AY 2021/2022

Politecnico di Milano

Implementation Document

Ottavia Belotti Alessio Braccini Riccardo Izzo

Professor
Elisabetta Di Nitto

Version 1.0
January 23, 2022

Contents

1 Introduction

The code can be found in the official project repository on GitHub at the link: `https://github.com/AlessioBraccini/SE2-Belotti-Braccini-Izzo`.

1.1 Purpose

This document aims to describe how the implementation and integration testing took place. Implementation is the last step of the DREAM application development cycle. Testing, instead, means check that the critical parts of the application works in a correct way, as described in the DD document.

1.2 Definitions, Acronyms, Abbreviations

- ACID: Atomicity-Consistency-Isolation-Durability
- API: Application Programming Interface
- CSRF: Cross Site Request Forgery
- DBMS: DataBase Management System
- DD: Design Document
- HTTP: HyperText Transfer Protocol
- JS: JavaScript
- ORM: Object-Relational Mapping
- REST: REpresentational State Transfer
- RASD: Requirements Analysis and Specification Document
- UI: User Interface
- URL: Uniform Resource Locator
- WSGI: Web Server Gateway Interface
- ASGI: Asynchronous Server Gateway Interface

1.3 Revision History

- Version 1.0:

1.4 References

- Django Framework: `https://www.djangoproject.com/`
- REST Framework: `https://www.django-rest-framework.org/`
- Vue.js: `https://vuejs.org/`
- Axios: `https://axios-http.com/docs/intro`

2 Development

2.1 Implemented Functionalities

Given the three types of user, we decided to implement th functionalities for Policy Maker users and Agronomist user. In particular:

Policy Maker

- Farmers ranking
- Visualization of humidity sensors and water irrigation systems data as graphs
- Retrieving agronomists' reports about steering initiatives

Agronomist

- Farmers ranking tailored on the agronomist's district
- Uploading reports concerning steering initiatives
- Creation and updating of daily plans
- Help requests inbox
- Weather widget

Figure 1: Django Framework and REST Framework logo

2.2 Adopted Development Frameworks

Model-View-Controller paradigm

2.2.1 Programming Language

The programming language of choice for the DREAM backend is Python.

- **Pros:**
  - \+ **Allow fast development**: Python is a very high-level programming language, so the built-in functions and reliable third-parties libraries allow us to spend the development time focusing on implementing the desired custom behaviors rather than building basic and non-functional features that are shared among most web apps.
  - \+ **Readability**: Python's syntax is very clear and human-friendly.
  - \+ **Large community and widely spread among WebApps**: being used a lot by the web app developers' community, the internet is full of help for troubleshooting. Moreover, this has resulted in a very extensive support through libraries dedicated to web developing and integration with frontend frameworks.
- **Cons:**
  - \- **Speed**: being an interpreted language, Python suffers of slower computation in comparison to other compiled languages (e.g. C++, Java, etc.).

For the client side we chose JavaScript, a text-based programming language, that allows to build interactive web pages. It handles the user's interaction with the elements present on the page. Alongside JavaScript, we used HTML and CSS to give structure and style to the page.

2.2.2 Django Framework

Django is Python web framework whose key is rapid development without sacrificing well structured code. In fact, the framework encourages to follow the MVC pattern and embeds it into its Django apps structure.

It is **secure**, since it provides the developers security tools (e.g. automatic user's passwords encryption, CSRF protection, SSL/HTTPS support, etc.) that have been implemented by experts in the field. See more about security in section 2.2.3.

It is **supportive**, given that it has lots of database operation support thanks to the *Object-relational mapping* (ORM) feature which easily maps DB tables to *Django Model* components (Model in MVC pattern) within the code, basic queries can be done transparently and in a very readable fashion without losing in performance. Moreover, it allows to configure the *Admin Interface* from where we, as backend developers, can easily manage the database, like monitoring at a glance the entries in all the DB tables and create new ones for testing purposes. A part from the DB integration, Django allows the creation of custom *endpoint URLs* from where the frontend will make the backend calls. Meanwhile the *Django View* components come to be a dedicated place for managing HTTP requests, acting as the Controller in MVC design. Django would also support *Template* components as its Views in MVC pattern, but in this project it hasn't been used because of the different framework choice to handle the presentaion part (see section 2.2.4).

Furthermore, Django is **extensively documented** and **scalable** since it's being used in very demanding professional contexts, being one of the frameworks of choice of well known social media platforms that experience high loads of requests due to their intensive usage from their user base.

2.2.3 Django REST Framework

We decided to pair REST framework to our Django backend since REST allows even more security features.

We used its authentication policies, in particular the **REST Token-based authentication** that ensures an authorized clients' connection to the backend, so that every interaction with the application server is safe from an authentication point of view: unknown users can access none of the backend's functionalities.

We used the *Djoser* REST library to handle users login and signup. The choice has been taken to make sure that such standard operations are carried out without flaws, especially in handling sensitive data (i.e. passwords). Moreover, *Djoser* is compatible with custom User Models, so it enables us to delegate the authentication phase to it even with our custom User model.

Furthermore, Django REST gives an extensive support in building *Django Views*, from basic skeleton function (`@api_view`) for managing all types of HTTP requests, up to more functional and request-specific functionalities.

2.2.4 Vue.js

Vue.js is an open-source model–view–viewmodel front end JavaScript framework that allow to create user interfaces and single-page applications. Vue.js features an incrementally adaptable architecture that focuses on declarative rendering and component composition. The core library is focused on the view layer only. Advanced features required for complex applications such as routing, state management and build tooling are offered via officially maintained supporting libraries and packages.

We used this framework to build up the client-side rendering of pages because its easy usage as it integrate in a single .vue file, also called components, the HTML, CSS and javascript part.

In order to communicate with the backend we use the javascript library Axios. It is a promise-based HTTP Client for node.js and the browser. On the server-side it uses the native node.js http module, while on the client it uses XMLHttpRequests. It transforms in an automatic way the json reply that arrives from the server in vue ready XMLHttpRequests.

2.3 API Integration

2.3.1 OpenWeatherMap API

To allow the user to retrieve the weather information we use this external api service that let us know in real time the weather condition of a specific

territory. This return us not only the basics information but also more specific ones.

2.4 DataBase

Our database system of choice is PostgreSQL, a well known object-relational database. It is reliable, robust, ACID-compliant and ensures high performance.

3 Source Code

3.1 Backend Structure

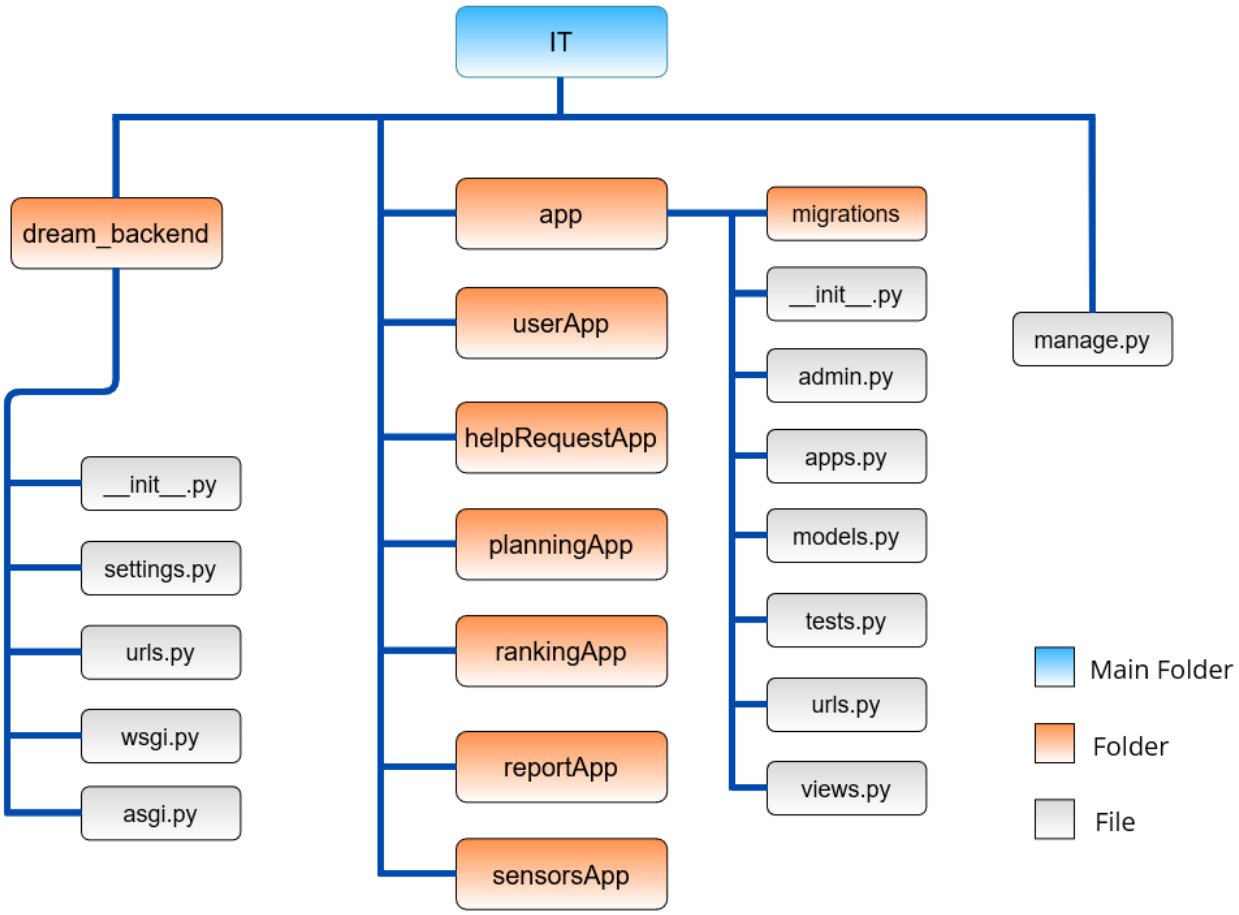

Figure 2: Backend structure

The back-end of the application follows the common Django project structure as shown in the figure above. This section describes the structure of the project and the role of each app.
Here is represented the structure of the back-end:

- **IT**: root directory, container for the project
- **dream_backend**: folder that contains the configuration files of the project
- **__init.py__**: it tells the Python interpreter that the directory is a Python package
- **settings.py**: main setting file for the Django project, used to configure all the applications and middleware, it also handles the database settings
- **urls.py**: URL declarations for the Django project, it contains all the endpoints that the website should have
- **wsgi.py**: entry-point for WSGI-compatible web servers to serve your project, it describes the way how servers interact with the applications
- **asgi.py**: entry-point for ASGI-compatible web servers to serve your project, ASGI works similar to WSGI but comes with some additional functionality
- **migrations**: Django's way of propagating changes to the models into the database schema, when changes occur this folder is populated with the records of them
- **admin.py**: used for registering the Django models into the Django administration, it allows to display them in the Django admin panel
- **apps.py**: common configuration file for all Django apps, used to configure the attributes of the app
- **models.py**: it defines the structure of the database, it allows the user to create database tables for the app with proper relationships using Python classes. It tells about the actual design, relationships between the data sets and their attribute constraints
- **tests.py**: used to test the overall working of the app through unit tests
- **views.py**: provide an interface through which a user interacts with a Django website, it contains the business logic of the app

- **manage.py**: command-line utility for executing Django commands; these includes debugging, deploying and running

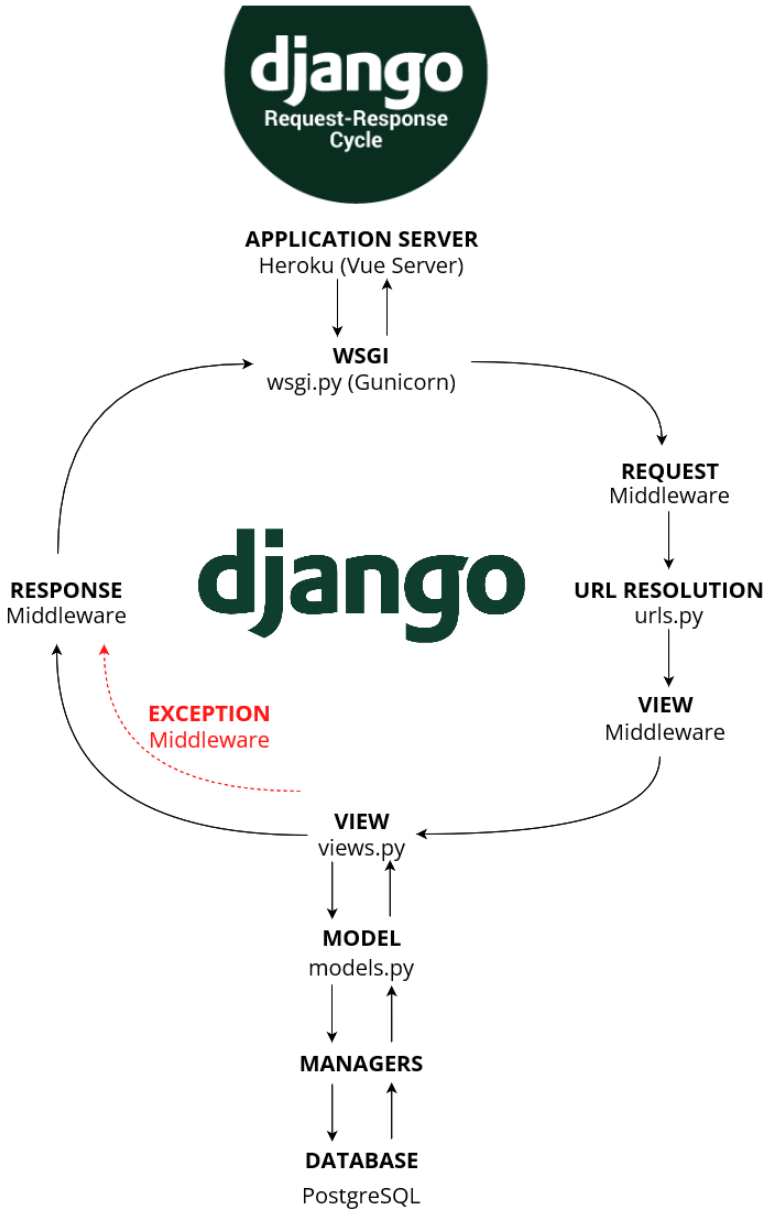

Figure 3: Django Request-Response Cycle

All the functionalities of the website are managed by different apps, these are the ones implemented:

- **app**: implements common functionalities that cover the entire application, it has one endpoint:
  - **farms_list**: handled by *FarmView* view, it manages GET requests by returning a list of farms
- **userApp**: implements the custom authentication system of the application, it contains all the models regarding the user. It has one endpoint that is used to check if a user is authenticated or not
- **helpRequestApp**: implements the functionalities about the help requests with two endpoints:
  - **help_request**: handled by *HelpRequests* view, it manages GET requests by returning the list of help requests and POST requests to reply to an existing requests
  - **help_request_by_id**: handled by *HelpRequestByID* view, it manages GET requests by returning a single help request associated to the unique id specified as a parameter
- **planningApp**: implements the functionalities about the daily plan with two endpoints:
  - **daily_plan**: handled by *DailyPlanView* view, it manages GET requests by returning the list of dates for the daily plan and POST requests to upload a new daily plan
  - **update_daily_plan**: handled by *UpdateVisits* view, it manages GET requests by returning the list of farmers in the current daily plan and POST requests to perform an update of an already existing daily plan
- **rankingApp**: implements the functionalities about the ranking with two endpoints:
  - **rank_farmers**: handled by *RankFarmers* view, it manages GET requests by returning the ranking as a list of farmers

  - **profile_info**: handled by *ProfileFarmers* view, it manages GET requests by returning the profile info of a selected farmer

- **reportApp**: implements the functionalities about the steering initiatives with two endpoints:
  - **steering_initiatives**: handled by *SteeringInitiativeView* view, it manages GET requests by returning the list of reports uploaded to the app and POST requests to upload a new report
  - **download_reports**: handled by *DownloadReport* view, it manages GET requests by returning a specific reports based on some parameters

- **sensorsApp**: implements the functionalities about the sensors (humidity and water irrigation) with two endpoints:
  - **humidity**: handled by *Humidity* view, it manages GET requests by returning a list with humidity and temperature values for each district
  - **water_irrigation**: handled by *WaterIrrigation* view, it manages GET requests by returning a list of water quantity values for each district

3.2 Frontend Structure

The front-end web application is contained into the dream_frontend folder of the IT directory. Of course, following the four tier architecture described in the Design Document, the front-end web application can also be deployed to a dedicated web server, which will then make requests to a different backend server. Here is represented the structure of the web app:

4 Testing

4.1 Backend Testing

5 Installation

5.1 Requirements

5.2 Backend Installation

5.3 Frontend Installation

6 Effort Spent

Student	Time for implementation
Ottavia Belotti	80h
Alessio Braccini	80h
Riccardo Izzo	80h