DREAM

Data-dRiven PrEdictive FArMing in Telangana

Implementation & Test Deliverable - ITD

POLITECNICO
MILANO 1863

Christian Grasso - Filippo Lazzati - Chiara Magri
Year: 2021/2022

Installation rules: `https://github.com/Chiara-Magri/GrassoLazzatiMagri`
Code: `https://github.com/Chiara-Magri/GrassoLazzatiMagri/tree/main/DREAM`

Contents

1 Introduction

The **Implementation & Test Deliverable** (ITD) is the document that contains implementation details of the system and notes about the tests performed to check the verification (that is, checking whether the implemented product is "right").

1.1 Purpose

The main goal of this document is to explain and motivate the reasons behind the choices performed in the implementation phase of `DREAM` and justify any possible difference between the out-and-out system and the system described in the *RASD* and *DD* documents[1]. Moreover, this ITD aims to show how the testing phase has been performed over the implemented `DREAM`, the procedure adopted and the main test cases along with their outcome.

1.2 Document structure

The structure of this document is organized in the following way:

- section 2 presents the requirements and functions introduced in the previous documents (*RASD* and *DD*) that have been actually implemented in the software (with motivations for including them and excluding others if applicable);
- section 3 is about the framework adopted: the reasons behind the choice and a brief overview of advantages and disadvantages;
- in sections 4 and 5 there is the presentation of the structure of the source code along with the entity-relationship model of the data;
- section 6 shows testing choices;
- sections 7 and 8 are about the effort spent by each member and the references.

[1] RASD = Requirement Analysis and Specification Document; DD = Design Document.

2 Requirements/Functions implemented

Since the software presented is a demo, we have decided to implement only the most important features of DREAM while ruling out others. In particular, we have decided to implement the features for farmers and agronomists and to avoid to implement the features for the policy makers because they are less suitable to the presentation of the demo.

2.1 Requirements implemented

The requirements we have decided to implement are shown in the following paragraphs.

Authentication This function has been implemented because it is fundamental for every web application to provide a service to authenticate the users. A user will never provide his data to a website that is not secure and does not recognize him.

Forum We have decided to implement the forum function because one of the main goals of DREAM is to put in touch various farmers to share information.

Suggestions We have decided to implement the suggestions functionality because for a farmer is helpful to have a suggestion about what to do, and in DREAM suggestions are important. It should be remarked that the suggestions provided are generated according to the algorithm presented in the *Design Document - 2.8* and that the training of the network has been done with a randomly generated dataset.

Requests and Replies Another main functionality that DREAM offers is that of providing help to those farmers that need it. Every time a farmer has some doubts or faces some problems, he can ask for help to an agronomist or a best-performing farmer through an help request. Agronomists and best-performing farmers can then answer through help replies.

Visualization of weather forecasts Every farmer and agronomist that needs to know the weather forecasts for the next days must be able to look at them through DREAM. This is one of the main goals of DREAM, therefore we have implemented it. The demo shows random forecasts generated for some Telangana's cities.

Insertion of production data Every farmer must provide production data that give information about their lands. This is central in DREAM, because this data is used by policy makers and agronomists to rank the farmers the to conduct analytics.

Daily plan As far as the agronomists' job is concerned, they need to schedule their visits to the farmers in an efficient way, that allows less-productive farmers to be visited more often than the others. `DREAM` allows to schedule their visits.

2.2 Requirements not implemented

On the other side, we have decided to do not implement the following main features of `DREAM`.

Ranking of farmers The ranking of the farmers conducted by the policy makers (and the agronomists) has not been implemented because we have considered it not too relevant for a demo of `DREAM`.

Analysis of initiatives This functionality allows the policy makers to understand whether the initiatives involving agronomists and best-performing farmers have a good impact on the work of the farmers. We have decided to avoid to implement it because, as previously said, policy makers' requirements are less suitable for the presentation of a demo of the system.

Sensors and water irrigation system These kinds of functionality are strictly related to the analytics function. The data that arrives from soil sensors and water irrigation system are mainly used in `DREAM` to provide accurate analytics and improved suggestions. Given the missing availability of the real sensors and that this kind of data can be easily added to the suggestions functionality without changing the logic of the code, we have opted for avoiding to implement it.

3 Adopted development framework

For our implementation, we used the `PHP` programming language and the Symfony development framework. The reasons for these choices are highlighted below.

3.1 Programming language

PHP `PHP` is the most used programming language worldwide for backend web development. It powers around 78% of all websites whose backend programming language is known[2].

The language itself is usually not considered a *modern* programming language, mainly because of some inconsistencies and lack of features such as a more solid typing system (albeit the language is still in active development and the latest versions are trying to fill its gaps).

However, as of today, the *ecosystem* around the language is still unmatched, with a large amount of solid frameworks and libraries which manage to improve the overall development experience and allow for very fast prototyping.

Python `Python` is the data science programming language. It allows to easily manage data structures for representing matrices and it is plenty of libraries and modules to work with data. Since for the "Suggestions" functionality of `DREAM` we have designed to exploit a neural network, we have decided to exploit some `Python` scripts for generating the datasets (clearly randomly, since we do not have real data) and for creating the neural network and for training it. It is not the best decision to adopt a second programming language, but it is better than programming a neural network in `PHP`.

3.2 Framework

The Symfony framework is an open source web application framework with a focus on simplicity and development speed. It is a very solid project with more than 15 years of development, and it is still being actively developed and improved (in our implementation we used version 6, released in November 2021).

Symfony follows many common architectural patterns, such as Model-View-Controller and Object Relational Mapping for database interaction (using the Doctrine[3] library) and provides developers with a standardized architecture to develop web apps in a streamlined way.

Symfony automates many common and repetitive tasks in web development, such as user authentication, routing, database interaction, data validation, cache management, localization, forms, emailing, testing and more. This allowed us to focus on the business logic of our application without having to worry about data correctness or security problems.

[2] https://w3techs.com/technologies/details/pl-php
[3] https://www.doctrine-project.org/

4 Structure of the code

The code in the repository is organized like every `Symfony` project. Inside the project directory *DREAM*, there are the following directories:

- **bin**, that contains the executable files;
- **config**, that contains the configuration files of all `Symfony` applications;
- **sample_datasets**, that contains a script to generate sample data for weather reports and forecasts (since sensor data reading has not been implemented). After running the generation script, these can be imported by using the commands `bin/console app:setup:populate-forecasts` and `bin/console app:setup:populate-reports`.
- **suggestions**, that contains training data for the neural network used for suggestions and the Python scripts to generate suggestions.
- **src**, where all the code is located;
- **templates**, that contains the *HTML* templates exploited for the implementation of `DREAM`; in `Symfony`, the templating language adopted is Twig;
- **tests**, where we have put all the unit tests performed;
- other non-relevant directories.

With regards to the **src** directory, it is organized in the following way:

- **Command**, that contains all the commands that we have developed to initialize the database. In particular, just for the goal of presenting a working demo of `DREAM`, we have placed in this folder a Python script for the generation of sample datasets to use to initialize the database;
- **Controller**, where all the controllers are implemented. According to the `Symfony` documentation, a *controller is a PHP function you create that reads information from the Request object and creates and returns a Response object. The response could be an HTML page, JSON, XML, a file download, a redirect, a 404 error or anything else. The controller runs whatever arbitrary logic your application needs to render the content of a page* (by extension, Controller also refers to the class containing such functions). Therefore, we have placed in this folder all the controller classes we have created that allow `DREAM` to provide its functions;
- **Entity**, that contains Object-Oriented representations of the tables in the database. As previously mentioned, `Symfony` exploits `Doctrine` to interact with relational (and not) databases in an Object-Oriented way. This is the place where all the classes modeling database entities live;
- **Form**, that contains all the `PHP` classes used to create forms and retrieve data from them. All these classes extend the *AbstractType* abstract class;

- **Forum**, that contains a service exploited by the forum functionality;
- **Repository**, where all the Repository classes are put. Again, according to the `Symfony` documentation, we have that a repository is *a PHP class whose only job is to help to fetch entities of a certain class*. Therefore, every Repository class contains helper methods that allow to query the database in a more Object-Oriented way;
- **Security**, that contains a service used to authenticate the user;
- **Twig**, utilities for frontend rendering.

5 Entity-Relationship model

We have decided to provide a conceptual model of the data through an Entity-Relationship diagram before actually create the database. The conceptual model of the data devised is:

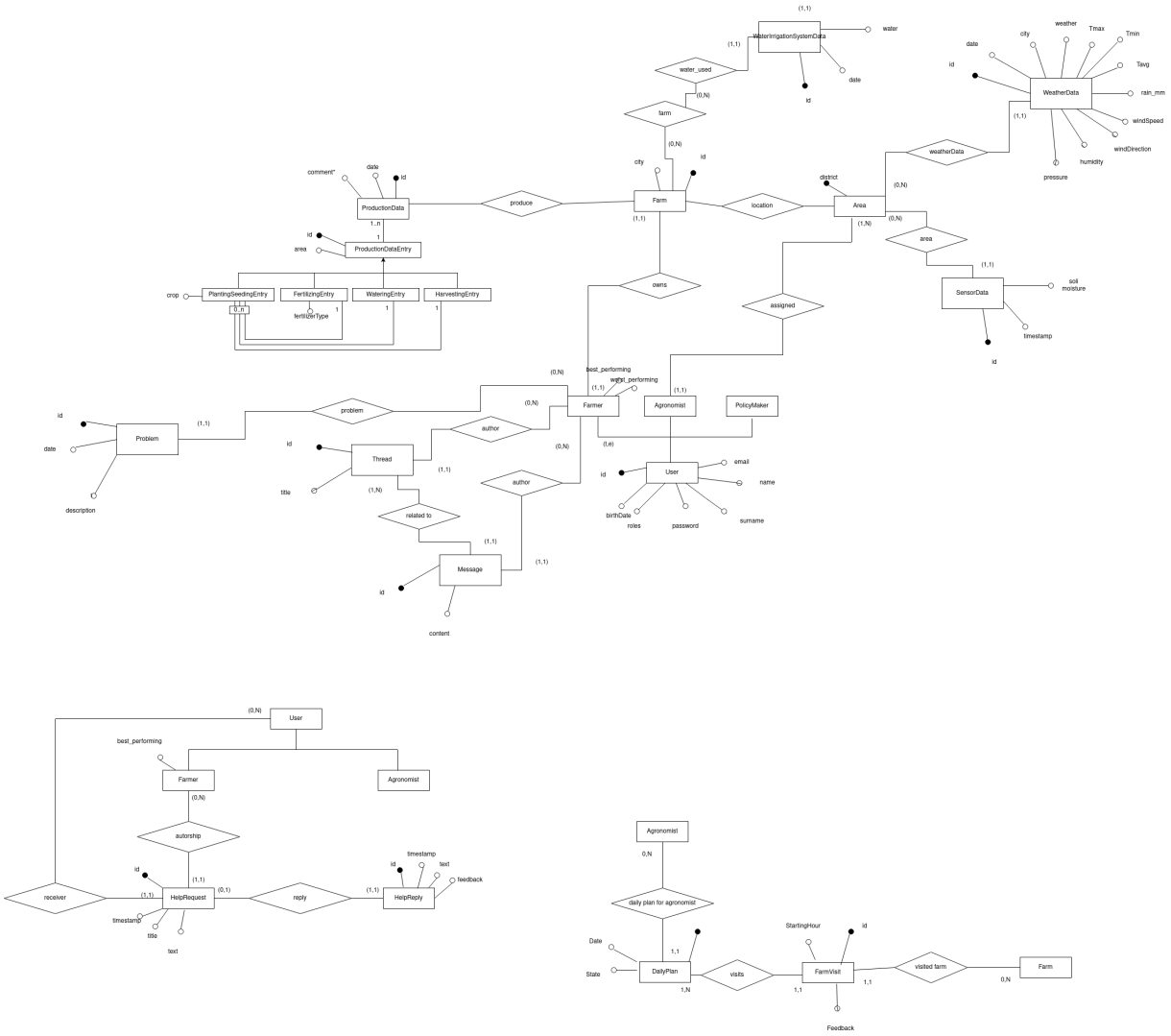

Figure 1: The Entity-Relationship model of DREAM.

The main entities of the diagram are:

- **User**, that represents a user of `DREAM` and is identified by an id; there

are three kinds of users: **Farmer**, **Agronomist** and **PolicyMaker** that inherit from **User**;

- **Thread** and **Message**, that are used to save forum data; a thread has a title and a message has a content; both of them are related to **Farmer** through an "author" relationship;
- **Problem**, that represents a problem faced by a **Farmer** and inserted into `DREAM`; it has a date and a description;
- **Farm**, which is owned by a farmer and stands in a city. Every farm receives **FarmVisit**s by an agronomist (described by date and starting hour) and uses some water; the use of water is saved through **WaterIrrigationSystemData** entity;
- **Area**, that represents a certain district of Telangana; every agronomist is assigned to a certain area and soil sensors data (**SensorData**) is assigned to a certain area;
- **WeatherData**, that represents both the weather reports and the weather forecasts.
- **ProductionData**, that represents a single insertion of data from the farmer into the system. A production data is made of various **ProductionDataEntry** (characterising a certain area), which is one of four different types: **PlantingSeedingEntry**, that says the crop planted, **FertilizingEntry**, that says the fertilizer used, **WateringEntry**, that says that the considered area has been watered, and **HarvestingEntry**, that says that the considered area has been entirely harvested;
- **HelpRequest** and **HelpReply**, that represent respectively the requests and the responses made/received by the farmers who need help;
- **DailyPlan**, characterised by a state and a date and related to a certain agronomist, which is composed of multiple **FarmVisit**s.

6 Testing

`Symfony`[4] exploits an independent library called PHPUnit as testing framework. Every test is a `PHP` class ending with "Test" that lives in the **tests/** directory of the project.
In `Symfony`, the following tests definitions are adopted:

- **unit tests**, to test individual units of source code (usual definition seen during the lectures);
- **integration tests**, to test combination of classes and commonly interact with `Symfony`'s service container;
- **application tests**, to test the behavior of a complete application. They make HTTP requests (both real and simulated ones) and test that the response is as expected.

These definitions are more or less (some differences occur) the same of those provided during the classes, except for **application tests** that we know as **system tests**.
The system has been tested following the test plan provided in the *Design Document*. In particular:

- as far as **integration tests** are concerned, as written in the *Design Document*, it is of minor importance for our system because every module is rather independent of the others, except for the `RankingService`, which requires the development of other services to be completed, but since we have decided to avoid the implementation of the `RankingService` because not considered one of the main services that `DREAM` has to offer, therefore we have not provided **integration tests**. Or better, we have provided two different **integration tests** according to the definition used by `Symfony` to test the "DailyPlanController". As a matter of fact, unit testing for the "DailyPlanController" is mandatory because it contains the whole logic for the generation of the daily plan, which is rather complex (it is explained at *Design Document - 2.8 Algorithms*). Therefore, we have written two *KernelTestCases*[5] to test its logic;
- with regards to **application tests**, they are the main kind of test we have adopted for testing our system. In `Symfony` there are three main types of **application tests**:
  1. *WebTestCase*, to run browser-like scenarios, but do not execute Javascript code;
  2. *ApiTestCase*, to run API-oriented scenarios;

[4]The Symfony official documentation about tests can be found at `https://symfony.com/doc/current/testing.html`.

[5]according to Symfony, they are integration tests, but according to the definition seen during the classes, they are unit tests.

3. *PantherTestCase*, to run e2e (end-to-end) scenarios, using a real-browser or HTTP client and a real web server.

It is clear that *ApiTestCase* are useless in our case and we avoid *PantherTestCase* because the amount of JavaScript in the pages is rather limited and the interaction between the user and the system is limited to some simple gestures, therefore *WebTestCase*s are enough. More specifically, we have written:

- *ForumTest*, to verify that the "ForumController" shows all the threads in the database with the proper content;
- *ForecastsResultsTest*, to check that "WeatherForecastsController" shows the proper forecasts (even though in the actual system these forecasts will not be taken from an internal database but from the **Telangana**'s website;
- *ProductionDataTest*, to check the correctness of the queries used to determine the currently planted crops;
- tests in the *DailyPlan* directory for daily plan generation and queries used in the daily plan section.

- The suggestion section, since its implementation is based on a neural network, cannot be tested in a rigorous way; however, we can use the value of the loss function as a metric for the *correctness* of the results. After training, this value settled at $1.505e-04$, meaning that suggestions are relevant:

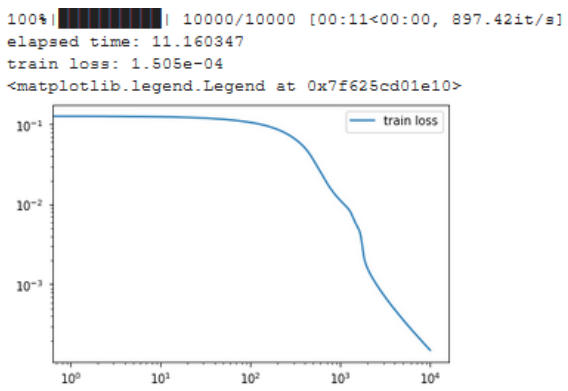

Figure 2: The history of the loss function used for the suggestions during training.

7 Installation instructions

In order to install DREAM, you need to have running on your pc both **Python** and **PHP** and also a database instance like **postgresql** or **mysql**.

7.1 Database installation

You can download **postgresql** at https://www.postgresql.org/download/ or, if you prefer **mysql**, you can download it at https://www.mysql.com/downloads/ or install it with a package manager of your choice.

7.2 Python installation

First of all, you can install python for instance from this website https://www.python.org/downloads/ and then you need to install the modules Numpy, Pandas and JAX, which can be installed through the following commands:

```
python3 -m pip install numpy
python3 -m pip install pandas
python3 -m pip install jax
```

or, simply, through:

```
pip install numpy
pip install pandas
pip install jax
```

if you have just one version of python installed (at least python 3 is required).

7.3 PHP, composer and Symfony installation

You need to install:

1. PHP 8.0.2 or higher, which can be downloaded at https://www.php.net/downloads.php, and these PHP extensions (which are installed and enabled by default in most PHP 8 installations): Ctype, iconv, PCRE, Session, SimpleXML, and Tokenizer;
2. Composer, which is used to install PHP packages;
3. you have also to install Symfony CLI; this creates a binary called symfony that provides all the tools you need to develop and run your Symfony application locally.

You are suggested to add both PHP and Symfony CLI to your *path* environment variable.

7.4 Start DREAM

Finally, you need to:

1. clone the repo `https://github.com/Chiara-Magri/GrassoLazzatiMagri` in a directory of your choice;
2. Enter the implementation directory `DREAM/`. All the following commands and actions must be performed inside this directory.
3. create a file **.env.local** which will be used for environment setup;
4. For email configuration, enter the following line inside the `.env.local` file (these are the credentials for the sample account we used for testing)

   ```
   MAILER_DSN="gmail+smtp://dream.sw.eng.2.project@gmail.com:dream-password@default"
   ```

   followed by the following line in case you have `POSTGRESQL` running on your pc (db_user and db_password are respectively the username and password of your database and 13 represents the version installed):

   ```
   DATABASE_URL="postgresql://db_user:db_password@127.0.0.1:5432/dream?serverVersion=13"
   ```

   or, in case you are using `MySQL` (same meaning for db_user and db_password):

   ```
   DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/dream?serverVersion=5.7"
   ```

   You also need to setup the path the Python 3 executable on your system:

   ```
   PYTHON_PATH=python3 # adapt for your system
   ```

5. You may also need to enable the appropriate database driver in the PHP configuration. To do so, find the file "php.ini" in the PHP installation folder, and remove the comment from the line corresponding to your database:

   ```
   ;extension=pdo_mysql
   ```

   that is, remove the semicolon ';' from it:

   ```
   extension=pdo_mysql
   ```

   (for Postgres, uncomment the line `extension=pdo_pgsql`).

6. make `Composer` install the project's dependencies into vendor/:

   ```
   composer install
   ```

7. initialize the database instance through the commands:

   ```
   php bin/console d:d:c % create database
   php bin/console d:s:u -f % synchronize schema
   ```

8. start the server:

```
symfony server:start
```

9. call the commands *app:setup:populate-database* and *app:setup:populate-areas* to populate the database with sample data (in particular Telangana areas, needed to create a farmer account):

   ```
   php bin/console app:setup:populate-database
   php bin/console app:setup:populate-areas
   ```

10. you can now open a browser and go to `localhost:8000/` to connect to `DREAM`.

7.5 Run tests

In order to be able to run the tests, you also need to follow the next steps (in the project directory `DREAM/`:

1. add the line that you previously added to your **.env.local** file:

   ```
   DATABASE_URL="..."
   ```

   to the file **.env.test.local** (that you need to create);

2. run the commands to initialize the test database (it will be called "dream_test"):

   ```
   php bin/console --env=test doctrine:database:create
   php bin/console --env=test doctrine:schema:create
   ```

3. you are now ready to run tests through the command:

   ```
   php ./vendor/bin/phpunit
   ```

   You can specifiy a test name after the command to run that specific test, e.g. `php ./vendor/bin/phpunit tests/Forum/ForumTest.php`.

8 Effort spent

Table 1: The time Christian Grasso has spent on the implementation part.

Effort spent	
Initial organization	2h
Authentication	5h
Forum	14h
Production data	32h
Document	4h
Test	9h
Final recap	6h

Table 2: The time Filippo Lazzati has spent on the implementation part.

Effort spent	
Initial organization	2h
Suggestions	33h
Weather forecasts	12h

Document	12h
Test	7h
Final recap	6h

Table 3: The time Chiara Magri has spent on the implementation part.

Effort spent	
Initial organization	2h
Daily plan	35h
Help request / reply	13h
Document	2h
Test	14h
Final recap	6h

9 References & Tools

- Symfony, the adopted framework;
- Doctrine, the `PHP` libraries for database storage and object mapping;
- draw.io: another tool to draw UML diagrams;
- Overleaf: Latex editor;
- Github: to share the code;
- Numpy, Pandas and JAX for implementing the suggestions;
- PHPStorm as IDE.