your Windows user environment:

```
SETX WAM_CONFIG_PATH "<drive letter>:\<path to your WAM repo>\.config\config.cfg"
SETX WAM_APPS "<name of wam app 1>,<name of wam app 2>"
```

1.4.3 Configuration file

Configuration file located under the path given by *WAM_CONFIG_PATH* is loaded within *settings .py*. The file is read in using python's configobj package. The file should contain following sections:

- *[WAM]*: general config for the WAM-Server,
- *[DATABASE]*: with at least one default database connection, which will be used as django's database,
- *[CELERY]*: if celery is needed
- *[<APP_NAME>]*: Multiple sections containing config for each app

See minimal example config_file:

```
[WAM]
    DEBUG=False
    ALLOWED_HOSTS=127.0.0.1
    SECRET_KEY=<secret_key>
    DJANGO_DB=DEFAULT

[DATABASES]
    [[DEFAULT]]
        ENGINE = django.contrib.gis.db.backends.postgis
        HOST = localhost
        PORT = 5432
        NAME = wam_database
        USER = wam_admin
        PASSWORD = wam_password
    [[LOCAL]]
        ENGINE = postgresql
        HOST = localhost
        PORT = 5432
        NAME = wam_database
        USER = wam_admin
        PASSWORD = wam_password

[CELERY]
    HOST = localhost
    PORT = 5672
    USER = <USER>
    PASSWORD = <PASSWORD>
    VHOST = <VHOST>
```

Note: the indent level is important in the configuration file. Keywords are specified within `[]`, they are analog to the key in a python `dict`. The nesting level of a keyword depends on the number of square brackets.

1.5 Deploying server without apps

Even without adding apps into WAM, you can follow these steps to deploy the WAM server locally.

Make sure the *postgresql* service is running

```
sudo service postgresql start
```

Then, run the following commands

```
python manage.py makemigrations
```

```
python manage.py migrate
```

```
python manage.py createsuperuser
```

After the last command, follow the instructions inside the terminal and use the same values for user and password as the *USER* and *PASSWORD* fields of the *Configuration file*.

Finally access to the WAM server with

```
python manage.py runserver
```

1.5.1 Example app

From the root level of the WAM repository, you can clone the app *WAM_APP_stemp_mv* with

```
git clone https://github.com/rl-institut/WAM_APP_stemp_mv.git
```

For the time being you have to rename the app folder *stemp* and set your environment variable *WAM_APPS* to *stemp*

1.6 Deploying server with custom apps

Requirements:

- *urls.py* which includes *app_name* equaling the app name and an index page, which is loaded as landing page by default

Additional setups:

- *settings.py* can setup additional parameters for projects *settings.py*.

If your app requires the use of additional packages, you should list them in the settings.py of your app (not the settings.py file form wam core) in the following way

```
INSTALLED_APP = ['package1', 'package2']
```

Then, wam core will manage the packages' installation and avoid duplicate installations between the different apps.

- *app_settings.py* contains application specific settings and is loaded at start of django server at the end of *settings.py*. This file may include additional database connections, loading of config files needed for the application, etc.

Warning: Avoid using config variables for packages in your app as it may override or get overridden by package config of other app!

- *labels.cfg* (uses configobj) supports easy adding of labels to templates via templatetags (see *Labels*)

To install the required packages for each app run

```
python install_requirements.py
```

from the root level of the WAM repository. Then follow the procedure described under *Deploying server without apps*

1.7 Celery

Celery is included in WAM. In order to use celery in an app, follow the setup.

1.7.1 Setup

- *Message Broker* has to be running.
- Celery must be configured to use the message broker (see configuration).
- Run celery from WAM root directory via command (**environment_** variables have to be set!):

```
celery -A wam worker -Q wam_queue -l info
```

- After running above command, celery searches for *tasks.py* in every app and "activates" all tasks in it.
- WAM-apps are now able to run celery *Tasks*!

1.7.2 Tasks

Celery tasks are easy…

You simply have to create a *task.py* in your app and put *from wam.celery import app* at beginning of the module. Then, "normal" python functions can be turned into celery tasks using decorator *@app.task*. See minimal example:

```
from wam.celery import app


@app.task
def multiply_by_ten(number):
    return number * 1000
```

Afterwards, the task can be imported, run and controlled from elsewhere:

```
from app_name import tasks


running_task = None


def start_task():
    # Running task has to be stored, in order to get results
    running_task = tasks.multiply_by_ten.delay(10)


def task_is_ready():
    # Returns True if task has finished
    return running_task.ready()


def get_results():
    # Returns results of the tasks
    return running_task.get()
```

Note: Input parameters and output results of celery tasks must be pickleable or jsonable. In order to use Django models within the celery task you should exchange primary keys (see also https://oddbird.net/2017/03/20/serializing-things/).

See http://docs.celeryproject.org/en/latest/userguide/tasks.html for more information on tasks.

CHAPTER 2

Helpers

Little helper functions and classes for several use-cases.

2.1 Group Permissions

A TemplateView-mixin is provided, which checks if user is logged in and if user is member of certain groups. To check group permissions on a TemplateView, simply inherit from GroupCheckMixin as well and define groups to check.

Example:

```
from django.views.generic import TemplateView
from utils.permissions import GroupCheckMixin

class MyView(GroupCheckMixin, TemplateView):
    groups_to_check = ['group1', 'group2']
    template_name = 'my_template.html'
```

2.2 Labels

A templatetag *label* is provided to easily load labels into templates. To do so, labels can be configured in a file *labels.cfg* within application folder. Section, subsection, etc. are supported (use *:* to separate sections) to organize labels within needed structure.

Example:

```
<!-- Loads label-templatetag -->
{% load labels %}
```

(continues on next page)

(continued from previous page)

```
<!-- Tries to load label 'description' from section 'general' and subsection 'landing_
↪page' from labels.cfg -->
{% label 'general:landing_page:title' %}
```

with *labels.cfg* as:

```
[general]
    [[landing_page]]
        title = Das ist die Startseite!
```

Additionally, the label template tag supports two attributes:

- *safe* (boolean, default=False): Can be set to allow import of html-code
- *app* (str, default=None): Can be set explicitly to load labels from given app (Must be set, if template tag is requested from widget or form template)

Note: The *labels* templatetag uses requested path to specify for which application a label is requested. Thus, path *stemp/index/* will try to load labels from application *stemp*. If template tag is used within widget or form template, app attribute has to be set. If no label is found or given (sub-) section is not found, *None* will be returned.

2.3 Feedback Form

A feedback form is available which can be used in all apps. The feedback is send via e-mail using an Exchange account. Required configuration parameters for the Exchange account are *WAM_EXCHANGE_ACCOUNT*, *WAM_EXCHANGE_EMAIL* and *WAM_EXCHANGE_PW*. They must be set in the *[WAM]* section of the *config.cfg* file, see *Configuration file* for details.

To use the form, just add the view to your urls like

```
# my_app/urls.py

from utils.views import FeedbackView

admin_url_patterns = [
    path(<path to other view>),
    ...,
    path('feedback/', FeedbackView.as_view(app_name='<my app name>'), name='feedback')
]
```

Make sure you have the parameter `email` set in your *app.cfg*, example:

```
# my_app/app.cfg
category = app
name = ...
icon = ...
email = 'address_of_app_admin@domain.tld'
```

This address is used to send feedback messages for the app. If the Exchange account or app admin's e-mail address is not configured correctly, the user will be redirected to an error page.

2.4 Customizing Admin Site

With custom admin site, it is possible to add app-specific views to admin pages. The WAM-backend is configured to search for list *admin_url_patterns* in *urls.py* for every app. Those urls are internally added in AdminSite.get_url() and are afterwards available on admin site.

An additional view for admin site can now integrated via:

```
# my_app/urls.py

from my_app import views

admin_url_patterns = [
    path(
        'my_url',
        views.MyView.as_view(),
    ),
]
```

Afterwards, this view would be accessible (**to all users!**, see example below for admin-only-access) under *. . . /admin/my_url*.

An additional example can be found in Stemp Tool MV

```
from wam.admin import wam_admin_site
from stemp import views_admin

admin_url_patterns = [
    path(
        'stemp/manage',
        wam_admin_site.admin_view(views_admin.ManageView.as_view()),
        name='manage'
    ),
]
```

Please notice the wrapping of custom view into *wam_admin_site.admin_view* function - this will guarantee admin-only access!

CHAPTER 3

API Changes

List of API-Changes which have to be regarded when using WAM-Django-Backend.

3.1 7667dc9 (29.11.2018)

Renamed utils.utils to utils.shortcuts

3.2 8ad8e54 (26.11.2018)

Now, entire engine name has to be set in config-file. Example for case of django backend:

```
# In config.cfg:

 [DATABASES]
     [[DEFAULT]]
         ENGINE = django.db.backends.postgresql  # instead of postgresql
         HOST = localhost
         PORT = 5432
         NAME = wam_database
         USER = wam_admin
         PASSWORD = wam_password
```

3.3 551866e (19.11.2018)

In order to customize/edit admin site per app, the default admin site from *django.contrib.admin.site* is exchanged with *wam.admin.wam_admin_site*. This new *AdminSite* class is used as default admin site for WAM-backend. Thus, **your models will not appear in admin site**, until you changed your code to use *wam_admin_site* instead. Fortunately, this is very easy. . .

Instead of this "normal" looking admin.py file in app-folder:

```
from django.contrib import admin
from my_app import models

admin.site.register(models.MyModel)
```

Simply import wam_admin_site and register your model there:

```
from wam.admin import wam_admin_site
from my_app import models

wam_admin_site.register(models.MyModel)
```

With this little change, everything works as expected. But now, you are able to extend WAM-admin site! See *Customizing Admin Site* for more information.

Links:

General https://docs.djangoproject.com/en/2.1/ref/contrib/admin/#module-django.contrib.admin

Custom https://docs.djangoproject.com/en/2.1/ref/contrib/admin/#customizing-the-adminsite-class

Admin-URLS https://docs.djangoproject.com/en/2.1/ref/contrib/admin/#django.contrib.admin.ModelAdmin.get_urls

CHAPTER 4

Contribute

Contents

- *Contribute*
  - *Idea*
  - *Getting Started*
    - *Workflow*
    - *Code Linting*

4.1 Idea

If a user want to improve the existing code, here is a description of the workflow and coding practices we would like to enable.

4.2 Getting Started

4.2.1 Workflow

4.2.2 Code Linting

CHAPTER 5

Continuous integration

CHAPTER 6

Edit documentation

6.1 Idea

If a user want to improve the documentation, here is how to produce html files locally before pushing the changes to the repository.

6.2 Getting Started

First, you have to install the sphinx package:

```
pip install sphinx
```

Then, from the root of the WAM repository, run the following command to compile your documentation:

```
make html
```

The html file will be available under doc/_build/html/index.html.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search