AY 2021/2022

Politecnico di Milano

Implementation Document

Ottavia Belotti   Alessio Braccini   Riccardo Izzo

Professor
Elisabetta Di Nitto

Version 1.0
January 30, 2022

Contents

1 Introduction **1**

1.1 Purpose 1

1.2 Definitions, Acronyms, Abbreviations 1

1.3 Revision History 2

1.4 References 2

2 Development **3**

2.1 Implemented Functionalities 3

2.2 Adopted Development Frameworks 4

2.2.1 Programming Language 4

2.2.2 Django Framework 5

2.2.3 Django REST Framework 6

2.2.4 Vue.js 6

2.3 API Integration 7

2.3.1 OpenWeatherMap API 7

2.4 DataBase 7

2.5 Heroku 7

3 Source Code **8**

3.1 Backend Structure 8

3.2 Frontend Structure 13

4 Testing **15**

4.1 Unit Testing 15

4.2 System Testing 17

4.3 Post-deployment Testing 18

5 Installation **18**

5.1 Requirements 18

5.2 Backend Installation 19

5.3 Frontend Installation 21

6 Effort Spent **21**

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`
- Heroku: `https://devcenter.heroku.com/categories/reference`
- PostgreSQL: `https://www.postgresql.org/docs/14/index.html`

2 Development

2.1 Implemented Functionalities

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

Policy Maker

- Farmers ranking in ascending/descending order of the whole region or grouped by district
- Visualization through graphs of data about humidity and water irrigation sensors
- Download agronomists' reports about steering initiatives

Agronomist

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

For the details about these functionalities please refer to the RASD document.

2.2 Adopted Development Frameworks

Model-View-Controller paradigm

Figure 1: Django Framework and REST Framework logo

Figure 2: Vue.js Framework logo

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 authorized users 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 a famous frontend JavaScript framework that allow the creation of user interfaces and single-page applications. 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 of the easy usability as it integrate in a single .vue file, also called components, the three main blocks of a web application paradigm.

The structure of a Vue file is intuitive, it is compose by three blocks:

- HTML: the part where the structure of the component is created.

- JavaScript: where is placed the interactive part, or better the logic, of the component.
- CSS: the part where is placed the style of the HTML structure, it can be scoped to the actual component or not.

In order to communicate with the backend we use the JavaScript library Axios that is a promise-based HTTP Client for Node.js. On the server-side it uses the native Node.js http module, while on the client it uses XML-HttpRequests. Axios killer feature is that it can transform in an automatic way the json reply that arrives from the server in JavaScript ready XML-HttpRequests.

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 place in the World. This return us not only the basics information but also more specific ones like pressure or wind speed and direction.

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. Furthermore it is supported excellently by Django. We decided to use PostgreSQL as Database system, Chiedo

2.5 Heroku

We choose to host our project in a server in order to make it reachable by everyone that have a browser and an internet connection. The hosting service we choose is Heroku, a platform as a service (PaaS) that enables developers to build, run, and operate applications entirely in the cloud.

We created two different server, one for the backend and one for the frontend. Heroku give also the possibility to register a personalized domain for reach the website We choose:

- `dreamapplication.herokuapp.com` as frontend domain
- `appdream.herokuapp.com` as backend domain

3 Source Code

3.1 Backend Structure

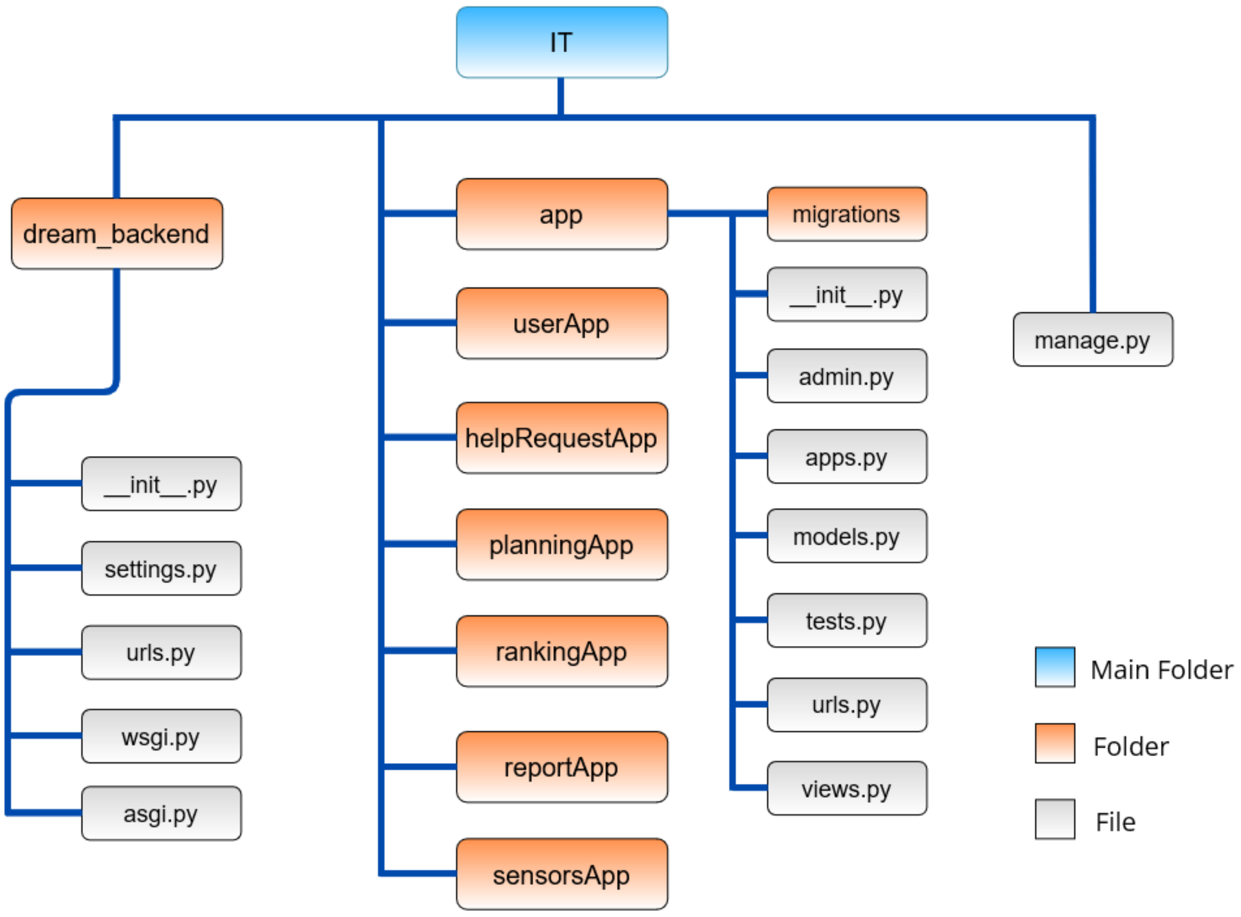

Figure 3: Backend structure

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

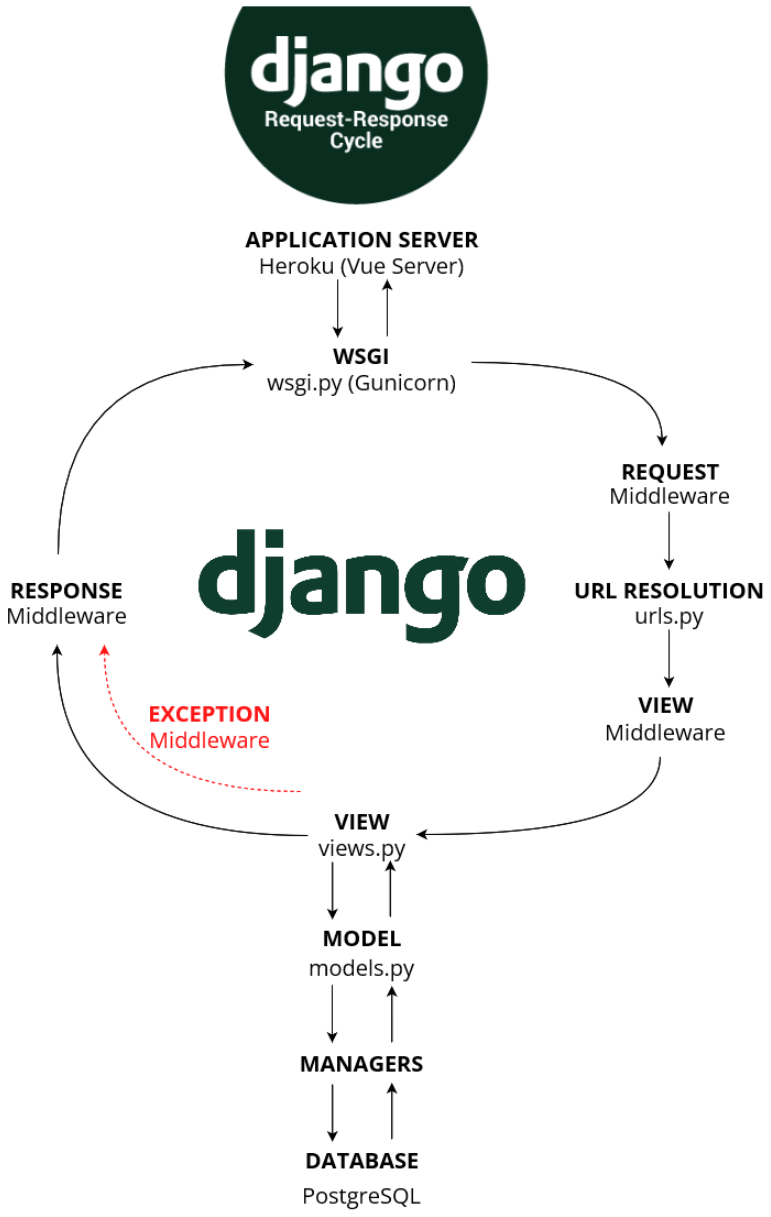

Figure 4: Django Request-Response Cycle

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 frontend web app is contained into the *dream_frontend/dream_ app* 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:

- **node_modules**: here are stored all the installed dependencies of the project listed in the *package.json* file. It's generated automatically every time the project is set up.
- **public**: in this folder is present the *index.html* file. This is the point where all the generated code will injected in order to displays it on the browser. During the building phase the files in this directory don't change.
- **src**: this is the main folder of the project. It is divided in several sub folder:

  - **assets**: it contains all the images needed by the front-end like the icons of the weather or the logo in the login page.
  - **router**: it contains the JavaScript file that list all the routes used to navigate the frontend.
  - **views**: it contains all the vue component needed by the frontend. They are divided in category, e.g. Policymaker or Agronomist, in order to better separate the different logic.

There are other several files needed by Vue to run the project. The most important are:

- **package.json**: it contains the list of all the dependency installed in the project.
- **server.js**: a simple JavaScript file needed by Heroku in order to start the web server and accept the incoming requests.
- **config.js**: it contains the url of the backend server. It can be set by the final user in order to specify if the server has to run local or online.

When the build process is finish the web ready files are put in the *dist* directory.

4 Testing

In this section we described how we tested the application following the general guidelines given in the Design Document. The testing process has been divided into unit testing and integration testing. We decided to test only the backend because most of the logic resides here. In any case, the system testing and the post-deployment testing phases also covered the functioning of the frontend.

4.1 Unit Testing

For each app in the back-end we wrote the test cases in tests.py file using **unittest**, a built-in module from the Python standard library.

helpRequestApp

- Attempt to get the list of help requests by a user through a GET request on *help_request* endpoint
- Attempt to get the list of help requests in case of an invalid user
- Attempt to reply to an existing help request through a POST request on *help_request* endpoint, verify the correct deletion of the old help request which is replaced by the reply
- Attempt to reply to an existing help request in case of an invalid user
- Attempt to get a single help request given the id through a POST request on *help_request_by_id* endpoint
- Attempt to get a single help request given the id in case of an invalid user

planningApp

- Attempt to upload a new daily plan through a POST request on *daily_plan* endpoint
- Attempt to upload a new daily plan with an invalid region
- Attempt to upload a new daily plan with an invalid date

- Attempt to upload a new daily plan with an duplicated farmer
- Attempt to upload a new daily plan with an invalid user (not an agronomist)
- Attempt to retrieve the daily plans through a GET request on *daily_plan* endpoint
- Attempt to retrieve only the daily plans associated to the current user
- Attempt to update a daily plan through a POST request on *update_daily_plan* endpoint
- Attempt to update an old daily plan
- Attempt to remove a daily plan
- Attempt to update a daily plan before creating it

rankingApp

- Attempt from an agronomist to get the ranking with "descending" order specified in the parameters through a GET request on *rank_farmers* endpoint
- Attempt from an agronomist to get the ranking with "ascending" order specified in the parameters
- Attempt from a policymaker to get the ranking with "descending" order and district among the parameters
- Attempt from a policymaker to get the ranking with "ascending" order and district among the parameters
- Attempt from a policymaker to get the ranking with different districts
- Attempt from a policymaker to get the ranking with "ascending" order but without the district among the parameters
- Attempt from a policymaker to get the ranking with "descending" order but without the district among the parameters

- Attempt to get informations about a farmer in the ranking through a GET request on *profile_info* endpoint
- Attempt to get informations about a farmer in the ranking in case of an invalid user

reportApp

- Attempt from an agronomist to upload a new report through a POST request on *steering_initiatives* endpoint
- Attempt from a policymaker to upload a new report
- Attempt to upload duplicated reports
- Attempt to get the list of reports through a GET request on *steering_initiatives* endpoint
- Attempt to get a report that has been deleted

sensorsApp

- Attempt to get data about humidity sensors through a GET request on *humidity* endpoint
- Attempt to get data about water irrigation sensors through a GET request on *water_irrigation* endpoint

There is a total of 32 tests with a percentage of success of 100%, this establishes the stability of the system. All the tests can be verified by running "python manage.py test" in the root directory.

4.2 System Testing

A part of the testing phase was also dedicated by manually testing the app through the web browser and with Postman, a tool for making HTTP GET/-POST requests with custom parameters. In this way we tested the authentication phase and all the functionalities provided by the system. All the tests have been performed successfully.

4.3 Post-deployment Testing

After the deploy the application has been extensively tested on a daily basis. This allowed to test the entire system and the interactions between the components both on front-end and back-end. We also invited a couple of testers in order to stress the app.

5 Installation

As a web application we chose to deploy it on Heroku, so a fully running version of the software is available at: `https://dreamapplication.herokuapp.com/`.

We strongly suggest you to use the deployed version instead of installing the software locally, since this would require some changes both in backend and frontend source code. Mind that, running the backend locally requires some adjustments to deal with static files. In fact, to be able to deploy the backend on a Cloud Application Platform such as Heroku, the storage of static files (i.e. Steering Initiatives reports uploaded by agronomists) has been managed through Google Drive.

However, if you wish to install it on your machine you can follow the guide below.

5.1 Requirements

Node.js is required in order to run the frontend, please install it if you don't have it on your device.

To run the backend, **Python 3.10** or a compatible version is mandatory. Moreover, `pip` command must be installed too, in order to resolve the project's dependencies.

- **Python and Pip**
  - Download Python 3.10 or equivalent from `https://www.python.org/downloads/`
  - Install latest version of Pip referencing `https://pip.pypa.io/en/stable/installation/`
- **Node.js**

5.2 Backend Installation

1) Download the latest zip archive from the Relase Page on the project's GitHub repository

2) Extract all the files in the same folder

3) Create a `.env` file in the same folder with the following fields and fill it:

   i. `SECRET_KEY` is the key used by Django to manage authentication and hashing messages. You can set your own.

   ii. `DATABASE_NAME`, `DATABASE_USER` and `DATABASE_PWD` are credential to access your local PostgreSQL database. If you wish to use a not local PostgreSQL database, you can also provide the `DATABASE_URL` variable.

   iii. `LOCAL_STATIC_FILES` can be set either to `True` or `False` to indicate if you want to store static files locally or on Google Drive.

   * If `True` (or not set), then the *Steering Initiatives* reports uploaded to the app will be stored in 'generated/reports'
   * If `False`, then the reports are going to be stored on Google Drive

   iv. `GOOGLE_DRIVE_STORAGE_JSON_KEY_FILE_CONTENTS` and `GOOGLE_DRIVE_STORAGE_SERVICE_MAIL` must be set if `LOCAL_STATIC_FILES=false`. This happens because a Google Drive API is going to be used to store static files. You have to:

   * Create a service account on a project in Google Developers Console: this will be your service email
   * Generate a secret service key a store it in a safe place. Copy and paste its content in the JSON Key variable

```
SECRET_KEY=your_backend_secret_key

DATABASE_NAME=your_database_name
DATABASE_USER=your_database_admin_username
DATABASE_PWD=your_database_password

LOCAL_STATIC_FILES=true/false
```

```

# if LOCAL_STATIC_FILE=false then the following
   variables must be set as well
GOOGLE_DRIVE_STORAGE_JSON_KEY_FILE_CONTENTS={
   your_JSON_key_file_content_for_Google_Drive_API}
GOOGLE_DRIVE_STORAGE_SERVICE_EMAIL=your_project.iam.
   gserviceaccount.com
```

4) Save `.env`

5a) If you are on a Unix-like system (Linux or MacOS), you can run the Makefile in the *dream_pkg* via `make run` command: this will install all the dependencies, update the database schema and start the backend server. However, if you wish to do that manually, you can follow the commands below:

```
# create virtual environment
dream_pkg:~$ python3 -m venv IT/venv

# activate venv
dream_pkg:~$ source IT/venv/bin/activate

# install dependencies
dream_pkg:~$ pip install -r IT/requirements.txt

# update database schema
dream_pkg:~$ python3 IT/manage.py makemigrations
dream_pkg:~$ python3 IT/manage.py migrate

# start backend server
dream_pkg:~$ python3 IT/manage.py runserver
```

5b) If you are on Windows, run the following commands:

```
# create a new virtual environment
C:\dream_pkg\> python -m venv c:\dream_pkg\IT\venv

# activate venv: 2 options
C:\dream_pkg\> IT\venv\Scripts\activate.bat  #for cmd
   .exe
```

```
PS C:\dream_pkg\> IT\venv\Scripts\Activate.ps1  #for
    PowerShell

# install dependencies
C:\dream_pkg\> pip install -r IT\requirements.txt

# update database schema
C:\dream_pkg\> python IT/manage.py makemigrations
C:\dream_pkg\> python IT/manage.py migrate

# start backend server
C:\dream_pkg\> python IT/manage.py runserver
```

5.3 Frontend Installation

Firstly you have to specify in the config.js file (contained in *IT/dream_frontend/dream_ application*) if you want to run the backend online or locally on your device. Inside the file you will find a configuration string and you have to insert the url of the chosen backend (by default is set to online backend but some choices are already available).

1 Open a terminal and move to *dream_ app* folder

2 Run `npm run build` command

3 Move to *dist* folder

4 Run `serve dist` command

5 Follow the instructions

6 Effort Spent

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