

**DREAM project by Federica
Tommasini & Alessandro Verosimile**



POLITECNICO
MILANO 1863

Implementation and Test Document

Deliverable:	ITD
Title:	Implementation and Test Document
Authors:	Federica Tommasini, Alessandro Verosimile
Version:	1.0
Date:	05-February-2022
Download page:	https://github.com/federicatommasini/TommasiniVerosimile.git
Copyright:	Copyright © 2022, Federica Tommasini and Alessandro Verosimile – All rights reserved

Contents

Table of Contents	3
List of Figures	4
List of Tables	4
1 Introduction	5
1.1 Purpose	5
1.2 Definitions, Acronyms, Abbreviations	5
1.3 Revision history	5
1.4 References	5
2 Development	7
2.1 Implemented functionalities	7
2.2 Adopted development frameworks	8
2.2.1 Application logic tier	8
2.2.2 Client tier	9
2.2.3 Data tier	12
3 Source code	13
3.1 Backend structure	13
3.2 Frontend structure	14
4 Testing	15
4.1 Application Server Testing	15
4.2 HTTP Requests Testing	16
4.3 Mobile Application Testing	17
4.4 Test of Communication between all components	17
5 Installation instructions	18
5.1 Java Installation	18
5.2 Maven Installation	18
5.3 MySQL Installation	18
5.4 Eclipse Installation	18
5.5 Xcode Installation	19
5.6 Mobile application and Server on different devices	19
6 Effort Spent	21
7 References	22

List of Figures

1	Swift DTO class	9
2	A section of the storyboard of DREAM.	10
3	Example of HTTP request in Swift.	11
4	Example of successful test in Spring.	16
5	Example of successful GET request from Postman.	17
6	Application properties file.	19
7	IPv4 on command prompt.	19
8	IP changes in Xcode.	20

List of Tables

1	Table of acronyms	5
2	Table of versions	5
3	Effort spent by student 1	21
4	Effort spent by student 1	21

1 Introduction

1.1 Purpose

The purpose of this document is to expose how the DREAM application, whose design has already been discussed in the RASD and DD documents, has been implemented and tested. In particular, there will be presented the main functionalities that the application provides and analyzed the decisions taken regarding the frameworks and programming languages that have been utilized. Furthermore, there will be described the whole implementation process and there will be defined several test cases that have been executed in order to assure the correct functioning of the application with respect to the requirements defined in the RASD document.

1.2 Definitions, Acronyms, Abbreviations

Acronym	Description
DD	Design Document
RASD	Requirement Analysis and Specification Document
API	Application Programming Interface
DBMS	Database Management System
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
IPv4	Internet Protocol version 4
DNS server	Domain Name System server
LLVM compiler	Low Level Virtual Machine compiler
Java EE	Java Enterprise Edition
AOP	Aspect Oriented Programming
DTO	Data Transfer Object
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
JPA	Java Persistence API
JDK	Java Development Kit
JRE	Java Runtime Environment
IDE	Integrated Development Environment

Table 1: Table of acronyms

1.3 Revision history

Version	Date	Description
1.0	05/02/2022	First version

Table 2: Table of versions

1.4 References

- I&T Assignment A.Y. 2021-2022
- Github repository Data4Policy <https://github.com/UNDP-India/Data4Policy.git>
- Git hub repository for Keychain Swift: <https://github.com/evgenyneu/keychain-swift>
- Apple developer documentation: <https://developer.apple.com/documentation>

- Spring framework documentation: <https://docs.spring.io/spring-framework/docs/current/reference/html/>
- Spring Data JPA - Reference Documentation <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference>

2 Development

2.1 Implemented functionalities

It has been decided to implement the functionalities related to the farmer's side of the application. In particular, these functionalities have already been defined in the RASD document and consist in:

- **Registration:** A farmer can register into the application inserting his personal information and other information related to his farm. If the email address inserted by the farmer is not already associated with an existing account, the registration process is successful and the user is redirected to the Home view.
- **Login:** A farmer can log into the application through his email and password, the same he inserted during the process of registration; if the credentials are correct he will be redirected to the Home view of the application. Here all the main functionalities of the application are visualized and, clicking on specific buttons, the farmer can access a specific one of them.
- **Visualize meteorological forecasts:** From the Home view, a farmer can access the section related to the meteorological forecasts and visualize the ones relevant to the location in which his farm is placed, accordingly to the date he selects.
- **Exchange suggestions:** From the Home view, a farmer can access the section related to suggestions: here he can visualize the ones previously posted in the system by other farmers, regarding the best practices to use in different situations, filtered by the geographical area, the utilized products or the environmental conditions. In addition, he can also post a suggestion in the system, in order for it to be visualized in a second moment by other farmers.
- **Visualize an evaluation:** From the Home view, a farmer can access the profile section. Here he can visualize the evaluation he received from a policy maker, expressed through the definition of a status, which indicates if he's working well or not, with respect to other farmers in similar conditions. In the profile section he can also visualize all his personal information and the ones related to his farm and update a personal profile image.
- **Update production quantities and used products:** From the profile section he can also access to two specific sections: in the first one he can update the list of the products he uses, in the second one he can update the product he cultivates and the related production quantities of the last year.
- **Request for help:** From the home view, a farmer can access the section dedicated to help requests. Here, he can easily request for help describing in detail the problems he is facing in a proper form. Such help request will be addressed to another farmer through the system.
- **Answer to help requests:** A farmer can visualize a request for help, which has been addressed to him by the system, in the message section and he can answer to it sending a response message.
- **Visualize messages from a policy maker:** In the message section, a farmer can also visualize the messages that policy makers send to him to give some indications or suggestions.
- **Discuss a topic:** From the Home view a farmer can access to the forum section in which he can either visualize or create a discussion forum about some topics of his interest. In such way, he can take inspiration from other farmers' opinions or write his own.

In order to provide all these features, the services described in the DD document that are related to these functionalities have been implemented. Their description will be deepened later. The only one that has been ignored is the Copy manager, the one in charge of retrieving data about meteorological forecasts, soil measurements, water consumption and agronomist information. Such service has been simulated through the generation of random values and the insertion of them into the database in order to test all the features that need the presence of such values.

2.2 Adopted development frameworks

For what concerns the application logic tier, it has been implemented through the Spring framework, which relies on the Java programming language. The Java programming language is widely used among developers because of its advantageous features: for instance, it is platform independent because it relies on the Java Virtual Machine to execute the bytecode, to which the source code is converted by the Java compiler. In addition the fact that it is an object-oriented language leads to an higher degree of code's reuse and to an higher modularity, which allow the developers to easy modify any part of the system. Finally, for the data tier, it has been utilized MySQL, which is a relational database management system.

2.2.1 Application logic tier

Spring framework

Spring is an open source framework that provides support for the implementation of Java EE applications. It is widely used as a framework because it allows the developers to focus on the logic of the application, handling the infrastructure. Spring consists of several modules that provide different features. These modules are grouped into five main components:

- Core container: it includes the bean factory and application context containers, which are in charge of managing the life cycle of the beans through dependency injection.
- Data access: which provides interfaces to map java objects with data retrieved from different kinds of data sources.
- Web: it contains the web-servlet module that include a model-view-controller implementation that can be used for the development of web applications.
- AOP (Aspect Oriented Programming): it is used in the Spring framework to provide declarative enterprise services, such as declarative transaction management.
- Test module: it supports the testing of Spring components with JUnit or TestNG

The components described above provide several libraries and annotations useful to manage the logic of the application server. The model view controller architecture included in the Spring framework consists in a dispatcher Servlet, which is in charge of receiving all the view's requests and dispatching them to the appropriate controller which, in turn, calls the business logic's services that satisfy those requests and processes a response exploiting the model objects.

In order to understand better how the logic of the application is handled by the application server, it is now presented a more detailed description of the system. First of all, the tables of a relational database can be mapped as java objects through the @Entity annotation; in such way, retrieving data from the database and saving them into it will be easier because it will consist in creating plain java objects and calling specific methods defined in appropriate repositories. These repositories are interfaces that extend default ones and that define methods that, through the @Query annotations, let the developers specify which data they need to retrieve from the database. Through the usage of these methods, there can be defined some java classes annotated as Services that will be in charge of processing data to provide the needed functionalities. These services will behave as intermediaries between the entities and the controllers. The latter are responsible for receiving the HTTP requests sent by the client tier and identifying them using an url path, categorizing them as GET, POST, PUT or DELETE requests through the @RequestMapping annotation and analyzing the parameters attached to such requests. The parameters can be of two types: simple variables passed through url paths or data transfer objects (DTO), which are encoded in JSON format and allow the communication between client and server. These DTO objects are defined in the Model package as java objects and map the entities to the data needed by the user interfaces to properly provide the requested functionalities to the user.

2.2.2 Client tier

As already mentioned, the client side has been developed as an iOS application. The tool used for the implementation and testing is Xcode that uses Swift and Objective-C as programming languages and the Storyboard technology for the implementation of the User interfaces' graphic. Using this technology the various components of the mobile application are divided into some groups:

- **view controllers:** they are in charge of manage the user interfaces, the data which populate them and the communication of these data between the various user interfaces. They also manage the actions to be executed when a button is pressed or a text field is edited and so on. Furthermore, they send all the http requests (GET, POST, PUT, DELETE), to the application server in all the situations in which the mobile application needs some information from the server or has to upload some information in the database. These requests are managed by a library of functions, written in swift, that allow every type of request with or without a response from the server. A more detailed argumentation about this topic will be discussed in a later section about HTTP requests.
- **entity structures:** for each entity that is useful for the realization of the mobile application, it has been created a relative structure with all necessary attributes. All these structures are DTO elements that, in swift, are "codable" in order to be encoded in JSON. Also this topic will be detailed in a later section about HTTP requests.

```
struct FarmerDTO: Codable{
    var id: Int!
    var email: String
    var name: String
    var surname: String
    var status: String
    var pw: String
    var farm: FarmDTO!

    init (email: String, name: String, surname: String, status: String, pw: String){
        self.email = email
        self.name = name
        self.surname = surname
        self.status = status
        self.pw = pw
    }
}
```

Figure 1: Swift DTO class

- **graphic components:** some prototype elements, such as cells, defined in order to populate some structures of an indefinite number of elements, such as tables or collections, in a coherent way.
- **Info.plist file:** a file xml in which all the configurations and the permissions of the application are explicated. In particular they consists in: the authorization for the specific IP address of the device on which the server is placed (it sometimes changes, so it is necessary to update it when it happens); the permission for accessing to camera and photo library on the devices in which the application will be downloaded and run.
- **Main.storyboard:** the storyboard on which all the user interfaces are designed and all the related graphic components are created. The storyboard will be then translated as a xml file.

Storyboard

The Storyboard technology is one of the two alternatives way of coding in Xcode for developing iOS applications. With respect to SwiftUI, it allows to create and manage the user interfaces in a faster and more intuitive way. All the graphic elements (button, label and so on) have to be dragged and dropped in the related view controllers to create the related variables that will be managed by such controllers or to create some handler associated to specific actions on that elements. In this way it is possible to change the values of these components with the data retrieved from the server or obtained by previous interfaces, and perform some actions in order to send request to the server or visualize some alerts to manage some unexpected situations or simply to move among the various interfaces.

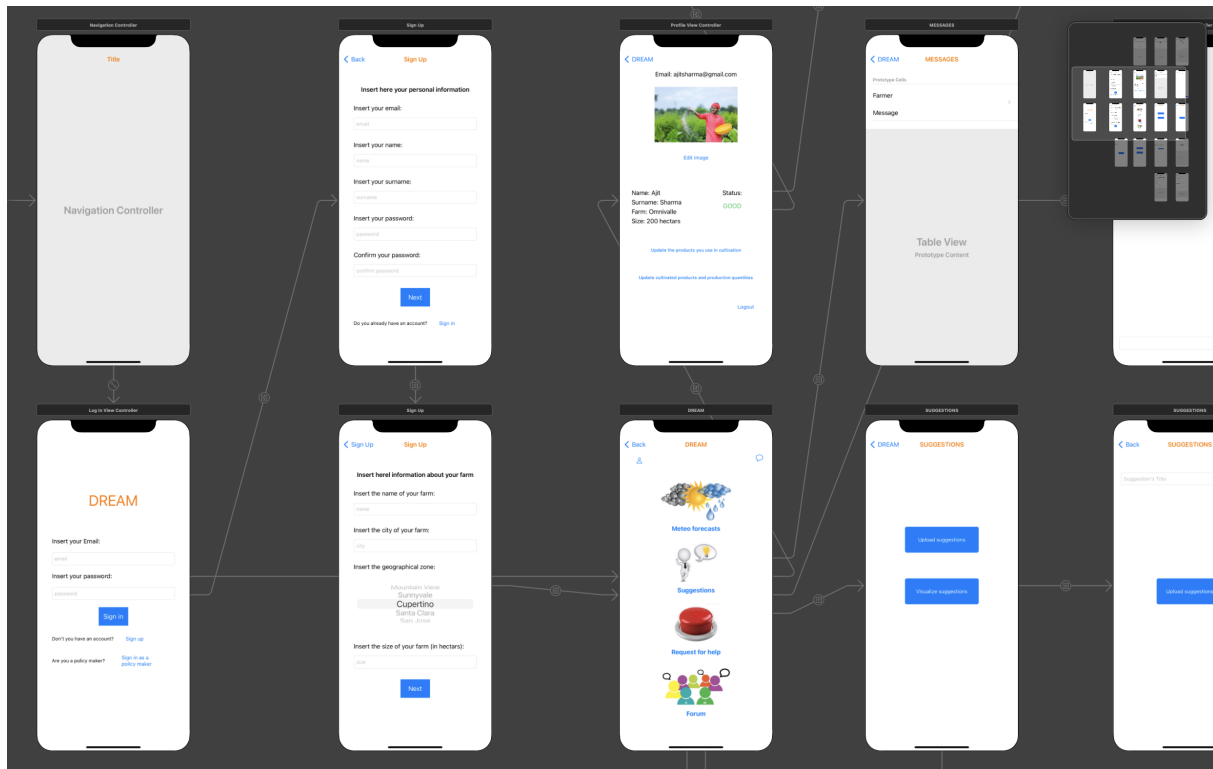


Figure 2: A section of the storyboard of DREAM.

Keychain

Keychain represents a simple way to save sensitive data in local memory. It is a sort of database in which all data are encrypted and it is principally used to save passwords. Keychain works through the Swift framework Security and, in DREAM application it has also been used a Github repository which simplifies the operations that involve this tool. Through this repository the elements that have to be saved with Keychain can be stored and retrieved with simple get and set functions specifying the key and the type (that can be string or bool) of the element to store or retrieve. In DREAM mobile application it has been used in order to simplify and speed up the process of login if the user chooses to save his credentials. The last email and password used for logging into the application are saved in local memory (if the user gives the authorization for doing it) and at the next opening of the application he can choose to log in faster using them with an auto-complete of the text fields.

HTTP Requests

The whole communication with the server is managed by a library of functions which perform all the actions needed to send and receive the data in a correct and secure way. In order to send and receive data, they, as already described, have to be encoded in JSON and, therefore, all the structures of the

mobile application have to be DTO objects which are "codable". The name of the classes in the mobile application, the name and the types of the variables in the classes have to correspond to the ones in the model of the application service, in order to assure that such JSON data can be sent and received in a correct way. The library who manages the HTTP requests contains functions for all the possible methods (GET, POST, PUT, DELETE).

- **GET request:** In a GET request it is necessary to create the URL concatenating a string containing "http://", the IP address and the port number and another string containing the path that allows the application server to identify what is requested. After the set of the timeout interval of the request, the set of the method to "GET" and the initialization of a session the request can be sent. When the response is received, the data have to be decoded from JSON and, if the decoding ends with success, a completion function can be executed. Such function is asynchronous with respect to the thread that calls the get request and allows to retrieve the decoded values and make them available for the controllers of the mobile application.
- **POST and PUT request:** In POST and PUT requests most of the actions that have to be executed are similar to a get request. However, before sending the request, in POST and PUT requests the body of the request has to be encoded in JSON. After that, the request can be sent. There are two types of functions for POST and PUT requests, one for the case in which it is needed a response, and one for the case in which it is not needed.
- **DELETE request:** In a DELETE request it is needed only to create the URL, set timeout interval of the request, set his method to "DELETE" and it can be sent. No response is needed because no data has to be retrieved in this case.

```
private static func postOrPutJSONRequestWithResponse<U : Codable, T : Codable>(_ path : NSString, _ httpBody : U?, responseType : T.Type, _ mode : String,
completionFunction : @escaping (T?, Int) -> Void) {
    guard let url = NSURL(string: "\($ip)\($path)") as URL? else {return}
    let dateFormatter = DateFormatter()
    dateFormatter.dateFormat = "yyyy-MM-dd"
    let encoder = JSONEncoder()
    encoder.dateEncodingStrategy = .formatted(dateFormatter)

    let request = NSMutableURLRequest(url: url)
    request.timeoutInterval = 10
    request.httpMethod = mode

    if let httpBodyData = httpBody {
        do {
            let requestData = try encoder.encode(httpBodyData)
            request.httpBody = requestData
            request.addValue("application/json", forHTTPHeaderField: "Content-Type")
        } catch {
            print("Error encoding JSON")
            return
        }
    }

    let configuration = URLSessionConfiguration.default
    let session = URLSession(configuration: configuration)
    let dataTask = session.dataTask(with: request as URLRequest, completionHandler: { (data, response, error) -> Void in
        if let e = error {
            print(e)
        } else {
            guard let responseCode = (response as? HTTPURLResponse)?.statusCode else {return}
            guard let httpData = data else {return}
            do {
                let decoder = JSONDecoder()
                let dateFormatter = DateFormatter()
                dateFormatter.dateFormat = "yyyy-MM-dd"
                decoder.dateDecodingStrategy = .formatted(dateFormatter)
                let data = try decoder.decode(responseType, from: httpData)

                DispatchQueue.main.async {
                    completionFunction(data, responseCode)
                }
            } catch {
                DispatchQueue.main.async {
                    completionFunction(nil, responseCode)
                }
            }
        }
    })
    dataTask.resume()
}
```

Figure 3: Example of HTTP request in Swift.

2.2.3 Data tier

The data tier has been realized in MySQL using the tool MySQL Workbench. The application tier retrieves and saves all data through it. All the configuration properties regarding the MySQL database are specified in the application.properties file of the application server.

3 Source code

All the source code implemented for the DREAM project can be found in the github repository at the link: <https://github.com/federicatommardini/TommasiniVerosimile/tree/main/DeliveryFolder/ITD>.

3.1 Backend structure

For what concerns the backend code it has been used the structure of a Maven project, in which the java files are grouped accordingly to their functionalities in different packages. The source main folder contains the following ones:

- **com.TommasiniVerosimile.Dream:** it contains the main class of the Spring application, annotated as `@SpringBootApplication`, which, in turn, contains the main method needed to properly launch the project.
- **com.TommasiniVerosimile.Dream.bean:** it contains the java classes that define the entities that are mapped to the database's objects through the JPA APIs. These classes make usage of the annotations: `@OneToOne`, `@OneToMany`, `@ManyToOne` and `@ManyToMany` in order to define the relationships between the java objects that represent the relationships between the database's tables.
- **com.TommasiniVerosimile.Dream.controller:** it contains the java classes (annotated as `@RestController`) that define the restful controllers that handle the client's requests.
- **com.TommasiniVerosimile.Dream.modelDTO:** it contains the java classes that define the data transfer objects mapping them to the entities of the application, in order to use such objects in the communication with the client tier.
- **com.TommasiniVerosimile.Dream.repository:** it contains the interfaces that extend JPA repositories, which provide already a lot of useful features, defining additional query methods for the communication with the specific database of the application.
- **com.TommasiniVerosimile.Dream.service:** it contains the java classes that represent the business logic of the application. These classes are annotated as `@Service` and are called by the controllers in order to process data from or directed to the client tier.
- **com.TommasiniVerosimile.Dream.config:** it contains the class "DBPopulation" that is in charge of initialize the database inserting into it some dummy data for farmers, meteorological forecasts of the last month and the next 15 days, suggestions and so on. In such way it is possible to test the application logging into it with the data of one of the inserted farmer and check the correct functioning of the application in all his sections. The initialization takes place every time the server is restarted and all modifies executed through the mobile application are forgotten.

There is also another folder containing java files, namely the source test folder which contains all the java files needed to perform unit and integration testing that will be properly addressed in a later section of this document.

In addition there is a folder dedicated to resources, which contains an "application.properties" file that specifies some useful information for the configuration of the database connection and others regarding the configuration of the connection with the client, defining the IP address of the server and the port used to run the application process.

Finally there is a "pom.xml" file which specifies all the maven dependencies of the project besides other standard configuration properties.

3.2 Frontend structure

For what concerns the frontend code, it has been divided in folders. The principal folder is named "DREAM" and contains several files:

- **Main.storyboard:** the file in which all the user interfaces are designed. When the application is executed it will be translated in a xml file.
- **AppDelegate:** The AppDelegate is effectively the root object of the app, and it works in conjunction with UIApplication to manage some interactions with the system. It initializes your app's central data structures, configures the app's scenes, responds to notifications originated from outside the app and responds to events that target the app itself.
- **SceneDelegate:** The SceneDelegate is used to manage life-cycle events in one instance of your app's user interface. This interface defines methods for responding to state transitions that affect the scene, including when the scene enters the foreground and becomes active, and when it enters the background.
- **Info.plist:** This is an information property list file which contains a collection of key-value pairs that specifies how the system should interpret the associated bundle. Some key-value pairs characterize the bundle itself, while others configure the app, frameworks, or other entities that the bundle represents. Some keys are required, while others are specific to particular features of the executable.
- **Assets:** In the assets, images of various dimensions for the application's logo are requested.
- **Launchscreen.storyboard:** It is a storyboard file with only one screen. It represents the interface that is showed at the beginning of the application for few seconds.
- **ViewController folder:** It is a folder in which there are all the view controllers of the mobile application.
- **Classes folder:** It is a folder in which there are all the DTO classes of the mobile application.
- **ViewCell folder:** It is a folder in which there are all the classes of the cells concerning tables and any other graphic structures. In this classes all the components of such cells are defined.
- **Util folder:** It is a folder which contains two libraries, one in charge of managing all the HTTP requests and one, taken from the Github repository <https://github.com/evgenyneu/keychain-swift>, in charge of managing the storing of sensitive data through Keychain.
- **Images folder:** It is a folder containing all the static images used in the image views of the application.

There are also other two folders:

- **DREAMTests:** In this folder all the tests applied in order to verify the consistence of the application components are included. Their description will be detailed in the next chapter.
- **DREAMUITests:** In this folder all the tests applied in order to verify the consistence of the actions regarding the interface elements of the mobile application are included. Their description will be detailed in the next chapter.

4 Testing

4.1 Application Server Testing

All the unit and integration tests are executed in a coherent way with respect to the plan exposed in the DD document. They are performed through JUNIT5, one of the most popular unit-testing frameworks in the Java ecosystem. Once the correct interaction between the model and the database has been tested, all the other tests are related to the controllers that are the ones who manage all the actions that can be performed on the server. Through the testing of the controllers it is possible to test the correct work also of all the services and repositories. Firstly, some unit tests for all the methods of the controllers which have no dependencies to other methods are executed.

- **DiscussionForumControllerTest:** For this controller two tests are performed: the "discussionForumTest", that checks the correct insertion of a discussion forum in the database and the correct fetch of it; the "forumPostTest", that checks the correct insertion of some posts written by a farmer on a specific discussion forum.
- **FarmerLoginControllerTest:** For this controller three tests are performed: the "registrationTest", that checks the correct insertion in the database of a farmer who registers; the "checkRegCredentialsTest", that checks the correct fetching of all the emails and the searching among them of a given one, as it happens during the process of registration; the "loginTest", that checks the correct fetching of all emails and passwords and the search among them of a given couple, as it happens in the login process.
- **MessageControllerTest:** For this controller two tests are performed: the "getAllMessagesTest", that fetches all the messages received from a given farmer and checks if the id of the receiver is equal to the one of the farmer for all the messages; the "addMessageTest", that uploads a message from a farmer to another one and then checks if it is present in the list of messages received from the second farmer.
- **InformationControllerTest:** For this controller five tests are performed: the "getMeteoForecastTest", that, after retrieving all the meteorological forecasts for a certain city in a certain date, checks if they really refer to that city and that date; the "updateUsedProductTest", that adds a used product to a given farmer and then checks if there is in the database a product with the same name associated to that farmer; the "deleteUsedProductTest", that deletes a used product related to a given farmer and then checks if there is any product in the list associated to that farmer with the same name; the "addProductionForAFarmerTest", that adds a production quantity to a given farmer and then checks if in the database there is a product quantity related to that farmer with the same product and the same quantity; the "deleteProductForAFarmer", that deletes a product quantity related to a product for a given farmer and then checks if there is any product quantity in the list associated to that farmer related to the same product.
- **SuggestionControllerTest:** For this controller two tests are performed: the "addSuggestionsTest", that uploads a suggestion written by a given farmer and then, checks if in the database there is a suggestion with the same text uploaded by the same farmer; the "getSuggestionsTest" that, for a given farmer, retrieves the suggestions filtered firstly by geographical zone, then by used products and, for each suggestion, checks if it is really uploaded by a farmer which, respectively, has the same geographical zone and at least one used product in common.

Then, some integration tests are executed in the same order already exposed in the DD document. For every service, the related controller is tested with the ones with which has some dependencies. For example a relevant one is the **HelpRequestFormControllerTest** that has not been executed yet. The HelpRequestFormController needs to be tested with the MessageController. For what concerns the HelpRequestFormControllerTest one test is performed, the "addRequestFormTest", that uploads an help

request of a farmer to the database and checks if an help request with the same text from the same farmer is found in the database. At the end of all these integration tests a unique test involving all the previous ones has been executed and no errors are detected for the application server.

Finished after 9,754 seconds

Runs: 5/5

Errors: 0

Failures: 0

InformationControllerTest [Runner: JUnit 5] (0,629 s)

deleteUsedProductTest() (0,457 s)

deleteProductForAFarmerTest() (0,052 s)

getMeteoForecastTest() (0,039 s)

updateUsedProductTest() (0,034 s)

addProductionForAFarmerTest() (0,042 s)

Failure Trace

Figure 4: Example of successful test in Spring.

4.2 HTTP Requests Testing

After checking the correct functioning of the whole application server with some dummy parameters, some tests with parameters sent by means of HTTP Requests are necessary in order to check the proper work of the controllers when they receive the path and the body of a request. In order to perform this tests Postman has been used. Postman is an application used for API testing. It is an HTTP client that tests HTTP requests, utilizing a graphical user interface, through which we obtain different types of responses that need to be subsequently validated. Using Postman it has been checked if every method of the controllers of the application server that receives an HTTP request works properly. Postman allows to visualize the response of the server in JSON format, the same that the mobile application will receive and decode. For every method which returns a response, the validity of it has been guaranteed through several attempts. For every method which does not return a response, the proper functioning of it has been checked directly on the database.

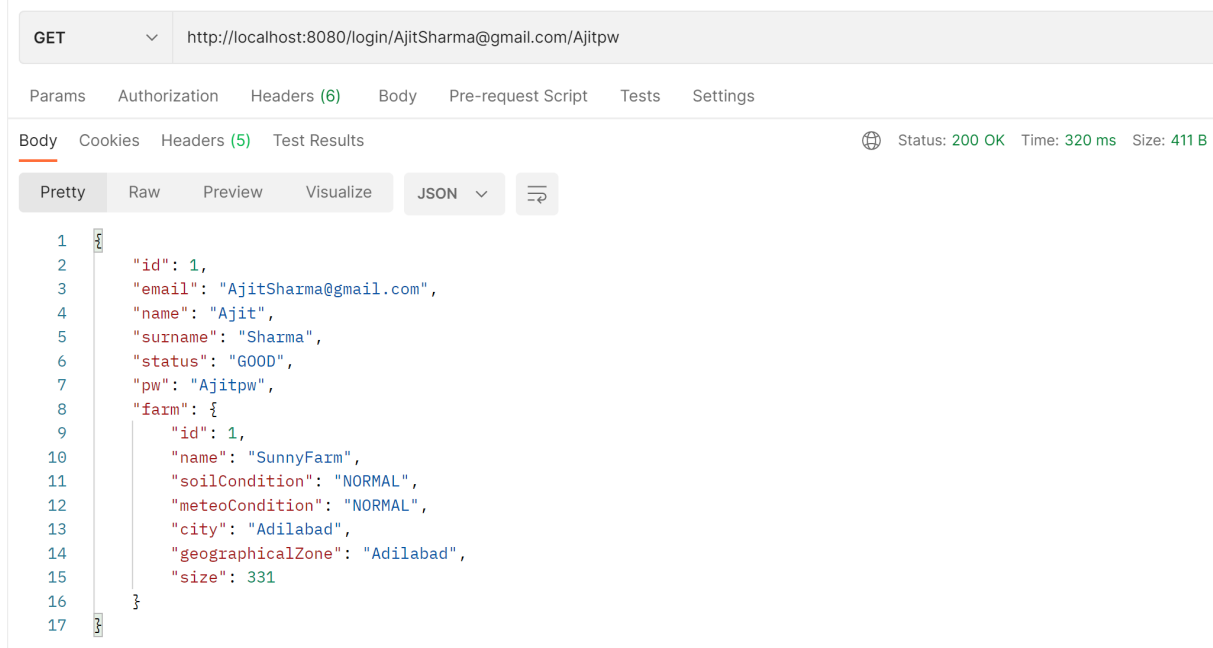


Figure 5: Example of successful GET request from Postman.

4.3 Mobile Application Testing

For what concerns the mobile application, several tests are executed during all the process of implementation substituting the effective values of the database with some dummy parameters. The proper work of every controller and the correctness of the transitions between views and of the passage of parameters between different controllers have been checked. Also in this case all the checks that have to be executed on some transitions by means of some requests to the server are simulated. Furthermore, some tests about the correct storing of email and password in the Keychain store are performed in order to guarantee the correctness of auto-complete function in the login view. When the correctness of all the functionalities of the mobile application is assured, it is possible to continue with the testing of all the components of our system together.

4.4 Test of Communication between all components

When all the components of the system are individually tested, it is possible to perform a complete test of the whole system. The dummy parameters of the mobile application are now substituted with the values retrieved from the application server through GET HTTP requests, the values uploaded by the farmers (help requests, messages, posts on a forum, values of production quantities and so on) are uploaded on the application server through POST HTTP Requests and the values deleted by the farmers (for example some products he does not use anymore) are deleted in the database by the application server through DELETE HTTP requests. All the possible actions which can be executed in the mobile application are tested several times in order to discover possible bugs and the coherence of what is shown in the application with respect to the values stored in the database has been checked. Three videos of the whole functioning of the DREAM application, one for a good farmer, one for a bad farmer and one for the registration process, can be found at the following Github link:

<https://github.com/federicatommassini/TommasiniVerosimile/tree/main/DeliveryFolder/ITD/ApplicationDemo>

At the end of this process the mobile application has been distributed to a substantial number of people in charge of testing it and communicating if any bugs are found.

5 Installation instructions

Since the mobile application has been developed only for iOS platform, in order to test the application it is needed that the one who wants to download it uses a device with macOS. Consequently the installation's instructions are exposed only for macOS both for the application server and the mobile application. If someone prefers to run the application server on a different device with respect to the macOS device on which the mobile application runs, has to download Eclipse and MySQL Workbench for this other device. Some settings of the mobile application and application server have to be changed in this situation. These will be explained later.

5.1 Java Installation

In order to run the project it is required to have Java installed on the device you intend to use. The java version which has been used to test the project is Java 16. You can download such version of the Oracle JDK and JRE at the following link: <https://www.oracle.com/java/technologies/javase/jdk16-archive-downloads.html>.

5.2 Maven Installation

The installation of Maven it is required to import the project on your IDE. The version of Maven which has been used to create and manage the project is apache-Maven 3.8.4, which can be downloaded from the following link: <https://maven.apache.org/download.cgi>.

5.3 MySQL Installation

As already mentioned, the DBMS which has been chosen to utilize is MySQL; therefore, in order to run the application, the installation of the MySQL server and MySQL Workbench is needed. If you are using MacOS you can follow the links: <http://dev.mysql.com/downloads/mysql/>, <http://dev.mysql.com/downloads/workbench/> While, if you are using Windows you can follow the link: <https://dev.mysql.com/downloads/installer>. During the installation process will be required the insertion of a password for the root user, which you will have to remember because it will be needed later to access MySQL tools.

Once you have completed the installation, you have to open MySQL Workbench and create a database named "dream", through the command **"create database dream;"**. Such database will be automatically populated with the first execution of the java application. If you don't want to re-initialize the database at the successive runs and keep all the changes of data, you have to update a configuration of the database: in the application.properties file, at the line 7, substitute "create" with "update".

5.4 Eclipse Installation

The IDE that has been used and that is recommended is Eclipse, but other Java IDEs that support Maven projects can be used. The Eclipse IDE for Enterprise Java and Web Developers can be downloaded from the following link: <https://www.eclipse.org/downloads/packages/>. Once you have downloaded the .zip or .tar.gz file, you have to extract its content and click on the Eclipse executable file. If you are using MacOS and you have downloaded the .dmg file, you have to proceed opening the file and then, in the appearing window, moving the Eclipse executable to the Applications folder and open it in order to continue the installation.

Once you have installed your chosen IDE, you have to import the project that you find in the github repository at the following link: <https://github.com/federicatommadini/TommadiniVerosimile/blob/main/DeliveryFolder/ITD/DreamServer.zip>. Once you have imported it, you will have to change the two application properties files, that you can find one in the resources folder and one in the src/main/resources folder, inserting the password of the root user of your DBMS. Then, you will be ready to execute the java application right clicking on the DreamApplication.java file in the com.TommadiniVerosimile.Dream package.

5.5 Xcode Installation

Download the Xcode project from this Github link: <https://github.com/federicatommasini/TommasiniVerosimile/blob/main/DeliveryFolder/ITD/DREAMApplication.zip>. Then, open Xcode (if it is not already installed, do it from App store) and, on the top bar, click on **file -> open** and choose the project downloaded. When the project is opened on Xcode, on the top of the Xcode page, choose the device on which you want to run the application through the simulator (it is suggested to run it on iPhone 11, the one that has been used for the testing). Alternatively, if you have an iPhone, you can connect the iPhone to the Mac through an USB-C-Lightning cable. Then, on your iPhone, go to settings -> general -> VPN Device Management. Here you find a DEVELOPER APP. Click on it and give the authorization. Now you can run from your Mac the application and test it on your iPhone.

If you followed these instructions and you have the mobile application, the application server and the MySQL database installed on the same device, you can test the whole system.

5.6 Mobile application and Server on different devices

Instead, if you choose to run the application server on a different device, there are some changes to perform in order to allow the communication between them. Firstly, it is important to highlight that, in order to avoid long and difficult configurations, the two devices have to be connected to the same LAN. Once checked this, on the DREAM project on Eclipse (or any other IDE you choose to utilize), go to the application.properties file, uncomment the lines 11 and 12 and substitute the IP address with your private IPv4 address.

```
1 spring.datasource.driver-class-name=com.mysql.jdbc.Driver
2 spring.datasource.url=jdbc:mysql://localhost:3306/dream
3 spring.datasource.username=root
4 spring.datasource.password= yourPassword
5 spring.jsp.show-sql=true
6 spring.jpa.generate-ddl=true
7 spring.jpa.hibernate.ddl-auto=update
8 spring.jpa.database-platform = org.hibernate.dialect.MySQL8Dialect
9 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect
10 spring.jpa.properties.hibernate.globally_quoted_identifiers=true
11 #server.address=192.168.31.39
12 #http://: http://192.168.31.39:8080
```

Figure 6: Application properties file.

You can find it writing on the command prompt:

- **ipconfig**

you find the IP address in the section "Wireless Lan adapter Wi-fi", IPv4 address.

```
Wireless LAN adapter Wi-Fi:

Connection-specific DNS Suffix  . : 
Link-local IPv6 Address . . . . . : fe80::747c:fa4e:c3b6:5be6%4
IPv4 Address. . . . . : 192.168.8.103
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.8.1
```

Figure 7: IPv4 on command prompt.

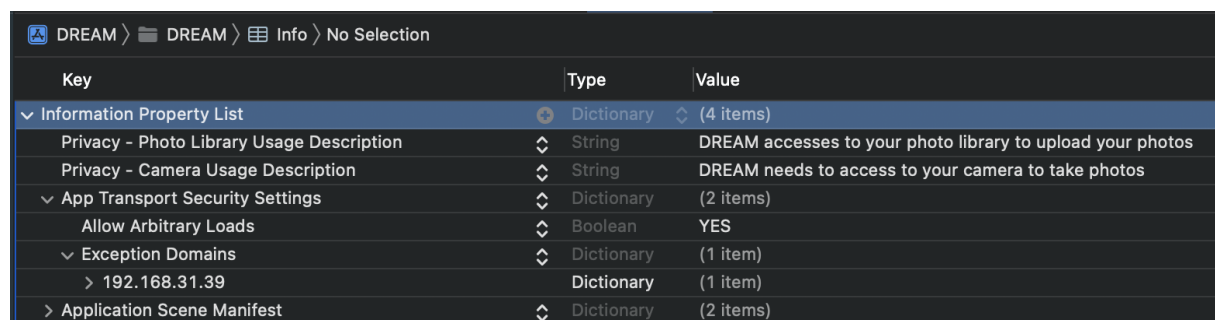
In order to be easily identified through an IP address, such address have to be static. To set such configuration (if you are using Windows) you have to go to the control panel in the "Network and Internet" section and then to the "Network and Sharing Center". Here, you have to click on the link related to your current Wi-Fi connection; in the new window you have to click on the "Property" button and then select "Internet Protocol Version 4(TCP/IPv4)" and click again on the "Property" button. Here you have to set the IP address, the subnet mask and the default gateway to the values that you have observed in the prompt using the "ipconfig" command; as default DNS Server you can write the address of the default gateway.

Instead, on the Xcode project, it is needed to insert the same IPv4 private address of the device on which the server is placed, in two files. Firstly, in the DREAM folder, click to the Util folder, then open the file Lib.swift. At the line 11, substitute the substring "localhost" with the IPv4 address. Then, in the DREAM folder, open the Info.plist file. Click on **"Information Property List" -> "App Transport Security Settings" -> "Exception Domains"** and you can find a dummy IP address. Substitute it with the previous IPv4 address.

```

7
8  import Foundation
9
10 struct Lib {
11     static let ip = "http://192.168.31.39:8080"
12

```



Key	Type	Value
Information Property List	Dictionary (4 items)	
Privacy - Photo Library Usage Description	String	DREAM accesses to your photo library to upload your photos
Privacy - Camera Usage Description	String	DREAM needs to access to your camera to take photos
App Transport Security Settings	Dictionary (2 items)	
Allow Arbitrary Loads	Boolean	YES
Exception Domains	Dictionary (1 item)	
192.168.31.39	Dictionary (1 item)	
Application Scene Manifest	Dictionary (2 items)	

Figure 8: IP changes in Xcode.

Now you can run the application performing the communication between the two devices.

6 Effort Spent

- Student 1

Topic	Hours
DB configuration and development	2:00h
Application server development	15:00h
Mobile application development	35:00h
Java tests	8:00h
Overall tests	5:00h
I&T Document development	10:00h
Document organization	5:00h
Overall system report	5:00h
Total effort spent	85:00h

Table 3: Effort spent by student 1

- Student 2

Topic	Hours
DB configuration and development	2:00h
Application server development	35:00h
Mobile application development	15:00h
Java tests	10:00h
Overall tests	5:00h
I&T Document development	8:00h
Document organization	5:00h
Overall system report	5:00h
Total effort spent	85:00h

Table 4: Effort spent by student 1

7 References

- All the screenshots of the application server code are derived from Eclipse IDE
- All the screenshots of the mobile application code are derived from Xcode IDE
- All the screenshots of http requests are derived from Postman