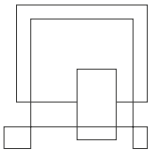


# SEP3

## Fresh Fitness

Students:

Jaser Ghasemi 267243

Modaser Ghasemi 267251

Yasin Issa Aden 267276

Supervisor:

Christian Flinker Sandbeck

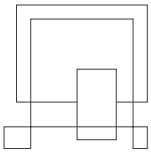
Ole Ildsgaard Hougaard

Number of characters

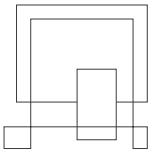
Software Engineering

Third Semester

19-12-2018



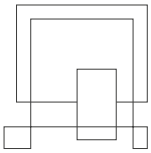
# Table of content





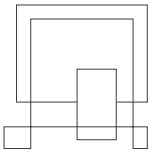
## List of figures and tables

Optional



## Abstract

*During this semester project, the group decided to make a fully functional software application for Fresh Fitness. The project filled the requirements for the SEP3. The main requirement for SEP3 was to create a heterogeneous 3-tier-architecture, which was connected to an external API. For the process of the project, the SCRUM and unified method were to be used. In addition, the project report was separated into some different phases following the waterfall approach. The first phase was the analysis phases, here the problems formulated in the background description were analyzed. Secondly, the design phase was dedicated to solving the analyzed problem. Thirdly, in the implementation phase, the design had to be implemented. Finally, the testing phase required that the system should be tested. It was following these phases that the system was developed and structured in this report. The final result was an user-based system created using a 3-tier-architecture using two languages Java and C#.*



# 1 Introduction

Fresh fitness is a fitness chain that wants to offer people the best equipment's and facility in order to help people maintain their active lifestyle. FF, wants to expand their brand and reach out to everyone in Denmark. They have decided to build a new center in Horsens, and they want to connect its member for events in their centers throughout Denmark. In order to reach this goal, they want to build a subscription platform where the customers are able to sign up for different workout sessions throughout all the Fresh fitness centers in Denmark. For users to be able to register for all events they need the subscription for all centers. On the other hand, the user with the subscription of the local center is only able to register for events, at that center

To summarize, Fresh fitness desires a system where its member can keep track of their workouts and progress overtime. Secondly the users should be able to see the workout activities available and be able to join. Lastly the admins should be able to manage the member and activities.

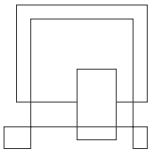
Questions to be answered are the following:

1. How will the system be designed?
2. How will the system, which are coded in different languages be able to communicate?
3. How will the system be maintainable?

The delimitations for this project is:

1. The user will not be able to see other member profiles(privacy)
2. No implementation of security

Overall, this report will go through the group's attempt at implementing this system with all the required steps following the waterfall approach template.



# 2 Requirements

The requirements for the system was based on background description from Fresh fitness. Fresh fitness is a fictional company created by the group to come up with some relevant requirements.

## 2.1 Functional Requirements

| ID | Functional Requirements | Position | Priority | Est. hours |
|---|---|---|---|---|
| 1 | The user must be able to login so that their information is secure | Product Backlog | High | 30 hours |
| 2 | The user must have a profile, so they can track their information | Product Backlog | High | 30 hours |
| 3 | The user must be able to add/edit/delete workouts to their log, so they can track their progress | Product Backlog | High | 50 hours |
| 4 | The user must be able to see available activities to be able to join or leave activities | Product Backlog | High | 60 hours |
| 5 | The admin must be able to add/delete/edit members, so they can manage the members and their info | Product Backlog | High | 40 hours |
| 6 | The admin/instructor must be able to add/delete/edit activities, so they can manage the activities | Product Backlog | High | 45 hours |

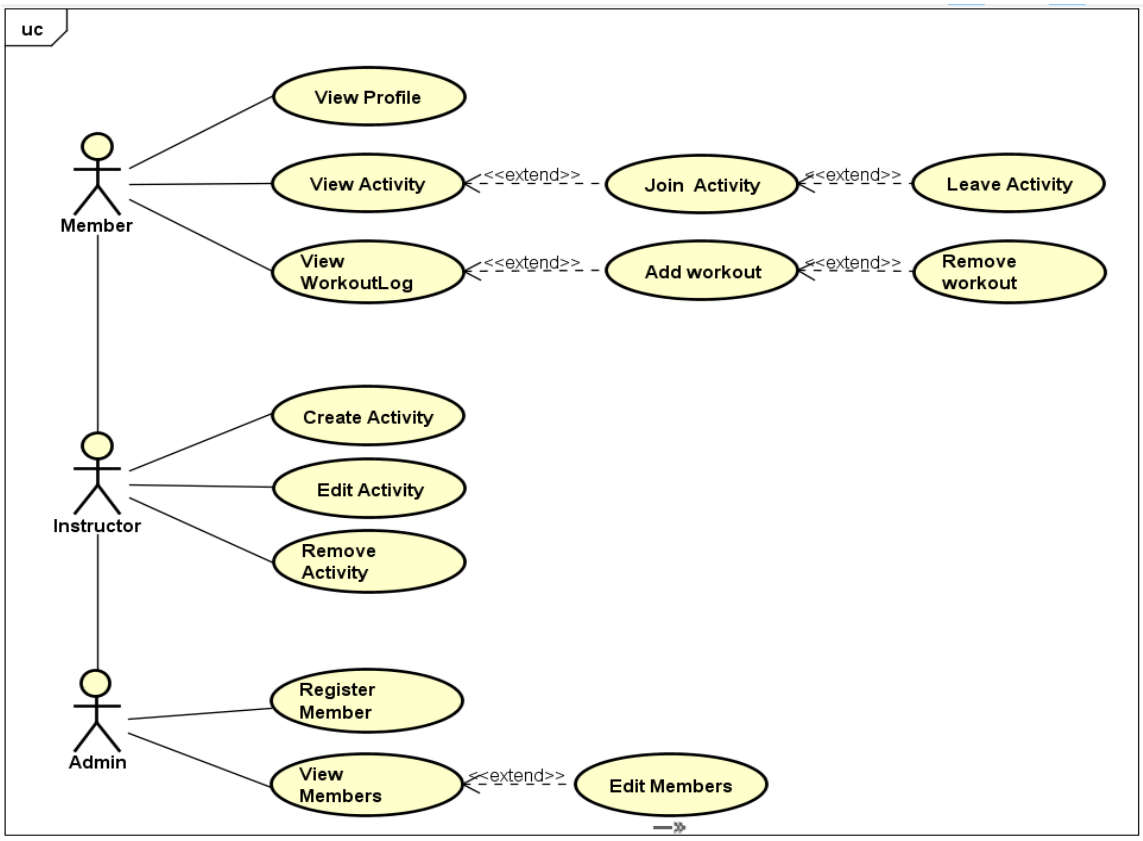
## 2.2 Non-Functional Requirements

| ID | Non-Functional Requirements | Position | Priority |
|---|---|---|---|
| 1 | The system must be stored in a database | Product Backlog | High |
| 2 | The system must be implemented in Java and C# | Product Backlog | High |

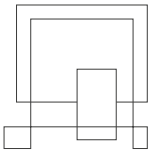
# 3 Analysis

In the analysis phase it is important analyze the problems Fresh fitness faces in an attempt to make the system they requirements. In additions, analyzing the requirements as a foundation to making the use cases, domain model and an analysis class diagram are all important steps in this phase.

## 3.1 Use cases & Activity diagram



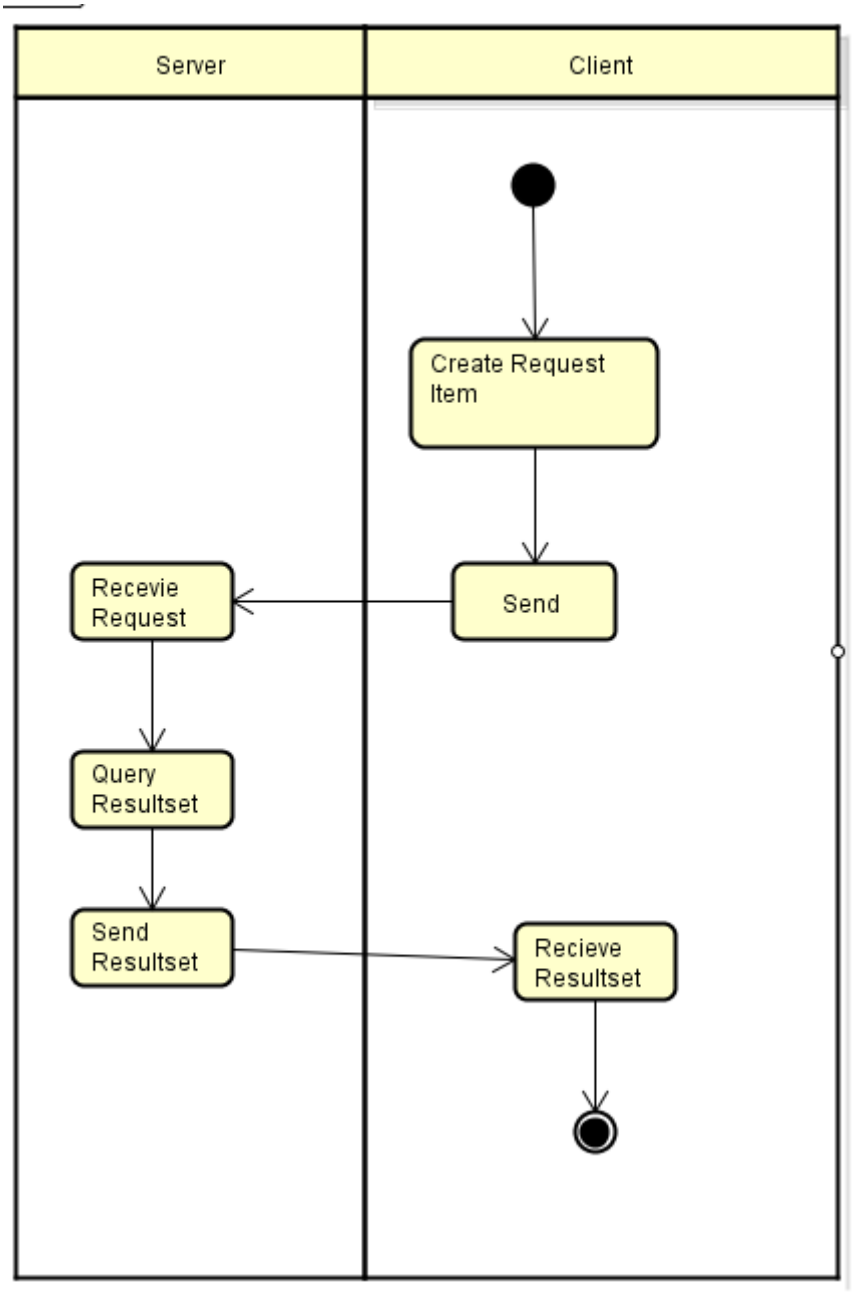

In the use case diagram above all the use cases for the systems three actors Member, Instructor and Admin are illustrated. The user should be able to view their profile, which is done by entering the profile tab. Here all their important profile information is displayed. Secondly, the Member should also be able view available activities and be able to join and leave these. This is shown by having the extend line, which means the member can join after viewing an activity. If they change their mind, they also have the option to leave already joined activities. Finally, the Member should be able to view their workout log,



were they can see/store all they workouts done sorted by month. The Member should be able to add a workout and the number of repetitions they did for a day to their log. Finally, the member should also be able to remove these workouts from their workout log.

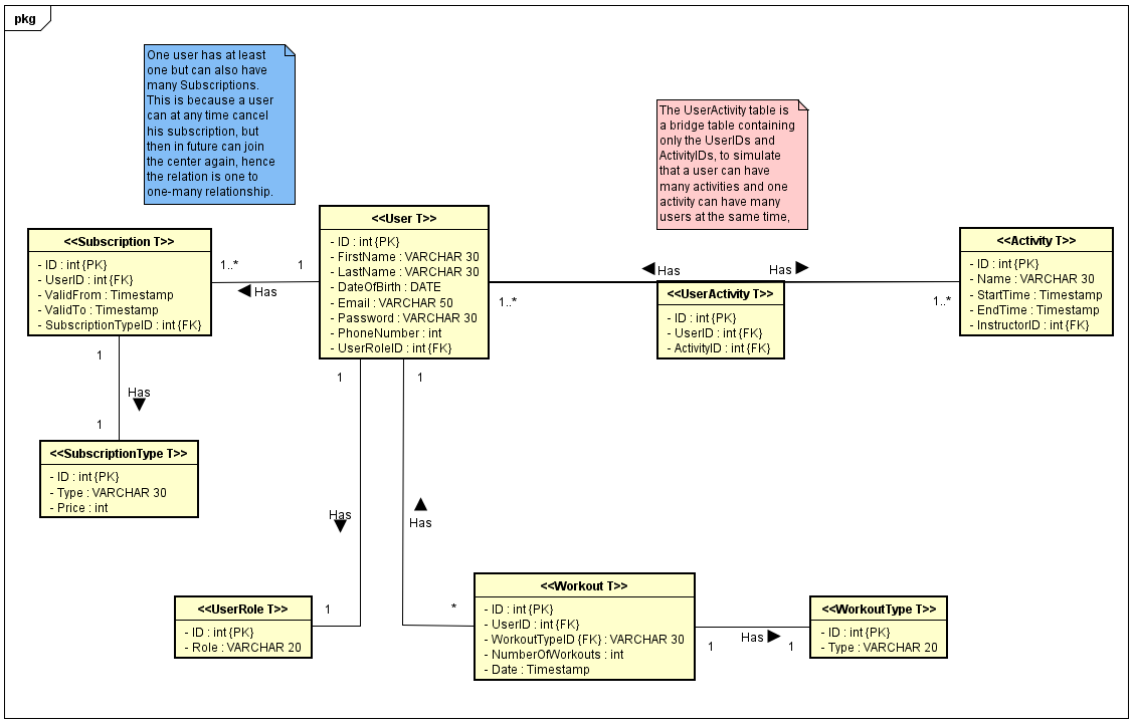
The second actor is the Instructor. The Instructor can do everything the member can, but they can also create, remove and edit activities. They have a separate GUI where there is tab for managing activities. They can save the changes that now will be shown to the Members.

The last actor is the Admin. The Admin can do everything that both the Member and Instructor can, but they can also View, Register and Edit Members to the system. The have a separate GUI where there is a tab for managing Members along with all the previous tabs. The can decide the Members roles (Member, Admin and Instructor) along with their subscription type.



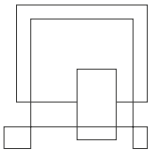

The activity diagram above illustrates the main sequence for the API in our system for both the client and server side.

## 3.2 Domain model




In the diagram above the domain model for the system is illustrated. All the tables are marked with a T, which indicates these are the entities. This model is also the ER-diagram for the database, therefore the IDs functioning as primary keys and foreign keys are also illustrated. The central entity in the diagram is **User**, since this is a user-based system. The **User** has multiple instance fields, which is ID(PK), FirstName, LastName, DateOfBirth, Email, Password, PhoneNumber and UserRoleId(FK) that references **UserRole**. All these fields cover the Users information. As mentioned before this entity has a **UserRole** that indicates the role the user has. It can either be Member, Admin or Instructor, whom all have different User interfaces. The Admin is for example supposed to manage the members, and the Instructor is supposed to create activities.

The next entity is **Subscription**, ID(PK), UserID, ValidFrom, ValidTo and SubscriptionTypeID(FK). This entity is responsible for having the user's subscription duration and type. This entity is also connected to **SubscriptionType**, which is responsible for having the type of Subscription and its price. The system has two forms of subscription regular and premium, which have different privileges and hence prices.



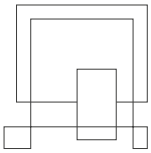
The **Workout** entity is important for the functionality of the **Users** workout log. This entity has the fields ID(PK), UserId(FK) that refences User, WorkoutTypeID(FK) that references WorkoutType, NumberOfWorkouts and Date. **Workout** is responsible have the workout information to be stored in the User workout log for them to keep track of their progress. The **Workout** entity is also connected to **WorkoutType,** which indicates the type of workout. Another important entity is **Activity**, which has the field ID(PK), Name, Startime, Endtime, InstructorID. This entity is responsible for having the important information regarding the activity. The next table is **UserActivity** this is the bridge table that is connected to both **User** and **Activity**. It contain UserID(FK) and ActivityID(FK) to simulate that a **User** can have many **Activities** and an **Activity** can have many users. This table also exist to avoid redundancy in our database, since this is a many to many relationships.

## 3.3 Threat model:

### 3.3.1 STRIDE

Now that an overview of the system is made the possible threats will be analysed. To prioritize which attacks the group should be concerned about, in order to protect their system. They started thinking about what the attacker wants to achieve by attacking their system. The STRIDE classification was then used to go through all the attacker's possible goals:

**S**poofing identity: means an attacker impersonates another user.
**T**ampering: means an attacker manipulates the data without being detected.
**R**epudiation: means an attacker can deny something they actually did.
**I**nformation disclosure: means an attacker can access data they should not see.
**D**enial of service: means an attacker can deny other users' access to the system.
**E**levation of privileges: means an attacker can have more rights in a system than they should have.



### 3.3.2 Means of the Attacker

Since the system was using an API, it was vulnerable for common attacks on network transmissions. To understand the approach of the attackers the group decided look at the means in which they could attack:

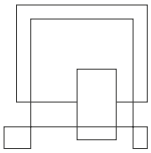
**Passive attacks**: which encapsulate ***eaves dropping*** (the attacker listens in and looks at the information sent), and ***traffic analysis*** (the attacker only looks at who is sending and how much is sent).

**Active attacks:** that summarizes ***replay*** (the attacker resends an old message), ***blocking*** (the attacker stops a message from arriving), and ***modification*** (the attacker changes the data sent, and injects a new message of his own).

### 3.3.3 EINOO

The group followed the ENIOO classification because it helped them to understand who the attacker could be and where the possible attack could take place.

**E**xternal attacker: The attackers who are not legal users of the system.
**I**nsiders: Attackers who are registered as a user to the system and have an amount of access to start off with.
Secondly, we can start looking at where the attack is coming from.
**N**etwork attacks: Attackers can only listen and might be able to modify network traffic by using *active* and *passive* attacks.
**O**ff-line attacks: The attacker get an unauthorized access to information by a disk or other media. The attacker can steal/modify the given information.
**O**n-line attacks: Attacker breaks into the system and observes how the running program is handling sensitive information. The attacker might be able to read secret keys from the RAM or modify what is show on displays to users.

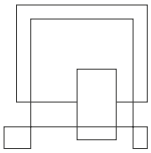


### 3.3.4 TPM

In order to prevent an attack, the TMP can be used to analyse and interpret the mistakes that can be in the group framework.
**Threat model**: The attack is possible the threat model was incomplete. The group demised certain attacks that became relevant.
**Policy:** The security policy expresses something different from what the group intended. It could for instance be too difficult.
**Mechanism:** The security mechanisms can be avoided.
When making the threat model the group focused on some threats, they thought to be relevant regarding the system, but it could very well be the wrong focus of the relevant threats. To combat this TMP model can be used.

### 3.3.5 Risk assessment

Now that the threat model is identified, the next phase is how the attackers can threaten the system. The system has a login Authentication process, so for the attackers to spoof they would need to bypass this. So, the Authentication process is a vulnerable spot. Secondly, the system has different user roles, with different privileges, which are a member, an instructor and an admin. If the attackers get access to an admin account, they can threaten to wipe-out the whole system, and effectively achieve all the STRIDE goals. If the attackers find a way to bombard the system with a lot of request at once they shut down the server and make it unavailable to everybody (DDOS attack). A big concern regarding the system is the fact that passwords in the database is not encrypted. When the client makes a request the user info is visible, however the system makes use of validation for each request and the server only respond with the specific request to hide additional info.

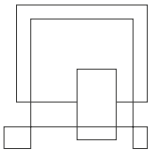


## 3.4 CIA Triad model

The CIA Triad-model is used in IT security, and stands for Confidentiality, Integrity, Availability and Authenticity. These guidelines are used to secure an IT system the best possible way.

**Confidentiality**: Means that unauthorized users do not have access confidential information.
**Integrity**: Means that only authorized users can change data and information.
**Availability**: Means that the system will continue to be available to the users.
**Authenticity**: Means that the requesting targets are authenticated and are able to prove it.

The first threat identified can be accomplished by attackers using Brute force. When unauthorized users (Attacker) want to access the system, they need to access the system as a legal “user”. For them to do that the clearest way is to find out a user’s login information. A common way to do that is trying to guess the user login. Every user won’t have an equally secure password. Some attackers have software that can guess a weak password in seconds. To prevent this problem, a software can only allow a limited of guesses up to a day this prevents hackers to endlessly trying to guess (Brute force) the password, by incrementing the alphabets and numbers. Some attackers even have a list of common weak passwords, the attackers will first to use them and if that does not work, they will try to use Brute force. Back in 2014 iCloud suffered a major leak of celebrities’ private data (**I**nformation disclosure) exactly because of this problem. On their “find my phone” access point hackers were able to have an unlimited number of guesses, making them eventually get into some celebrities’ accounts. This attack will compromise the system’s Confidentiality, Integrity and Authenticity.



The second way for attackers to threaten the system is by **D**istributed **D**enial **O**f **S**ervice attack (DDOS)[1]. This accomplished by Attackers sending malicious data or requests from multiple systems. A lot of request will be sent at once to overwhelm the server and shut it down. For example, the systems API could have so many requests that it would crash while its under demand, or the database would be hit with a high volume of queries. This would cause the servers internet bandwidth, CPU and RAM capacity being overwhelmed. This attack would compromise the system's Availability.

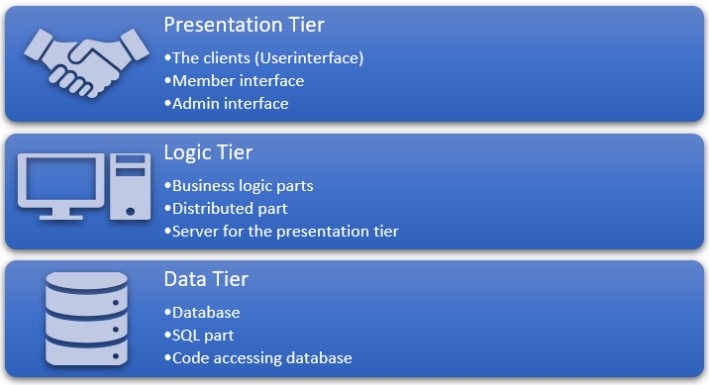
# 4 Design

In the design phase the structure of the system is decided. First and foremost, the system has a 3-tier-architecture with the connection to a PostgreSQL database. The 3-tier-architecture is central in the system alongside the GUI (Graphical User Interface) is also important piece of the software for the user to interact with the system. In addition, the system should also be heterogeneous.

## 4.1 3-tier-architecture

Requirements for this project was to make a system consisting of 3-tier architecture and that is heterogeneous, meaning that it would be coded in different languages in this case Java and C#. Furthermore, consume and expose a web service, and a protocol should be designed and used for sockets. Lastly, a GUI should be made for each client.

[1] https://www.csoonline.com/article/3222095/network-security/ddos-explained-how-denial-of-service-attacks-are-evolving.html

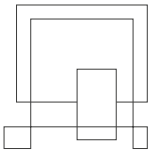



## 4.2 REST API

One of the requirements was that the system should be made using a REST API. REST stands for Representational State Transfer, which means it works the same way as a website. API stands for Application Programming Interface and allows a software system to connect with a remote application over the internet through a series of routes and endpoints. With an API a request can be made with a client, and responses are received via HTTP protocol as GET and POST from a server. APIs are an interface, which means they define a way for two entities to communicate. A client and a server that is independent of each other can communicate. With the API the applications communicate back and forth using routes. Routes consist of[i]:

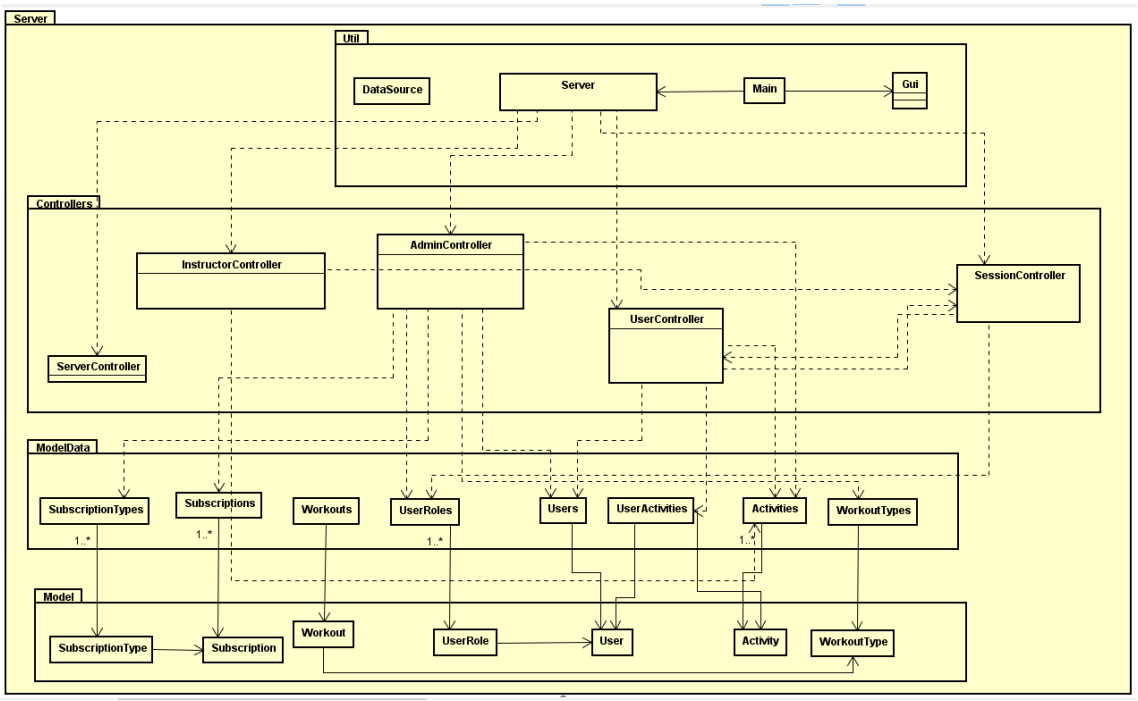
1. Verb: a method that corresponds to HTTP methods. The method includes: Get, post, put, delete, head, trace, connect and options.
2. The path: determines the routes URI(s) the route should listen to and response for.
3. The callback: is a handler function that is invoked for a given verb and path to generate the response. In other words, it is whatever the route should return.

The communication back and forth are accomplished by web services. Web services are defined as a collection of technological standards and protocols. An API is great to use when you have two applications written in different programming languages and gives



them the possibility to communicate with each other. An API helps translating between the systems through either XML or JSON. In our case JSON was used. API are mostly invisible to web surfers and run the background to provide a way for applications to communicate.

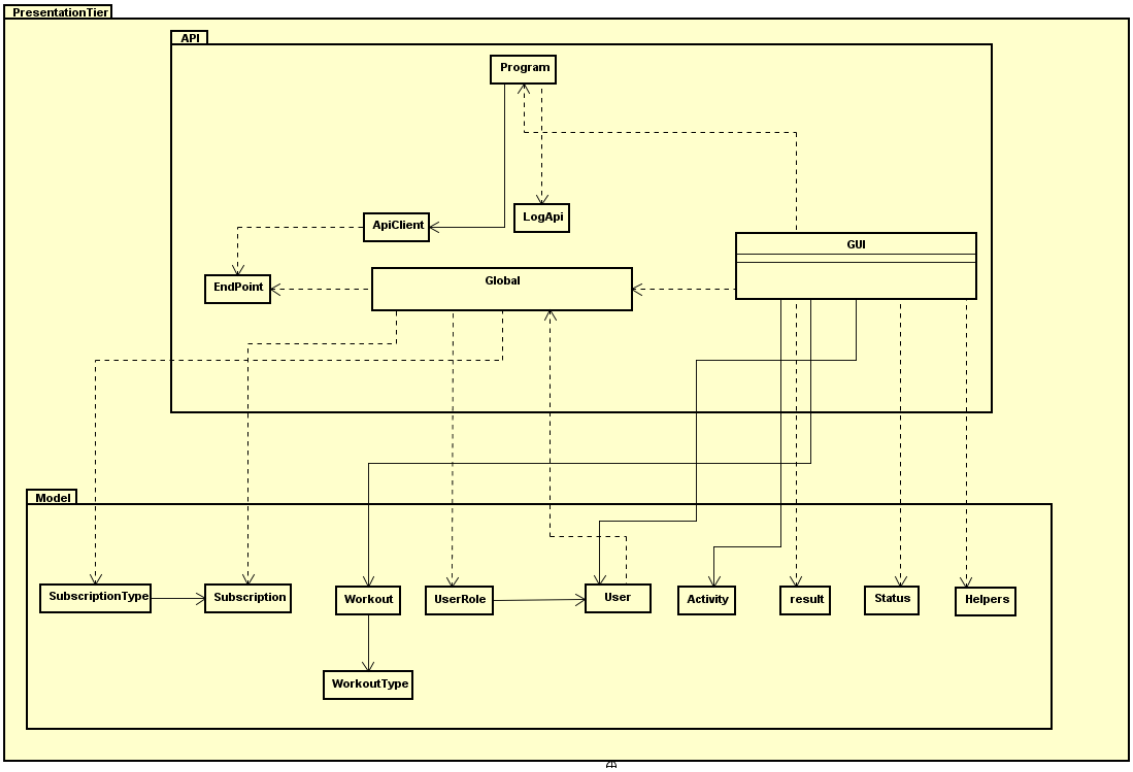
## 4.3 Data- and Logic-tier



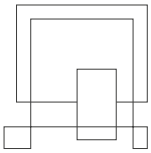

The diagram above shows the classes of the system separated into the four different packages creating an overall convenient structure for the system. The **Model package** has all the classes responsible for the business logic in the system. In the case **User**, **UserRole**, **Workout**, **WorkoutType Subscription**, **SubscriptionType** and **Activity**. These are also the classes that the domain model is based on. This is first and foremost the **logic-tier**. The second package is **ModelData** and is responsible for having all the method with the appropriate SQL queries for every class. The first method in these classes is getDataFromDataBase(), this method establishes a connection to the database and initializing variables into the temporary list of the class object. In addition, the methods accessing/manipulating databases for the respective classes is also here.

The is the system's **Data-tier**. The third package consist of **Controllers** for the system that responds to the request from the Presentation-tier (GUI). In these classes the routes for each corresponding classes' is made. Specifically, the callbacks for the routes are here. The last package is **Util.** The first class hers is **DataSource** and is responsible for connecting to the database, this is the class that the ModalData classes uses to establish a connection. The second class is the **Server**, here all the routes verbs, paths for the API made. Then there is the **GUI**, which is used for debugging and checking the API**.** Lastly**,** there is **Main**, which runs the **Server** and **GUI.**

## 4.4 Presentation-tier




In the diagram above the presentation-tier is illustrated. It consists of two packages. The first package is **Model,** which is identical to the server sides except for three new classes **result**, **Status** and **Helpers**. **Model** is also implemented in the presentation-tier to have the objects that can accept the data from the API and hence communicate with the server. All responses from the server have at least one field from the **result** class, so it

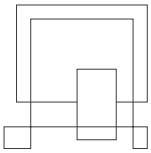


was made into a model instead of checking all the time. In other words, it can be a Status, Message or Data. The **Status** class is a small boolean that checks if the user is logged in or not. **Helpers** is a small class that returns the IDs for either an admin or an instructor.

The next package is called the **API**. This package is responsible for establishing the connection of these, and the server and client to communicate the API was used. The first class here is **Endpoint.** The **Endpoint** class consist of entry points for the routes, this is shown be having the corresponding URI for the routes. Every endpoint needed is made, so it can be used conveniently in the **GUI** to create the requests without having to write the entire URI. The **APIClient** class establishes through the endpoints a connection to the server. The client can now through the GUI request from the server and gets a response. The **LogAPI** class is used to log exceptions. The **GUI** class is responsible for handling all the request and responses through endpoints, making the forms and button handlers. The **Global** class has additional methods that's used in **GUI**. Lastly, the Program class is responsible for running **APIClient** and **GUI**.

## 4.5 Security mechanisms

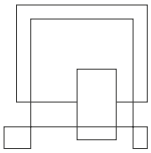
Now that the threats and possible attacks are identified, the preventive measures must be discussed. The first attack identified was using brute force to find the passwords of the users. To protect against this the Authentication process should be optimized to prevent the attacker from detecting another user's login information. For example, a user would only have 3 wrong guesses and would have to wait 10 min before trying again, if the users guess wrong again **message authentication** could be used. A message could be sent to the user's private mobile phone to login. Message authentication is a process to validate the identity of the originator, by sending message with its integrity protected. Alternatively**, a symmetric message encryption** could also be used here, where the key to decrypt the message is only know to sender and receiver. Symmetric encryption is characterized by having the same key to encrypt and decrypt know to both parties. **Asymmetric encryption** is characterized by having a different key to encrypt and decrypt. One of the keys is public key that is freely available anyone, who might want to



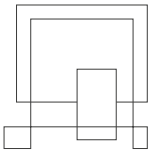
send an encrypted message, and the other key is a private to decrypt the message. This adds an extra layer of security compared to symmetric encryption, which can be compromised if the attacker finds the key. Another way to secure a message is with the use of **MAC**. MAC stands for Message Authentication Code, which uses a small fixed-sized code to authenticate a message. This is used to confirm if the message came from the original sender and has not been altered.

The software system has admins who possesses special privileges. They can manipulate data and have access to all the members confidential data, so this relies on Authorization. So, if the Admin account is hacked the attackers can do a lot damage, including ***T**ampering, **I**nformation **d**isclosure, **d**enial of services and **e**levation of privileges*. When designing the system giving all information to Admin was our choice, but from a security aspect probably wasn't the wisest, since the entire systems security is relying on the Admin honesty. The question is now how the different users in the system are protected against each other. To add the extra security measures here **Digital signatures** could be used. For example, payment related data and social security number could require a digital signature to get access to. In addition, when the user what's to confirm a payment.

The second attack identified are DDOS. Preventing DDOS completely is never possible, but there are preventive measures. Having multiple distributed servers can be effective against DDOS. This is for example how Facebook can handle these massive amounts of data and request. There multiple servers that all are connected to a master server and works as "slaves" for that server, and if it were to break down another of the "slave" server would take the master servers place.  Another way for attackers gets into access to the systems API. Luckily our API secured by our authentication through a password. Secondly some form of encryption was used whenever the HTTP protocol was accessed. So even though the attackers would have access to our API routes they would not be able to see the content. This was achieved by using HTTPS where their S in Hyper Transfer Protocol stands for secure. This involves the SSL certificate, which stands for Secure Socket Layer. That creates a secure encrypted connection between the webserver. Using https is therefore more secure. Since the system was using API, it was



vulnerable for common attacks on network transmissions. The passive attacks can be done by the attackers being connected to the same internet as the server and following the protocol. Today there are sniffing tools that allows attackers to filter data packages and collect them. So, now if the attackers have all data packages, they can analyze them. The network protocol is arranged in layers in the following order, HTML on top of HTTP on top of TCP on top of IP. Wireshark can decode all of that, and just show the HTML and collect all the packages at the right order. The attackers can now save that to a file or read it off a screen. If they collect everything, they can reassemble the original file and ***eavesdrop***. To protect against this, a form of encryption should be applied to all good data. For example, SSL/TLS, secondly it is extremely important that the internet connection the webserver uses is private and not public to everyone. The problem with passive attacker is that they are hard to detect. To protect against ***traffic analysis***, sending the same message to many severs to hide the intended receiver and randomizing the length of packages can be a useful method, secondly another method is known as ***steganography***, which means there are methods to hide information inside a message, which looks different on the outside. Active attackers are a lot easier to detect but harder to prevent. Usually active attacks can be prevented with a firewall. The firewall is a protective wall between the computer on the local network and another network controlling the incoming and outgoing network traffic. The firewall sets the rules and boundaries regarding which traffic is allowed through and which isn't. a lot of active attackers can function as man in the middle attack where they secretly relay or alters the communication between two parties, in this case networks.



# 5 Implementation

## 5.1 Implementation of Data-tier

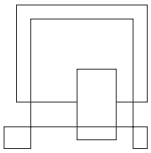
To first establish a connection to the database the HikariCP was used, which is a fast JDBC connection pooling framework. This framework utilises a maven dependency to function. The central class when using this framework is DataSource.

```
public class DataSource {

    private static HikariConfig config; // Declare private HikariConfig variable
    private static HikariDataSource ds; // Declare private HikariDataSource variable

    static {
        Properties props = new Properties(); // Make properties variable
        props.setProperty("dataSourceClassName", "org.postgresql.ds.PGSimpleDataSource"); // Set JDBC driver to PostgreSQL one
        props.setProperty("dataSource.user", "postgres"); // Set login username
        props.setProperty("dataSource.password", "ecq65tah"); // Set login password
        props.setProperty("dataSource.databaseName", "postgres"); // Set database
        props.setProperty("dataSource.portNumber", "5432"); // Set PostgreSQL server port
        props.setProperty("dataSource.serverName", "localhost"); // Set PostgreSQL server location
        props.setProperty("dataSource.currentSchema", "freshfitness");
        props.put("dataSource.logWriter", new PrintWriter(System.out)); // Return logs to System.out

        config = new HikariConfig(props); // Initialize new config
        ds = new HikariDataSource(config); // Initialize HikariCP pool
    }
```

The code snippet above illustrates how this connection is setup. HikariConfig is the configuration class used to initialize a data source. Secondly, HikariDataSource object declares the data source that is going to be used. Next a Properties object is created to set all the property info from our database. This props object is now initialized in the config, and the datasource object takes the config to now pool from the database. This framework was efficient to establish a connect to the database in very few lines.

```
public static Connection getConnection() throws SQLException {
    return ds.getConnection();
}
```

The next method called getConnection() gets the connection to the database, and is later used by the classes in **ModelData**.



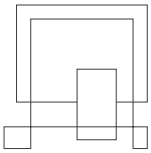
```
private static void getDataFromDataBase() {
    // Initializing users variable
    try {

        // All values from `users` table.
        String SQL_QUERY = "select * from users;";
        Connection conn = DataSource.getConnection(); // Getting connection to database
        PreparedStatement pst = conn.prepareStatement(SQL_QUERY); // Preparing the query
        ResultSet rs = pst.executeQuery(); // Executing query

        List<User> temp = new ArrayList<>(); // Initializing temporary users list -temp one so while updating there still exists

        while (rs.next()) { // Loop through all returned rows
            User user = new User(rs.getInt("id"), // Setting it's ID
                    rs.getString("email"), // Setting it's email
                    rs.getString("firstname"), // Setting it's first name
                    rs.getString("lastname"), // Setting it's last name
                    rs.getInt("phonenumber"), // Setting it's phone number
                    rs.getTimestamp("dateofbirth"), // Setting it's date of birth
                    UserRoles.getUserRoleById(rs.getInt("userroleid")) // Setting it's Role
            ); // Creating a new user

            user.setPassword(rs.getString("password")); // Setting it's password

            temp.add(user); // Add user to temporary list
        }

        users = temp; // Assigning temporary users list to users variable
    } catch (SQLException error) { // Catch any SQL errors
        System.out.println("[Error] Couldn't initialize Users data! Reason: " + error.getMessage()); // Show it to the console
    }
}
```

The code snippet above is from Users in **ModelData**, and illustrates how a List of User objects are made using the data from the database. The first line shows how the SQL query is called to get the data, then a connection is made to the database and the query is prepared and executed. A temp List of users is now made and filled out with the user data from the database, by looping through the returned rows and setting the corresponding info to user objects and then adding these objects to the List.

## 5.2 Implementation of API

```
// Activities
public static Route activityList = (Request request, Response response) -> {
    return Result.superUltraJsonData(true,
            Main.getServer().getGson().toJsonTree(Activities.getActivities())); // Return user in JSON
};
```

In the code snippet above is from **UserController** the callback route is made for activityList. Since this a Route it is specified with the parameters Request and Response. The response is returned inside, which in this case is Activities.getActivities(). This is the method takes all the activities from the database and returns them as list of activity objects. However, for this be viewed in the API it needs to be return as a JSON. The following lines makes sure the data is returned in JSON format in the API, and hence is readable to the client.



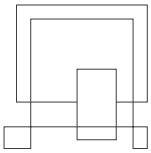
```
secure("keystore.jks", "aBKzWP2QfZx4XwHdFjtYUj56", null, null);
System.out.println("Starting API Server at " + Main.getConfig().getProperty("server.port") + " port!");
port(Integer.parseInt(Main.getConfig().getProperty("server.port"))); // Sets Spark server port
init(); // Initializes Spark server
```

```
// Admin routes
path("/admin", () -> {
    before("/*", (o,a) -> SessionController.isLoggedInAs(o, Utils.getAdminRole())); // Check if user is an admin for all

    path("/activities", () -> {
        get("/", AdminController.activityList);
        post("/add", AdminController.activityAdd);

        path("/:id", () -> {
            put("/edit", AdminController.activityEdit);
            delete("/delete", AdminController.activityDel);
        });
    });
```

In the code snippet above from the **Server** the code snippet above the spark server port is initialized. The first line makes the API HTTPS instead of HTTP. The next snippet shows how the Admin routes verb and paths are made. First the path is defined “/admin” followed by the verb before, which checks if the user is an admin for all requests. The next path is defined as “/activities” and has the verb get, which uses the callback defined in the previous snippet as AdminController.activityList. This route will now display all activities in JSON format in the following URI.

```
class Endpoint
{
    private readonly string route;
    //private static readonly Uri serverUrl = new Uri("https://localhost:8080"); // LOCAL Server
    private static readonly Uri serverUrl = new Uri("https://api3.mplauncher.pl:8888"); // GLOBAL Server

    // Endpoints
    public static readonly Endpoint STATUS = new Endpoint("/status");
    public static readonly Endpoint LOGIN = new Endpoint("/login");
    public static readonly Endpoint LOGOUT = new Endpoint("/logout");

    // Admin
    public static readonly Endpoint AdminActivitiesList = new Endpoint("/admin/activities/");
    public static readonly Endpoint AdminActivitiesAdd = new Endpoint("/admin/activities/add");
```

Snippet above is from the client-side in the class **Endpoint** and shows how the endpoints are made. First the URI for the server is initialized, since the group is using a dedicated server that is available 24/7 the URI is “htts://api3.mplauncher.pl:888”. Secondly the endpoints are illustrated. The Endpoints consist of entry points for the routes, this is shown be having the corresponding URI for the routes. They will later be used in the GUI to create the request for the server.

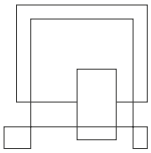


```
using RestSharp;

namespace FreshFitness.API
{
    class ApiClient
    {
        private RestClient client;

        public ApiClient ()
        {
            client = new RestClient(Endpoint.getServerUrl());
            client.RemoteCertificateValidationCallback += (sender,certificate,chain,sslPolicyErrors) => true; // Ignore SSL/TLS error
            client.UserAgent = "Jas-o-client LOL";
            client.CookieContainer = new System.Net.CookieContainer(); // We need it because server uses it for authenticationB
        }
```

Snippet above shows **APIClient** class creates a client for the server using the endpoint to get the servers URI. To setup **APIClient** RestShap was used. When using RestSharp the classes define how the data should be returned, in an attempt to remove the pain of having to parse XML or JSON. RestSharp uses classes as a starting point looping through each publicly-accessible, writable property and searching for the corresponding element in the data returned. This is also the reasoning behind identical model classes for both client and server.

```
private void refreshUserActivities()
{
    var Areq = new RestRequest(Endpoint.UserActivities.ToString(), Method.GET);
    Program.getApiClient().getClient().ExecuteAsync<result>(Areq, Aresponse =>
    {
        if (Aresponse.IsSuccessful && Aresponse.Data.status)
        {
            userActivities = JsonConvert.DeserializeObject<List<Activity>>(Aresponse.Data.data);
            refreshVisibleActivities();
        }
        else
        {
            // TODO: Error while getting user activities
        }
    });
}
```

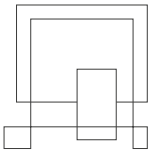
The snippet above illustrates how the endpoint creating in **Endpoint** class now is used to request data. The first line creates a request for the usesActivities to a string, using verb get. The next line the response from the Server and depending on whether successful or not show all the userActivities by converting the response to JSON for the system to use.



# 6 Test

We used our functional requirement to make test cases out of and with the test cases we had results, where we listed the outcome of a test case by giving it a pass or fail. User had to be able to fulfil a given test case, accordingly to our case descriptions without any sort of errors. The majority if the code was first tested on console and later the functionality was added to the GUI.

| Test caes | Test case description | Status |
|---|---|---|
| **User login** | The user must be able to login so that their information is secure | Passed |
| **View profile** | The user must have a profile, so they can track their information | Passed |
| **Add workout**<br>**Edit workout**<br>**Delete workout** | The user must be able to add/edit/delete workouts to their log, so they can track their progress | Passed |
| **View activities**<br>**Join activities**<br>**Leave activities** | The user must be able to see available activities to be able to join or leave activities | Passed |
| **Register member**<br>**Remove member**<br>**Edit member** | The admin must be able to add/delete/edit members, so they can manage the members and their info | Passed |
| **Create activity**<br>**Remove activity**<br>**Edit activity** | The admin/instructor must be able to add/delete/edit activities, so they can manage the activities | Passed |



**User Login**

This test will be performed when trying to login to the system. The user enters their username and password and press login. The database now needs to recognize the user for them to be logged in to the system. This is done by the method Called authenticate in the **UserController**. The method will compare username and password from the GUI's text fields to the data in the database.

**Results**

This test has been successfully completed without any errors, which means the user input gotten from the GUI text field and the database.

**View profile**

After logging in the user should be able to their profile with their personal information in the profile tab.

**Results**

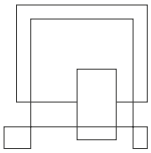
Test has been successfully completed without any error.

**Add/delete/edit workout**

After logging the users should be able add/delete/edit the different workouts to/from their log sorted by the date.

**Results**

Test has been successfully completed without any error.



**Add/delete/edit members**

After logging the admin should be able add/delete/edit the different users to/from the system.

**Results**

Test has been successfully completed without any error.

**Add/delete/edit activities**

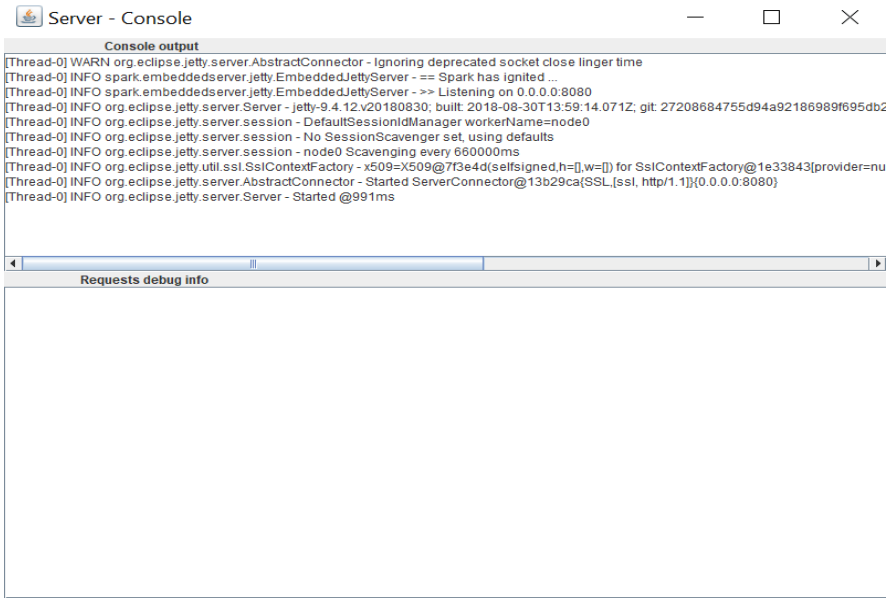
After logging the admin/instructor should be able add/delete/edit the different activities to/from the system

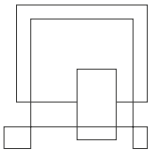
**Results**

Test has been successfully completed without any error.

**Console test**

When testing the system GUI console was made showing the spark request, in addition to the debug info. This was helpful for the group to detect were exactly the error was located and its message. When testing the system, the agile approach was used to continuously as each route was made.



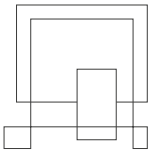


# 7 Results and Discussion

The result of this project is a functional user-based fitness app, where the users can login and track their workout progress and join and leave activities. The user can with this system keep track of their own information and subscription. The admin has separate user interface to manage the user's information and activities as well. Lastly, the instructor also has a separate user interface to manage activities. This system can also extend to other user-based system that has activities.

The group started the very first steps towards our system by doing a meeting, and in this meeting our product owner and SCRUM master were able to set the boundaries of the system, the product owner shared his vision about the chat-system and the necessary features for it, which lead to the creation of the user stories and backlogs, and these were very essential because they were the backbone of what we thought would be required for our system and they helped us to prioritize the features and stay on track.

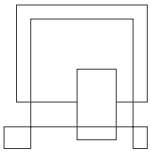
Every sprint did not go as smoothly as the group planned, sometimes they would run into errors and bugs along the way. The design was also changed continuously as we progress further. In addition, testing was done continuously through the whole process. Ones the functionality was made it was also individually tested.



# 8 Conclusion and Project future

When designing the system, it was important to keep it simple and follow the SOLID principles. A 3-tier-architecture fits perfectly into the system. It would organize all the classes in separate packages making the system more structured.

The system was fully functional with all the methods needed to fulfil all the requirements. The main problem the group had was adding the functionality of the code to the GUI. When testing the system all the use cases were tested. The most important and complicated ones were fully implemented. Despite this, the group had a decent amount of use cases which was not implemented. This was due to the flaws in sprints the group previously made. In conclusion, we can draw many lessons from this semester project. The group learned how a proper system would be managed by using SCRUM and unified process.



# 9 Sources of information

Baeldung, 2018 Building an API with the Spark java Framework, [Online]
https://www.baeldung.com/spark-framework-rest-api
[Accessed 19/12/2018]

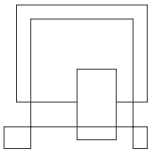
Baeldung, 2018 Introduction to HikariCP, [Online]
https://www.baeldung.com/hikaricp
[Accessed 19/12/2018]

Github, 2017 Getting Started, [Online]
https://github.com/restsharp/RestSharp/wiki/Getting-Started
[Accessed 19/12/2018]

Github, Need help with RestSharp, [Online]
https://github.com/restsharp/RestSharp/wiki/Recommended-Usa
[Accessed 19/12/2018]

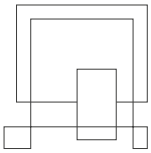
Github, Deserialization, [Online]
https://github.com/restsharp/RestSharp/wiki/Deserialization

Coulouris et al., Distributed system concepts and design fifth edition [Book]



# 10 Appendices

Check the folder Appendices



## Appendix A Project Description

Insert the original Project Description here

# Project Description

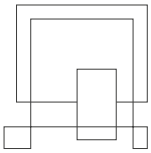
## Fresh Fitness

Group 8:

Jaser Ghasemi - 267243
Yasin Issa Aden - 267276
Modaser Ghasemi - 267251

Supervisors:

Ole Ildsgaard Hougaard
Jakob Knop Rasmussen
Christian Flinker Sandbeck
Erland Kertil Larsen
Line LIndhardt Egsgaard
Jan Munch Pedersen

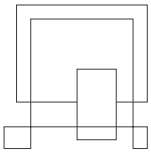


## Background Description

Humans have always sought to find the ways that make life easier. We have supermarkets, we use public transport and communicate via our phones. All these inventions share one common thought, which is to make life easier for us. Back in the stone age physical strength and stamina were vital for the survival of the human species. In order to hunt, travel and deal with the hardships of life being physically fit was important (History, 2018). Nowadays, physical activity is primary happening in the gym and for sports. Physical activity is a trend today because people realize how important it is to be active and have an environment where they can be active without being disturbed by the weather (Telegraph, 2014).

Fresh fitness is a fitness chain that wants to offer people the best equipment's and facility in order to help people maintain their active lifestyle. FF, wants to expand their brand and reach out to everyone in Denmark. They have decided to build a new center in Horsens, and they want to connect its member for events in their centers throughout Denmark. In order to reach this goal, they want to build a subscription platform where the customers are able to sign up for different workout sessions throughout all the Fresh fitness centers in Denmark. For users to be able to register for all events they need the subscription for all centers. On the other hand, the user with the subscription of the local center is only able to register for events, at that center (Activenetwork, 2018).

Currently, fresh fitness does not have a platform in place that offers their users a convenient way registers for workout sessions and track the members that are attending these sessions. Tracking the sessions and members is now by pen and paper, which is big time waste for the employees and difficult for the members keep track of.



Given the obstacles fresh fitness faces, a new software system is needed. First and foremost, they need to make a system where admins can register members and create workout sessions. Secondly, the users must be able to see the sessions available and be able to join. Lastly, fresh fitness must be able to manage the sessions and members and keep track of the members joining each session.

## Purpose

A fitness-based registrations system where the users are able to keep track off and join different workout sessions depending on their subscription.

## Problem Statement

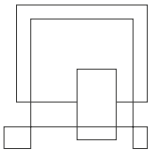
The overall challenge is to create a system where users can have their own profiles, join workout sessions and keep track off upcoming sessions. The admin should be able to manage all this. This requires a lot of data to be transferred, stored and shared. The system will also be a distributed heterogeneous system. The system's components will be located on different networked computers, which then can communicate and coordinate their actions by passing messages. Furthermore, a client-server-based system needs to be implemented, which will use different languages.

Questions to be answered are the following:

1. How will the system be designed?
2. How will the system, which are coded in different languages be able to communicate?
3. How will the system be maintainable?

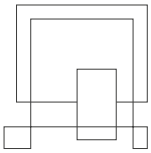
## Delimitations

- The users will not be able to look at other users' profiles(privacy)



## Choice of model and method

| **What?**<br>**Partial problem** | **Why?**<br>**Why study this problem** | **Which?**<br>**Which models/theories are expected to be used to solve the problem?** |
|---|---|---|
| How will the system be designed? | The interaction between different languages and the distributed part of the system.<br><br>The reason to study this problem is that it is an obstacle that hard to overcome, due to the fact you have a system that has two components that are build using different languages.<br><br>Furthermore, the distributed part of the system is also a challenge to design. Which architectures and design patterns will be used? | When design the system is important that is done in an efficient and structured way. Most importantly that the system should also be designed so that it is maintainable.<br><br>To do that using UML diagrams to make the system more structured will be used. Along the UML the SOLID principals will also be useful for creating a strong design. Furthermore, the architectures and design patterns used is also important.<br><br>To store all this different information, some sort of database would be very efficient. In this case, a database made in software programs such as PostgreSQL/pgadmin4 could be very useful, since it is easily transferable to Java. |
| How will the system, which are coded in | These different components need to share different data and messages with each other. The data also needs to be leak proof. | To accomplish this webservices like .net will be used. Furthermore, the parts using different |



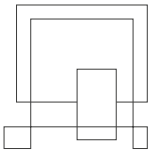
| | | |
|---|---|---|
| different languages be able to communicate? | How will they be able to communicate through webservices? How will they languages be translated to understand each other? | languages needs to translate using XML. |
| How will the system be maintainable? | The system must be easily extended and maintained later. Furthermore, the system should be ready for the organization's own management, without any bugs or complications. | Using UML diagrams to make the system more structured and designing the system.<br><br>Along the UML the SOLID principals will also be useful for creating a strong design. Furthermore, the architectures and design patterns used is also important. |

## Time schedule

Total work time: 825 hours

Total work time per students: 275 hours

Deadline: week 51 19/12-2018



## Risk assessment

| Risk categories | Probability | Impact | Effect | Response |
|---|---|---|---|---|
| Group member gets sick | Medium | High | Reduced quality of the project.<br><br>Higher workload for other members. | Redistribute group work. |
| A member breaks the group contract | Medium | Medium | Wasting the groups time. | Warnings or expulsion from group. |
| Not attending group/SEP meetings | Low | High | lack guidance/help from supervisors.<br><br>Reduced quality of the project. | Use the given resources on studynet.<br><br>Sending mails to supervisors.<br><br>Plan new meetings. |

# Sources of information

History, 2018, Stone Age [Online], Available at: <https://www.history.com/topics/pre-history/stone-age> [Accessed 20-09-2018]

Telegraph, 2014, Fitness fanatics: Why are all my friends suddenly 'training' like they're professional athletes? [Online], Available at: <https://www.telegraph.co.uk/women/womens-health/10817153/Fitness-fanatics-Why-are-all-my-friends-suddenly-training-like-theyre-professional-athletes.html> [Accessed 21-09-2018]

ActiveNetwork, 2018, The Top 10 Benefits of Online Registration for You and Your Participants [Online], Available at: <http://www.activenetwork.co.uk/event-management-resources/articles/top-10-benefits-of-online-registration.htm> [Accessed 20-09-2018]