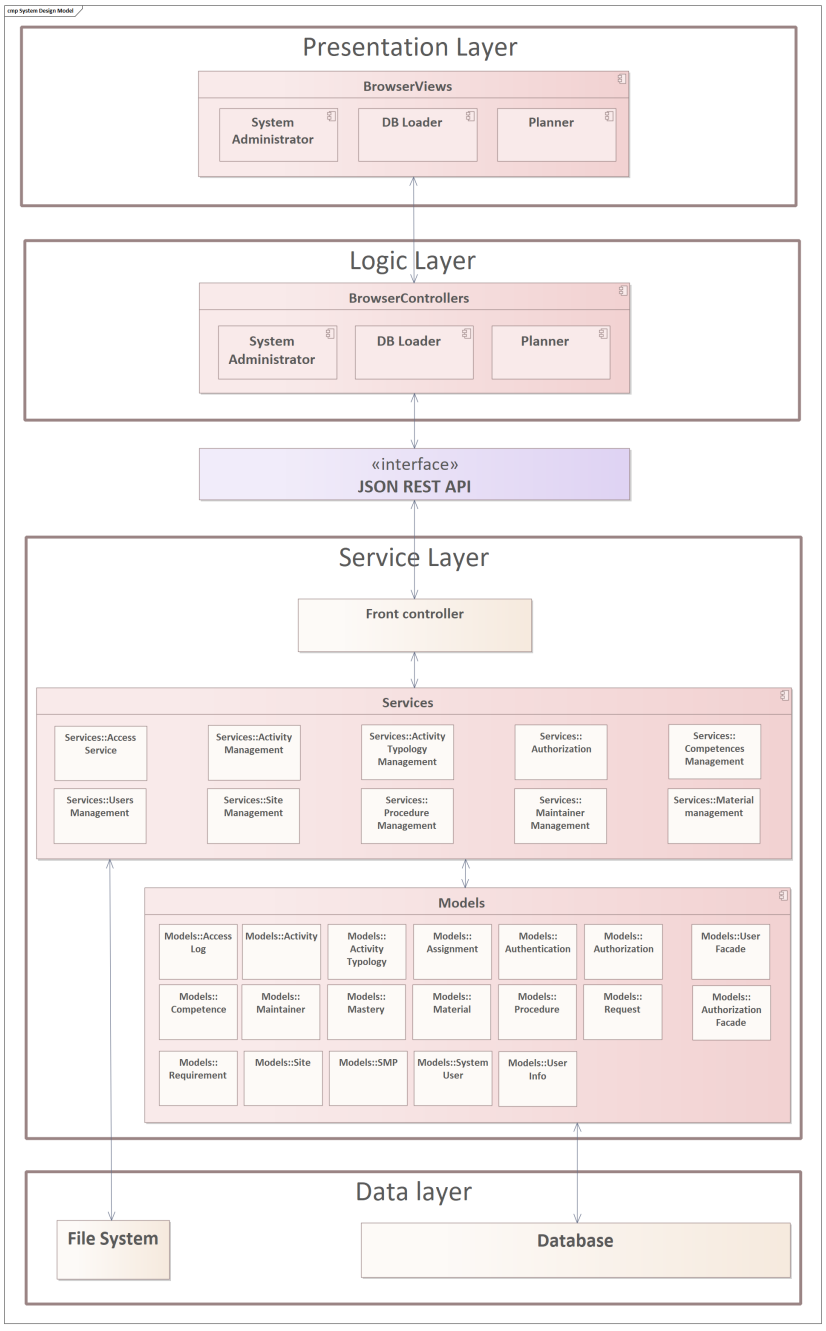
Application Architecture

Team 9

December 2020

1 Architecture

Our application is based on a client-server architecture, where the interactions are made using a REST API, that exchanges data using JSON format:



2 Frontend architecture

The frontend of our application consists of two principal components: a **controller** and a **view**.

The **view** is responsible for the logic of data's presentation: it manages the GUI's construction, consisting only of static content (HTML) and that represents the means by which the users interact with the system. The views were realized by using, in addition to the common languages (HTML and CSS), the Bootstrap framework, for the rapid development of personalized and responsive GUI. The GUI created is made up of several views that allow you to access the application data in multiple ways. Starting from a first view common to all, the login page, each user will be able to access all the views associated with their role:

- the *System Administrator* will be able to access the views dedicated to user management and to view the access log;
- the *Planner* will be able to access the views dedicated to the management of activities and their assignment;
- the *DB Loader* will be able to access the views dedicated to entering the database and viewing information relating to activities.

Entering data into the view, the execution of the processes requested by the user after capturing the inputs and the choice of any screens to be presented are delegated to the controller.

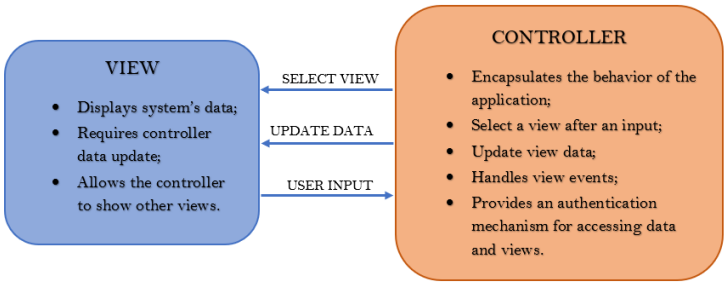
The **controller** implements the application control logic: made almost entirely using the *jQuery* Javascript library, it deals with the manipulation of the data coming from the view, the insertion of the data received from the server, the management of the events, the choice of the screens to be displayed and user authentication.

The data entered by the user in the view are validated using a *regular expression mechanism* and subsequently sent to the server that will take care of updating the database. The display of the data in each view is subject to the request for information from the server, which will take care of taking it from the database. The requests made by the controller to the server, both GET and POST data, are made with *Ajax* asynchronous requests that use a callback mechanism to manage the response received, updating the view on success or managing any type of error.

In addition, each request provides a *client authentication* mechanism, implemented by passing a token in the header of each packet destined for the server. This token is received after the user logs into the application and is kept by the controller until it expires or until the system user explicitly logs out.

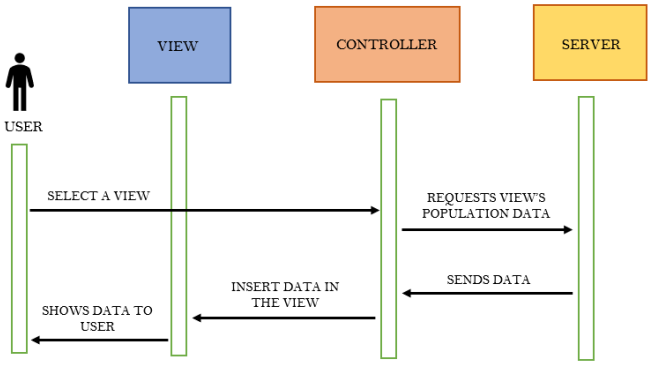
The controller is also responsible for controlling the access permissions to each view: by checking the token and the role played by the user, the controller allows or does not allow the display of a particular view.

The following figure represents an interaction diagram, which highlights the responsibilities of the two components:

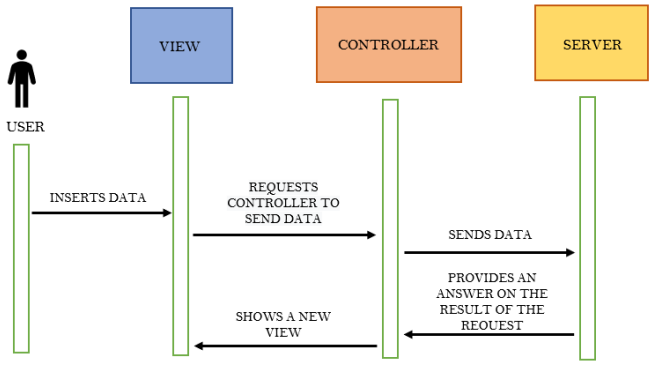


With the use of these two components we want to try to make the application modular and based on responsibilities, in order to increase the maintainability of the system as much as possible. Let's now highlight two of the possible operating scenarios of the two components:

- The first scenario is the user's request for a particular view and all the information it will show:



- The second scenario is the insertion of data by the user in the view and the request to send it to the server using the appropriate buttons:



3 Backend architecture

The server-side of our application was made using the *PHP language* as it seemed to us the most simple one for the context, especially for the connections to the database. For the data layer, we have used the *DBSM PostgreSQL* since we were already familiar with it. The application was deployed using *Apache HTTP Server*.
The server-side architecture of our project uses the **Front Controller pattern**, a specific representation of the Model-View-Controller pattern. We have 3 different components in our architecture, namely:

- The *front controller*, which takes the clients' requests and uses the services to elaborate them;
- The *services*, which communicate with the file system or the models, in order to give the requested data;
- The *models*, which abstract the shared database from the services.

The **front controller** is the only entry point of our server. It accepts requests from the clients, using a REST API where the data is in JSON format, and then it chooses what service (or services) must be used in order to satisfy the request.

The front controller uses a class implemented by us that allows the routing of the requests. This class (called *REST*) gives the possibility to define routes for requests of type GET, POST, DELETE and PUT. It also implements the possibility to create middlewares, that the registered routes can call before managing the actual request. For example, the middlewares that we used in our project were related to the authorization process. So, all the requests that require the same authorization, use the same middleware (so that code repetition is avoided). Considering that each request from the clients is managed from a different process, we chose to make the REST class a **Singleton**.

The **services** receive instruction from the front controller and compute the results, that will then be returned to the front controller itself, which will finally respond to the client that has made the request.

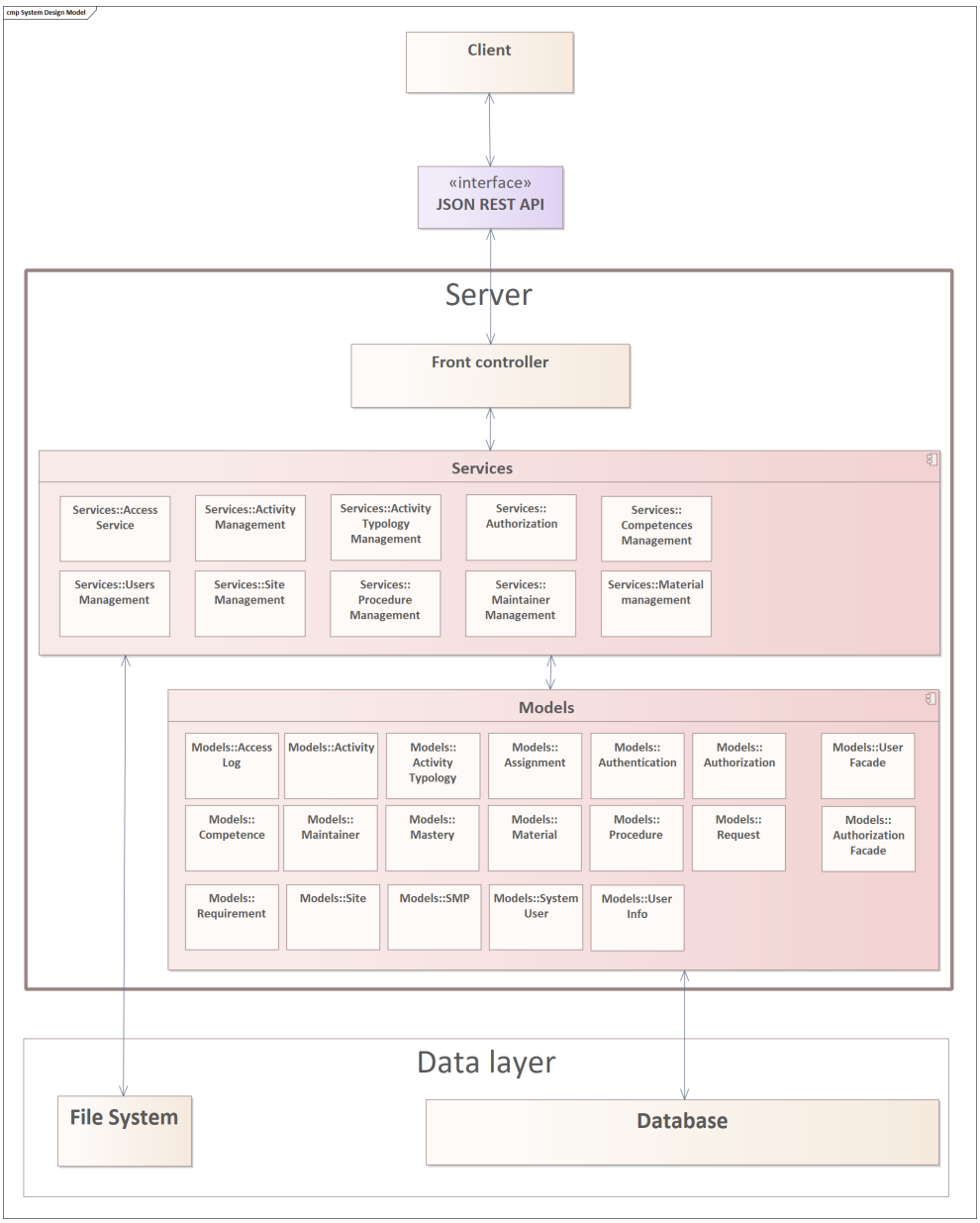
Each service communicates either with the File System or the shared database (or, eventually, both). In order to create a new layer of abstraction between the services and the database, we chose to introduce another component, that is the **models**. Each model is simply a class that represents a table on the database and that gives to the services a convenient interface to manage the data inside the table itselfs.

Have a model's class for each database's table is useful because each edit on the database structure is localized inside the code, in order to make it more maintainable. However, many services require the use of several models at the same time, and these interactions can create an inconsistent database's state if they are not well managed. In these cases, we used the **Facade pattern**.

This pattern hides the complexities of the larger system and provides a simpler interface to the client (the services in our case). It typically involves a single wrapper class that contains a set of members required by the client. These members access the system on behalf of the facade client and hide the implementation detail. In our specific case, each Facade class that we built facilitated the communication between the services and multiple models at once, without creating inconsistencies.

3.1 Server architecture design

In the image below we can see a graphic representation of the server architecture:



3.2 Simple interaction client-server

In the image below we can see a simple interaction between the client application and the server one described above: