iLift

A solution for gyms and sport centers for a better interaction with their users

Documentation

Table of Contents

1. Introduction
2. System specification
   1. System description
   2. Main architecture
   3. System features
   4. Behavior between components of the system
   5. Database specification
   6. Testing techniques
3. Deployment specification
   1. Deploy Server-Side
   2. Deploy Client-Side
4. Introduction

We have implemented a solution which aims to resolve problems regarding gym administration such that both the end-user, which is the gym user and the gym administrator can spare time and organize better their schedule on the gym, respectively to make their gym registers easier to use and understand.

Many gyms or sport clubs nowadays are trying to find solutions in order to consume less time and energy on administrating their schedules and to have a clear vision on how used is their gym, but also to have a payment roll so that the administrator would know who needs to pay in order to use their gym. We have compared some gym websites which have the same purpose and we found out that most of them are hard to use and generally they need extra computer skills in order to put them to full use. We though it may be a good idea to make an improvement and make all the process of scheduling courses, handle full training rooms and paying operations much easier, while on the same time much faster than just transforming the job of writing whole pages of gym registers every day into a digitalized job, where the administrator would have to work the same amount and have the same result as keeping paper records.

For example, we have made the process of adding a weekly schedule as simple as selecting a tick. We found out that, statistically, gym administrators would want to schedule a course for a long period of time weekly, twice a month or monthly. So, we added this feature, while keeping the old fashioned way (e.g. add a schedule only for this week), just by selecting the interval on which the administrator would want to schedule to the current week.

Also, we estimated that normally, an end-user would want to do the following operations most often: make an appointment to a course, get a subscription to the gym and verify if the schedule has changed. We managed to do these operations as simple as possible, such that an end-user would not have to do more than 3-4 clicks (excluding the login if he is not already logged in) in order to make one of the operations described above.

For making a subscription to the gym, the end-user is required to pay at the gym. However, in the future, this feature can be extended so that a gym user can pay directly online via secure payment to subscribe to the gym. For this project, we have decided that the user should pay at the gym for the subscriptions/course packages he wants, and the administrator can accept the payment on the website immediately.

In the following technical documentation, we will present the system architecture constructed in order to make everything work, as well as the behavior between components, the database and how the system can be deployed to a gym which accedes in using it for more efficient administration.

1. System Specification

2.a. System description

The system is composed of the front-end component, which is implemented in React, and of the back-end component, implemented in PHP. For data persistence, we have used a MySQL database.

There are two roles which can be used to access the system, the *“user”* role and the *“administrator”* role. In order to access user/administrator features, one must be logged in with a corresponding account which has this role in the database. We have used JWT tokens encrypted with a 256bit key, such that an user cannot access administrator features. For this purpose, on every call on an interface on back-end, if this interface includes admin features, firstly the token is verified so that it will match the role found in the database for the given user. The JWT token is also used in some cases for identification, so that we wouldn’t have to store a *“user\_id”* in the front-end, which would also be a security hazard, as one could identify himself to the system as someone else. The token is validated for this purpose on every single call that involves user/administrator features.

There are some features that are available even if one is not logged in, thus requiring no token. Those calls are not validated on the back-end for a valid token, and are generally very simple operations, for example “get all courses”, “get all trainers”, “see feedbacks for a course” and “see feedbacks for a trainer”.

For security reasons, we also stored hashed passwords in the database. In case of a database leak, there is little chance that someone will easily un-hash any of the passwords in the database. We have used the ‘ripemd160’ algorithm for hashing passwords.

For any operation made on the website, the front-end component needs to call the back-end component, which in turn makes operations on the database, depending on the request. The server responds with a JSON, for example for a call on the server interface */get\_all\_courses.php* would return a JSON of the form:

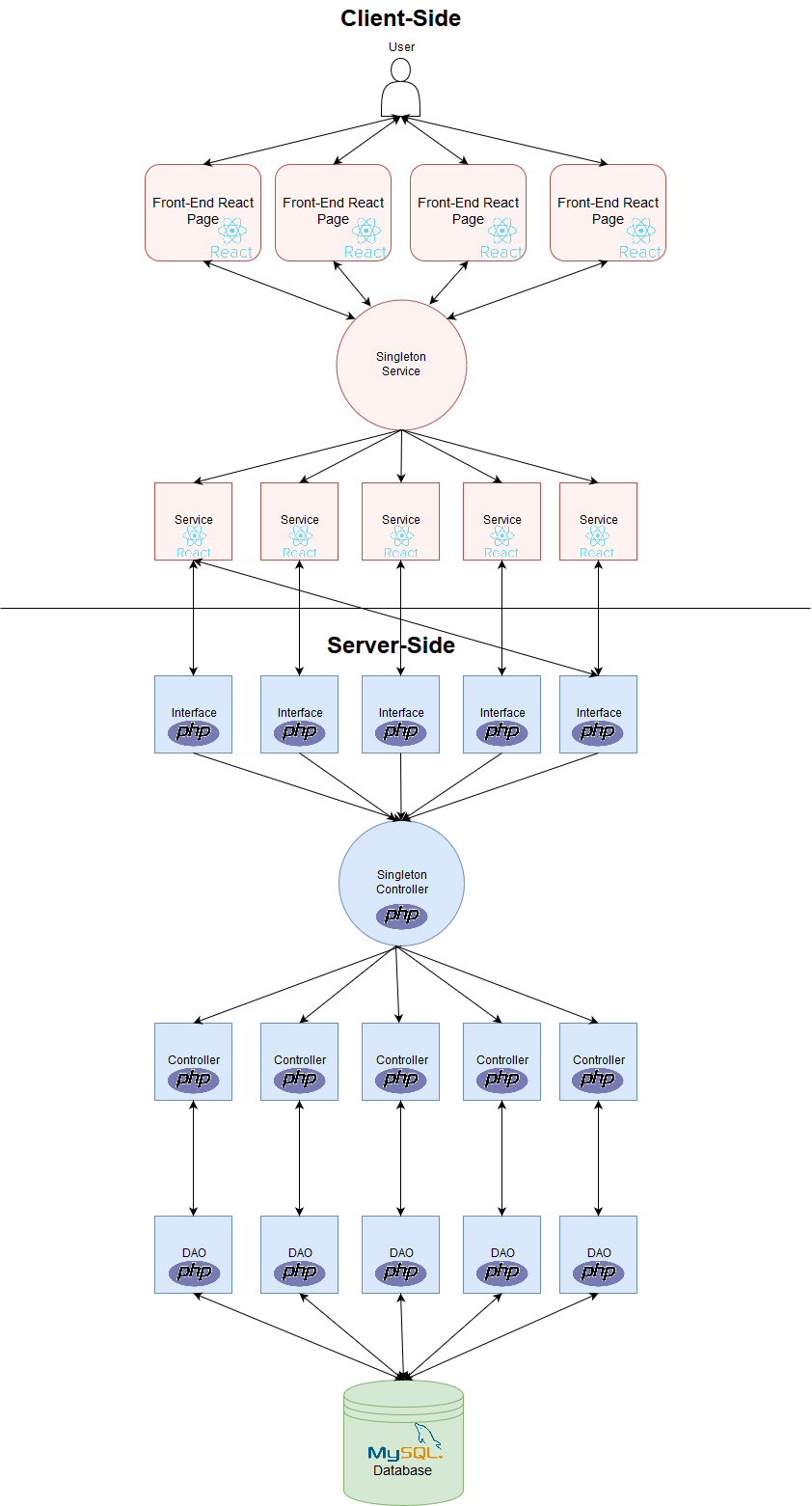
*{"answer":"Success","courses":[{"id":2,"name":"Boxing Course","description":"Do you want to learn box? This is a great opportunity! ","url\_photo":"..\/uploads\/phpE817.png","number\_of\_feedbacks":9},{"id":3,"name":"Cycling course ","description":"Cycling can be very fun! Come to us! ","url\_photo":"","number\_of\_feedbacks":0}]}*

For communication between front-end and back-end we have used a strict protocol: every JSON response of the server must have an “answer” field, which returns the answer to the server call. This can be “Success”, “Error” or “Warning”. For “Error” and “Warning” answers, the server is guaranteed to include in the response JSON a “reason” field, which describes the exact motive for which that answer was given. The possible reasons will be described in the behavior between components of the system chapter of this documentation. For “Success” answers, the JSON will contain the result of the operation, it might be a list, an element or a descriptive message, depending on the back-end interface called by the front-end. It is worth noting that all calls to the server must be POST calls.

2.b. Main Architecture

We have modelled an architecture diagram, which describes how the system is linked together. Firstly, we will show it in Figure A, then we will explain it.

Figure A – Architecture Diagram



All the front-end components are written in React. The user will access a React front-end page, redirected by the React Router, which shows a page generated by react. It is easy to note from Figure A that the flow of data between the user and the front-end component is bi-directional, the user can send information to a page, or retrieve information. When the page loads, or the user takes an action on the current page, the Singleton Service will be called. This service keeps references to all services, so that each service is instantiated only once and every component from front-end will call the same service independently. This service gets initialized when the React front-end component is deployed and remains the same. The singleton service is used by the front-end page to retrieve information, but also for updating fields, as requested by the user, so the data flow between this service and the interfaces is bi-directional.

As said above, the Singleton Service will give control to a specific service, depending on which functionality is called. For example, when the user wants to see the schedule for the current week, on the page load, a call to the Singleton Service will be made, which gives control to the Schedule Service. As the singleton service gives control directly to a specific service, the information flow is only on one direction, specifically, the singleton services gives data to a given service. The service is then guaranteed to retrieve the response to the front-end page which must react to the response correspondingly, such as show an error message if the server responded with an error message.

The services are divided depending on functionality, so that a service which has the job to do operations on the schedule won’t affect any other component. This makes the services easier to use, as their name give the functionality for which they were implemented. The services are the cross-line between the front-end component and the back-end one. They are the main link between these components and all the calls to the back-end server must be made through these services.

The back-end server is exposing interfaces to front-end which are used to call to retrieve or update information. The services are calling these interfaces through HTTP POST calls. It is easy to note that a service does not necessarily have a single interface which resolves the call given by the service. Also, the data-flow is bi-directional, so the service can retrieve information, but can also give data to the interfaces, so that these data can be then persisted in the database.

The interfaces are done by the idea of “an interface should resolve one problem at a time”, so, for example, for a CRUD implementation, we have implemented 4 interfaces. The reason behind this is to create more abstractisation in the server-side implementation, and also to simplify the logic. The interface serves as an entry-point to the server-side, and represents a façade-like design pattern, as the interface will make the entire implementation logic and will call the controllers when needed, in order to make the operation requested.

The controllers are called from within a singleton controller, which respects the same pattern as the singleton service used in the front-end component. Mainly, it gives control to the needed specific controller, and the requested controller must return after the operation is resolved the response to the caller interface.

The controllers are operating on objects which are retrieved from database. The objects are retrieved from/pushed to the database using Data Access Objects (DAOs). For this purpose, every controller keeps a DAO which works on objects of the controller type. The controllers can send object to the DAOs to be inserted/updated/deleted from database, but can also retrieve objects from the database using calls to the DAOs. The data access objects are working directly on the database and are used to create and execute SQL queries.

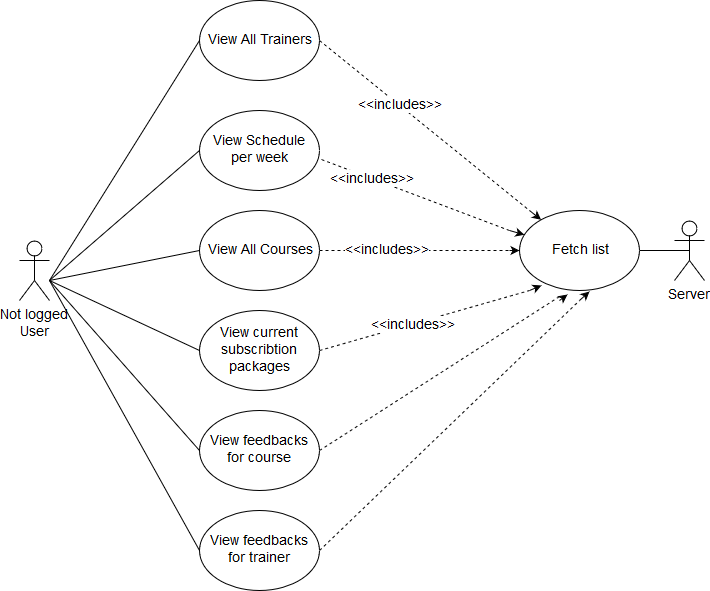
In conclusion, the architecture of the system is done in such a way, so that when a normal user makes an operation on the system, there are maximum seven components which are accessed until the operation is done and the response is retrieved. This ensures a fast utilization of the system. By using some known design-patterns (server-client, façade, singletons), we have reduced the complexity of the system, thus making the logic behind the system simple to understand and easy to use.

2.c. System features

From the beginning, we have divided the features of the system by the user role which uses them. On this section, we are presenting the use-case diagrams which we have to create to describe the features of the three main roles of the system: not logged user, logged in normal user and logged in administrator.

\*) *Not logged user*:

Figure B – Not logged user use case



The “not logged user” has access to the following features:

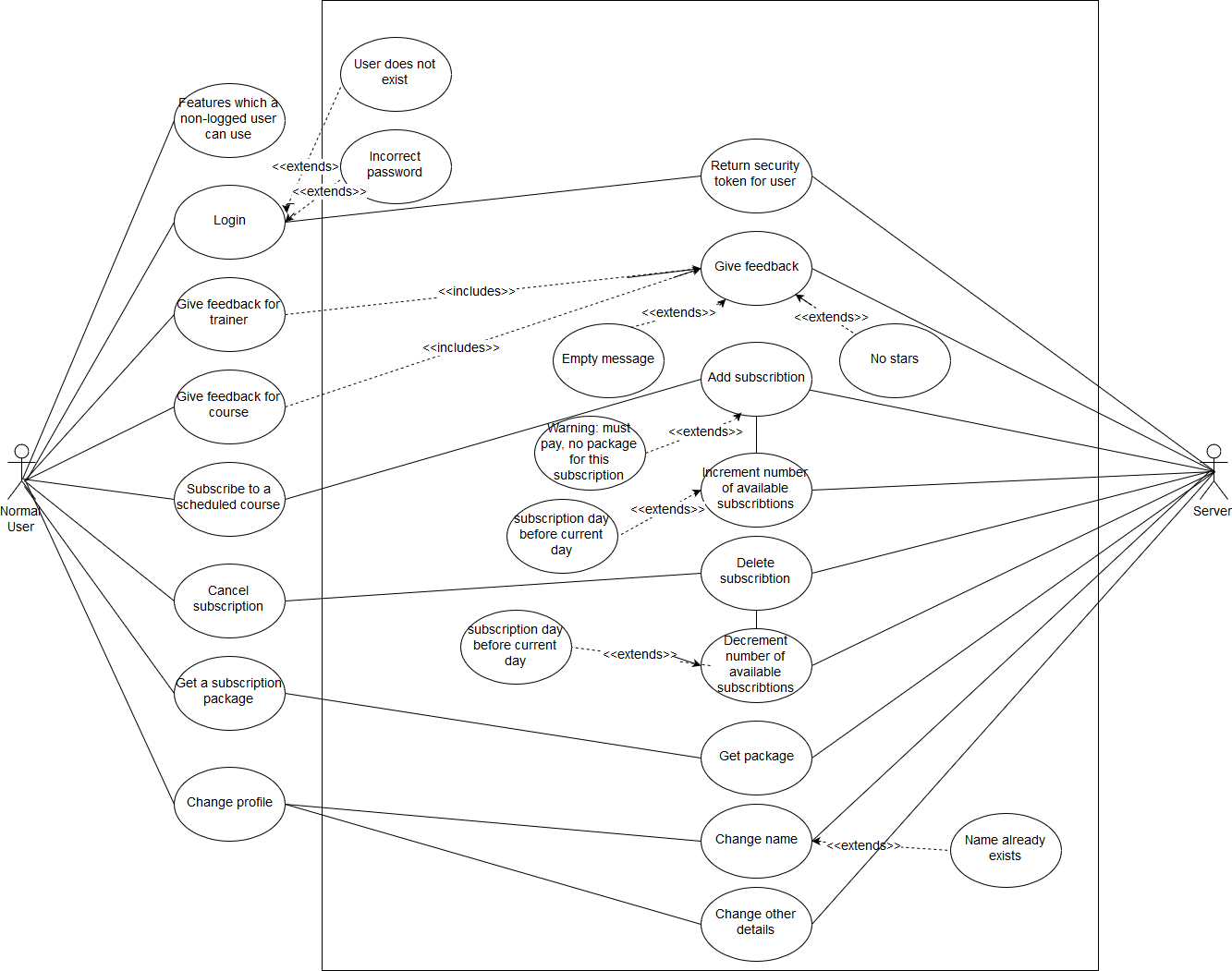
* *View all trainers* – it is useful for a user which has not yet visited the gym/sport club to see the available trainers and their description
* *View Schedule per week* – an user which wants to go to the gym, before enrolling, would probably want to see if the gym schedule suits his personal schedule
* *View all courses* – useful to see what courses does the gym offer
* *View subscription packages* – an user should see which are the pricings and what does the gym offers before enrolling
* *View feedbacks for courses* – the user tends to be more trustful when he can see that other users are saying that courses are done well on the gym he wants to enroll
* *View feedbacks for trainers* – the same idea as feedbacks for courses

\*) Logged in normal user

A normal logged in user, or “gym/sport club user” has access to the features for a non-logged in user, but in addition, he can do the following, after he uses the login form to confirm his identity:

* Give feedbacks – both for trainers and courses. Note that, if the user does give an empty feedback or somehow manages to give a number of stars which is not accepted by the server (between 1 and 5), an exception will occur and an error message will be given to the user.
* Subscribe to a scheduled course – this can be done by firstly having an active subscription package, otherwise the user will be warned, and he cannot go to the scheduled course until he does not get a package. If he has a package containing this course, the number of courses available for this user is decremented, only if he is subscribing to a course which follows the current date.
* Cancel a subscription – the user cannot go to a scheduled course, so he cancels the subscription. To avoid problems, the number of courses available in the package is re-incremented (so the user is not paying for the course if he does not want to come), only if the cancel is done before the day of the course.
* Get a subscription package – the user requests a subscription package and, when he pays at the gym, the admin will confirm the payment and the package becomes available
* Change his/her profile – useful interface for the user to see the current profile, which means the subscription he made, his current available packages for which he paid, his packages for which he must pay. The user can also change his password from here.

Figure C – Logged in normal user use case



\*) Logged in as administrator

The administrator should have control on the gym courses, trainers, schedules, promotional packages presented, but also on the users. We will get each of these, one by one, and we will present what features we offer. We have also given to the admin the possibility of registering new users on the site. This has been done in order to avoid unwanted users, or bots which access the site and, for example, give not useful feedbacks.

* Course control panel: - Create, Update, Delete courses
* Trainers control panel: - Create, Update, Delete trainers
* Users control panel: - Create, Update, Delete, users
  + Create and Update operations on users may give the exception “User with this name already exists” and the error message is shown
* Feedbacks control panel: - Delete feedbacks
  + We offer only the option to delete feedbacks, as there might be users which use bad words, or there might be users which are writing non-sense on the feedback page.
* Schedule control panel: - Add new schedule, Edit one schedule entry, Delete one schedule entry
  + Adding a new schedule implies: for a given interval <start date> - <end date>, add schedule entries from <period> to <period> from <start hour> to <end hour> in the <room> given, with the <trainer> given, teaching <course>. This can give exceptions as “Training room occupied on selected date”, “Trainer occupied on selected dates and hours”, or “Invalid schedule hours”. This will result in (<end date> - <start date>)/<period> scheduled entries.
  + Updating schedule entries can give the same exceptions as adding schedule entries. These are treated and shown to the user.

Figure D – Use case for administrator



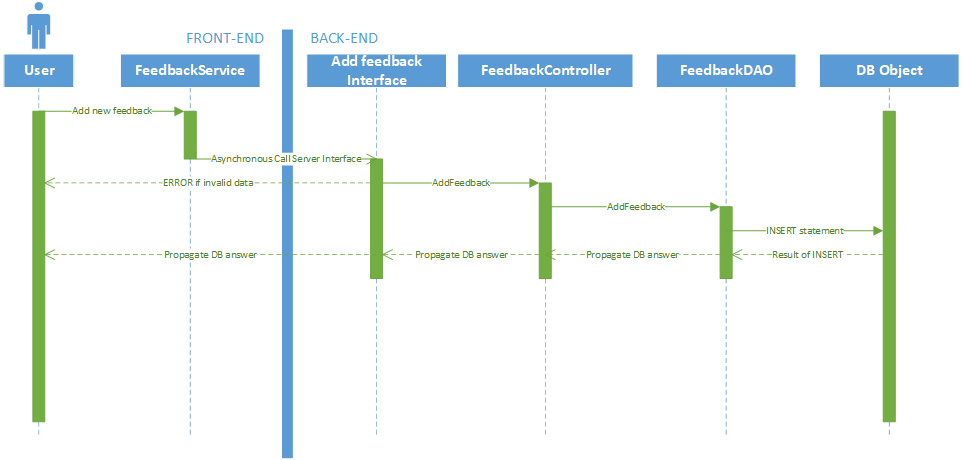
2.d. Behavior between components of the system

The components of the system are communicating using HTTP POST calls, mainly the flow of the data consists of: the front-end component is making the HTTP POST request to a back-end server interface, firstly completing the parameters as documented. The server is responding to the front-end component by using a JSON response formatted by the following users: every response has a field “answer”, which can be “Success”, “Warning” and “Error”. If the answer is “Warning” or “Error”, the server is guaranteed to return another field, called “reason”, which consists of a descriptive message why this has happened. The front-end component is expected to treat warnings and errors explicitly, by showing in the user interface layer the reason returned by the server. If the “answer” field is “Success”, the other fields contain information requested by the front-end component, depending on the interface called. The front-end services must know exactly which interface respond in which format in case of Success, so that the reported answer can be treated and converted to React objects by this component.

The server propagates an error from within the lower layer it occurred to the most upper layer (e.g. interfaces), where it is sent as a JSON to the caller service. For example, if an error occurs while inserting into the Database, the error is propagated upper through DAOs and then through Controllers, up to the interface. Then a JSON of the form *{“answer”: “Error”, “reason”:”DB Insert Error”}* would be returned.

The React Services implemented are alerting the users of a Warning or Error occurred on a server call, by making use of the *alert(message);* call from JavaScript.

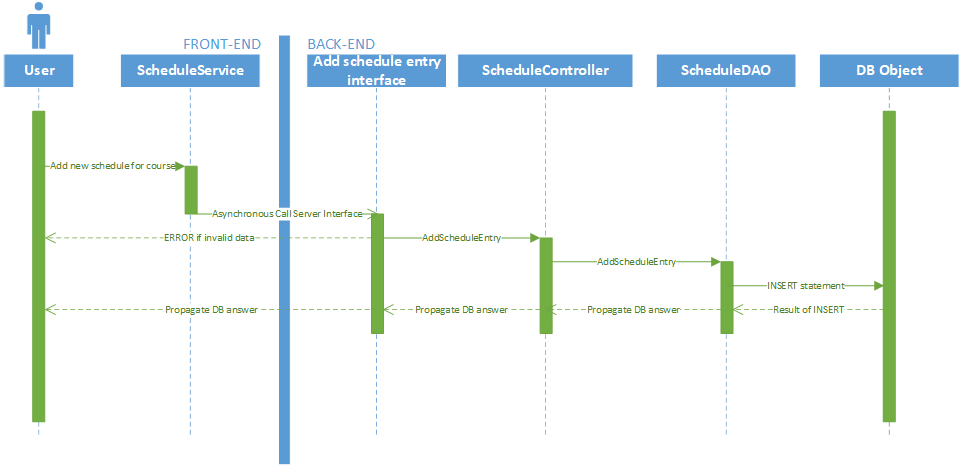
In this sub-chapter, we will present and explain a few sequence diagrams in order to further visualize the behavior between front-end and back-end components, and also we will present an UML Class Diagram representing the server architecture, which shows more explicitly how the components are linked together inside the server, and show more visually how and why the server sends an error upper, until it reaches the caller. We will also present some server interfaces, how to call them, and what errors could they return.

Figure E – Sequence diagram describing adding a new feedback

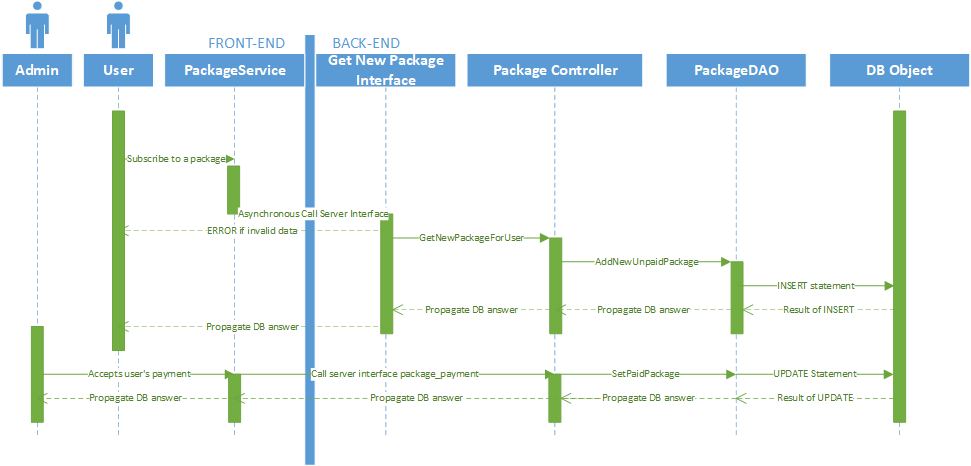
We can see in the upper diagram the following components: the User, which is the main actor, the Feedback Service, the server interface which handles the Add Feedback call, the Feedback Controller, which controls operations on feedbacks, the Feedback DAO, which access and process data-base objects, and finally the DB Object, used to handle directly the DB operations.

The user and the DB Object have been represented with a continuous instantiation, as we have considered the fact that “the user is on the site continuously and the database can answer to operations continuously”. When the user presses the “Add Feedback” button, a call to the Singleton Service is made, which gives immediately control to the Feedback Service. The feedback service is making an asynchronous call to server interface, which returns with a Promise object immediately. The server interface gets the request and processes it, while validating the given parameters, and the token. If any of the validations fails, a response with “Error” is given back directly to the user, and the user is notified of what the problem was with his input in the “reason” field. If the validation succeeds, the interface is giving control to the Feedback Controller service, which then calls the Feedback DAO. The feedback DAO makes the operation on the Database Object, which could return an error if any constraint fails in the database. The error is then propagated up to the interface, where the interface puts the “Error” tag in the “answer” field and the returned error text in the “reason” field.

We present another sequence diagram in the sequence diagram from figure F, now describing the Add Schedule operation. This operation makes approximatively the same road as the Add Feedback operation, but the services, interfaces, controllers and DAOs are different because of abstractisation.

Figure F – Sequence diagram describing adding a new schedule

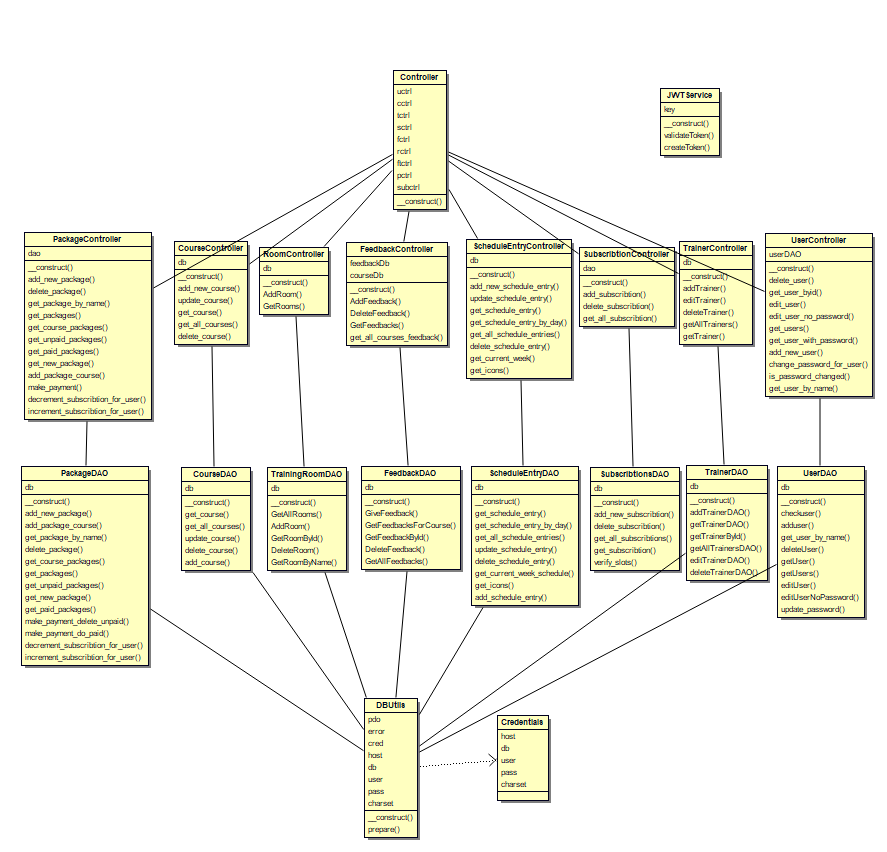
Another interesting sequence diagram is the “New subscription package” operation, on which both the Administrator and the User must make some operation in order for the whole operation to be completed. Firstly, the users will request a new package, which calls the server and adds the package to his Unpaid Packages. Then the administrator will accept the payment when the user pays at the gym, and the package will be updated and added to the Paid Packages of the paying user. This ensures some security regarding the payment, as the user cannot have a package until the administrator has accepted his payment, and the administrator would not accept the payment and therefore make the package available to the user, until the user hasn’t really paid. This behavior is described in Figure G.

Figure G – Sequence diagram describing getting a new package

After we have shown the behavior of components when communicating between front-end and back-end, we are presenting the relations and behavior of the components present in the back-end, as well as how are they linked, and how they together are resolving requests and also giving implementation details. The UML class diagram of the server component is shown in figure H for this purpose. On the upper-most layer of the diagram, there are the Controller and JWT Service classes. These classes are used by all the interfaces, the Controller is a singleton which is a gate-way to other functionalities which are implemented inside other Controllers. This Singleton Controller has references to all the application controllers: Package Controller, Course Controller, Room Controller, Feedback Controller, Schedule Entry Controller, Subscription Controller, Trainer Controller and User Controller, which in turn reference DAOs containing their name. Each DAO has a reference to a singleton DB Utilities object, which depends on the Credentials class. The Credential Class is like a “credential manager”, you just have to replace the credentials to the database here, and all the DAOs will use the database object constructed in DB Utilities by connecting with those credentials.

The JWT Service is a utility manager class for working with JWT tokens. This service will encrypt and decrypt tokens, raising exceptions if any error occurs while decrypting. This module uses the *firebase/php-jwt* utility for work with JWT tokens.

Figure H – UML Class Diagram for Server Components



Next, we will present some interfaces. More specifically, we will present the interfaces exposed by the server in order to manipulate Trainer objects.

/interface/add\_trainer.php

* adds a new trainer to the database
* called using POST
* parameters:
  + token – a valid administrator token
  + name – the name of the trainer
  + description – the description of the trainer
  + photo – a multipart/form-data file object representing the photo of the trainer
* possible errors:
  + *“Invalid token given”*
  + “Not a valid format! Only JPG/JPEG/PNG/SVG accepted!”
  + *“File too large”* – we accept only maximum 5 MB files
  + *“Name or description empty”*
  + *“Unknown reason for file transfer failure, try again”*

/interface/delete\_trainer.php

* deletes a trainer
* called using POST
* parameters:
  + token – valid administrator token
  + id – the id of the trainer to be deleted
* possible errors:
  + Invalid token given
  + The given id does not exist

/interface/edit\_trainer.php

* updates a trainer given the fields
* called using POST
* parameters:
  + token – valid administrator token
  + id – the id of the trainer to be updated
  + name – the updated name
  + description – the updated description
  + photo – optional – the edited photo, if not given, then the photo will not be changed
* possible errors:
  + *“Invalid token given”*
  + “Not a valid format! Only JPG/JPEG/PNG/SVG accepted!”
  + *“File too large”* – we accept only maximum 5 MB files
  + *“Name or description empty”*
  + *“Unknown reason for file transfer failure, try again”*

/interface/get\_all\_trainers.php

* retrieves the list of all trainers in a JSON format
* called using POST
* parameters: None
* possible errors: None
* the returned JSON will be of the form:
  + answer – should be Success
  + trainers – list

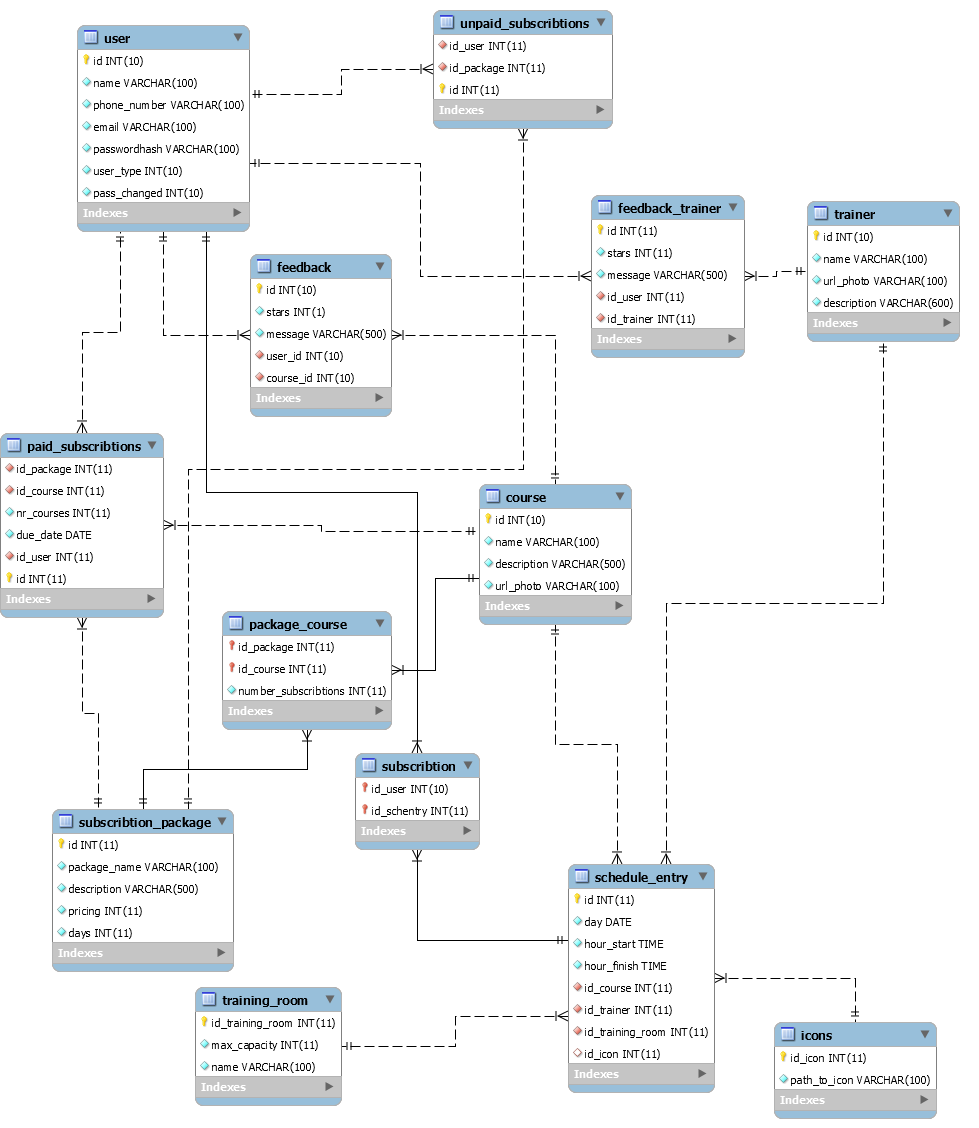
2.e. Database specification

The database was made in MySQL, using the phpMyAdmin tool for creating it. It comprises data about every entity present in the system. The diagram of the database can be seen in the Figure J, and in Table I we have presented the relationships between entities.

Table I – relationships between entities

|  |  |  |  |
| --- | --- | --- | --- |
| From which table | Type of relation | To which table | Observations |
| user | one-to-many | unpaid\_subscript. |  |
| user | one-to-many | feedback\_trainer |  |
| user | many-to-many | schedule\_entry | through subscribt. |
| user | one-to-many | feedback |  |
| user | one-to-many | paid\_subscriptions |  |
| trainer | one-to-many | feedback\_trainer |  |
| trainer | one-to-many | schedule\_entry |  |
| course | one-to-many | feedback |  |
| course | one-to-many | schedule\_entry |  |
| course | many-to-many | subscription\_pckg | through pckg\_c. |
| course | one-to-many | paid\_subscriptions |  |
| subscription\_pckg | many-to-many | course | through pckg\_c. |
| subscription\_pckg | one-to-many | paid\_subscriptions |  |
| subscription\_pckg | one-to-many | unpaid\_subscript. |  |
| training\_room | one-to-many | schedule\_entry |  |
| icons | one-to-many | schedule\_entry |  |
| schedule\_entry | many-to-many | user | through subscribt. |

Figure J – Database Diagram describing relationships between entities



2.f. Testing techniques

For testing purposes, we have used unit testing on the server component to test the controllers. They are testing each function of the controllers which are called by the interface. This is done because if anything fails on this testing phase, integration of the interfaces cannot continue. We have also used integration testing to test the link between the front-end component and the back-end component, specifically to test the link between the react services and the interfaces which respond to their requests.

To show the test results, we have created a react component routed by /testing. This will run sequentially the integration tests, by calling interfaces and verifying their results, while mocking data, and then will make a call to the server interface /interface/run-unit-tests.php which will run all the unit tests on the server-side and return a JSON with the results.

For every test, we showed a green color if they passed, or a red if they failed. We have also used manual testing to verify if the front-end components are put in place, because we couldn’t find a good automated way to fulfill our expectations and which suited our components.

1. Deployment specification
   1. Server-Side

We recommend the usage of XAMPP for deploying the server-side. First, open XAMPP and start Apache and MySQL services. Put the server part (ServerApp folder in the solution) in the C:\xampp\htdocs folder. Launch phpMyAdmin by clicking Admin in the XAMPP control panel, next to the MySQL label. Create a new user, or, if you already have one, this step doesn’t need to be done. Create a database named ilift and select it. Click on Import and select the ilift.sql file in C:\xampp\htdocs\ServerApp\db. Now you can close phpMyAdmin. Open the file credentials.php from C:\xampp\htdocs\ServerApp\db and enter your username and password for MySQL (the above created user, if not any existing). The server is fully deployed after this step. Make sure that the 80 port of the machine where you deploy the server is opened and accepting requests.

* 1. Client-Side

For deploying client-side you need Node.js along with Node Package Manager (npm). Get the folder ClientApp from the solution and open a command line in ClientApp\ilift. Run in the command line “npm install”. Then open {CLIENT\_ROOT\_FOLDER}\WebApp\ClientApp\ilift\src\services and open ServiceCredentials.js into a text editor. Edit the SERVER\_PATH variable to point to http://<your server ip here>/ServerApp, surely completing your server ip here with the ip of the machine where the Server-Side deployment has been done. Now go to ClientApp\ilift and run npm start. The client is now deployed and can be accesed on http://<machine ip>:3000.