Flapjack Documentation – Backend

Contents

1 Introduction

The backend is written in Python, using the Django framework as a base for the API and WebSockets architecture. Django Rest Framework library is used to build a powerful and flexible REST API using the Django models, while Pandas, Numpy and SciPy libraries are used to process the data before sending it to the frontend. The API provided by the backend, in addition to serve the frontend with the services it needs, can be used for more sophisticated analysis by connecting to existing libraries and software.

This document aims to describe the application's key structures and functions for development purposes.

2 Requirements, Building and Executing

2.1 Requirements

This app is built with Django framework using Python, and deployed in containers via Docker. All dependencies are handled via the Dockerfile and the docker-compose.yml within the project source.

As such, the only pre-requisite for running the application is a modern version of Docker and Docker Compose.

2.2 Environment Variables

To be able to protect secret keys or differentiate between development and production environments, an environment file must be created within the project root, and named .env.prod for production environment and .env for development environment. These files must have the following fields:

```
DEBUG="<Django Debug Mode>"
SECRET_KEY="<Django Secret Key>"
DJANGO_ALLOWED_HOSTS="<Django Allowed IPs>"
SQL_ENGINE="<Django Database Engine>"
SQL_DATABASE="<Django Database Name>"
SQL_USER="<Django Database User>"
SQL_PASSWORD="<Django Database Password>"
SQL_HOST="<Django Database Host>"
SQL_PORT="<Django Database Port>"
```

In addition to .env.prod, running the application in a production environment requires a file named .env.prod.db to expose the database credentials to the PostgreSQL database Docker service. This file must have the following fields:

```
POSTGRES_DB="<Django Database Name>"
POSTGRES_USER="<Django Database User>"
POSTGRES_PASSWORD="<Django Database Password>"
```

2.3 Building

As mentioned previously, dependencies are downloaded and installed in a docker container when it is built. To build the application, run:

```
docker-compose -f <docker-compose path> build
```

Where `<docker-compose path>` is `./docker-compose.dev.yml` for development environment and `./docker-compose.prod.yml` for production environment.

2.4 Running

After creating the environment variables and building the container, it is necessary to setup the database before running the application by running the Django migrations. This can be done running the following command:

```
docker-compose -f <docker-compose path> run flapjack_api python manage.py migrate
```

Where `<docker-compose path>` is `./docker-compose.dev.yml` for development environment and `./docker-compose.prod.yml` for production environment.

After running the migration, the application can be run with:

```
docker-compose -f <docker-compose path> up
```

3 File Structure

The application folders are organized as follows:

- `accounts`: Contains the serializers and endpoints for authentication.
- `analyze`: Contains the analysis engine.
- `flapjack_api`: Contains the project setup.
  - `channels_middleware.py`: Contains the middleware used to authenticate WebSockets connections.
  - `routing.py`: Contains the API WebSockets' routes.
  - `settings.py`: Contains the Django project main settings
  - `urls.py`: Contains the API HTTP's routes.
  - `routes.js`: File containing the web page's routes. It contains an array of routes with the following format:
- `plot`: Contains the plotting WebSockets' methods.
  - `consumers.py`: Contains the plotting WebSockets's setup.
  - `plotting.py`: Contains plotting auxiliary methods.
  - `routing.py`: Contains the plotting Websocket's routes.

- `registry`: Contains the project's models, REST API endpoints and data upload WebSockets' methods.
  - `consumers.py`: Contains the data upload WebSockets' setup.
  - `models.py`: Contains the database models.
  - `permissions.py`: Contains the permissions setup for each model.
  - `routing.py`: Contains the data upload Websocket's routes.
  - `serializers.py`: Contains the serializers for each model used to serialize data exposed in endpoints.
  - `upload.py`: Contains the data upload auxiliary methods.
  - `util.py`: Contains general auxiliary methods.
  - `views.py`: Contains each model ViewSet and FilterSet.

4 Django Rest Framework

As told in section 1, Django Rest Framework library is used to easily build a flexible and powerful REST API. The project make use of the following DRF features:

- ViewSets: Used to define each model's endpoint behaviour.
- FilterSet: Used to define available filters for each endpoint in addition to the `django-rest-framework-filters` library.
- routers: Used to provide the url of each model endpoint based on its ViewSet

The API make use of some global configurations provided by Django Rest Framework. Those configurations are defined in `/flapjack_api/settings.py` and are the following ones:

- `DEFAULT_FILTER_BACKEND`: Set the default filter used by any endpoint to the Django Rest Framework's provided one.
- `DEFAULT_PAGINATION_CLASS`: Set the default pagination class to the Django Rest Framework's provided one.

- `PAGE_SIZE`: Set the default pagination size to 100.
- `DEFAULT_AUTHENTICATION_CLASSES`: Set the default authentication system to use the JWT standard.
- `DEFAULT_PERMISSION_CLASSES`: State that each endpoint requires authentication to access it as its default permission rule.

5 JSON Web Tokens

The API and its WebSockets endpoints make use of the JWT standard for authentication. This standard works using 2 types of tokens; the `refresh token` and the `access token`. When registering or login in, both of them will be provided.

Only the `access token` is needed to access the others endpoints. It has to be provided in every request as a HTTP header as follows:

```
Authorization: Bearer <Access Token>
```

The `access token` expires after 5 minutes. In order to get a new one, a refresh endpoint is exposed with no authentication required. This endpoint returns a new `access token` by providing a `refesh_token` in the HTTP body as follows:

```
{
    "refresh": "<Refresh Token>"
}
```

6 Endpoints

6.1 Authentication

There are 3 endpoints exposed by the API:

1. Register: create a new user and password.

```
// POST /api/auth/register/
// Example request body
{
  "username": "JohnDoe",
  "password": "topsecret",
  "password2": "topsecret",
  "email": "john@doe.com"
}

// Example response body
{
    "refresh": "<Refresh Token",
    "access": "<Access Token">
}
```

2. Login: get a `refresh` and `access token` by providing an existing user and its password.

```
// POST /api/auth/log_in/
// Example request body
{
  "username": "JohnDoe",
  "password": "asd123"
}

// Example response body
{
    "refresh": "<Refresh Token",
    "access": "<Access Token">,
    "username": "JohnDoe",
    "email": "john@doe.com"
}
```

3. Refresh: get an `access token` by providing a `refresh token`

```
// POST /api/auth/refresh/
// Example request body
```

```
{
    "refresh": "<Refresh Token",
}

// Example response body
{
    "access": "<Access Token">
}

```

6.2 Models

By default each model has all the endpoints provided by Django Rest Framework's ViewSet. Those are:

- `GET`: for item retrieving.
- `PUT`: for item update.
- `PATCH`: for item partial update.
- `DELETE`: for item destroy.

To access any model's endpoint, the Authorization HTTP header must be included in the request as told in section 5.

Some examples of requests to the main endpoints are described below:

6.2.1 Studies

- Filter/search/list studies: displays studies that are whether owned, shared with the user or public.

```
/* GET /api/study/

Available query parameters:
- is_owner: bool
```

```
- search: string
*/

// Example response body
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 1,
            "is_owner": true,
            "shared_with": [],
            "name": "Dummy Study",
            "description": "Testing API",
            "doi": "",
            "public": false
        }
    ]
}

```

- Create study: create a new study by providing its name, description and whether is public or not.

```
// POST /api/study/

// Example request body

{
    "name": "Dummy Study",
    "description": "Testing API",
    "public": false
}

// Example response body
{
    "id": 1,
    "is_owner": true,
    "shared_with": [],
    "name": "Dummy Study",
    "description": "Testing API",
```

```
    "doi": "",
    "public": false
}
```

- Get study: display information about a specific study.

```
// GET /api/study/1/

// Example response body
{
    "id": 1,
    "is_owner": true,
    "shared_with": [],
    "name": "Dummy Study",
    "description": "Testing API",
    "doi": "",
    "public": false
}
```

- Partial update study.

```
// PATCH /api/study/1/

// Example request body
{
    "shared_with": ["another@mail.com"]
}

// Example response body
{
    "id": 1,
    "is_owner": true,
    "shared_with": [
        "another@mail.com"
    ],
    "name": "Dummy Study",
    "description": "Testing API",
    "doi": "",
    "public": false
}
```

```

```

- Destroy study.

```
// DELETE /api/study/1/

// Example response status
Status: 204 No Content

```

6.2.2 Assays

6.2.3 Vectors