

DECEMBER 19, 2018

# PROJECT REPORT

A SEP3 PROJECT REPORT

## Students

**Christian Sørensen, 267142**

**Cristian Guba, 254104**

**Flemming Vindelev, 251398**

**Ionel-Cristinel Putinica, 266123**

**Mihail Rumenov Kanchev, 266106**

## Supervisors

**Christian Flinker Sandbeck**

**Erland Ketil Larsen**

**Jakob Knop Rasmussen**

**Jan Munch Pedersen**

**Line Lindhardt Egsgaard**

**Ole Ildsgaard Hougaard**

## Table of Contents

1. Figures .....	3
2. Tables.....	4
3. Abstract .....	5
4. Introduction.....	6
4.1. Initial Architectural Design .....	6
4.2. Final architectural design .....	7
4.3. Multi-Language System .....	9
5. Analysis .....	9
5.1. Conception of the Core System .....	9
5.2 Security Analysis .....	10
5.3. Requirements .....	14
5.4. Use Case .....	15
5.5. Sequence Diagrams .....	20
6. Tier 3 - Database.....	24
6.1. Design decisions .....	24
6.2. Security .....	28
6.2. Web services.....	29
6.3. Use of Entity Framework and LINQ .....	29
6.4. Database Conclusion .....	30
7. Tier 2 - Middleware .....	31
7.1. MVC .....	33
7.2. Connections .....	35
7.2.1. Server Socket .....	35
7.2.2. Connection Threads.....	36
7.2.3. Why Sockets?.....	38
7.3. Database web service consumption with a Jersey client class.....	38
7.4. Middleware Testing.....	39
7.5. Designing the back-up server .....	39
7.6. Middleware conclusion .....	40
8. Tier 1 – Application .....	41
8.1. Design Decisions .....	41
8.1.1. Razor Pages.....	42
8.1.2. Project concerns .....	42

8.2. Application Security.....	42
8.3. Page model .....	45
8.4. Socket Connection.....	46
8.5. Application Conclusion .....	48
9. Testing .....	48
9.1. System Testing.....	48
9.2. Unit Testing.....	51
10. Result .....	53
11. Discussion .....	54
12. Conclusion .....	54
13. Sources of Information .....	55
14. Appendices .....	55

## 1. Figures

Figure 1: First architecture diagram .....	6
Figure 2: Updated architecture diagram .....	<b>Error! Bookmark not defined.</b>
Figure 3: Use case diagram.....	15
Figure 4: Sequence diagram for login.....	20
Figure 5: Sequence diagram for getting the rides .....	21
Figure 6: Sequence diagram for joining a ride.....	21
Figure 7: Sequence diagram for leave ride.....	22
Figure 8: Sequence diagram for create a ride .....	22
Figure 9: Sequence diagram for deleting a ride .....	23
Figure 10: Updated EER diagram.....	25
Figure 11: UML for database .....	26
Figure 12: Code for delete ride.....	27
Figure 13: Database controllers .....	27
Figure 14: Example of the login validation in the database tier.....	28
Figure 15: Code for registering a user .....	30
Figure 16: Code example of a LINQ function.....	30
Figure 17: UML for middleware.....	32
Figure 18: Console of the middleware .....	33
Figure 19: Middleware controller.....	33
Figure 20: Middleware model .....	34
Figure 21: Middleware socket .....	35
Figure 22: Middleware connection thread.....	36
Figure 23: Middleware handling a request .....	37
Figure 24: Consumption of web service .....	38
Figure 25: Middleware handling a request part 2 .....	38
Figure 26: Code of middleware test .....	39
Figure 27: UML for application.....	41
Figure 28: Application stylesheet setup .....	41
Figure 29: Application lack of MVC.....	42
Figure 30: Code for authentication .....	43
Figure 31: Code for authentication part 2.....	43
Figure 32: Code for authorization .....	43
Figure 33: Code for input validation.....	43
Figure 34: Code for input validation part 2 .....	44
Figure 35: Code for exception handling .....	44
Figure 36: Code for message display in view.....	44
Figure 37: Code for application model .....	45
Figure 38: Code for application model part 2 .....	45
Figure 39: Socket connection .....	46
Figure 40: Socket connection part 2.....	47
Figure 41: Leave ride socket protocol .....	47
Figure 42: User test .....	51
Figure 43: Ride test.....	52
Figure 44: RideList test .....	52

## 2. Tables

Table 1: Possible risks .....	12
Table 2: Requirements .....	14
Table 3: Use case for edit user .....	16
Table 4: Use case for delete user .....	16
Table 5: Use case for find a ride .....	17
Table 6: Use case for leave a ride .....	17
Table 7: Use case for monitor system .....	18
Table 8: Use case for cancel ride .....	18
Table 9: Use case for edit profile .....	18
Table 10: Use case for rate user .....	19
Table 11: Use case for offer a ride.....	19
Table 12: Use case for edit ride offering .....	20
Table 13: Test case for create a ride.....	48
Table 14: Test case for login .....	49
Table 15: Test case for create a ride.....	49
Table 16: Test case for logout .....	49
Table 17: Test case for join a ride.....	49
Table 18: Test case for leave a ride .....	50
Table 19: Test case for delete ride .....	50
Table 20: Test case for cookies.....	50
Table 21: Result high priority .....	53
Table 22: Result low priority.....	53

### 3. Abstract

The modern society has entered a mobile age, where transport becomes a necessity, thus, this distributed system has been designed from scratch in order to make transport to and from the workplace much easier for the working class. This project report has the purpose of describing all the methods, stages and iterations that went into the implementation of this project, using Java and C# as the main programming languages. The system has been fully designed with the end-user experience in mind, giving him the option to either create rides for other users to join, or to join the rides created by other users. The entire system is maintained by an admin, and, for a better stability, it has been designed on 3 tiers: a database tier, a middleware tier and a server tier.

## 4. Introduction

System architecture is the skeleton of software. It is the first problem the group tackles with and the first design that needs to be determined in order for a quality product to be produced ultimately. The team was tasked to build a three tiered, heterogeneous system, which on itself sets some boundaries over what the group can come up with, but still opens up a window for creativity and uniqueness.

### 4.1. Initial Architectural Design

The core concept of a multi-tiered system is to physically separate concrete parts of the system, applying the Loose Couple principal to the general system components and opening up an opportunity for independent functionality of all the tiers. Taking this in consideration, after an overlook on everything learned through the semester, the team ended up with the following initial architectural design.

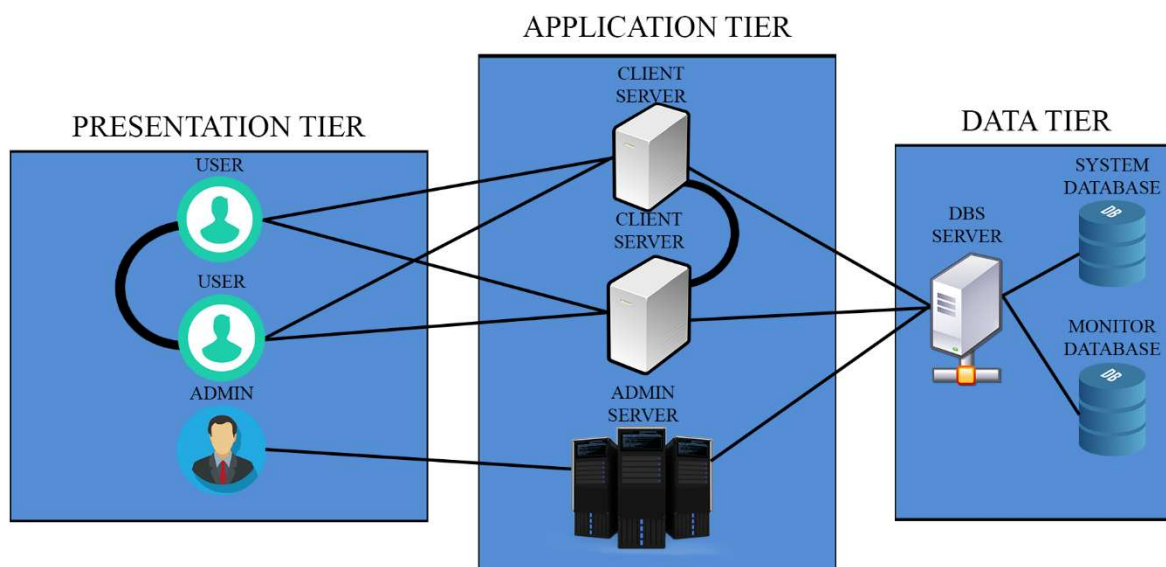


Figure 1: First architecture diagram

With the multitier design in mind the team came up with some interesting concepts that eventually got cut in order for a final working product to be presented.

After analysis the initial architectural design was established, having two databases serving system runtime data and system monitor data respectively. One database server was planned to invoke methods on them while making the connection to the middleware tier through web services exposure.

*Find more about the database tier and the dropped features in the database tier section.*

The middleware tier of the system follows up by consuming the addressed web services, transforming the extracted data into a proper model and executing the business logic on the same model. A backup server as well as an admin server were planned but never reached the final solution. *Find more about the dropped server concepts in the middleware section.*

The presentation tier was planned to include two types of accounts with different authorization and a peer-to-peer type of connection amongst clients that would send raw data to the middleware through a socket connection and invoke methods through a built protocol. *Find more about the presentation tier in the front-end section.*

## 4.2. Final architectural design

Ultimately the project ended up on a different note that generally took inspiration from the first concept. The core design was kept but some of the ideas proved to be extremely ambitious given the time span for the project.

### Overview

Eventually the majority of the creative decisions did not find their way to the final solution. Although many ideas did not get included, the team can proudly announce that the final architectural design has paved the road for the dropped features to be implemented in the near future with ease *(as the project will move on to release for local use).*



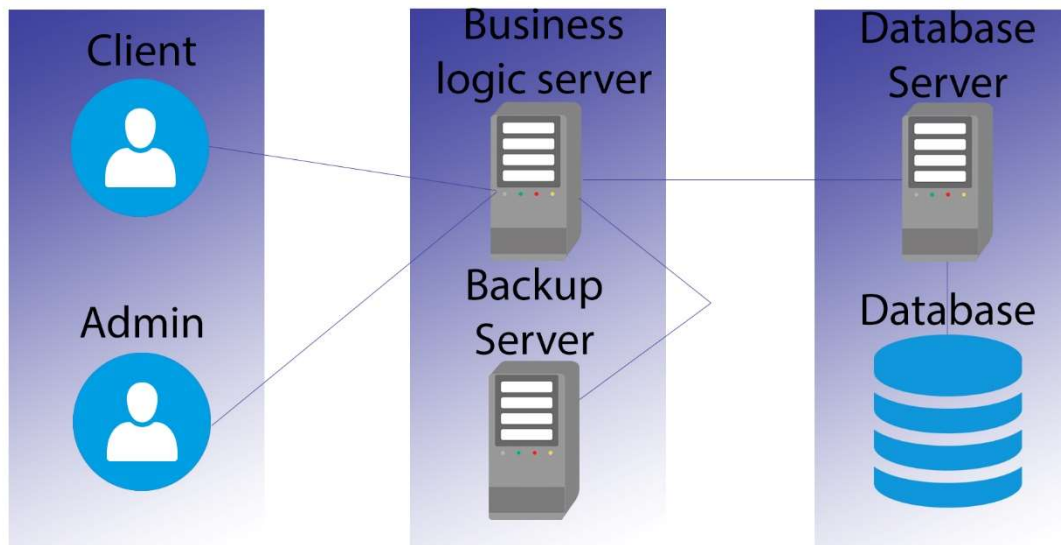


Figure 2: Updated Architecture diagram

Database tier now holds only one Database. The reasons of that change are obvious. Although having multiple databases proves to be a more secure way of distributing data, having the data stored in the same database but in different schemas and being hashed is more recourse friendly. *Find more about the database tier in the database tier section.*

Middleware tier now excludes the Admin server, which served the purpose of a secure connection to the back-end. The team decided that the server can be worthless if replaced with a clever usage of socket protocols. *Find more about the middleware tier in the middleware section.*

Presentation tier now does not include a peer-to-peer type of connection between clients, as the team set on a design that did not involve any direct communication between users. *Find more about the presentation tier in the presentation tier section.*

### 4.3. Multi-Language System

*This is a brief examination of the system analysis and system design. For more in-depth look of the different parts, read the dedicated sections.*

The opportunities of programming language usage are endless and many ingenious designs can be exploited in order to avoid awkward solutions that correspond to the set requirements, but the team settled on a more streamlined approach, assigning some value to the ultimate decisions.

The initial idea was to completely avoid the Java language in order to keep the code consistent while using the Java language for a socket that triggers the back-up middleware servers. The idea was eventually scrapped, because the team decided that it would not be humorous to the examiners.

**The final approach sets on the following choices:**

- Razor pages would provide a simple GUI and Security.
- Entity Framework and LINQ would keep the code consistent.
- Asynchronous programming will prove useful for a system that takes time to return a response.

Taking these statements into consideration, the team decided to use C# for the Presentation tier and the Database tier. In order for all project requirements to be completed Java language was picked for the middleware tier.

## 5. Analysis

*It is recommended to read the project description, which describes all the thoughts this project is built upon. And generally it contains a lot of foundational analysis for this project. The project description can be found in appendix 1.*

### 5.1. Conception of the Core System

It is the first time the team tackles a challenge of this size. Initially the vision of the system was very hard to grasp. Plenty of analysis was done taking inspiration from similar platforms but most of the proposals either felt like exact copies of already existing systems or did not reach the goal. As a result of the difficulties in the analysis phase, the team conducted a prototype market research that had the aims to set the system vision and motivate the group members. The addressed market research not only boosted group

motivation but gave some meaning to the overall project proving to the team that if the system was implemented properly, it could be deployed as a full time platform and help student society by providing crucial services.

From the conducted research, multiple students were asked about potential features that they would love to have in an imaginary system. A huge part of the feedback proved useful and ignited series of ideas which set the vision of the platform. *To see the survey used for analytic purposes, please see appendix 2.*

Most of the concerns of all students indirectly involved questions about the domain model and security. After documenting the responses, the statistics of the collected data outputted the following mode. People were not worried about the look or the functionality of the platform, as they did not care about the system. They all agreed on a certain feature that we have not previously discussed – a rating system.

In order for a similar platform to work without monetizing is if there is a type of a proof that the person you are dealing with is reliable.

That is why the team analyzed and designed initially a rating system that eventually did not get implemented in the final product due to requirements priority. Although the rating system is of crucial importance to the product from a market standpoint of view, it did not hold any value to the project quality as it just repeated some of the features the team has already implemented. The previous state leaves a conclusion that although the team did not implement this system, as for many other dropped features, room was left for it and the design for it was done furthermore paving the road to implementation in the near future.

## 5.2 Security Analysis

System security is one of if not the most important aspect of a system because it directly correlates to personal data and directly affects all users as well as the developers and the system. In our case, the system we made offers a social service to people in the form of a car sharing web application. The damages to such system can affect everything, from a system crash to a complete loss of personal data.

Many issues needed to be addressed in order for a quality outcome. Authentication, authorization and input validation are some of the many responses the team came up with. Some of those ideas made it to the final product, some of them stayed in their design state. *Learn more details about them in the Database and Application tier sections.*

- Starting from the very back-end of the system, Database tier faces many threats and needs preventative measures. For example, if an unauthorized person gets access to the database information, he would know all passwords, mobile phones and emails of everybody that uses these services. In order for this to be avoided, information needs to be hashed before it gets saved to the database. The damage of such attack would be loss of authentication and security breach.
- Aside from keeping information in the database enact, the database needs a constant input validation in order for a damaging user input to be avoided. For instance, if a user includes a database

command in one of his names, there is a possibility of that command invoking inside the database code and deleting tables worth of data. Damages of such attack would be loss of system data.

- Another security problem, concerning the database tier would be the web service that gets exposed from it. Having a web service constantly listening for potential calls, proper authentication needs to be introduced to the Middleware, which authorizes itself in front of the Database server, before being allowed to invoke any methods from the web service. If a hacker gains access to this web service, he would be able to do whatever he wishes with all the system data, for as long as he wants.
- Going to the Middleware, the exposed socket from it does not require any authentication for a socket connection to be established. If a breacher gets access to the IP Address and Port number of the Socket Server, he can take the Identity of anyone he wants and interfere with their personal data.
- The front-end is where the most of our system security will be focused as this is where our clients get a direct access to our system. An Identity system takes care of the login and assigns cookies to users in order to establish authentication.
- Authentication checks at the beginning of every page make sure that unauthenticated users do not have any access to parts of the system that require an identity. This protects users' personal information and acts as an account system.
- Authorization can be implemented if more than one accounts are introduced to the system. Different accounts can gain different types of authorization which will provide access to content, unavailable to unauthorized users.
- Cookies will make sure that security information gets carried over the different web pages of the system, in order to keep the authentication and authorization parts consistent.
- HTTPS type of connection will make sure that users communication with the system will stay protected and will keep its integrity authenticating the system to all potential browsers.

The social aspect of the system leads to a conclusion that the team should mainly focus to keep the identity of all users secure. An attacker can benefit a lot from stealing ones identity.

If he manages to steal an account he can act as if he was that person and inflict chaos upon the system. He can create inexistent rides and fool people into joining them. He can take over somebody else's rating and use it for his own sake.

A brute force attack can be foreseen. There is no denial of service currently in the system, but a design of such feature was done and it is documented in the application section of the report.

### Authenticity

- a) Threat Model - A user can create multiple secondary accounts
- b) Objective – Creating fake/misleading reviews
- c) Incident likelihood and incident consequence – The chances of this kind of attack happening are high, and the consequences would be that the rating system will become impractical, which might lead to a need for it to be removed, since a lot of misleading reviews could be created

### Elevation of privilege

- a) Threat Model- An attacker can manage to get access to the admin account
- b) Objective – Access to features that only users with admin privileges have access to
- c) Incident likelihood and incident consequence – The chances of this kind of attack happening are low, and the consequences can be system-breaking, because the now-attacker with admin-privileges has the ability to edit/delete user's profiles, edit/delete rides, see confidential information, etc.

### Input sanitization/validation (Legal inputs)

- a) Threat model- User inputs data not supported by the system
  - b) An attacker can crash or make the application to misbehave by inputting data not supported by the system, or typing in, for example, as an email, a statement that would make the database crash or delete itself.
- Incident likelihood and incident consequence – The chances of this kind of attack happening are relatively low, but the consequences are major, because, using the proper input, the user could delete all the data in the system.

Table 1: Possible risks

Risk number	Risk description	Likelihood(1-5)	Consequence(1-5)
1	Brute force log-in	5	5
2	The attacker uses eavesdropping and decrypts messages to get private data.	5	4
3	The attacker affects the system with illegal inputs	5	5
4	An attacker creates multiple accounts to change ratings.	4	2
5	The attacker gets access to admin privileges	3	5

*This part of the report has the purpose of discussing methods of preventing different kinds of attacks.*

**Risk prevention 1.** In the case of an attacker tries to brute force another's user login details, a lock-down on that account will come into play, thus partially preventing the attack (in the event that the attempt to brute force the log-in details has not been successful after the first few attempts).

**Risk prevention 2.** To prevent eavesdropping attacks, HTTPS over TLS has been implemented into the system, which serves as authentication of the accessed website and also the protection and integrity of the exchanged data while it is in transit (by encrypting it), thus making eavesdropping a more laborious process for the attacker.

**Risk prevention 3.** In order to prevent this kind of outbreak, whenever a user that is not logged in tries to access a page which contains a feature for logged-in users only, he will be automatically redirected to the "Log-in" page.

**Risk prevention 4.** This security problem can be prevented by checking the user's input. This could be by using methods for only allowing certain types of characters and numbers as an input. In this way the system is prepared and something like a wrong input will not be possible.

**Risk prevention 5.** To be able to prevent the user from making multiple accounts, each account has to contain something that designates the unique identity of the user, such as a linking the account with NemID, or something such as a phone login (a phone number is used to confirm the creation of the account, and a code is sent to the user for confirmation).

### 5.3. Requirements

Table 2: Requirements

Functional Requirements	
High Priority	The user should be able to create an account
	The user should be able to log-in
	The user should be able to find a car to work
	The user should be able to post an offer for a ride to work
	The user should be able to cancel the ride
	The user should be able to change their password
	The admin should be able to edit/delete user profiles
	The admin should be able to monitor the system
Low Priority	The user should be able to edit the offer
	The user should be able to edit their profile
	The user should be able to rate/comment the driver/passenger
Non-functional Requirements	
The login should include denial of service	
The system will be heterogeneous	
The system will use 3-tier architecture	
The system will be coded in Java, C#, SQL, HTML and CSS	
The system will use a socket connection	
The system connection will be local	

## 5.4. Use Case

### Use case diagram

The use case diagram almost talks for itself. It shows the different functions that were planned to be in the system. As well as who should have access to them. *To get a better view of the diagram, see appendix 3.*

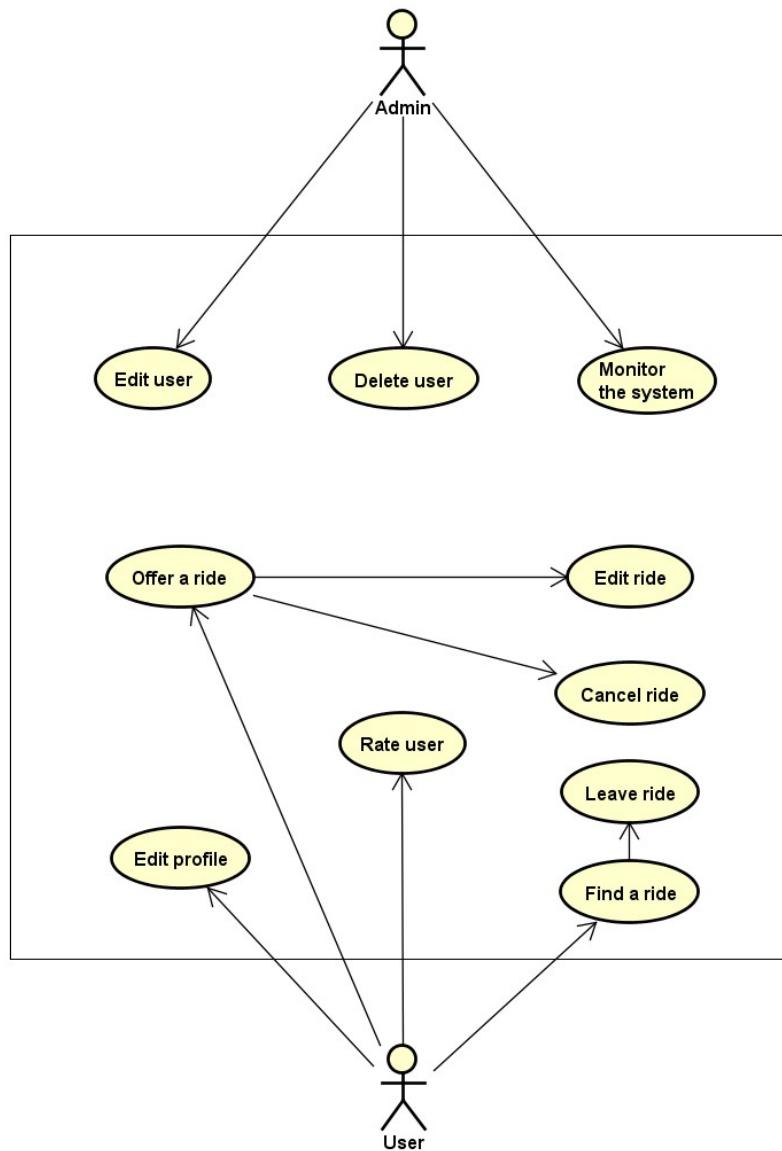


Figure 3: Use case diagram



## Use Case Descriptions

The following use case descriptions explain how each function was planned to work, and put some detail on each use case shown in the diagram above. *To get a better look at each use case description please see appendix 4.*

*Table 3: Use case for edit user*

<b>Use Case</b>	Edit user
<b>Actor</b>	Admin
<b>Pre-Condition</b>	There must be an existing user in the database for the edit to take place
<b>Post-Condition</b>	The data is stored in the system
<b>Base sequence</b>	1)Admin selects an user from the user list 2)Admin clicks the "Edit user" button 3)Admin inputs the new data for the user 4)Admin clicks the "save changes" button 5)The changes to the user are saved, data is updated all around the system
<b>Exception sequence</b>	If data is invalid: 3-5

*Table 4: Use case for delete user*

<b>Use Case</b>	Delete user
<b>Actor</b>	Admin
<b>Pre-Condition</b>	There must be an existing user in the database to be able to delete someone
<b>Post-Condition</b>	The user is removed from the system
<b>Base sequence</b>	1)Admin selects an user from the user list 2)Admin clicks the "Delete user" button 3)Admin clicks the "Confirm" button 4)The user is removed from the system, and data is updated all around it
<b>Exception sequence</b>	If data is invalid: 1-4

Table 5: Use case for find a ride

<b>Use Case</b>	Find a ride
<b>Actor</b>	User(Passenger)
<b>Pre-Condition</b>	There must be an existing ride in the system
<b>Post-Condition</b>	The passenger has joined a ride
<b>Base sequence</b>	1)Passenger clicks "Find a ride" button 2)Passenger writes specific requirements regarding the ride 3)Passenger clicks "Find" button 4)Rides are displayed on the screen 5)Passenger selects the ride 6)Passenger clicks "Join" button
<b>Exception sequence</b>	If data is invalid: 5-6

Table 6: Use case for leave a ride

<b>Use Case</b>	Leave a ride
<b>Actor</b>	User(Passenger)
<b>Pre-Condition</b>	The passenger must have applied for an existing ride in the system
<b>Post-Condition</b>	The passenger withdraws from the ride
<b>Base sequence</b>	1)Passenger clicks "My rides" button 2)Passenger selects a ride 3)Passenger clicks "Cancel" button 4)The changes are saved in the system and the driver receives a notification
<b>Exception sequence</b>	If data is invalid: 2-5

Table 7: Use case for monitor system

<b>Use Case</b>	Monitor System
<b>Actor</b>	Admin
<b>Pre-Condition</b>	The system must be running.
<b>Post-Condition</b>	System runtime details have been observed.
<b>Base sequence</b>	1) Admin selects to monitor the system. 2) System's details show up. 3) Admin selects which part of the system is to be observed.
<b>Exception sequence</b>	Server not found: 3

Table 8: Use case for cancel ride

<b>Use Case</b>	Cancel ride (Driver)
<b>Actor</b>	User(Driver)
<b>Pre-Condition</b>	The user must have created a ride.
<b>Post-Condition</b>	The offer is deleted and users are notified.
<b>Base sequence</b>	1) User goes to his profile. 2) User selects which ride is to be canceled. 3) User clicks "Cancel ride" button. 4) Ride offer is deleted. 5) Ride passengers are notified.
<b>Exception sequence</b>	If data is invalid: 2-5

Table 9: Use case for edit profile

<b>Use Case</b>	Edit Profile
<b>Actor</b>	User
<b>Pre-Condition</b>	The user must have made a profile in the system
<b>Post-Condition</b>	The user-data is changed
<b>Base sequence</b>	1)User clicks the "Edit Profile" button 2)User changes the information to their desire 3)User clicks the "Save Profile" button 4)The changes are saved, and the data is updated in the system
<b>Exception sequence</b>	If data is invalid: 2-4

Table 10: Use case for rate user

<b>Use Case</b>	Rate User
<b>Actor</b>	User
<b>Pre-Condition</b>	The user must have successfully completed a ride
<b>Post-Condition</b>	Rating will be submitted and data will be stored
<b>Base sequence</b>	1)User successfully completes a ride 2)User hits the "Rate" button 3)User gives rating (and leaves a comment) 4)User clicks "Save Rating" button 5)The rating is saved on the users profile for other users to see
<b>Exception sequence</b>	If data is invalid: 3-5

Table 11: Use case for offer a ride

<b>Use Case</b>	Offer a ride
<b>Actor</b>	User (Driver)
<b>Pre-Condition</b>	The user has an account.
<b>Post-Condition</b>	The ride is created and stored in the system.
<b>Base sequence</b>	1. The user selects that he wants to offer a ride. 2. The user enters the information about the car, the destination, date, time and amounts of seats. 3. The user clicks the save bottom and the ride is saved in the system, and automatically post the offer.
<b>Exception sequence</b>	Invalid data. Redo step 2 -3.

Table 12: Use case for edit ride offering

<b>Use Case</b>	Edit ride offering.
<b>Actor</b>	User
<b>Pre-Condition</b>	The user has already posted and created a ride offering.
<b>Post-Condition</b>	The information is going to match what the driver has planned.
<b>Base sequence</b>	<ol style="list-style-type: none"> <li>1. The user selects the ride he is offering.</li> <li>2. The user selects that it should be edited.</li> <li>3. The user changes the information that should be changed.</li> <li>4. The user selects save and the changes has been changed in the ride offering</li> </ol>
<b>Exception sequence</b>	<p>The data is not saved to the database.</p> <p>Redo step 1-5.</p>

## 5.5. Sequence Diagrams

The following sequence diagrams shows how the different methods behave in the system. It shows the sequence of which parts of the system it connects to, and what information it carries around at a given time. *To get better look at each of the sequence diagrams, please see appendix 7.*

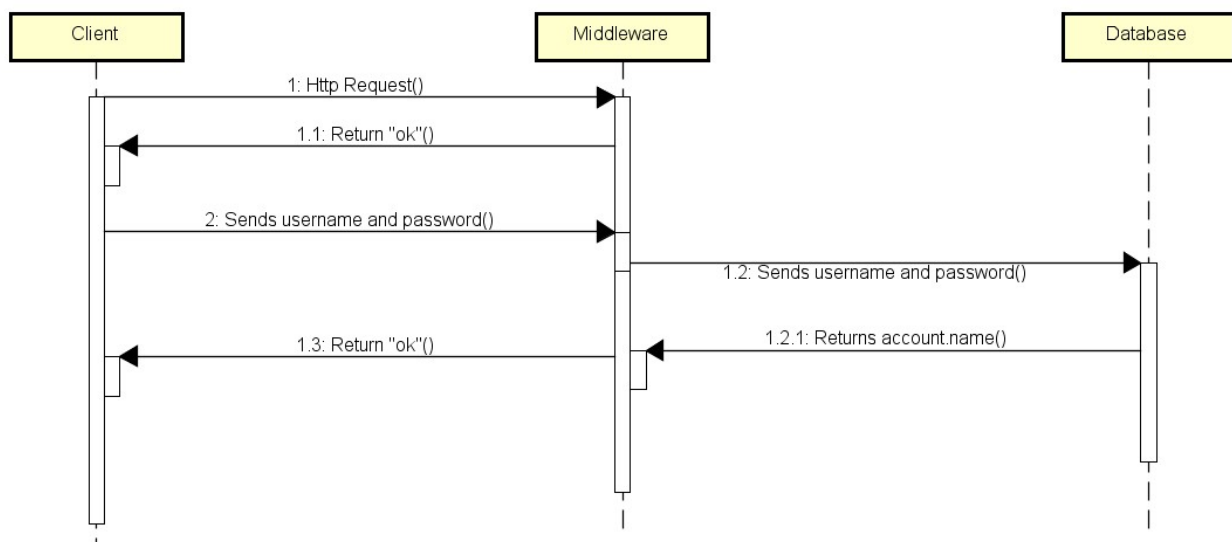


Figure 4: Sequence diagram for login

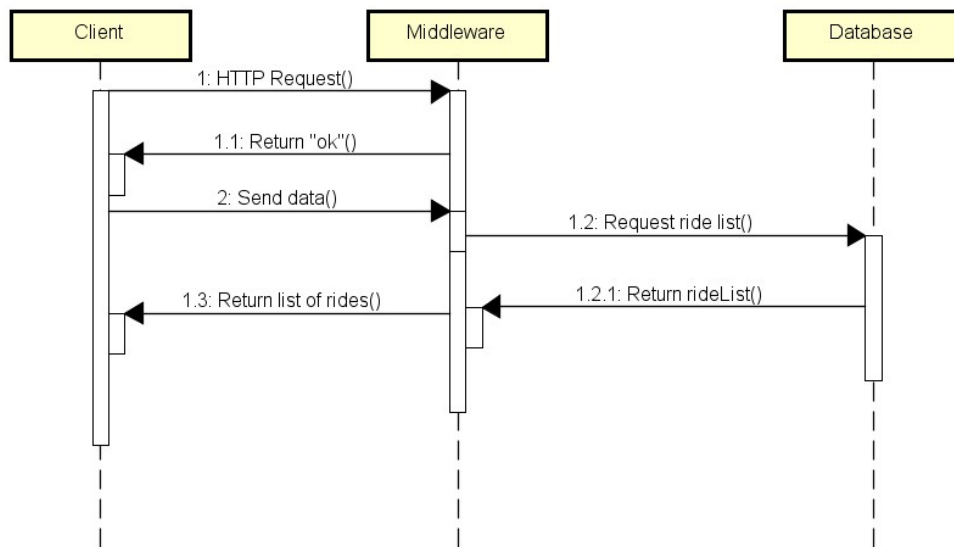


Figure 5: Sequence diagram for getting the rides

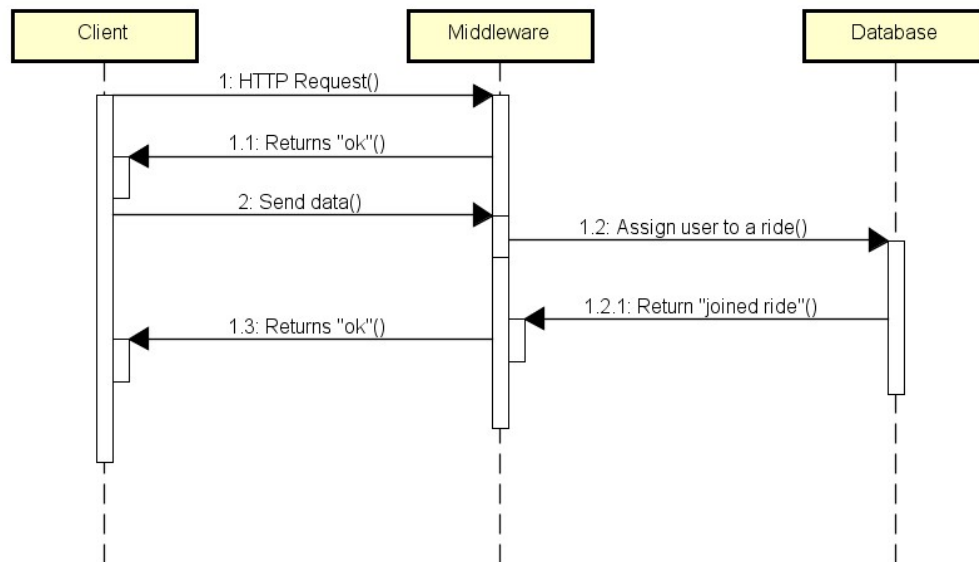


Figure 6: Sequence diagram for joining a ride

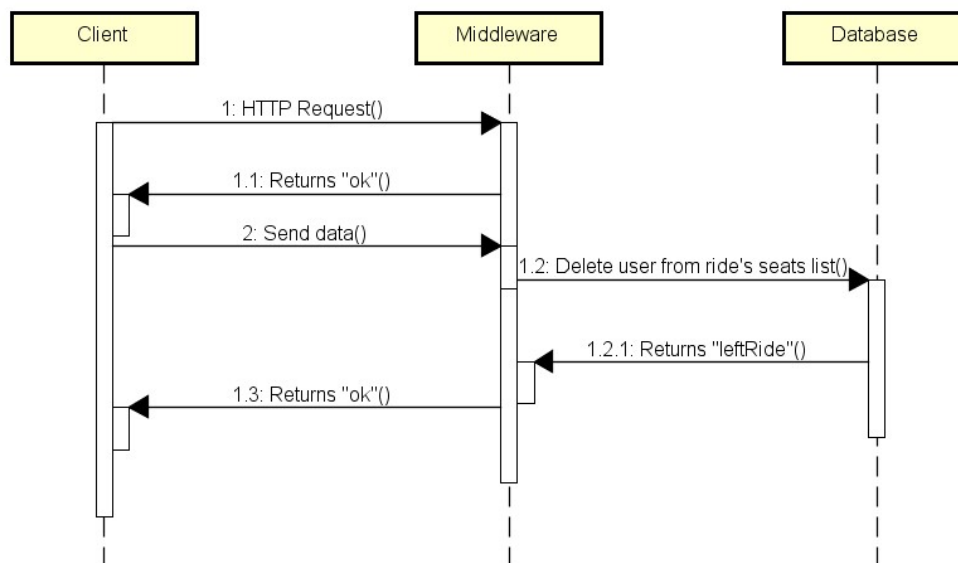


Figure 7: Sequence diagram for leave ride

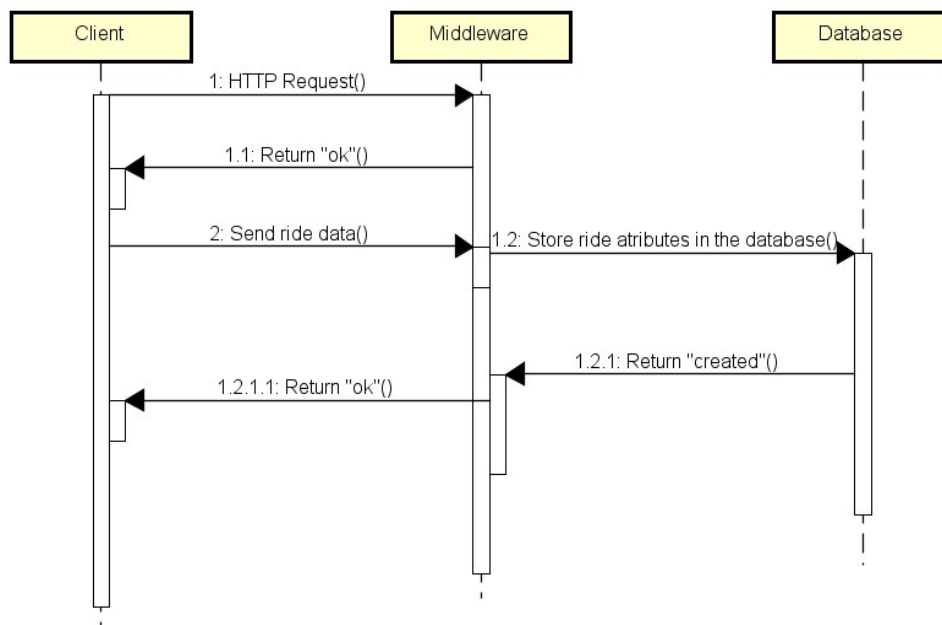
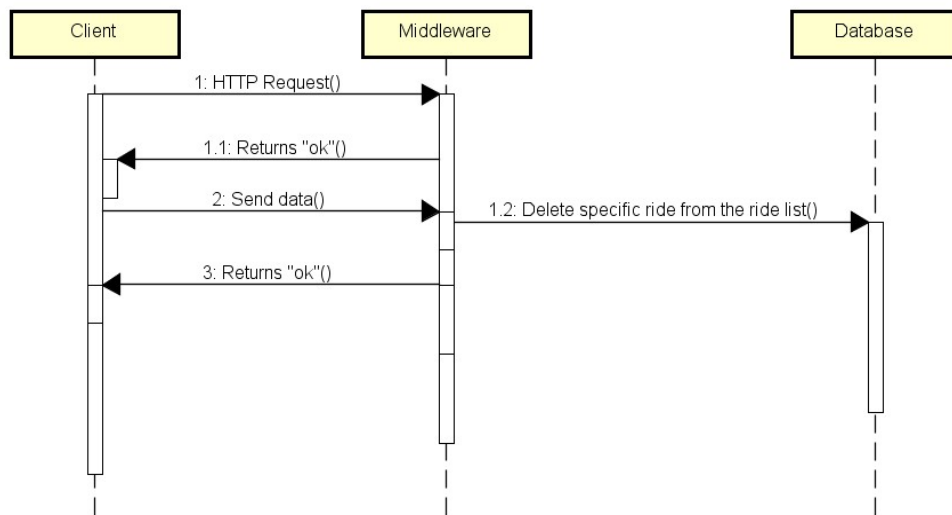


Figure 8: Sequence diagram for create a ride



*Figure 9: Sequence diagram for deleting a ride*



## 6. Tier 3 - Database

In this chapter the database tier is described and discussed in detail. All of the thoughts surrounding this tier, as well as discussions on the things which did not make it in to the system.

### 6.1. Design decisions

Below you see an Enhanced Entity Relational Diagram, which shows the different data that is stored in the database. It also shows some unimplemented attributes, which will be addressed later.

The design is rather straightforward. The major things stored in the database are users and rides. The users are made from basic attributes. We need attributes that represent user identity in order to bind them to a given ride as either a driver or passenger. The ride holds values like destination, time and date, which make it clear for every user if this ride is useful for them or not. The Seats table is what ties everything together as the seat entities are used to bind a passenger to a ride, as well as checking how many Seats are still available in a given ride. *To get a better view of the EER diagram, please see appendix 5.*

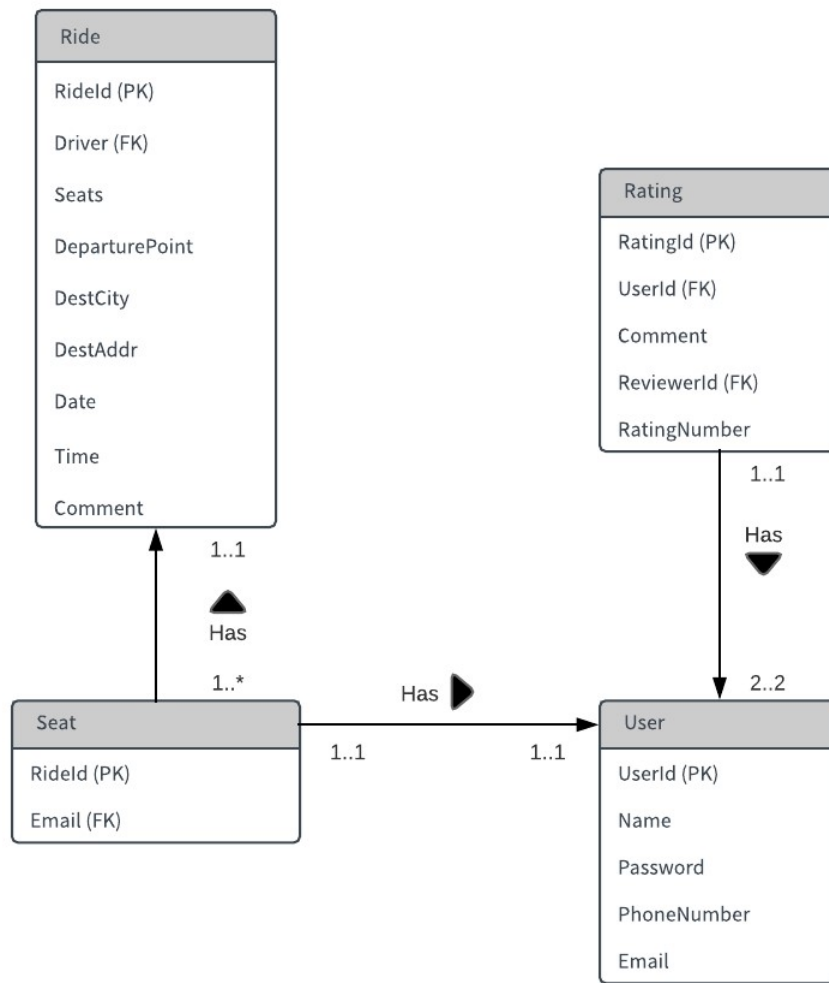


Figure 10: Updated EER diagram

The following is the UML diagram of the database tier. Each part of the diagram will be cut out and explained more in depth in the following chapters. *To get a better look at this diagram, please check appendix 6.*

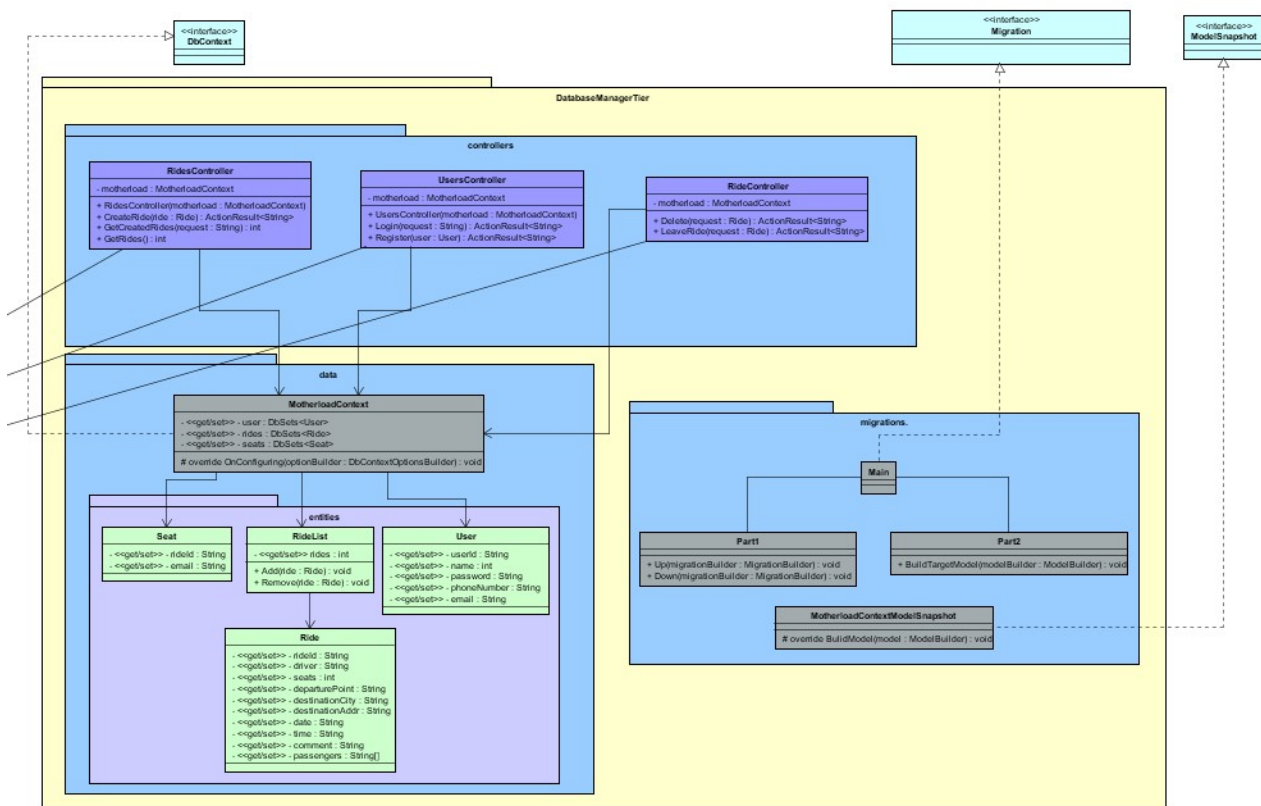


Figure 11: UML for database

## Unfinished work

Unfortunately the rating part did not make it to the implementation phase. Rating would have been something any user could give to other users. A nice to have feature which was not necessary for system functionality. This feature would however have been a nice addition, since you would be able to see if a driver/passenger had been unpleasant before. And from this knowledge you would be able choose if you want to participate in a ride.

Other than the missing table in the database, there are other features in this tier which are either missing or could be improved. One of those features is the idea of a back-up server as well as design pattern implementation as both would prove to be a huge increase in system performance and reliability.

Some poor implementations can be noticed if the CRUD functions of this tier are taken under inspection. Lack of consistency is seen amongst these methods, as for example the Delete function is used with delete, put and post methods respectively.

```

[HttpPut]
public ActionResult<String> Delete([FromBody] Ride request)
{
    try{
        var query = from r in motherload.Rides
                     where (r.Driver == request.Driver && r.Time == request.Time && r.Date == request.Date)
                     select r;
        Ride ride = new Ride();
        foreach(var q in query){
            ride = q;
        }
        motherload.Rides.Remove(ride);
        motherload.SaveChanges();
        return "ok";
    }
    catch(Exception e){
        return "bad";
    }
}

```

Figure 12: Code for delete ride

Due to time limitations in order to handle some mischievous exceptions, the team used different controllers instead of routes for most of the method calls. Another example of poor implementation.

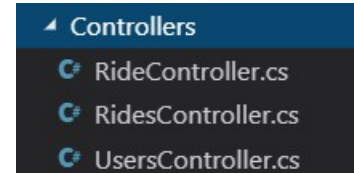


Figure 13: Database controllers

Although most of the major cases in which the runtime can go downhill are addressed, many exceptions are left unattended and the lack of validation before addressing the Database can prove to be fatal for the whole system.

Custom NotFound responses are built in order to shorten the data sent through the web service.

Different outcomes to the calls return different values, for instance the database tier makes the login validation for a user and returns a response, depending on if the email, password or both are incorrect.

```

[HttpGet("{Request}")]
0 references
public ActionResult<String> Login(string Request)
{
    try{
        string[] user = Request.Split(",");
        var account = motherload.Users.SingleOrDefault(Users => Users.Email == user[0]);
        if(account == null){
            return "notfound";
        }
        else if (!(user[1].Equals(account.Password))){
            return "password";
        }
        else{
            return account.Name;
        }
    }
    catch(Exception e){
        return "notfound";
    }
}

```

Figure 14: Example of the login validation in the database tier

## 6.2. Security

Security is a big topic in this semester and the database holds the utmost importance in the whole system. Having that in mind, the team had to discuss potential security breaches. The following topics were addressed as a result:

- Information needs to be encrypted before it is put in the database, in order to avoid data leakage in cases of database breach.
- Middleware requests need to be authenticated in order for the system to be sure that the requestant can be trusted.
- Information needs to be validated before it is stored inside the database in order to avoid potential crashes ("for instance somebody having the name of "drop users").

Sadly the team could not implement solutions to these security issues, but the topics were investigated thoroughly and a response to the respected threats was designed accordingly.

- Passwords need to be hashed and salted before they are put inside the database (as well as some personal information, to keep on par with the privacy policy).
- Authentication Tokens can be used for the middleware requests.
- Validation through if-statement checks can be executed on database writing requests.

## 6.2. Web services

Connection type is one of the big topics in this semester and designing the connections within the project was of high priority. Web services were chosen due to three reasons. Abstraction, because the components that are communicating through the mentioned services are written in different programming languages. Security, due to the statelessness of web services and the HTTPS type of connection. And finally the process intensity, as the alternative (sockets) involved a perpetuate connection, web services are stateless and require no constant connection.

## 6.3. Use of Entity Framework and LINQ

By utilizing EF and LINQ, the creation of a database as well as data access becomes quite simple. EF makes creating and updating a database really simple. By using the command “dotnet ef migrations add [migration name]”, you can create “snapshots” of the database. And by using the command “dotnet ef database update” you update the database with the latest migration added. You can also at any time revert to a previous migration, in case the database was update with a wrong model.

LINQ is a really strong and effective way of making queries. LINQ allows you to use C# in the query, this results in very specific and effective queries. As shown in the code below, there have been set a few conditions to the query in the form of boolean. The query changes the value on the boolean if the email or phone number of the user input already exists in the database. This means the user is not unique and therefore cannot register, since the system only allows one account per user. If none of the values already exists in the database, then the user successfully registers. This is just one of many ways to use LINQ.

```

[HttpPost]
public ActionResult<String> Register([FromBody]User user){
    //creates user object from the request
    //Checks if email or phone exists in database
    bool emailCheck = false;
    bool phoneCheck = false;

    emailCheck = motherload.Users.Any(User => User.Email == user.Email);
    phoneCheck = motherload.Users.Any(User => User.PhoneNumber == user.PhoneNumber);

    if(emailCheck){
        return "email";
    }
    else if(phoneCheck){
        return "phone";
    }
    //if everything is fine, adds new user to database and sends back "created"
    else{
        motherload.Users.Add(user);
        motherload.SaveChanges();
        return "created";
    }
}

```

Figure 15: Code for registering a user

But the topic being fresh, the integrated query language could not be utilized at its full potential. Although it is mainly used to access the PostgreSQL database, on further inspections it can be noticed that the code is reused on multiple occasions, contradicting the DRY principle and failing the main vision of an optimized code.

```

var query = from r in motherload.Rides
            where (r.Driver == request.Driver && r.Time == request.Time && r.Date == request.Date)
            select r;

```

Figure 16: Code example of a LINQ function

## 6.4. Database Conclusion

To put things into perspective, the team is happy with this tier. Learning how to set up a .NET database was one of if not the greatest advancements in our pool of database skills. With the introduction to EF and LINQ the whole process of database construction became more fluid and abstract. Having the whole component on a different tier opened up an opportunity to make changes and optimizations to the whole tier without crashing the system. Although the group has regrets on how the implementation phase got executed, we find that the gained experience will prove worthy in future projects.

### Subpar database design

Although the current Enhanced Entity Diagram tackles the matter efficiently, it would tremendously fail upon a system expansion. The data is mainly stored in Strings which greatly delimits the optimization part of the code. Having a Database as a way to store Data offers opportunities, which the team failed to utilize completely.

## 7. Tier 2 - Middleware

### Overview

The middleware tier is where the business logic of the system is located and serves as a bridge between the back-end and the front-end of the system. In this case the business logic is written in Java using the Model View Controller framework. The connection to the back-end is made possible through a web service consumption using JAX-RS (Jersey library). The connection to the front-end is established through sockets and protocols.



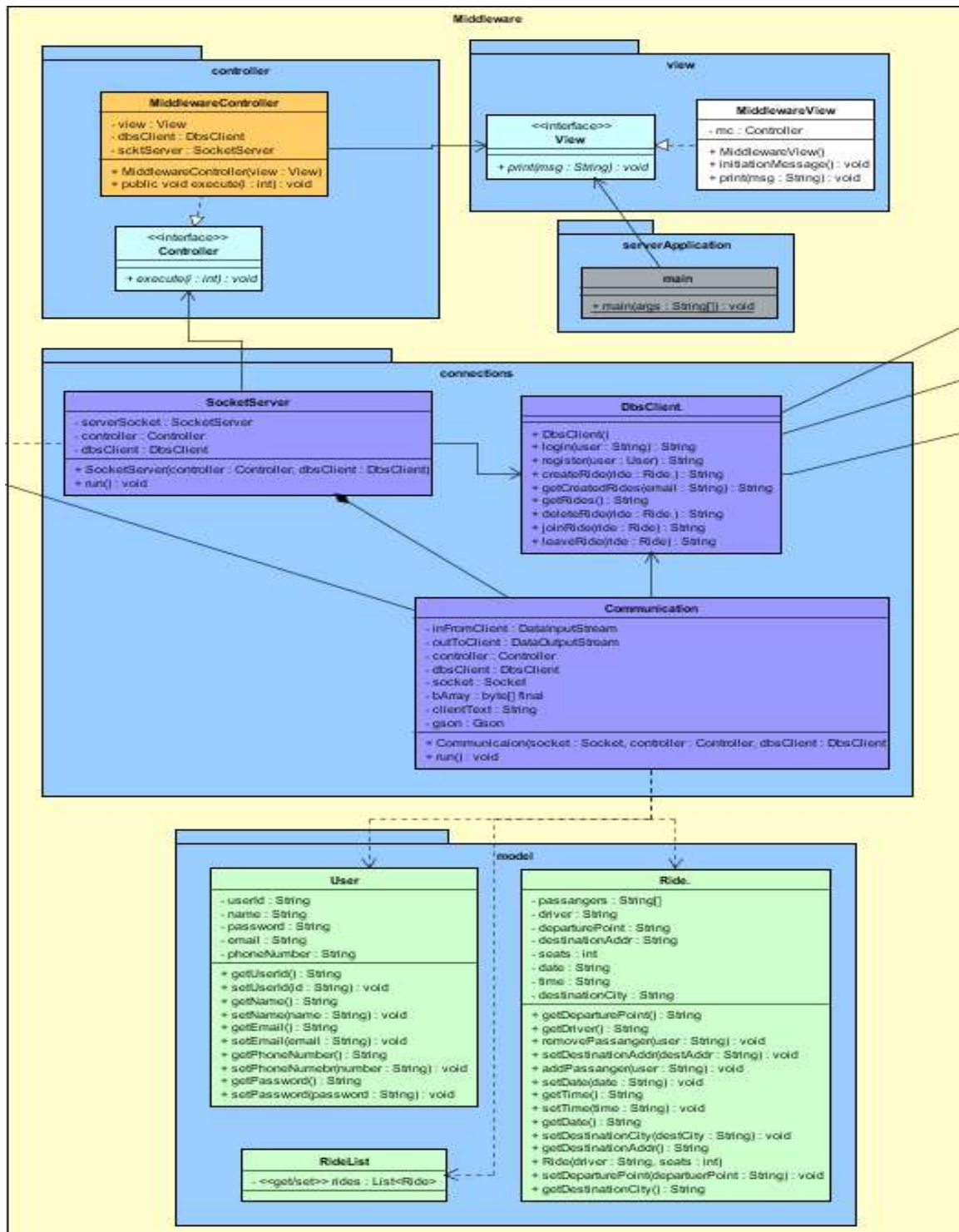


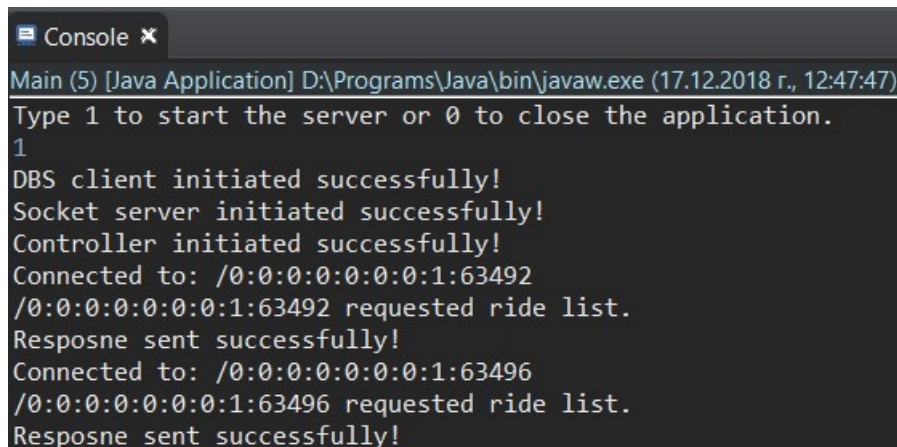
Figure 17: UML for middleware

Please visit appendix 6 for a cleared version of the UML class diagram.

## 7.1. MVC

### View

After the team decided to drop the monitoring database, we chose to keep the concept but use a different approach. The new idea was to implement the monitoring part as a Middleware View.



```
Console x
Main (5) [Java Application] D:\Programs\Java\bin\javaw.exe (17.12.2018 r., 12:47:47)
Type 1 to start the server or 0 to close the application.
1
DBS client initiated successfully!
Socket server initiated successfully!
Controller initiated successfully!
Connected to: /0:0:0:0:0:0:0:1:63492
/0:0:0:0:0:0:0:1:63492 requested ride list.
Resposne sent successfully!
Connected to: /0:0:0:0:0:0:0:1:63496
/0:0:0:0:0:0:0:1:63496 requested ride list.
Resposne sent successfully!
```

Figure 18: Console of the middleware

The concept stayed through most of the implementation but the outcome is not the complete version of the feature. The design of the view was to include name for every client, exact date and time the message was received, how much time it took for the function to be completed and much more detailed messages. Although the view features were dropped, the framework allows future implementations to proceed flawlessly.

### Controller

The controller has a purpose to invoke methods on the model and display results to the view. User input is irrelevant for the system at this tier and project state. That is why the only purpose that the controller serves in our project middleware is to display information to the view.

The execute method has a switch inside, that presents information in number of ways to the view.

```
package Controller;

public interface Controller
{
    public void execute(int i, String[] list);
}
```

Figure 19: Middleware controller

## Model

From the first glance of the Middleware Model it can clearly be seen that the team failed to keep the System Model abstract as it perfectly follows the same model as in the Database tier.

This can be fatal to the security of the system because it exposes the Domain Model.

The mistake is present to the team and is intentionally made for time saving purposes. Given more time or in future development, the Middleware Model will be changed to a completely different structure, which does not correlate to the Database Model.

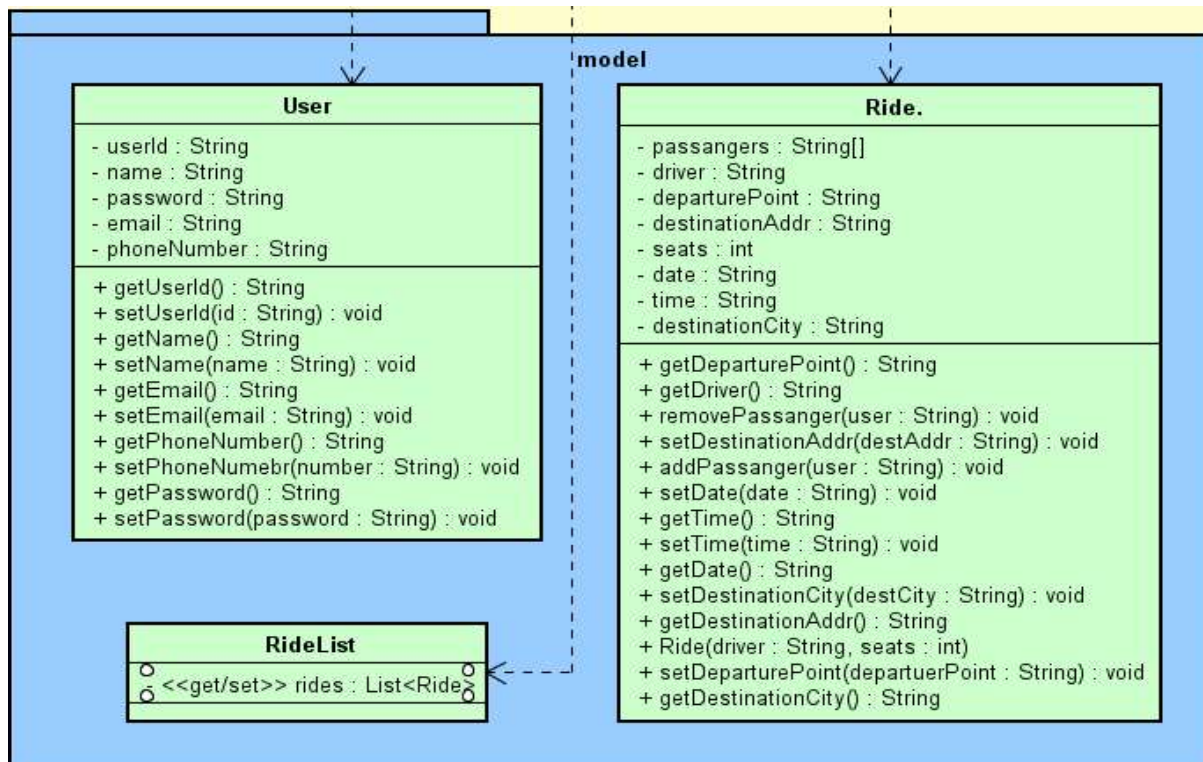


Figure 20: Middleware model

## 7.2. Connections

### 7.2.1. Server Socket

```
public SocketServer(Controller controller, DbsClient dbsClient) throws IOException{
    serverSocket = new ServerSocket(6969);
    this.controller = controller;
    this.dbsClient = dbsClient;
    this.controller.execute(0, new String[] {"Socket server initiated successfully!"});
}
//Run a loop that listens for connections.
//If a connection occurs, create a new thread and run the connection on it
//Proceed listening for other clients.
public void run() {
    while(true) {
        try
        {
            Socket server = serverSocket.accept();

            Communication clientConnected = new Communication(server, controller, dbsClient);
            clientConnected.run();
        }
    }
}
```

Figure 21: Middleware socket

The purpose of the SocketServer class is to open up a connection on a new thread whenever a client connects. It sets a port and runs on a Local host IP Address. Whenever a new client connects, the opened connection is given access to a database client, to invoke methods on the database and access to the controller, to display messages to the View whenever a request is received and completed.

There was not much testing done to this class and potential exceptions are poorly handled.

### 7.2.2. Connection Threads

```
public Communication(Socket socket, Controller controller, DbsClient dbsClient) throws IOException{
    inFromClient = new DataInputStream(socket.getInputStream());
    outToClient = new DataOutputStream(socket.getOutputStream());

    this.controller = controller;
    this.dbsClient = dbsClient;
    this.socket = socket;
    controller.execute(0, new String[] {"Connected to: " + socket.getRemoteSocketAddress().toString()});
}
//Runs a loop that listens for client requests
//and potentially responds to them.
public void run()
{
    boolean runny = true;
    while(runny) {
        try{
            int bInt = 0;
            //Tries to talk to client and if exception is caught
            //the thread is terminated and client connection
            //is closed.
            try {
                bInt = inFromClient.read(bArray);
                clientText = new String(bArray, 0, bInt, Charset.forName("ASCII"));
            }
            catch(SocketException socktE) {
                controller.execute(1, new String[] { socket.getRemoteSocketAddress().toString(), " disconnected from the server."});
                socket.close();
                runny = false;
            }
        }
    }
}
```

Figure 22: Middleware connection thread

With the initialization of every thread, new input and output streams are being initialized. Followed by granting access to the Controller and DbsClient classes. The socket is saved as a local variable. After the addressed thread gets called with the run() method, a loop starts iterating, listening for client requests. If it fails to talk to the client, the exception is caught and the controller is notified, the socket is closed and the loop is stopped.

If the client contacts the database, the byte stream is saved into an integer, which then gets saved into a String, transforming the bytes into ASCII symbols.

After the String gets successfully received, it gets compared by multiple if-statements. The first message of a sequence is always the type of the request, followed by a second message with the request data.



The following section shows an example of how a request gets handled.

```
if(clientText.equals("login")) {
    controller.execute(0, new String[] {socket.getRemoteSocketAddress().toString() + " requested login"});
    //Send a response saying that you received the login message
    //and the client can proceed with its request.
    String response = "ok" + "\r\n";
    byte[] responseBytes = response.getBytes("ASCII");
    outToClient.write(responseBytes);

    //Read the next message
    bInt = inFromClient.read(bArray);
    String request = new String(bArray, 0, bInt, Charset.forName("ASCII"));
    //Talks to database and saves the answer to string
    String answer = dbsClient.login(request);
    //Adds \r\n in the end in order to tell client
    //where to stop reading
    response = answer + "\r\n";
    responseBytes = response.getBytes("ASCII");
    outToClient.write(responseBytes);
    //Sends "Respond sent successfully" to middleware view
    controller.execute(0, new String[] {"Resposne sent successfully!"});
    socket.close();
    runny = false;
    break;
}
```

Figure 23: Middleware handling a request

After the system establishes that the request is of type “login” it precedes with an answer.

First a String is created and at the end of it a new tab and line are added. Later on they will be used by a String builder in the client to establish the end of the message. Every single response that is sent to the client must end with a new tab and new line “\r\n”.

After the String is done, it is turned into an ASCII Binary array which is then sent to the client.

Following the request approval, the client sends the request data. Same procedures are applied to the received String, which is passed to the Database Client, in order to request something from the database.

After the database response arrives, new tab and new line get added at the end of it and it gets sent to the client.

Finally the controller gets called with a message, saying that the request has been finalized, the thread loop gets terminated and socket connection gets closed.

Similar actions are applied to all other request with the difference being in the actions performed in-between the responses.

### 7.2.3. Why Sockets?

Initially sockets were planned in order to control the flow of users from the server to a potential second server, reducing the process demand on designated servers, but the feature got scrapped and the sockets stayed eventually, due to implementation advancement. The socket connection is not being used to its full extent as the connection gets terminated every time a request completes (or does not complete). The termination is a result of an exception handling in the communication between the front-end and the middleware, which initially was a temporal fix that grew into a system feature.

### 7.3. Database web service consumption with a Jersey client class

```
public class DbsClient
{
    private Client client;
    private Controller controller;

    //Constructor receives a controller instance
    //in order to invoke methods on it.
    public DbsClient(Controller controller){
        client = ClientBuilder.newClient();

        this.controller = controller;
        this.controller.execute(0, new String[] {"DBS client initiated successfully!"});
    }
}
```

Figure 24: Consumption of web service

Jersey library proved to be very handy when it comes to web services and Java. Although it was hard to find code snippets, through trial and error the team successfully figured out how to use Jersey efficiently. When a DbsClient class gets instantiated, a controller is passed to it just for initialization approval and a client gets built by the Jersey client builder.

*The following section shows an example of how a request gets handled.*

```
public String login(String user) {
    Response response = client.target("http://localhost:5000/api/users/" + user).request("text/plain").get();
    String answer = response.readEntity(String.class);
    response.close();
    return answer;
}
```

Figure 25: Middleware handling a request part 2

Following the code from the Connection class, the example shown is of a login request. A Response object is created and a target is set to the respected route (/api/users in this case). The response is requested as a plain text and the whole invocation finishes with the type of CRUD method ("GET in this case").

When a response is received, it gets written to a string class and the response gets closed. If it happens that the response object needs to be used a couple of times, the object can be buffered.

## 7.4. Middleware Testing

The most testing done was on the model, involving JUnit testing and object consistency amongst tiers.

```
@Test
public void addPassengersToList() {
    this.ride.addPassenger("John");
    assertTrue(!(this.ride.getPassenger(0) == null));
}

// tests the method is full
@Test
public void isfullTest() {
    for (int i = 0; i < ride.seats; i++) {
        this.ride.addPassenger("John");
    }
    assertTrue(this.ride.isFull());
}
```

*Black box testing of the system can be found in a designated section of the report.*

Figure 26: Code of middleware test

## 7.5. Designing the back-up server

With the inception of the project there were some ambitious features planned for the middleware, back-up server being one of them. The design that the team came up with is as follows:

A separate server would run in the background, acting like a client and constantly making call-backs to the main server, in case callbacks do not reach the main server, the back-up server would initialize and start accepting all the clients from the main server. When the main server gets back online, it becomes a back-up server until the same situation repeats. That way downtime of the system is minimized and updates can also be applied on a running system without having to restart the whole entity.



## 7.6. Middleware conclusion

As a conclusion to the final look of that tier, the team stays neutral. Many test cases were not applied and a potential system crash is inevitable if forced. Some exceptions are handled but there are still plenty of cases where things can go downhill. The back-up server idea could not be accomplished but the design still lives on and probably will be implemented next year. Sockets were not utilized completely and still remain an issue that needs to be solved. Code remains consistent through most of the requests and the View does not print enough information when it comes to request sequence tracking and unique addresses.

Ultimately the group is happy with the presented Middleware and it completes all the critical requirements. The best feature is the loose coupling of the code, which allows us to add, remove and modify all parts of the system.

## 8. Tier 1 – Application

This chapter shows the implementation of razor pages, and the general setup of the pages. It also goes over the socket protocol used to connect to the middleware. Last but not least, there is discussions on testing and security.

### 8.1. Design Decisions

Since it can prove difficult to read the diagram, you can check the full UML diagram in appendix 6.

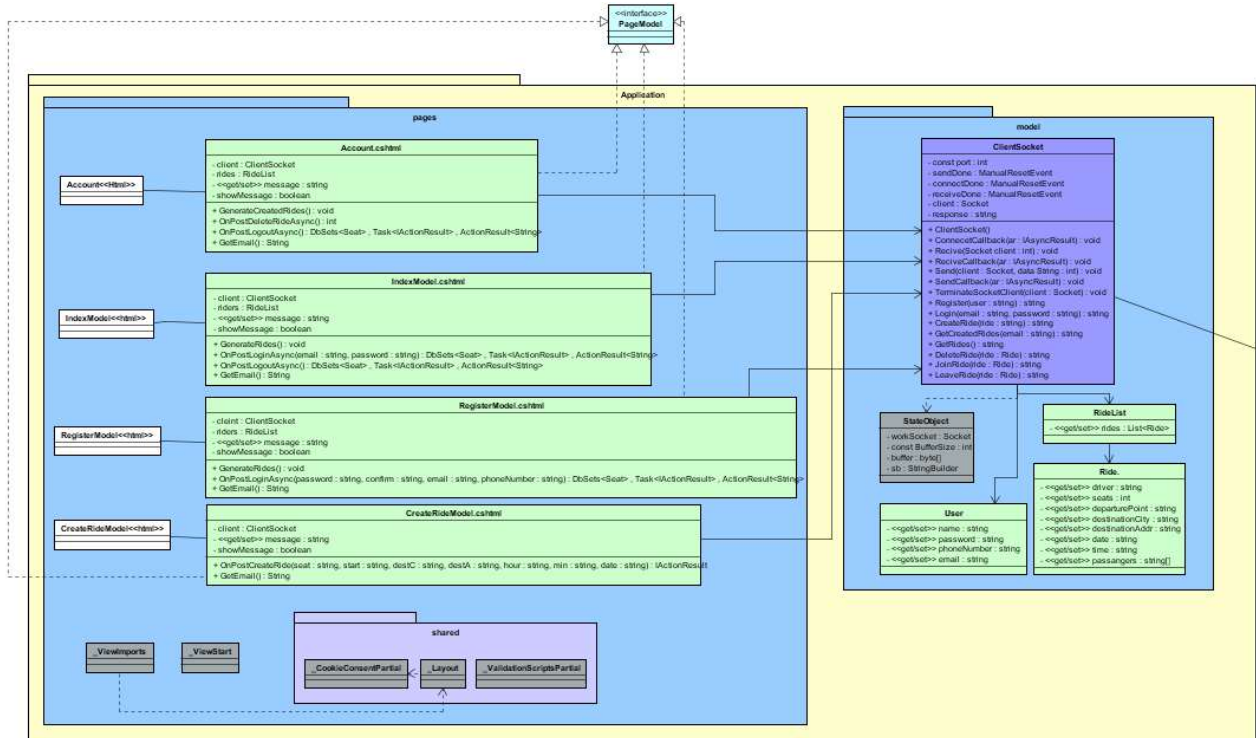


Figure 27: UML for application

The “Pages” namespace holds all visuals for the web pages, as well as the basic functionality like forms for collecting data from the users and the respective page models. Some of the pages such as “\_Layered” are general html code, which applies to all pages. In our case it is mainly used to load JQuery, Bootstrap and CSS libraries. The “Model” namespace holds parts of the system model (a topic that will be addressed in more detail in this section) and the socket protocol.

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
  <link rel="stylesheet" href="~/css/Login.css" />
  <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.3.0/css/datepicker.min.css" />
  <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.3.0/css/datepicker3.min.css" />
</environment>
```

Figure 28: Application stylesheet setup

### 8.1.1. Razor Pages

Razor pages have been chosen for the GUI part of the project, because of their loose coupling and separation of concerns. They offer a base level of security and provide useful libraries. Last but not least they give lots of options when it comes to GUI design as the usage of Bootstrap and JQuery is allowed.

### 8.1.2. Project concerns

Although the system passes requirements test cases in the black box testing, the code is not optimized. One of the goals set in the beginning was the DRY principle. Through a brief inspection it can be deduced that code is in fact repeated throughout the system and some of the logic even persists inside the Views instead of the Model as it should be.

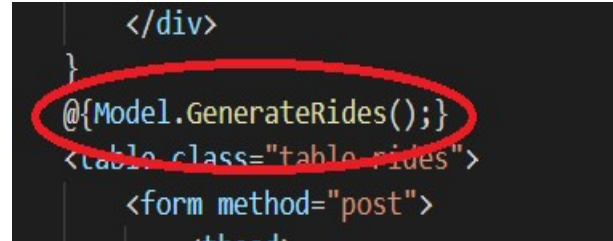


Figure 29: Application lack of MVC

Due to tunnel visioning the team did not make a full usage of the MVC framework in this tier and stored majority of the logic inside the model avoiding a controller class. A concern that will be addressed when the project is picked up in the future.

Using Razor Pages cannot always be a blessing as it comes with its negatives too. Being a web service that receives calls from cshtml pages, the method calls can be made from an attacker that is not using the system. There is no system implemented, that determines if the request is made by the system itself or by an unauthorized user.

Some responses from the Middleware get neglected completely, instead of being validated. A waste of resources that needs to be addressed in order for the code to be better optimized. Asynchronous methods appear throughout the whole tier but still some of them are left without an awaiter.

## 8.2. Application Security

There are multiple ways to secure the front end of a system. There is a small amount of authentication in the form of a login feature. Whenever the user logs in, new Claim identity gets created with his/her email and Name, which is then stored inside a Claim principle to be carried over cookies. In addition, by utilizing the login feature, one of the functions in the system called "Create a ride" has restricted access. Since a user has to log in before being able to create something in the system. This function is showed in the code below. *(Same design applies to the Register page, if the user is logged in)*

```
<div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
  <ul class="nav navbar-nav">
    @if(User.Identity.IsAuthenticated){
      <li class="active"><a href="CreateRide">Create a ride</a></li>
    }
  </ul>
</div>
```

Figure 30: Code for authentication

```
@{if(!User.Identity.IsAuthenticated){
    Response.Redirect("Index");
}}
```

Figure 31: Code for authentication part 2

Another way to protect the system is to Authorize user access to some parts of it. The lack of authorization is not unnoticed. Such feature was designed and planned to be implemented, but with the drop of some of the implementations, authorization felt more like a forced implementation than an implementation of necessity.

```
// Require authorization for all actions on the controller.
[Authorize]
```

Figure 32: Code for authorization

Input validation is one of the biggest types of security that needs to be implemented. Invalid input can cause a lot of trouble, from system crashes to data loss. The team designed a validation feature and tried to implement it inside the cshtml pages, but did not succeed. The design made use of a .Net library specifically made for Model validation.

```
[Required]
[StringLength(1000)]
public string Description { get; set; }
```

Figure 33: Code for input validation

Although the team could not use the Model validation library, a kind of validation still exists in the model, in order to avoid fatal exceptions.

```

if(string.IsNullOrEmpty(name)||string.IsNullOrEmpty(password)||string.IsNullOrEmpty(password)
||string.IsNullOrEmpty(password)||string.IsNullOrEmpty(password)||string.IsNullOrEmpty(password)){

    return Redirect("Register");
}

```

Figure 34: Code for input validation part 2

The final part of security in this tier was the way exceptions are handled. The team set a goal to handle exception that would otherwise crash system functionality, but could not finish the implementation and exceptions are handled by notifying the user of possible problems.

```

if(string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password)){
    Message = "Please fill in both email and password.";
    return Redirect("Index");
}

```

Figure 35: Code for exception handling

```

@if(Model.ShowMessage){
    <div class="alert alert-danger alert-dismissible" role="alert">
        <button type="button" class="close" data-dismiss="alert" aria-label="close"></button>
        @Model.Message
    </div>
}

```

Figure 36: Code for message display in view

The message that is displayed in the view is transferred through the pages with the help of [TempData] tag.

Denial of service was planned to be implemented, blocking the user from logging in after a couple of failed attempts but it got scrapped eventually due to priority reconstruction.

### 8.3. Page model

The purpose of the presentation tier is to display and send data to the backend. Usually it should not have access to the system Model. A principle that stays inconsistent in our case. In some methods the information that gets received stays in strings until the very end.

```
string time = hour + ":" + Minute;
string ride = User.FindFirst(ClaimTypes.Email)?.Value;
ride += "," + seats;
ride += "," + start;
ride += "," + destC;
ride += "," + destA;
ride += "," + date;
ride += "," + time;
ride += "," + comment;
```

Figure 37: Code for application model

But in some other cases a model is used and an object is created inside the presentation tier.

```
Ride ride = new Ride();
ride.Driver = Driver;
ride.Seats = int.Parse(Seats);
ride.Date = Date;
ride.Time = Time;
ride.DeparturePoint = DeparturePoint;
ride.DestinationCity = DestinationCity;
ride.DestinationAddr = DestinationAddr;
ride.Comment = Comment;
ride.passangers.Add(GetEmail());
```

Figure 38: Code for application model part 2

These practices need to be avoided in order for a three tier structure to be properly implemented.

Aside from the inconsistency in the model, the methods receive input from the View and send it to the socket class which takes on the responsibility of communicating with the middleware and returning a response.

Page models have some generic methods for displaying the email, error messages, etc. Model classes are also responsible for the majority of button actions that redirect the user to different views in the system.

## 8.4. Socket Connection

Socket type of connection was chosen initially for the communication between the Presentation tier and the Middleware tier. The reasons of that being the monitoring features that were planned in the design phase of the projects. Web services were not a good fit due to their statelessness. Sockets allow us to have a full control over every new connection.

The socket client is made to be asynchronous as the View highly depends on information extracted from the system. It uses two main methods, Send and Receive, which upon completing, notify the client.

```
private void Send(Socket client, String data) {
    // Convert the string data to byte data using ASCII encoding.
    byte[] byteData = Encoding.ASCII.GetBytes(data);

    // Begin sending the data to the remote device.
    client.BeginSend(byteData, 0, byteData.Length, 0, new AsyncCallback(SendCallback), client);
}

1 reference
private void SendCallback(IAsyncResult ar) {
    try {
        // Retrieve the socket from the state object.
        Socket client = (Socket) ar.AsyncState;

        // Complete sending the data to the remote device.
        int bytesSent = client.EndSend(ar);
        Console.WriteLine("Sent {0} bytes to server.", bytesSent);

        // Signal that all bytes have been sent.
        sendDone.Set();
    } catch (Exception e) {
        Console.WriteLine(e.ToString());
    }
}
```

Figure 39: Socket connection

Information addressing the connection is always displayed to the console. The client cannot determine when a stream of information finishes and that is why we had to also implement a protocol.



```

public class StateObject {
    // Client socket.
    2 references
    public Socket workSocket = null;
    // Size of receive buffer.
    3 references
    public const int BufferSize = 1024;
    // Receive buffer.
    3 references
    public byte[] buffer = new byte[BufferSize];
    // Received data string.
    2 references
    public StringBuilder sb = new StringBuilder();
}

```

Figure 40: Socket connection part 2

A StateObject class is used to set up a byte buffer and the size of the buffer, as well as a StringBuilder in order to progressively build a String from the received byte stream. Upon reaching the byte equivalent of new tab, new line “\t\n” the stream closes, the loop breaks, the StringBuilder is saved onto a local String inside the client and all other methods are notified that they can continue their work.

The usual protocol sends a String to the Middleware, specifying the type of request, and after it receives a response back, it sends the data it wants processed. After a response is received and saved back to the local String variable, the result is returned as a String.

```

public string LeaveRide(Ride ride){
    String request = JsonConvert.SerializeObject(ride);

    Send(this.client, "leaveRide");
    sendDone.WaitOne();
    sendDone.Reset();

    Receive(this.client);
    receiveDone.WaitOne();
    receiveDone.Reset();

    Send(this.client, request);
    sendDone.WaitOne();
    sendDone.Reset();

    Receive(this.client);
    receiveDone.WaitOne();
    receiveDone.Reset();

    return response;
}

```

Figure 41: Leave ride socket protocol



## 8.5. Application Conclusion

As a result of the tier implementation the team gained insight knowledge of a lot of .NET libraries (Model Validation, Async, etc). A lot of topics concerning security were very carefully examined and a solution to all of them was found. Although there are many features and code that need to be optimized the team acquired precious experience which expanded their understanding of Multi-tiered and Heterogeneous systems.

## 9. Testing

In this chapter, the system goes through various tests, to see if it is functional and delivers as promised. The testing is comprised of a combination of black box and white box testing. Which tests the system without looking at/utilizing the code and tests specific pieces of the code, respectively.

### 9.1. System Testing

The system test, checks if the system works as correctly from the users perspective. Does each function do what it promises and are the UI easy and understandable to read. The general goal for the system test is to test if the use cases work.

*Table 13: Test case for create a ride*

Description	Precondition	Expected Result	Steps	Results
A user wants to create a profile.	The person has the email and other personal information for creating a profile.	The user creates a profile which is added to the database, and should automatically get logged in.	<ol style="list-style-type: none"><li>1. The user presses the "Login" button.</li><li>2. The user presses the "Register" button</li><li>3. The user enter the corresponding information in the blank fields.</li><li>4. Presses the "Register" button when done.</li></ol>	Success, feature works as intended.

Table 14: Test case for login

Description	Precondition	Expected Result	Steps	Results
A user wants to login.	The user need to know the mail and password, which was used to create the account.	The user should get logged in and see a list of available rides.	<ol style="list-style-type: none"> <li>1. User presses "Login" button.</li> <li>2. User Enter email and password.</li> </ol>	Success, the feature works as intended.

Table 15: Test case for create a ride

Description	Precondition	Expected Result	Steps	Results
A user wants to log out.	User needs to be logged in to his/her account.	The user logs out of the system, and are still able to see the rides but the "Create a ride" button should be hidden	<ol style="list-style-type: none"> <li>1. User presses [Account name] in the top right corner.</li> <li>2. User presses "Logout" button</li> </ol>	Success, the feature works as intended.

Table 16: Test case for logout

Description	Precondition	Expected Result	Steps	Results
A user wants to create a ride.	User need to be logged in to his/her account.	The user creates a ride which is stored in the database and displayed under available rides.	<ol style="list-style-type: none"> <li>1. User presses "Create a ride" button.</li> <li>2. The user enters the corresponding information in the blank fields.</li> <li>3. The user presses the "Create ride" button.</li> </ol>	Success, the feature works as intended.

Table 17: Test case for join a ride

Description	Precondition	Expected Result	Steps	Results
A user wants to join a ride.	User has to be logged in to his/her account, and another user needs to have created a ride.	The user is added to the ride, and the "seats" number decrements.	<ol style="list-style-type: none"> <li>1. User finds a ride to join.</li> <li>2. User presses the "Join ride" button at the end of the ride.</li> </ol>	Success, the feature works as intended.

Table 18: Test case for leave a ride

Description	Precondition	Expected Result	Steps	Results
The user wants to leave a ride	User has to be logged in to his/her account, and have joined a ride created by another user.	The user leaves the ride, and the "seats" number increments.	<ol style="list-style-type: none"> <li>1. The user finds the ride to leave.</li> <li>2. The user presses the "Leave ride" button.</li> </ol>	Success, the feature works as intended.

Table 19: Test case for delete ride

Description	Precondition	Expected Result	Steps	Results
The user wants to delete a ride.	User have already created a ride.	The user deleted the ride, and the ride should disappear from their account page.	<ol style="list-style-type: none"> <li>1. The user presses the [Account name] button.</li> <li>2. The user presses the "Account" button.</li> <li>3. The user finds the ride they want to delete and presses the "Delete" button.</li> </ol>	Success, the feature works as intended.

Table 20: Test case for cookies

Description	Precondition	Expected Result	Steps	Results
The user wants to be logged in automatically.	User have an account and are logged in.	The user goes to the page and are already logged in.	<ol style="list-style-type: none"> <li>1. The user goes to any other site, or closes this site.</li> <li>2. The user goes back to this site.</li> </ol>	Success, the feature works as intended.

## 9.2. Unit Testing

The purpose of JUnit test was to check more deeply if every function in our project is working properly. Sometimes functions in the main program are executed well, but you cannot check every possible scenario like you can in a JUnit test program. Java was used to create the middleware of the project, which results in the following JUnit test.

The tests covered an array of different classes in the middleware being as follows: User, Ride and RideList. Every class tested passed with excellence, and all of them lives up to the requirements set. Passing a JUnit test means every step in a given method works as intended without failure. And therefore, if a mistake was to happen in the system, you know the method has probably been used incorrectly. The following pictures shows the result of all the tests, and the test code belonging to each of them can be found in appendix 8.

The first test was for the user class, it contained multiple tests which all passed successfully.

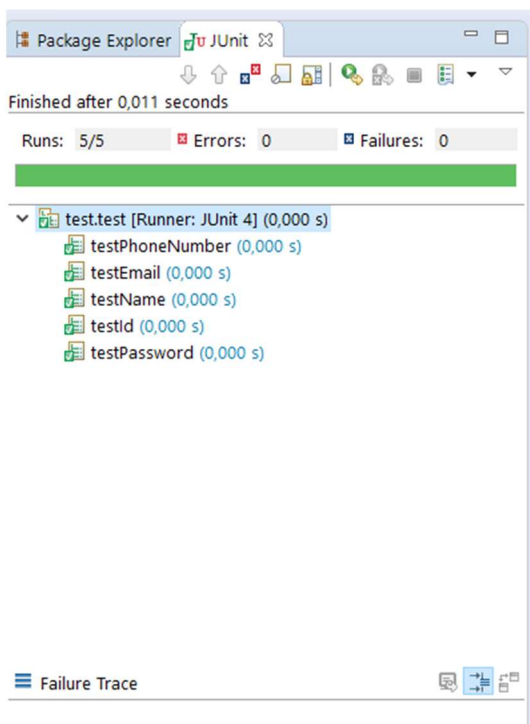


Figure 42: User test

The second test was made for the ride class, and again all tests passed.

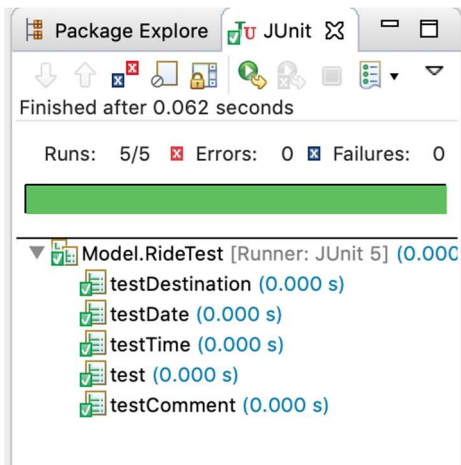


Figure 43: Ride test

The third and final JUnit test was made for the RideList class, which again proved to work as intended.

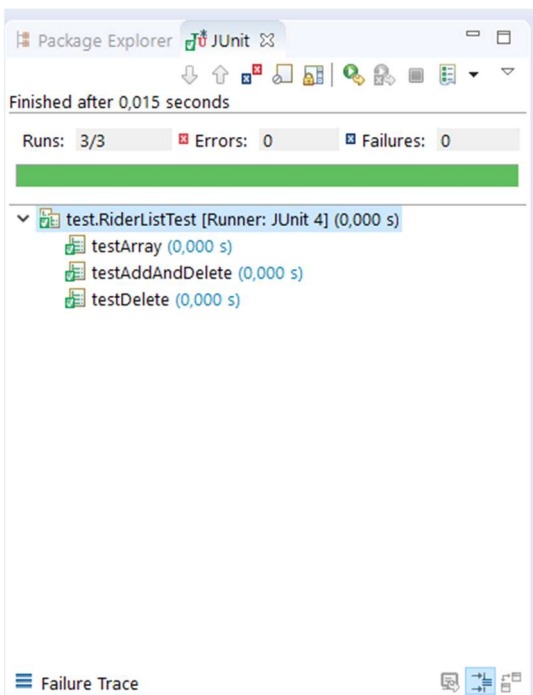


Figure 44: RideList test

## 10. Result

The “WeRide” System was developed with the purpose of meeting the requirements of a 3 tier system based on web api, storing, modifying and removing information. The final result of the system is a system which works, but unfortunately not fully implemented as seen in the tables below. The system and reports does however cover various required topics, set by the teachers as well as the students who worked on bringing the system to life. The system spoken of is a 3-tier heterogeneous system made in java and C#, with a custom socket protocol. And where security has been a big discussion point in the project.

*Table 21: Result high priority*

High Priority		
The user should be able to create an account	Implemented	Tested succesfully
The user should be able to log-in	Implemented	Tested succesfully
The user should be able to find a car to work	Implemented	Tested succesfully
The user should be able to post an offer for a ride to work	Implemented	Tested succesfully
The user should be able to cancel the ride	Implemented	Tested succesfully
The user should be able to change their password	Not implemented	Not tested
The admin should be able to edit/delete user profiles	Not implemented	Not tested
The admin should be able to monitor the system	Not implemented	Not tested

*Table 22: Result low priority*

Low Priority		
The user should be able to edit the offer	Not implemented	Not tested
The user should be able to edit their profile	Not implemented	Not tested
The user should be able to rate/comment the driver/pas-senger	Not implemented	Not tested

## 11. Discussion

When the implementation stage started, the main focus was put on the top-priority requirements, leaving the secondary ones aside. The main goal of the project was to have the top-priority requirements implemented, which would result in a better base version of the system. Throughout the project a lot of features got left out of the system, this mainly shows in the bad time management from the group.

The system showed up to be a larger mouthful than the team anticipated. Having to work on 3 different parts of the system to implement even the simplest of features, proved to take more time than predicted. Which caused the system to end up in a lackluster state.

However, it was not only negativity that came from this. The group succeeded in creating a functioning system which lives up to the requirements set by the teachers, and has the most important functionality set by the students. And more importantly, the group learned a lot about many different topics while working on this project. An important one being not to underestimate the project at hand, and carefully analyze how much work is actually related to a project before jumping into it.

Everything that went wrong in this project, is something that will be made right in the next. Improvement is key, but if nothing goes wrong you cannot improve. *It is recommended to read the user guide which can be found in appendix 9, before starting the system.*

## 12. Conclusion

The purpose of every semester project is to facilitate everything learned through the semester. The project work has to ignite interest in the targeted topics and challenge the students to come up with solutions. It strives to inspire them to dig deeper into the subjects. And it did all that for us. Although the project did not come out as we wanted it to, the group learned how to use LINQ and EF. Gained a deeper understanding of the exposing and consuming of web services. Faced against a heterogeneous system and built a communication amongst the tiers. Tackled with socket protocols and created one. Brainstormed security threats and then designed and implemented defenses. Used Razor Pages and dug deeper into .NET libraries. Learned more about principles and methodologies used in the professional environment and last but not least, we used our imagination to build system and features on top of everything learned, having fun in the process.

## 13. Sources of Information

[https://www.google.dk/publicdata/explore?ds=z5567oe244g0ot\\_&met\\_y=population&hl=en&dl=en#!ctype=l&strail=false&bcs=d&nselm=h&met\\_y=population&scale\\_y=lin&ind\\_y=false&rdim=area&idim=city\\_proper:014250&ifdim=area&hl=en\\_US&dl=en&ind=false](https://www.google.dk/publicdata/explore?ds=z5567oe244g0ot_&met_y=population&hl=en&dl=en#!ctype=l&strail=false&bcs=d&nselm=h&met_y=population&scale_y=lin&ind_y=false&rdim=area&idim=city_proper:014250&ifdim=area&hl=en_US&dl=en&ind=false) [Accessed September 20, 2018]

<https://www.dst.dk/en/Statistik/emner/priser-og-forbrug/biler> [Accessed September 20, 2018]

<https://www.uber.com/da/dk/> [Accessed September 20, 2018]

[https://en.wikipedia.org/wiki/Historic\\_roads](https://en.wikipedia.org/wiki/Historic_roads) [Accessed September 20, 2018]

<https://gomore.dk/> [Accessed September 20, 2018]

<https://en.wikipedia.org/wiki/Urbanization> [Accessed September 20, 2018]

## 14. Appendices

The following is a list of appendices which can be located in the same folder as this report.

- Appendix1\_ProjectDescription
- Appendix2\_Survey
- Appendix3\_UseCaseDiagram
- Appendix4\_UseCaseDescriptions
- Appendix5\_EERDiagram
- Appendix6\_UML
- Appendix7\_SequenceDiagrams
- Appendix8\_UnitTest
- Appendix9\_UserGuide