AALBORG UNIVERSITY

STUDENT REPORT

Software 7 - Bicycles and the Internet of Things

P7 Project by sw707f14

3-9-2014 to 19-12-2014

Participants:

Dennis Jakobsen
Erik Sidelmann Jensen
Lasse Vang Gravesen
Lars Andersen
Mathias Winde Pedersen
Søren Skibsted Als

AALBORG UNIVERSITY

STUDENT REPORT

Fourth study year
Software
Selma Lagerlöfsvej 300

Title: Software 7 - Bicycles and the Internet of Things

Theme: Internet of Things

Project period: P7, autumn semester 2014

Project group: sw707e14

Participants:

Dennis Jakobsen
Erik Sidelmann Jensen
Lasse Vang Gravesen
Lars Andersen
Mathias Winde Pedersen
Søren Skibsted Als

Supervisor:

Hua Lu

Copies: 8

Content Pages: 74

Appendix: 7

Total Pages: 88

Completed: 28-5-2014

Abstract:

The bicycle share system in Aalborg is generally subpar compared to other similar systems, to improve it would generally improve the experience of its users.

In other systems, such as the Gobike in Copenhagen the system uses GPS tracking, routing and other generally useful features. To create a website and a supporting system allowing for tracking, booking and other such features would improve the existing system and is attempted.

To do this we made a website, and supporting software for simulating stations and bicycles. We made a booking solution for the users of the system, and for the administrators we created pages to provide overview of the system on a whole.

The website ended up having many of the features already in other systems, and many new ones as well.

Foreword

This report was made at Aalborg University in the seventh semester of the Software study by the group sw707e14. The report was made as a part of the P7 project in the period 3-9-2014 to 19-12-2014. We discussed the current system with Aalborg Kommune, who's cooperation was helpful. The project was supervised by Hua Lu, whose supervision was much appreciated.

Dennis Jakobsen _____

Erik Sidelmann Jensen _____

Lasse Vang Gravesen _____

Lars Andersen _____

Mathias Winde Pedersen _____

Søren Skibsted Als _____

Contents

1 Introduction

In the current time of Danish politics, a heavy focused has been lain upon healthiness [1]. Additionally, a great focus has been placed on climate change, and how to resolve this [2]. A part of a solution to this is people bicycling more in urban areas. A way to make people bicycle more are bicycle sharing systems [3], which have appeared in several cities [4–7].

One of those systems that we focus on is Aalborg Bycykel. It is a system where several bicycle stations are placed around the city of Aalborg, and when you need a bicycle, you travel to one of those stations and pick a bicycle. Then when you are finished using the bicycle for the day you deliver it back to one of the stations. However, some immediate problems are associated with the current active system.

One of the problems with the system is that bicycles can easily get lost and there is no current way to locate the missing bicycles, other than user reports. Additionally, for a user to know if some bicycle is available, he has to walk to stations until he finds an available bicycle. Furthermore, if a user wants to be more certain that he can retrieve a bicycle in the near future, there is no way to ensure this other than retrieving a bicycle ahead of time.

These are some of the central issues that is sought to be resolved with the developed system described in the following chapters. In the developed system, we take other existing bicycle sharing systems into account [4–6]. On the basis of this, a booking and status system is developed for the users, and a tracking and statistics system is developed for Aalborg Kommune.

2 Analysis

This chapter includes analysis of the current system already in place in Aalborg, other existing systems, the problem definition, general requirements, and what the target audience for the system will be.

2.1 Current System

Aalborg Bycykel is a bicycle system where people in Aalborg are able to borrow bicycles to travel around the city. The system was started in September 2008 as part of the CIVITAS ARCHIMEDES project, which focuses on making bicycles more widely used [8]. The bicycles can be found in stations located around the city of Aalborg, for exact placement of the bicycles see Figure 2.1.

As can be seen, the bicycles are mostly located in the center of Aalborg, whereas fewer are placed in other areas.

In 2009, 135 bicycles was placed in Aalborg, and in 2012 this number had been increased to 200 bicycles [8].

In order to borrow a bicycle, you need to go to one of the bicycle stations, deposit 20 DKK to unlock the bicycle, then return it to a station when you are done with the bicycle [10], where you will then retrieve your 20 DKK, which makes the system free to use. The system is built on trust, because if people do not return the bicycles to a station when finished using them, the bicycles will gradually disappear.

Aalborg Bycykel has had, as of 2012, success with their system. According to a report, if there had not been bicycles more than half of the users would have walked and 5 percent would have driven in a car instead [8]. Furthermore, it shows that over three seasons the percentage of bicycles lost have been at maximum 11% [8].

As the bicycles are borrowed by depositing 20 DKK, it means that there is no additional monitoring of where the bicycles are located around the city, other than travelling around the city to locate the bicycles. This poses the problem that it can be difficult for Aalborg Kommune and the potential users to know which stations have bicycles. Another problem is if the bicycles are not returned to their stations after use, Aalborg Kommune has difficulty locating these misplaced bicycles.

On their website, it shows that a way for them of locating misplaced bicycles is through people reporting lost bicycles by way of SMS and voice mail, or by returning the bicycle themselves claiming the 20 DKK [11]. However, if a lost bicycle is not reported or returned by someone, the bicycle is practically lost. The company AFA JCDecaux is in charge of bicycle maintenance and storage during winter time and is also the company in charge of locating the lost bicycles [8].

2.1.1 Meeting with Aalborg Kommune

In order to gain more information about Aalborg Bycykel, a meeting with Aalborg Kommune was conducted. Through this meeting, various information was found, which is kept in mind when developing the system.

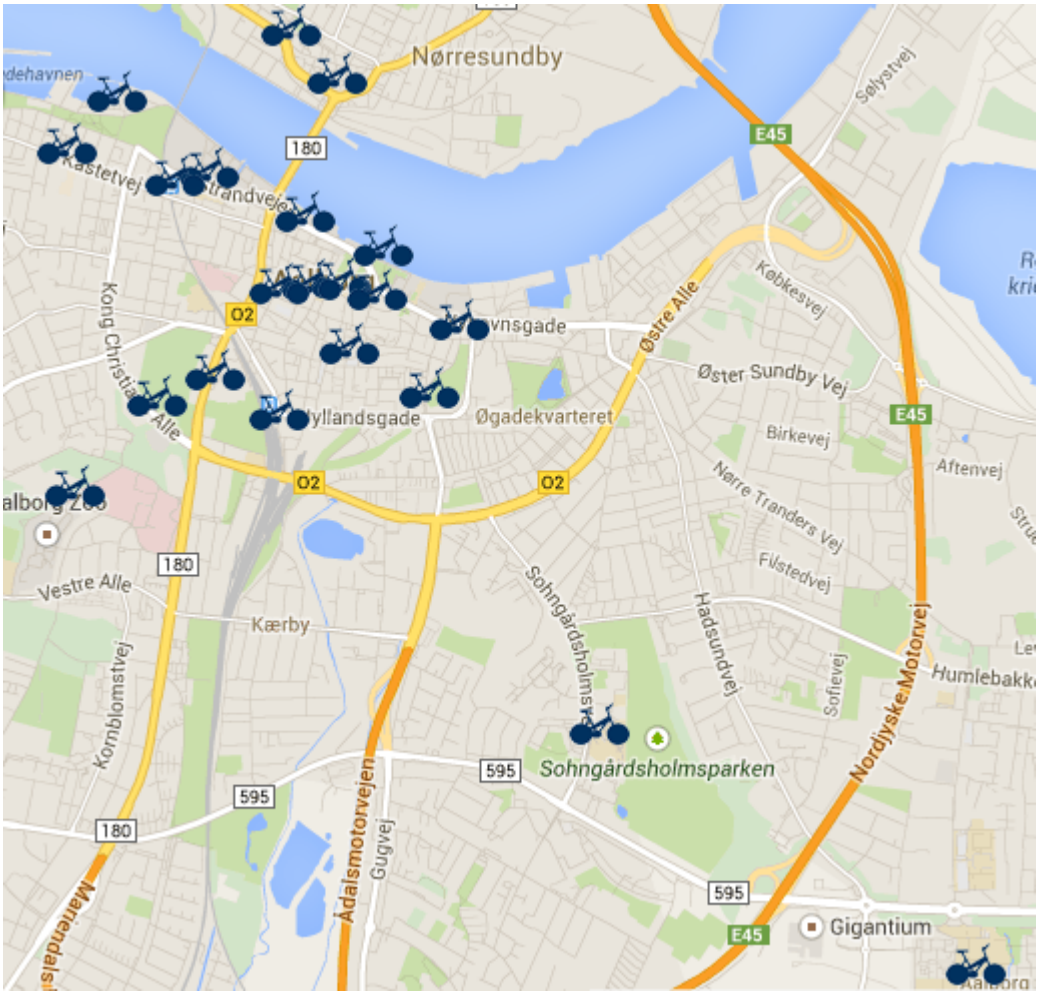


Figure 2.1: Bicycle locations [9].

For the overview of bicycles, it was found that Aalborg Kommune has no control over the bicycles when they are placed at docks in Aalborg during the season. The only information they get is if they see somebody bicycling in the city, or if a person makes a phone-call regarding a bicycle.

We also asked them if they had received any complaints, if so what those complaints consist of. This lead them to talk about how they are interested in gaining information about the usage of the bicycles. They said that the complaints they get is mostly about bicycles not being available, and also that they often see empty stations themselves. They thought that GPS tracking could be interesting, to monitor the bicycles and see how the bicycles are used, indicating that some administrator statistical page is useful.

Additionally, we asked them if they have had any thoughts on booking of bicycles. They told us that they had not thought of that but are open to the idea. However, they can also see some cons for such a system, namely that it could become too restrictive for the user since bicycles have to be made unavailable for free use because of bookings. They see that the usage of the bicycles

as much more spontaneous instead of something the user plans to do. However, when this was learned, the development of the booking part was well underway, and as such is kept as part of the system. But the system is also not strictly developed for Aalborg Kommune alone, and as such, the booking system can still be of use. Whereas, the administrator part with statistics is aimed to be used by Aalborg Kommune.

2.2 Other Existing Systems

A part of analysing and designing a city bicycle system is to investigate the current solutions on the market. Therefore an evaluation of existing systems is conducted. We found several existing systems implementing different aspects and features of a city bicycle system. These systems are listed below:

- Bycyklen in Copenhagen by Gobike
- Cibi by AFA JCDecaux
- Alta Bicycle Share by Alta

2.2.1 Bycyklen in Copenhagen by Gobike

The bicycle share system in Copenhagen is called Bycyklen and contains 1,860 bicycles and 100 docking stations [5]. Gobike is a Danish/Dutch company that designed this smart system with a bicycle they call Smart Bike. The Smart Bike is equipped with an information screen in the form of a tablet providing the user with an interface to lock the bicycle, select the level of assistance from the electrical system, navigate through the city via Global Positioning System (GPS) and explore new interest points such as cafés and shops. Furthermore, users of the bicycle system are able to check bus or train arrivals near their current location. The Smart Bike uses the tablet to send the location, who is travelling, and other statistics about the bicycle such as battery life to Gobike Admin. Gobike is currently looking at possibilities such as location based marketing, adjusting the traffic lights according to the cyclist based on the pattern of the bicycle trip, and the weather such as wind speed. Bycyklen has a very simple 3-step process to rent a bicycle.

1. Book a Smart Bike ahead from your computer or tablet, find a Smart Bike and log in via the on-board tablet.
2. Unlock the bicycle through the tablet, drive around in the city, possibly assisted by the GPS, paying by the hour.
3. Return the Smart Bike, lock the bicycle and log out of the system via the tablet.

The Smart Bike enables the user to lock the bicycle and securely park it during the bicycle trip.

2.2.2 Cibi By AFA JCDecaux

The Cibi bicycle system relies on SMS, where a booking of a bicycle is performed by sending an SMS to a special number with the ID of the bicycle slot in the docking station [4]. The user then receives an acceptance SMS and the bicycle is unlocked from the docking station. A bicycle trip is charged by the hour, and as with the bicycle share system in Copenhagen, a user can lock the bicycle during the trip. This is done through a wire lock which uses a code that the user was given in the SMS when booking the bicycle. The system also allows for checking the amount of bicycles at a station using special docks and chips on the bicycle [12].

2.2.3 Alta Bicycle Share

Alta Bicycle Share is a company that design, deploys, and manages bicycles in USA [6]. The company currently have projects ongoing in nine different cities in USA, with more than thousands of stations and over tens of thousands bicycles. Alta Bicycle Share believes that people get the best experience from the environment when the environment is sustainable and enjoyable. The bicycle stations are designed such that they can be placed everywhere in the cities without any preparation, since they get electricity from solar panels. Furthermore, the stations are using a cellular connection to upload their data to the main database, so the citizens can see if there are any bicycles at a given station. To rent a bicycle from one of Alta Bicycle Share's system, you have to use a card, which can be bought in shops near the stations. These cards then give access to any bicycle for a given time at any station, however, using the bicycle for a longer period of time can lead to a fee. The Alta Bicycle stations also allow for tracking how many bicycles are available at a given station.

2.2.4 Summary

The Copenhagen city bicycle system, Bycyklen, includes some good features such as a booking system that works using a tablet on the bicycle that is connected to the internet. It also includes a GPS used for navigation and tracking. One downside to this system, however, is that the bicycles are seemingly very expensive.

Cibi is a little different in that booking happens over SMS. At the same time it also allows for locking of the bicycle using a special code also sent over SMS. It also allows for retrieving information about the amount of bicycles at a station using a special dock and a chip on the bicycles.

Alta on the other hand uses a card to rent bicycles, and uploads information about bicycles at stations for the users of the system to easily get an overview of where bicycles are located.

	Aalborg Bycyklen	**Copenhagen Gobike**	**Cibi**	**Alta Bicycle Share**
Booking	No	On a tablet	No	No
Borrowing / unlocking	Deposit 20DKK	Login with tablet	SMS code deposit 300DKK	Keycard
Tracking	No	GPS	Dock + chip at stations	Dock at stations
Penalty	Loss of deposit	Continues payment	Loss of deposit	Fine per half hour overtime
Cost	Deposit, but otherwise free	First half hour free, then pay per hour	First hour free, then pay per hour	Depends on membership

Table 2.1: Comparison of bicycle systems.

A comparison of the different bicycle sharing systems examined can be seen in Table 2.1. As can be seen, Aalborg Bycykel is lacking a few features compared to the other systems and signifies a room for improvement. The improvements to be performed are then inspired by the other examined systems. However, Aalborg Bycykel is free to use as long as you deliver a bicycle back after use. This ease of accessibility of Aalborg Bycykel we find valuable, and is a value we

seek to keep in the developed solution. The knowledge gained is then kept in mind for defining the problem definition and the specification of the requirements for the new system.

2.3 Problem Definition

With the analysis of Aalborg Bycykel and other similar existing systems performed, it indicates that there is a lot of room for improvement. It is found that Aalborg Bycykel is very basic when it comes to technologies.

In other similar existing systems, GPS, SMS, booking, and card registration is used to manage the bicycle experience for the users. This gives inspiration to improvements that can be performed regarding Aalborg Bycykel, and leads to our hypothesis which is defined as follows.

It is possible to develop a system that makes Aalborg Bycykel more user friendly and manageable, within the context of Internet of Things.

In order to verify this hypothesis, the following questions have to be answered:

1. What are the requirements for a city bicycle booking and positioning system?
2. How can the booking and positioning system be designed and implemented?
3. Why should the developed system be used over the currently used system?

2.4 Requirements

A few system criteria have been set up that the different software and hardware solutions should conform to. In addition to this, there are software and hardware requirements listed with the highest priority first, such that the goal for the product is to fulfil the requirements in the listed order. Not all requirements will necessarily be fulfilled as time is limited, however, due to the fact that the simpler requirements might be easy to implement having more advanced ones could be necessary. We do not make any concrete decisions on the hardware, however, we determine some requirements for the hardware but how they are fulfilled is beyond the scope of this project.

2.4.1 System Criteria

Below is a set of system criteria that should be considered when choosing a solution for the system. These criteria should not be taken as requirements for the system, but as guidelines for the desired outcome of the system. In that, when choosing a solution these criteria should be considered and discussed to help choose the solution that matches them best.

- A chosen solution should not redesign the entire existing system but rather add an extension to the existing system.
- A solution should be easy to use by the end user and easy to access by everyone. It should require as few steps as possible in the process of borrowing a bicycle.

2.4.2 Software Requirements

These are the requirements that should be fulfilled by the software in prioritised order.

The system should be able to:

See how many bicycles are available at a given station
The users should be able to see how many bicycles are available at a given station.

Obtain data for statistical analysis
The system should be able to collect information and statistics about the usage of the system. The organization responsible should be able to use this information to, for example, perform analysis about where to redistribute bicycles and to determine if purchase of new bicycles is necessary.

Provide the option of booking a bicycle
The system should be able to book a bicycle and this booked bicycle should be locked and unavailable for everyone except the one who booked it.

Track bicycles
The data collected by the positioning system will be used to locate the bicycles if they are not returned to a station.

Predict availability of bicycles depending on placement and other variables
The position will be used for predicting the usage of the bicycles as well. For example to predict when a bicycle will be available at a given station again.

2.4.3 Hardware Requirements

In order to fulfil the requirements for the software listed above, a set of hardware requirements are necessary. These requirements specify a communication interface between software and hardware, and as long as the hardware implements these interfaces, the software solution should be able to perform the required tasks. The software requirements are listed in prioritised order, and depending on what requirements are implemented different hardware requirements need to be fulfilled. For example if the *Track bicycles* requirement is implemented, then the hardware on the bicycles should be able to transmit its position.

Interface between station and server
It should be possible to communicate the amount of bicycles docked in the station, and a possibility to lock and unlock docked bicycles.

Interface between bicycle and server
It should be possible to retrieve the position of each bicycle.

2.5 Target Audience

The target audience for this project are the regular users of Aalborg Bycykel, such as tourists and other people in Aalborg that do not have other means of transportation immediately available to them. Thus, we strive to develop a system that meets their needs. In that category goes low-cost and easy access for tourists. In general we will strive to develop the system for users who

are not particularly tech savvy, because if the solution works for them, it probably will for most other users as well.

Additionally, the organizations running Aalborg Bycykel are also part of the target audience, but that is in the context of managing the system, and not for the general use of the system, so this group is a secondary priority.

3 Suggested Solutions

Various suggested extensions for the current system are proposed. These solutions are then in turn examined and evaluated, in order to determine if a solution is preferred to others and gives valuable information for the software system. An exception to this is the Booking Software examination, which makes a concrete choice on what solution choose.

3.1 Availability of Bicycles

One of the requirements is that the users of the system need to be able to see how many bicycles are available at a given station. This section covers the suggested solutions for that requirement. We look at four different availability checking solutions:

- Camera based
- Dock based
- Chip based

3.1.1 Camera

A solution with little technological implementation required, is to use a camera such that users of the system can visually check if any bicycles are available at a given station.

This solution is good because it is an easy way to add some kind of availability checking without investing in a larger system.

Of course this solution has its downsides, because the information is not reliable in certain situations. One example of a situation where a camera would be insufficient would be if it was dark at the station. You also have to consider that the information transmitted by the camera can be deceiving, in that it could show a bicycle that does not really belong at the station and that would make the user come to the conclusion that there are bicycles at the station. The data is not concise in that it does not provide a standardised way of looking up the information, but rather leaves it up to the interpretation of the user.

3.1.2 Dock

Another solution that requires a bit more technological implementation is a *Dock* based implementation. The main idea is that when a bicycle is delivered at a station it is placed in a dock. A given dock, with some solution that could be a scale or some other type of detector, is then able to register if there is a bicycle located at it. Each station then contains several docks, and can read from each of them to provide a number for how many bicycles are located at that station.

This solution, unlike the *Camera* solution, provides exact information about how many bicycles are available at a given station with nothing up to interpretation for the user.

For this system to work, the bicycles would have to be placed into the docks to provide any correct information. That is, the system is not registering bicycles misplaced outside the docks. A potential problem could be if people park their private bicycles in the docks, as this could result in false positives.

3.1.3 Chip

A similar solution is to use chips, such as RFID chips. Such a solution is similar in that it provides the same information, namely the amount of bicycles at a given station. The solution differs, however, in the way this is registered.

One way to do this is be to take inspiration from a car park. The station would have one or more entrance gateways and one or more exit gateways that are able to read the chip. When entering the station with a bicycle, the total amount of bicycles will then get incremented, and when leaving with a bicycle it gets decremented.

Another way, depending on the distance these chips could be read from, is the possibility to deposit the bicycles at the station and the station would continually count the amount of bicycles it detects in the near-distance.

These two solutions have in common that they are easy to use, as you come and pick a bicycle and then return it when you are finished. On the downside it requires modifying all the bicycles in the fleet, furthermore, if using the car park idea, the stations can end up taking more space than the other solutions.

3.1.4 Discussion of Solutions

Among the above mentioned solutions to the problem of determining the availability of a bicycle, a preferred solution has to be chosen.

The impact of the solutions is measured on the amount of hardware needed to be added to the current system in order to get a working solution. For example, common to all the solutions is that they require a network connection available at every station because communicating with the system is necessary. Furthermore, each solution require different hardware and installations in order to provide the information needed.

Among the solutions with least impact are *Camera* and *Chip*. *Camera*, because only an addition of a camera at every station is required. *Chip*, because only an addition of a non-power-consuming RFID chip would be required.

The *Dock* solution requires an installation of multiple docks at stations, thus not being particularly low impact in terms of required hardware but it does come with some benefits. Specifically that it is easy for the user to use, and can be more easily extended than other solutions with more functionality such as forced locking.

The *Camera* solution is not considered further because of the possibly inconsistent interpretation by the user, possibly leading to situations where a bicycle is interpreted by a user watching the camera feed to be available, but in fact is not.

The *Chip* solution might be suitable if you only consider the availability of bicycles, however, since there is a requirement for booking of bicycles this solution is not considered further either.

The *Dock* solution provides the system with a simple way of determining the number of bicycles docked at each station. What this solution does not provide is a way of knowing the total number of bicycles currently in the system. Although a summation of the number of docked bicycles at every station would seem to provide this number, there is an uncertainty that bicycles are stolen or currently in use and not docked in any station. The *Dock* appears to be the best solution, though it does require some hardware installation.

3.1.5 Information Gain

Various solutions have been discussed, and from a software point of view, we need to know what information each solution can provide. The *Camera* solution gives access to a camera feed, which is not sufficient, based on the requirement which states you need to provide the amount of bicycles at a station. The *Dock* and *Chip* solutions gives information regarding how many bicycles are docked at each station, these solutions report this each time the amount of bicycles docked changes. The solution can, however, be modified to give the same information as the *Dock* and *Chip* solution, by reporting each time the amount of bicycles in the area changes.

While the *Dock* solution requires installation of the docks at the stations, it does not as such require modification of the bicycles which the *Chip* solution requires. The *Dock* solution is also very user friendly and easy to use, which is why we think it is the one that fulfils the criteria best.

3.2 Lock

There should be a way for the user to unlock a booked bicycle when reaching the station. This can be solved in several ways, some of these are described hereafter. We will look at the following solutions:

- Unlock on Time
- SMS
- Password Based
- QR Code
- GPS

3.2.1 Unlock on Time

This option is independent on whether or not the user is at the station at a specific time. When the specified time frame begins the booked bicycle unlocks and becomes available for anyone to take. This leads to problems in that a user arriving too early at the station will have to wait for the bicycle to unlock, and if the user arrives too late the bicycle might have been taken by another person. However, this solution is cheap since it only needs the lock on the bicycle, and the communication with the global system, which the other suggestions need as well.

3.2.2 SMS

Whenever the user is ready to get his booked bicycle, he sends an SMS to the system, and the system unlocks his bicycle. Compared to the previous suggestion, this suggestion does not make the bicycle available to everyone, therefore if the user is too late, he can still be sure that the bicycle is available at the station. Furthermore, if the user comes too early to the station, he is still able to unlock the bicycle therefore he does not have to wait for the bicycle to get unlocked. Moreover this solution is also cheap since it only needs to have a lock, some kind of receiver for the incoming SMS, and be able to communicate with the global system. However, this suggestion is not a free solution to the user since they have to pay for the SMS, which can become expensive if the user is not from Denmark.

3.2.3 Password Based

When the user arrives at the station he has to enter a password at the station to unlock a bicycle. To be able to do this it will require additional hardware at the station to make it possible to input a password to the system. Therefore this solution will increase the price of each station, as the additional hardware have to be at every station. However, it solves the problem that it can become expensive for the user, as this suggestion only costs something for the organizations behind Aalborg Bycykel.

3.2.4 QR Code

In order to unlock a booked bicycle the user has to scan a QR code located next to one of the locked bicycles at the station. This suggestion compared to the previous suggestions does not require any additional hardware and can therefore become cheaper for Aalborg Bycykel. However, it does require that the user has a mobile device that is able to scan a QR code and send the information over the internet to the system. This could, however, be as expensive for a foreign user as the *SMS* solution.

3.2.5 GPS

This suggestion uses the GPS of a device to see if the user is close to the station. If the user is close to the station the booked bicycle will get unlocked. To use this suggestion the user is required to have a mobile device which is able to send GPS information to the system. This would require that the user has an internet connection as well. Additionally this is very resource demanding for the users mobile device as the use of GPS prevent sleep mode of the mobile device [13], which results in significantly lower battery duration. Moreover, if the user passes by the station before he actually wants to use the bicycle there is a risk that the station will detect this, therefore unlocking the bicycle.

3.2.6 Preferred Solution

The chosen solution should be simple for the user, without requiring specific tools, while still being usable, as specified in Section 2.4.1.

The *Unlock on Time* solution is simple as it does not require anything from the user, it does, however, have the drawback that the user has to be at the station in time when the bicycle unlocks. It does not leave much flexibility for the user to arrive a little early or a few minutes late.

The *SMS* solution requires that the user has a phone capable of sending an SMS. It is expected that nearly everyone has a phone capable of sending SMS, but for tourists it might be a problem since not everyone is able to send an SMS to a foreign number or that it might be expensive. According to Danmarks Statistik 89% of people aged 16-74 are able to send and receive an SMS in the year 2013 [14]. Since the bicycles are also minded towards tourists, this could prove to be a problem.

To be able to use a QR code, the user would need some kind of scanner, for example a smartphone. 60% of the people aged 16-74 in the year 2013 was using internet on their phone, according to Danmarks Statistik [14]. With smartphones in mind, it is important that phones with an internet connection also usually has some kind of GPS capability. Because of the relatively low adoption of smartphones it may not be the proper solution.

The *Password Based* solution, however, appears to require more hardware installations at the stations, though it does solve the problems of the other solutions in that everyone can use it and the user is in control of when they get the bicycle.

3.2.7 Information Gain

The different solutions provide sufficient information to the system, to be able to register when a bicycle needs to be unlocked. Common to all but one solution is that it can be mapped to a password sent to the system, in order to unlock a bicycle. The exception being the *Unlock on Time* solution, where instead once the time runs out the lock unlocks.

3.3 Booking Software

In order to book a bicycle the user needs to interact with an interface. This interface could be in the form of a website or an application, each with their pros and cons.

3.3.1 Website

The users could make their booking through a website where the users could be asked to create a profile. The advantage of requiring a user to make a booking could be that the user, as well as Aalborg Kommune, would be able to see statistics about their bicycle usages. It does, however, have the disadvantage that it is slightly complicated to make the booking, since the user is required to login, although with modern browsers that capable of usernames and passwords, it does not necessarily have to be very complicated. If the user is not required to create a profile, then in combination with the ideas about unlocking the bicycles, discussed in Section 3.2, the user would still need to enter some information that can be used to identify him and his booking. It could be argued that in the long term it would be simpler to create a profile and login, rather than entering the same uniquely identifiable information for each booking.

3.3.2 Application

The application could be in the form of a mobile application. This would simplify the booking process for the users on the move, because the application could be optimised for easy access using touch screen gestures. The mobile application would exclude people without a smartphone unless both a mobile application and a website is developed. However, the mobile application could be in the form of a website optimised for phones, and thus allow people to easily use their phone, as well as their desktop computer for booking.

3.3.3 Chosen Solution

For this project it was decided to focus on only one application. We decided to focus on a website, since it can be optimised for use on mobiles phones, as well as for desktop computers. If the solution had been a dedicated mobile application it would limit the possible users to be people with smartphones, and possibly primarily Danish users because of internet roaming prices.

We chose that the better solution is for the user to create a profile in order to book bicycles. This allows Aalborg Kommune to keep better track of who was booking the bicycles and the user would not need to enter their user information each time they want to make a booking. The users who do not book a bicycle should be able to just borrow one of the bicycles that is

not already booked without creating a user. They would therefore not be needed to use the website, but should still be able to see how many free bicycles there are at a given station. This was decided because we believe it would be too inconvenient to create a profile if you quickly want to borrow an available bicycle.

3.4 Tracking of Bicycles

In order to counter the loss of bicycles, tracking of the bicycles could be used. With tracking, lost bicycles can be located and reused in the system. Furthermore, tracking can be used for analysis of the bicycling patterns, to see where most bicycles travel to, indicating new hotspots for potential placement of new stations. Additionally, tracking can be used to give an estimate of when a bicycle may arrive at a given station, opening up the possibility of providing waiting users with information about when a new bicycle might be ready for use.

The tracking system analysed is GPS.

3.4.1 GPS

GPS satellites are in orbit around the Earth. GPS is for example used for navigation purposes because it provides a reasonably accurate position of objects. For our purposes, however, it could be used to determine the position of a bicycle, though it would require some kind of GPS device being attached to the bicycle. This could then be used to find lost bicycles. GPS usually only works when outside, but as you are outside when you bicycle this is not a problem[15].

For each bicycle it is sufficient to have a GPS device used to determine the position and a power source for it. Then to report the position, some connection to the developed system would have to exist.

3.5 Chosen Solution

To provide the project with valuable information about how the system could work in the real world, we will also choose a solution for the hardware elements. This is despite the fact that only the chosen software solution, that is the website, the API, and some simulated hardware elements is implemented.

For the availability of bicycles, we choose to have docks since it also covers part of the hardware aspect of a different requirement, being able to lock. For locking we also choose to use passwords that need to be input at the station because it is something everyone can do once the booking has been done, as SMS, QR codes, and GPS require phones and might not be suitable for tourists. If it comes to tracking of bicycles we choose GPS.

The following provides an overview of the chosen solution using rich pictures.

The server-station relationship is shown in Figure 3.1. It depicts a central server, where the different bicycle stations are connected to. Furthermore, people can access the server to gain information e.g. bicycle count, but also to register bookings to the server, which will then communicate this to the given station(s) The stations also have the capability of handling booking by themselves, which is then shared with the central server. The server contains collected information from each station, such as bookings, amount of available bicycles at stations, and usage statistics. It provides booking and amount of available bicycles at stations through a website to the user. Usage statistics are provided to the facilitators of the system and gives them the ability to improve it.

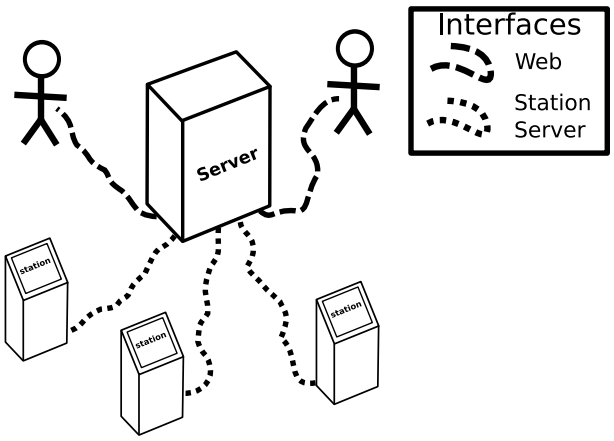


Figure 3.1: The server and the associated stations.

The dock that provide the locking/unlocking mechanism along with the bicycle detection ability have to talk to the station. This is for example when they need to unlock booked bicycle, the information needs to be propagated from the server to the station to the individual bicycle. This can be seen in Figure 3.2. The figure depicts a station and two docks connected to the station, which contains a bicycle each. On the display of the station, it can be seen that two bicycles are available for use. This is a sketch of how a station could look like when deployed.

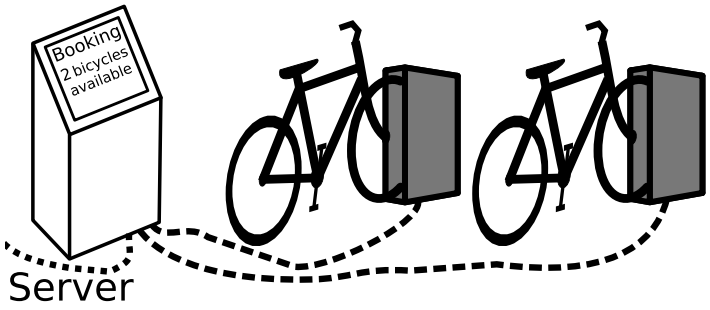


Figure 3.2: The station and the bikes.

4 Technologies

Some background material is needed for the development of a solution and this is examined and discussed. This includes describing the Internet of Things and what issues follow that, a description of the Model-View-Controller pattern used for the website, and AJAX(aka dynamic updating of websites).

4.1 The Internet of Things

This section introduces the Internet of Things (IoT), along with examples of how it is used. The section also introduces various hardware considerations to provide the context in which real-world applications for the IoT are developed.

The IoT is the concept describing the interconnection of uniquely identifiable things in the problem domain. The IoT also includes the virtual representation, the interfaces that allow for manipulation, and information retrieval regarding things [16–18].

Real usage of the IoT manifests as systems to provide some service, either by informing the user about things or allowing an intelligent system to manage those things. An example of this could be a 'smart home' that allows you to use a single device to manage the connected things in your home, such as the lights or the oven in the kitchen [19]. One society-wide use for it is the idea of a 'smart power grid', where the electricity usage is monitored and managed by an intelligent system that will e.g. redirect electricity if a cable has been cut somewhere in the system [20].

There are already a lot of things in the IoT, and by 2020 it is estimated that there will be upwards of around 26-30 billion things [21, 22]. This will likely require a shift to the IPv6 protocol as the amount of IP addresses are severely limited by IPv4 [23] given that the additions to IPv4, such as multiple devices sharing a single IP address, will at some point become insufficient.

One important aspect of the things connected to the IoT will be what technology they use to connect, here WiFi or mobile networking are obvious choices of communication means. The connectivity technology has to be low-power and cheap. Other than WiFi and mobile networking, it is possible to connect things to a server with a cable connecting to the internet, though that is not practical for things that must be mobile. Radio Frequency Identification (RFID) chips can provide relevant information to an outside observer about the thing itself [24] with active research in making it low-cost and low-power [25].

The geographical location in the IoT matters, especially for sensors where the location provides important context for the accessed information [26]. For example if there is a station for bicycles in a bicycle sharing system that needs to provide information regarding the amount of bicycles at the station, it is important for the usage of the information that it also provides the actual location of the station, if that is not otherwise known.

In order to give more detail on IoT, aspects about identification, communication, and software is given.

4.1.1 Identification

In order to uniquely identify the things in the IoT, different approaches can be taken, we give an example here.

One idea is to use the IP address of an object, which has relation to the previously discussed IPv4 versus IPv6 issue. With IPv6, this approach would have enough addresses to uniquely identify a large number of things. Given IPv6 has a theoretical possibility of $3.4 \cdot 10^{38}$ unique addresses [27], not having enough addresses would not be a problem in the foreseeable future.

When you have a unique address, you can use that to uniquely identify the given thing. An example of use is the power grid, where each power station can uniquely be identified with the IPv6 address, and as such, in case of malfunction in one of the grid connections, it would be possible to identify the stations lacking power.

4.1.2 Communication

In order for IoT to work, it is necessary to have a communication established. If that was not the case, the things would not be part of the IoT, as the central idea is that you can communicate over the internet.

One idea of communication for sensors is as follows. Each sensor has access to the internet to contact a web service of their given reading. However, it is unrealistic that each sensor alone is directly connected to the internet, and as of such, other alternatives can be performed. One such alternative is that a given thing consists of a communication device connected to several sensors and the internet. The communication device can then read from the sensors and contact the webservice.

However, the communication does not end here. The idea is that the communication is not limited to machine-machine communication, but is expanded to communication over the internet, such that the things of the IoT can be contacted from anywhere on the internet.

4.1.3 Software

Examples of Software that utilise the IoT are given to give a concrete idea of the power of IoT.

We here expand on the example of the smart power grid. Such a power grid uses the information it has about each section of the grid, to ensure that there is sufficient power reaching every part of the grid, even in cases where part of the grid malfunctions. This ability is achieved through the system being able to automatically reroute the power flow in the grid, so power flows through functioning areas, to reach the parts that would otherwise have been affected by the malfunction.

Another example is the mentioned home automation system. For such a system, the things of the house being the lock, coffee machine, lights, washing machine etc, is then the central parts. If the things of the house is connected to the IoT, it is possible to connect to those devices and control them over the internet. This has several advantages, which includes ensuring the door is locked, preparing coffee before you get home, and other actions you may want to do with the things of your house, even though you are at work or on vacation.

4.2 Model View Controller

The Model View Controller (MVC) is a design pattern widely used in programming to separate the program into different layers called 'Model', 'View', and 'Controller'. This provides the advantage that the code gains clarity and becomes easier to maintain as it separates responsibility to individual layers.

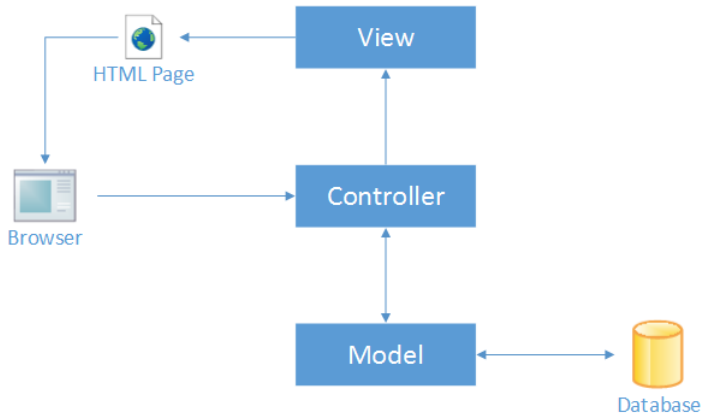


Figure 4.1: Illustration of MVC pattern.

An illustration of MVC can be seen in Figure 4.1.

If you look at the illustration, the 'Database' is in our case the relational MySQL database, where data is stored in tables.

The 'Model' in our case provides abstraction over the information in the database, it consists of several entities and a service for each such entity. This is to make the entities simply wrap data, while the services work on the data, and each implement an interface, requiring the services to implement Create, Read, Update, and Delete (CRUD) methods. The Service CRUD layer can also contain various other helper functionality, along the same lines as CRUD, e.g readAll. This is a departure from traditional MVC where behaviour for models is built into the model themselves.

The 'Controller' provides a set of 'actions' that the client points at. These actions then use CRUD methods in the 'Model' layer and uses views to show an representation of the model, e.g. a table or a map of stations. These views are then presented to the user through the browser. A controller may include multiple views to get a html-page generated for the client to view. As multiple views may be included by the controller, it may use separate views for the header, the body, and finally one for the footer. These multiple views allow for reuse of views.

It is important to note that there is a single model layer, consisting of multiple entities (e.g bicycle and booking) and their associated services. Additionally, it is a good practice to have multiple controllers, which each handle different parts of the website. In our case we for example have a home and user controller. With the user controller taking care of everything connected to user login, editing of profile, logout and so on, and some actions could be moved to a separate controller if it grows too large. The home controller takes care of the presentation of the front-page and the functionality of booking and unbooking of bicycles. As is evident, the controllers are split into different actions present on the website, which ensures better code clarity.

In addition to the better code clarity, as the website is organised in the way it is, working with the same model, the layout of the website can easily be changed, by including other views or developing additional controllers for other work routines. This is relates to the high modularity you gain with the MVC pattern, leading to high cohesion (elements in a module belong together to a high degree) and low coupling (low interdependence between modules).

However some things need to be loaded client-side for increased usability, which is where AJAX comes in.

4.3 Asynchronous JavaScript and XML

Asynchronous JavaScript and XML (AJAX) is a way to update parts of a website without user interaction. By using AJAX, parts of the websites can be updated without reloading the whole website every time, which makes the user experience of a website smoother. Even though XML is a part of AJAX, it is not necessary to use when using AJAX, other methods to send the data can be used, an example could be to use JSON. When using JSON, data is encoded and then displayed on a page by itself, which the javascript then decodes and can use to reload the affected parts.

The reason to use AJAX is to improve the usability of a website, as well as the performance of a page. An example of this is when rating a film on www.imdb.com [28] then when a user rates a movie, the website does not reload, but instead calls for an asynchronous update to the database giving the user an non-interrupted experience. AJAX should be used whenever the user interacts with something that does not need to update the entire page, but only update information on the database, this could be changing the password of a user or it should be used when updating only a part of the website, so that the website should not be refreshed before updating this part of the website.

4.4 Simple Object Access Protocol (SOAP)

The **S**imple **O**bject **A**ccess **P**rotocol (SOAP) is a protocol for sending structured messages over a computer network using XML.[29] The use of a SOAP helps structuring message passing, providing type information for contents.

SOAP is useful because at the same time as allowing transmission of information, it also allows implementation of functionality for what is to be done once that information has been received.

A reason to use SOAP over other methods for transmitting information is because it provides more possibilities for security and is generally considered more reliable, and because it provides a standard use that makes it easier for new developers to pick it up. SOAP is generally considered slower and much more verbose than other methods. Other methods on the other hand are generally more versatile when it comes to complex types, and is generally simpler, much less verbose and faster.

We chose to use SOAP because while other methods are less verbose, faster, and simpler, SOAP provides a standard protocol that while complex and verbose is a very big benefit.[30, 31]

5 Design

This chapter will present the design of the product, where a prototype is created hereafter the ER-diagram of the database, also going into what they administration tools need to be able to do, the architecture is explained, and how synchronisation between the central server and the client is going to be handled.

5.1 Website Prototype

In order to get a better idea of how the website should look like, various prototypes were drawn. As they are prototypes, they are by no means a representation of the final product, but more of a source of inspiration and brainstorming, to consider when developing the website. Throughout this section, prototype sketches for different parts of the website, is presented, explained, and discussed. Our primary inspiration is based on the current website for Aalborg Bycykel[7].

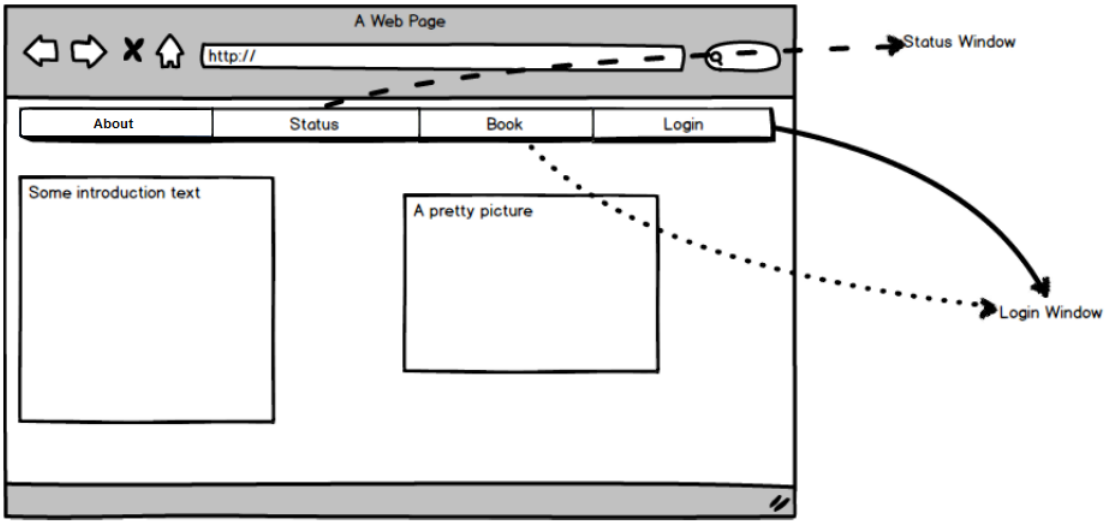


Figure 5.1: About page.

First, we take a look at Figure 5.1. This illustration presents a draft of the about page, it is structured as a standard about page with some description of the organisation behind the website. What is more interesting to see in this illustration is the first draft of the menu bar. The menu bar consists of links to about, status, book, and login, each of which have their page(s) illustrated hereafter.

Next is a presentation of the status page, see Figure 5.2. The main idea here is that the status page should be easily accessible and is what should be shown when navigating to the homepage. The reason for this is that the status page should easily be able to give you an overview of the status of each station. As can be seen from the illustration, this station detail can be found by selecting a station from a table, searching for it, or selecting the station on a map. When a station is selected, details will be shown for it, which can be useful to see if available bicycles

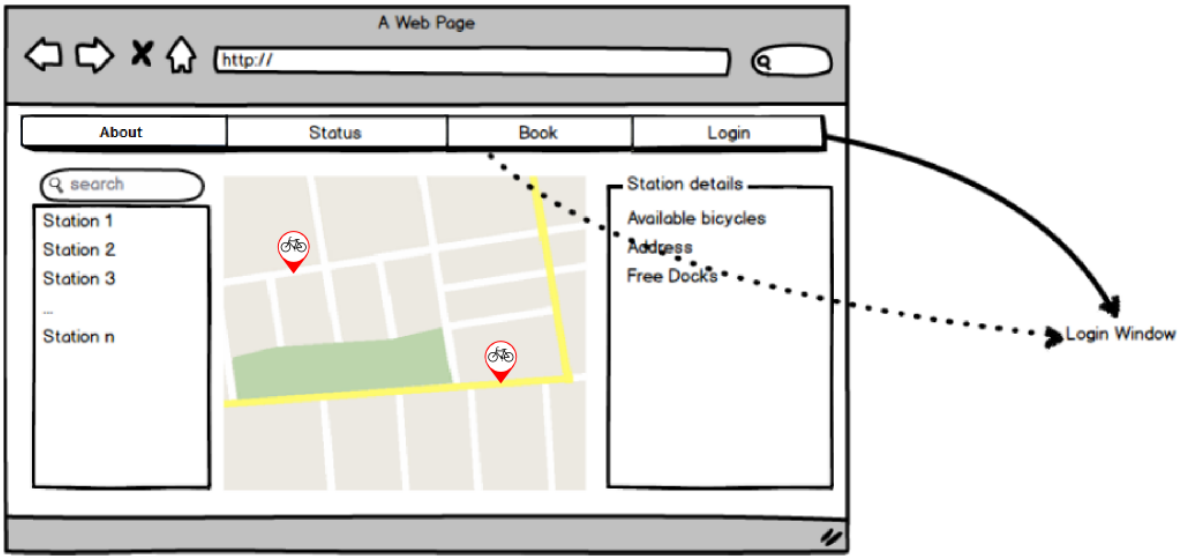


Figure 5.2: Status page.

exist at the station. In many ways, the status page is similar to the booking page, which will be presented hereafter.

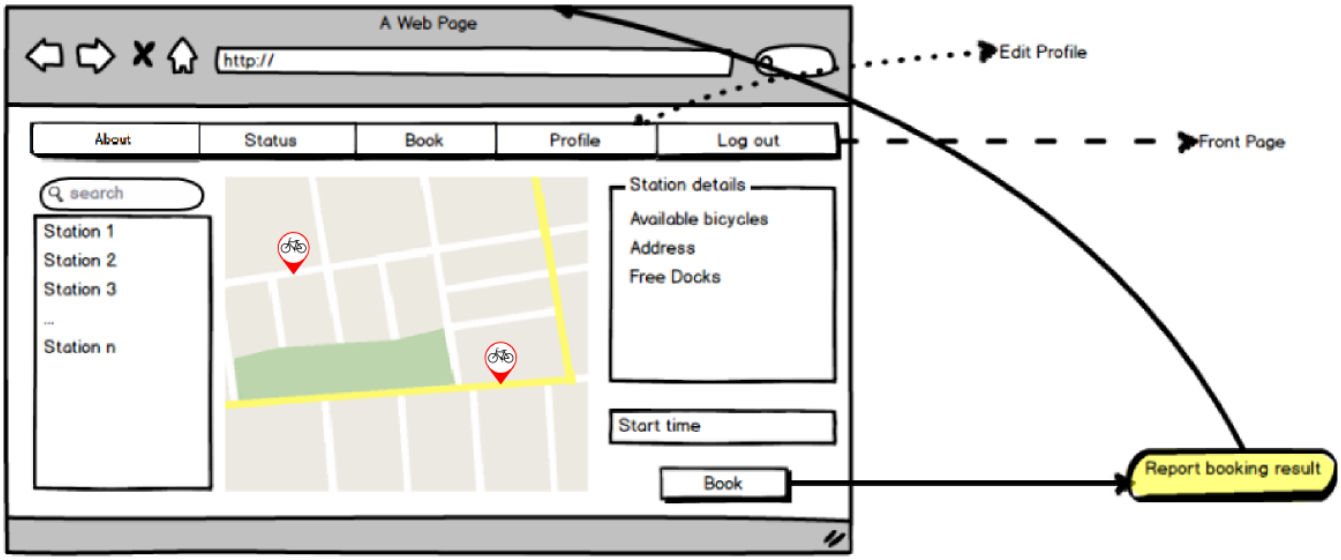


Figure 5.3: Booking page.

For an illustration of the booking page, see Figure 5.3. The booking page can only be navigated to if you are logged in, if not you are redirected to a login page. The booking page is otherwise the same as the status page, except you are able to book a bicycle at a given time. To make a booking, you would login to the website and navigate to the booking page. Thereafter you would select a station where you want to book a bicycle and select start time, which is the time

you expect to retrieve the bicycle. When you then click *Book*, the booking result will be handled and registered in the system if valid.

Prototype pages also exist for the profile login and management, but has been omitted here, as these pages are very standard pages. In that meaning one page being a login form and another page being a form with some fields to edit your password, phone, and email. What is interesting though, is that you can access your booking history page through the profile page.

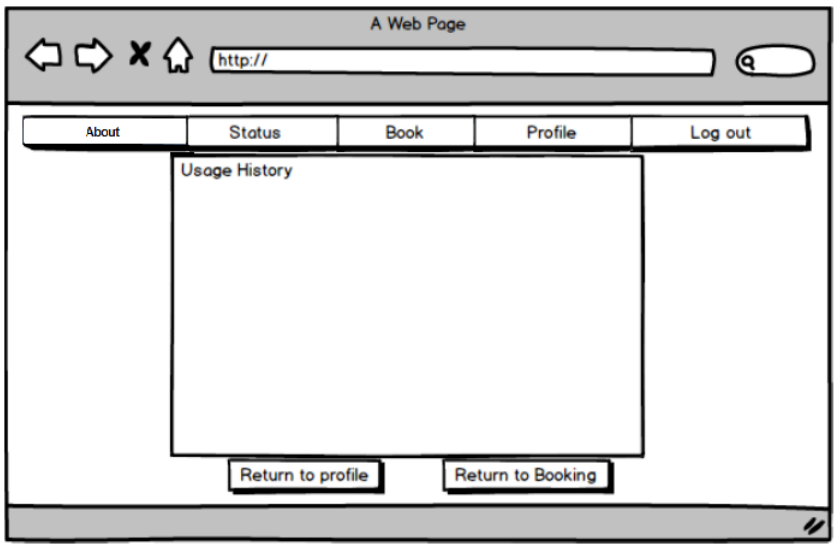


Figure 5.4: Booking history page.

The booking history page can be seen in Figure 5.4 and will show your whole booking history. The booking history is thought useful because it might be reassuring to show the user that their bookings have been handled correctly by the syste and to generally support users that think that their actions have not been properly registered. Furthermore, sometimes you see webshops provide purchase histories, and you can for this page consider a booking a metaphor for a purchase.

This ends the description of the prototypes for the system, and is what we first had in mind before implementing the system, and as of such you will later see some parts of this used for the implemented system.

5.2 Entity–Relationship Diagram

In order to have an idea of how the database handles bookings, location of bicycles, and their relation to the location of the stations, an ER Diagram was made, see Figure 5.5.

As seen in the diagram, the entities consists of bicycle, dock, station, booking, and account. An account consists of a username, email, which both must be unique, a phone number, password and a role which can be either a regular user or an administrator. These attributes are common for an account entity, with the phone number being special in that the idea is that they should, in the future, be able to receive the booking password over SMS.

In relation to this is the booking entity, where it can be seen that a user can have many bookings whereas a booking is registered to one and only one account. The booking then has a booking_-

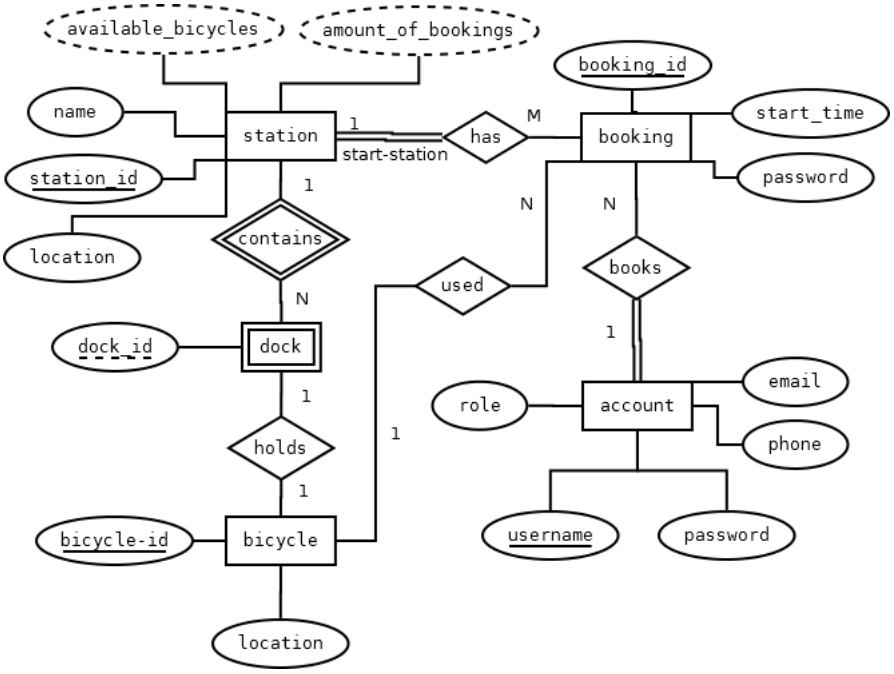


Figure 5.5: The ER Diagram for database overview.

id, to uniquely identify a booking, a start_time, such that a station knows when to lock a bicycle, and a password, used to unlock a bicycle at the given start_time if a correct password is entered. The idea is that if a booking is not used in a given time frame at the start_time, the booking will be removed, in order to free the bicycle for other to use.

A booking is also tied to a station, such that a booking is located at one and only one station, whereas a station can have many bookings.

A booking is also tied to a bicycle, such that when the booking is fulfilled it will identify which bicycle was taken.

A station has the attribute station_id, to uniquely identify the station. Furthermore, it has a name, which is thought to be used to give a meaningful description of a given station. We are aware that the name could be used as a primary key, but having an integer id as the primary key reduce storage requirements when using foriegn keys to the station id's.

In addition to this, a station has a location, which is used to easily place a station on a map, and can also be used for calculations such as distance between a given station and bicycle. A station also has two derived attributes, which are available_bicycles and amount_of_bookings, these are attributes that is beneficial to show to the user, such that they can see if it is possible to gather a bicycle at a given station.

Next is the dock entity, which is a weak entity, as it cannot exist without a station. This represents the real-life situation with stations located around the city, and each of these has a number of docks. The dock only has one attribute, which is the dock_id, used to identify a dock in combination with a station-id.

The interesting part of a dock is its relation with a bicycle. A dock may or may not hold a

bicycle, which is represented with the holds relation.

The bicycle entity then consists of a bicycle_id, to uniquely identify a given bicycle, and a location, which can be used to locate lost bicycles.

5.3 Administration Tools

In order for the administrator of the bicycle system to make reasonable decision about future actions/improvements towards the bicycle system, statistical data has to be collected and shown to the administrator. This will give him an overview of the usage of the bicycles and prepare him to possibly make a better decision or just provide him with a knowledge of how the system is used. A list of questions should be able to be answered or hinted by the system, however, the implemented features in the final system will likely be limited by available time:

FiXme Warning: Patterns? Most popular?

Which routes are used?
If routes could be determined, patterns could possibly be seen, e.g. if there is a particular route that is very popular.

Where is the most traffic of bicycles during some period?
An administrator would be able to see how many bicycles leaves and arrives at each station and thereby get an overview of where the traffic is high and low, providing an indication of where to put focus for relocation of bicycles and expansion of stations.

What is the current amount of bicycles at a given station?
This gives the administrator an idea of when bicycles leave each station.

How does the amount of bicycles at a given station change over time?
The administrator can choose a long time interval, providing a more general overview of when the activity at each station is high or low, also providing him with an overview of usage at each station.

Are there hotspots for bicycles?
If positions of bicycles could be determined, it would mean that different kind of patterns could be detected, for example detecting stagnant bicycle hotspots could provide valuable knowledge about where to add new stations.

Keeping a log of information about the usage as described above, requires additional database tables having timestamps as an important factor since the usage of bicycles is tied to some real life events that needs to be logged. In order to log the routes for a bicycle, a new relation is added to the database schema storing information about longitude and latitude and of course which bicycle it is for and when it was logged. In order to log traffic of bicycles between stations another relation is added to the schema having information about a start station and an end station for a trip along with times for each of the events (start of trip and end of trip). Last, a relation to store the count of bicycles at each station along with a timestamp, is added to the schema.

Putting all this together, also illustrating foreign key references, can be seen in Figure 5.6.

Addition and removal of bicycles, users, and stations is also part of the responsibility of the administration tools. For bicycles, they should be added to the system loosely and not directly to a station. We intend to do it like this because when the administrator adds a new bicycle,

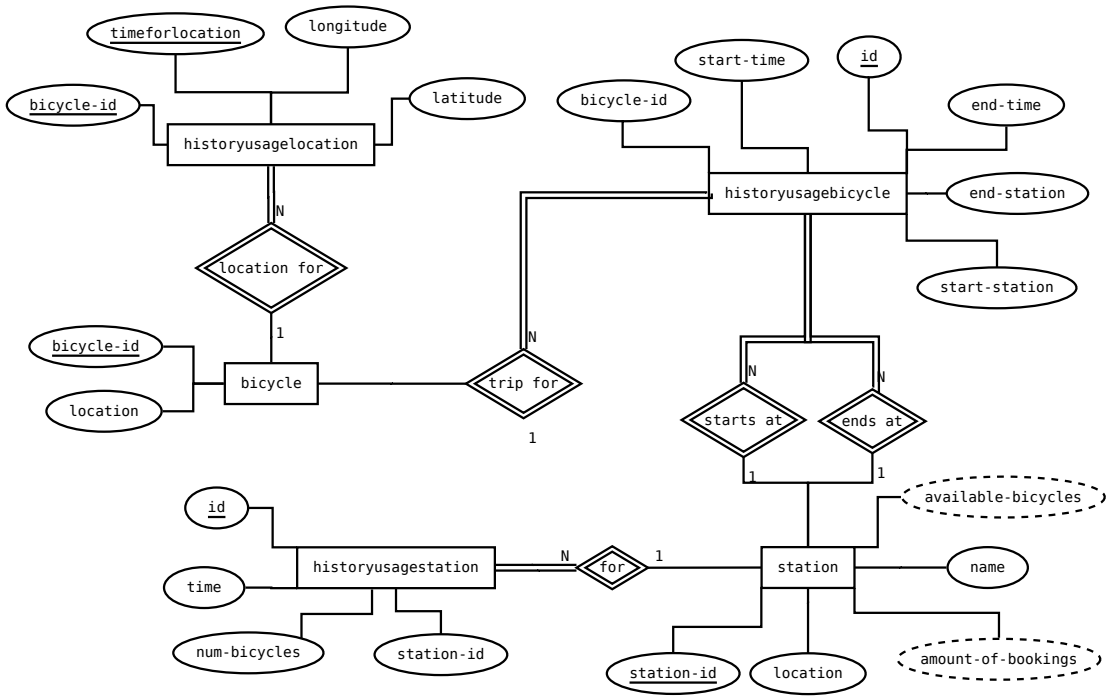


Figure 5.6: ER-diagram showing relations covering data logging.

they receive an id and the idea is that they use this id to program the bicycle such that it can be recognized by the system. For users and stations, addition does not have any such considerations to make. The dock is not part of the administration tools, since it is handled station side. For removal of stations, it is important that all bicycles at that station are released and that docks are deleted along with the station.

Since the logging tables use the other tables like the station and bicycle tables it is important to handle deletion of stations and bicycles. If deletion is not handled properly the logged data could end up being inconsistent. In our case we handle deletion by adding a column to the affected tables indicating if the row is deleted or not. When using the station and bicycles table on the website we make sure to only use the rows that is not deleted, while for the history data we use all rows.

An overview page of the stations should also be implemented, providing information about stations such as if they are online or not, where they are and what their status is with regards to usage.

5.4 Architecture

Before implementing the software system, it is a good idea to have a general overview of how the various parts of the system are connected. In order to gain this overview, an overall archi-

tecture diagram was drawn.

The overall architecture can be seen in Figure 5.7. In the illustration a single arrow is a one way communication, whereas two arrows are two way communication. The architecture is very similar to a client-server pattern, where the central database is the server, and the stations and bicycles are the clients. As we have a central booking system, and multiple simultaneous access points, the client-server pattern is applicable and appropriate.

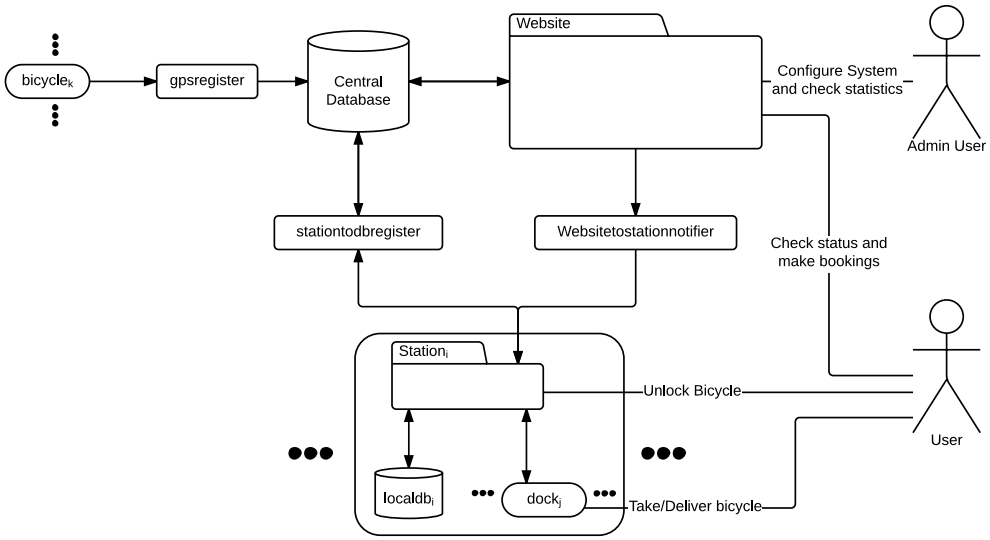


Figure 5.7: Overall architecture

As can be seen from the figure, the architecture consists of a central database as well as a website and three interfaces, named 'gpsregister', 'stationtodbregister', and 'websitetostationnotifier' , where 'gpsregister' and 'stationtodbregister' contact the central database. Additionally, multiple stations and their associated local database and docks exists. The reason each station needs to have its own local database, and not merely use the central one is to minimize the necessary network communication, as well as allowing the stations to remain operational in case of network failure. As of such, the local databases contains a subset of the central database, corresponding to the data involving the given station. Each dock is associated with a given station, and is used by the users to take or deliver bicycles.

Additionally, the three interfaces each serves a different purpose, as described here. The gpsregister interface, is provided for the bicycles to register their current position in the central database, as well as logging its path in another table. This data provided can be used for route mapping and positioning of bicycles.

The stationtodbregister interface serves the purpose of updating the central database of changes performed at the local stations. Furthermore, it enables the local stations to read their subset of the central database, which is used for booting of stations from scratch, when their local database is empty.

The websitetostationnotifier is a little different, in that it is an interface provided to the website, such that when the website changes data, e.g. create a booking, the involved station is notified by the interface.

The architecture for the website is the MVC pattern, see Section 4.2. There are other alternatives, such as Model View ViewModel or single pages doing everything, but we picked the MVC pattern because the idea behind it is very clear and easy to learn. A detailed overview of the architecture of the controller and model part of the website is provided in Appendix A.1 and Appendix A.2

Another part of the architecture is the user interaction. For the website, we distinguish between administrators and regular users. The administrator has access to an administrator webpage, where he can modify stations, bicycles, docks, and users. In addition to this, the administrator is also provided with some statistics pages about the usage of the system. The administrator also has access to the features a regular user has access to.

A regular user can interact with the system in several ways. The website allows for station status, in order to see how many bicycles and docks are available at any given station, but also the position of each station. In addition to this, the website also provides the option to book bicycles in advance for a given station, where the station locks a bicycle some time in advance. If you book a bicycle, you are provided with a password needed to unlock a bicycle at the station.

For the station interaction, two interactions are involved. If you have booked a bicycle at the station in advance, you use the password provided from the website to unlock a bicycle for usage, by entering this password in the station automata. The automata then tells you at which dock a bicycle is unlocked, and you can retrieve your bicycle. However, if you have not booked a bicycle in advance, you do not use the station automate, but instead check each dock for available bicycles. In that regard, we imagine a distinction between reserved unlocked bicycles and free bicycles, such that you do not on accident steal someone else's booked bicycle.

For the deliverance of bicycles, the interaction is the same whether the bicycle is booked or not. The process is easy, as you locate a free dock and then place the bicycle at that dock. The system will then automatically register that the bicycle has been returned to a dock at the station.

5.4.1 Station Software

While the figure of the overall architecture provides an overview of the system, the station component has its own architecture as well, which can be seen in Figure 5.8. In this figure, as in the previous, the arrows represent flow of information.

In the station software, the station is the main part and also the one that contains the UI. This part is responsible for all contact with the user at the station and is the one that is interacting with the hardware. *TCPListener* is a thread that listens for messages from the global system through the websitetostationnotifier and stores these messages as a *NetworkData* object, and tells this object to perform the action the message contains. The *NetworkData* object has the responsibility of adding a booking to the local database corresponding to the information the object contains and removing a booking with the ID it received. The *LockManager* is also a thread, and is responsible for locking a bicycle when a booking is close to its start time and unlocking said bicycle again if the booking's start time is exceeded by a certain limit. The last class is the *ServiceThreads* which is also running in a separate thread. This thread is responsible for reporting changes at the station, which are stored in the *ActionQueue* queue. As soon as there is something in the *ActionQueue* queue, the thread will try to report it to the online interface, repeating until it succeeds in transmitting it.

FiXme Warning: Hvad med stations? Det har vi ikke lavet, skal det laves?

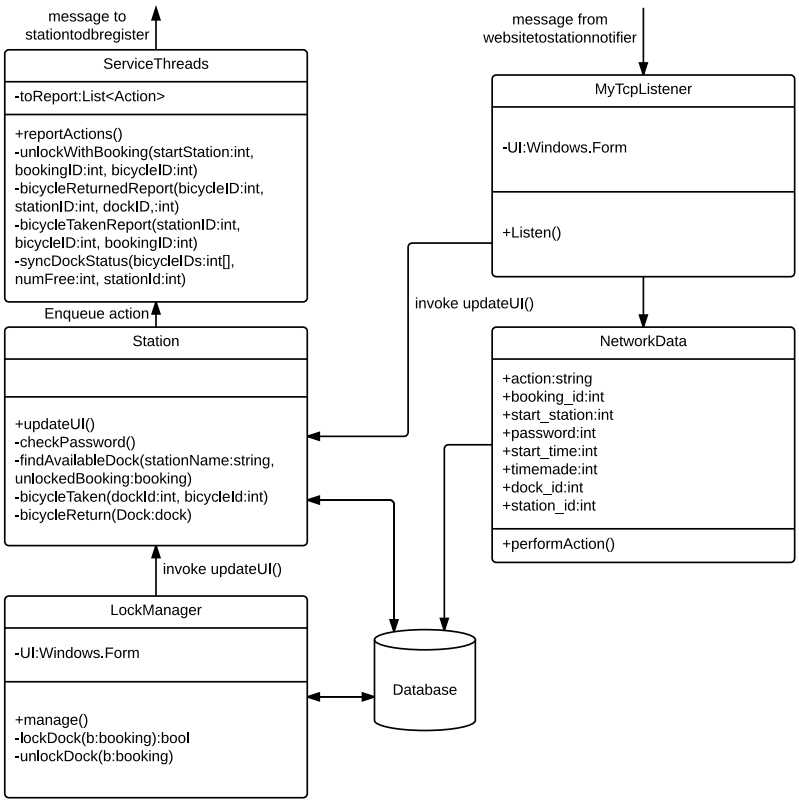


Figure 5.8: Station software architecture

Locking Considerations

Consider the scenario where a user u_1 makes a booking b_1 on the website at time t_1 wanting to reserve a bicycle for time t_3. Before time t_1 another user u_2 made a booking b_2 of a bicycle for time t_2. Both users made a booking at the same station, and the amount of available bicycles for that station was for both users 1. Under the assumption that a bicycle cannot be locked and thereby reserved before some amount of time, say t_{before}, before usage, this scenario is possible in the following way.

First consider this definition of a time t, it is a time represented as the number of seconds from Thursday, 1 January 1970 at 00:00:00 UTC, which is the Epoch time, as of such comparisons such as $>$ is just computed as usual on numbers. Let,

$$TSB(b) = \{(t_1, t_2) \mid t_1 = st(b) - t_{before} \wedge t_2 = st(b) \wedge t_1 \leq t_2\}$$

where,

$st(b)$ is defined as the start time of booking b.

$TSB(b)$ defines the Time Span Before of booking b.

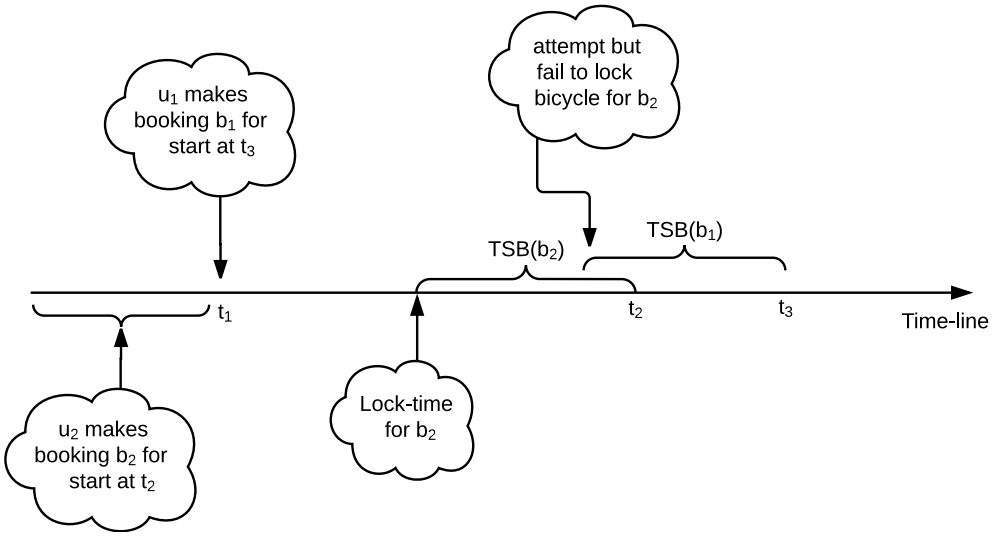


Figure 5.9: Booking scenario illustration.

Then the scenario is given if $TSB(b_2)$ starts before $TSB(b_1)$ and $TSB(b_2)$ starts after t_1, we have a situation as can be seen in Figure 5.9. As can be examined from the illustration, a problem occurs, under the assumption that no new bicycles gets delivered or taken from the given station in the given time-line, and where the amount of available bicycles before the lock-time of b_2 is 1.

The problem for user u_1 is then at time t_1 it appeared that a bicycle was available, but the bicycle will be locked for user u_2 (although not entirely certain, see Problem 1.2) before it will be locked for u_1 resulting in an availability count of 0 for u_1, thereby having no bicycle to lock for u_1.

The key aspect of why the handling of locking cannot be left as a responsibility of the database is to make the system less static, as requested by Aalborg Kommune, see Section 2.1.1. If we were to make this a responsibility of the database, it would be in the case where locking should be performed immediately, but if you book a bicycle for tomorrow that would mean a bicycle would be unavailable for other users for a whole day, which is highly undesired. This is a tradeoff between assureness of getting the bicycle booked and a dynamic system where more bicycles are in use.

5.5 Synchronisation

As the stations and the website are located on different addresses and communication is through TCP, you have to consider how to synchronise the various parts of the system, in order to avoid an inconsistent state.

Several types of such inconsistent information could happen if you are not careful.

An scenario that could lead to an inconsistent state, if not handled well, is what happens when a station fails to communicate its current state to the global system. A station maintains a local

database holding information about each current booking made at that station. When a booking has been processed, meaning that the user has taken the bicycle associated with the booking, the state of the local database changes in that the booking does not exist any more. This change in state is also communicated to the global database maintained by the global system. But when trying to communicate the change, the connection is interrupted due to a failure in the software at the station.

The problem is then, formulated with other words, which database will hold the correct state of the overall system? This depends highly on the direction of communication failure. In the described case above, the communication failure lies with the station software resulting in the local database of the station having the correct and current state of the system, which is when it comes to the state of the docks.

But if the direction of communication is the opposite, which would be involving bookings registered, the global system would have the correct booking information.

The desired behaviour of the system is to always be in a correct state, which requires that when a communication failure happens, the part participating in the communication holding the correct state will resend its information, or the one with the inconsistent reads the state from the consistent counterpart.

There are two cases for the communication between the station software and the global system (web-service).

Station to Global System
A communication failure in this direction will result in a faulty state of how many bicycles are available at the given station thereby resulting in a rather useless website for booking. Additionally, this can also involve the global system having the incorrect amount of docks registered, if such a change were performed at the station.

Global System to Station
A communication failure in this direction will result in a failing booking system, in that no booking made at the website will reach the station and thereby have no effect at all. In the end this will also give a faulty state of how many bicycles are available at the given station because the station will not know of the booking and thereby not perform the necessary actions to make it a booking.

6 Implementation

This chapter covers implementation details for the features covered in the design chapter.

6.1 User Interface

Based on the website prototypes in Section 5.1, a user interface is implemented. The website is implemented using HTML, CSS and JavaScript. Different JavaScript frameworks have been used for graphs, AJAX, animations and maps, including jQuery, AmCharts, D3, Google Maps Javascript API v3 and different plugins for jQuery. This user interface is then iterated upon based on the usability test, what follows is the final result based on these iterations. As can be seen from the prototype, the website has many pages, therefore only the essential ones are shown. The first we look at is the start page shown when an administrator is logged in, which can be seen in Figure 6.1.

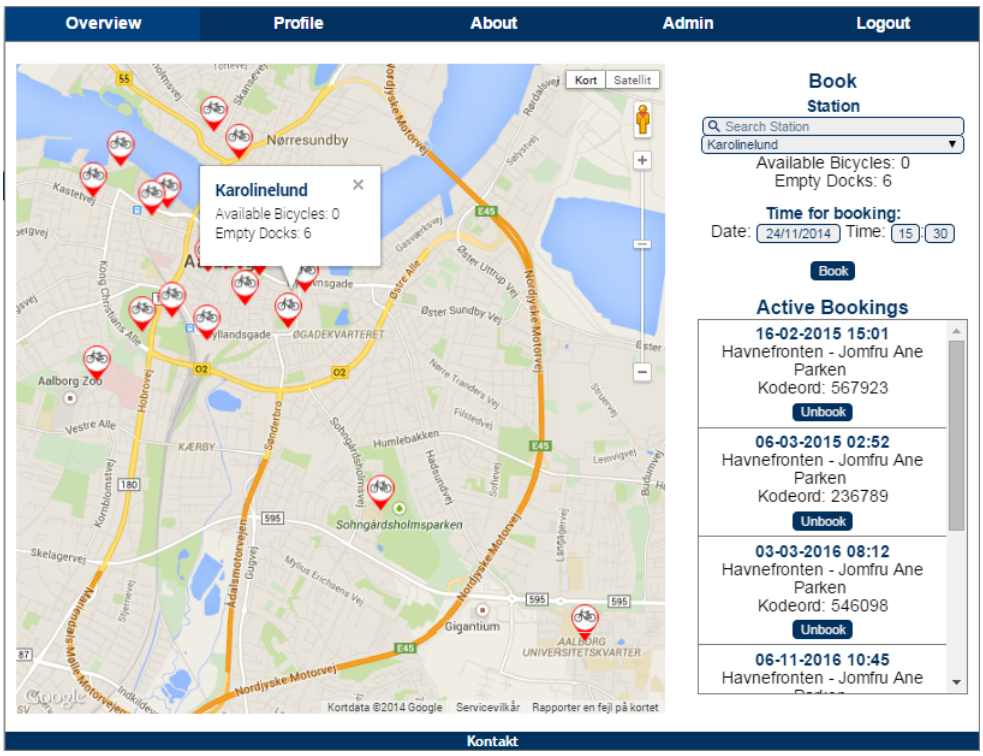


Figure 6.1: Overview page

In this figure we can see that something from the prototype has been changed, mainly the fact that the screen is now only split in two parts, where the map showing station locations covers two thirds, and the bookings/station details cover the last third. The map in the figure shows a map of Aalborg city with bicycle icons on it representing the station placements. The right side of the website is used for booking, where users can book a bicycle on a chosen station at

a specified time. The user can also see a list of bookings he/she currently has, and is able to cancel said bookings by clicking on the unbook button.

In Figure 6.1 the header contains five elements, of which the administrator element is only shown to administrators and directs the user to the administrator section of the site when clicked. The profile element in the header directs the user to their profile page, where they can change their profile information if so desired, as well as view the history of their usage of the system.

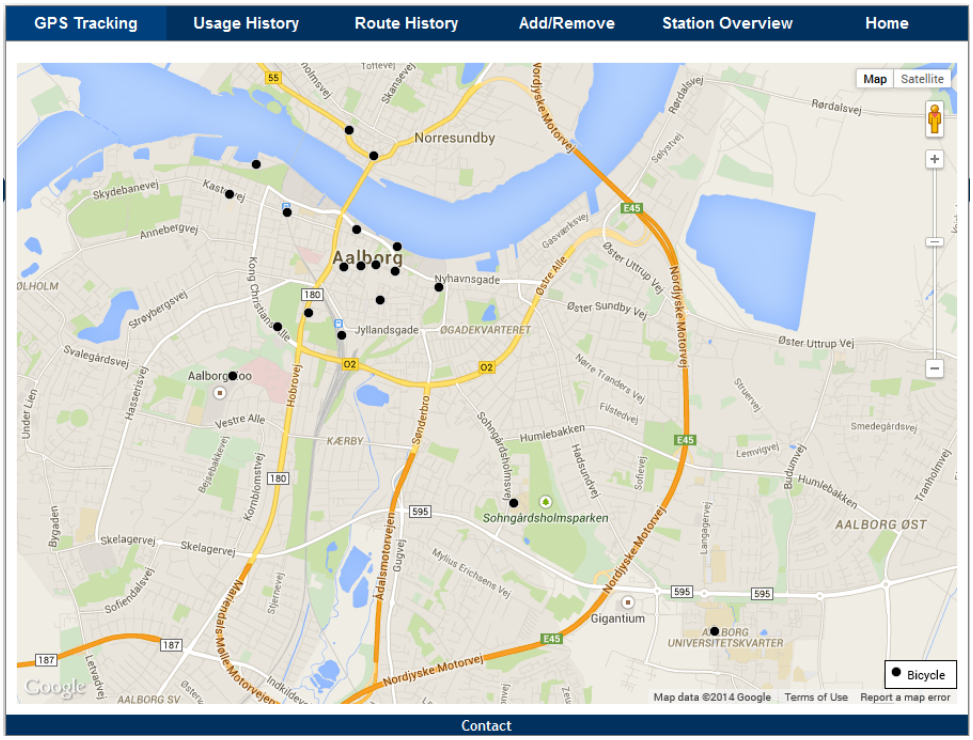


Figure 6.2: Administrator page

The administrator features are created for use by Aalborg Kommune, and they start on the page that can be seen in Figure 6.2. This page is almost just a map, however, the black dots on the map are used to represent the current position of bicycles. The header can be used to navigate to the various administrator tools, elaborated more in Section 6.9. The only header element that does not direct to another administrator site, is the home element, which sends the user to the overview page. Additionally, if an ordinary user tries to access an administrator page, via. the url, he will be redirected to the homepage.

For an overview of what the map of the site looks like, see Figure 6.3.

6.2 Website Structure

The website is structured via a modified MVC pattern, for an explanation of the general idea of MVC see Section 4.2. In order to make the usage of MVC smooth, a framework was used [32].

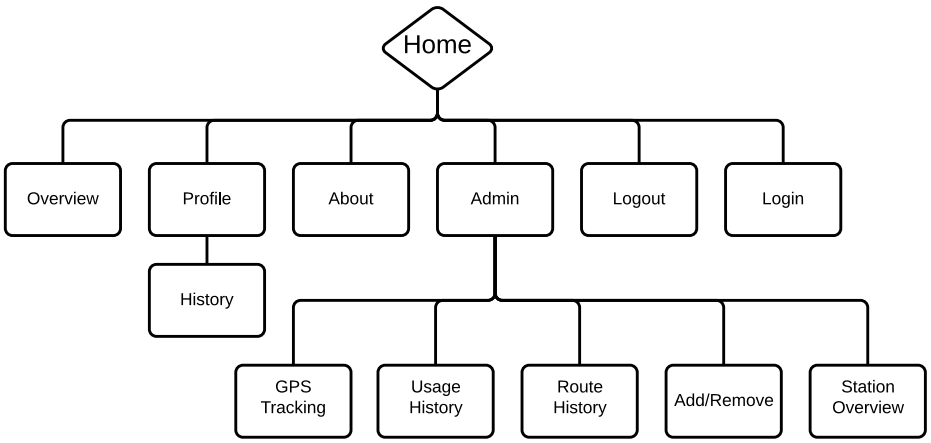


Figure 6.3: Sitemap

The framework provides a controller base class that the new controllers inherit from, mainly providing a database instance. It also provides an 'Application´ class that handles url parsing and navigation to the correct pages based on the url. In addition to this, it provides a directory structure. As an example, you can add a controller to the controller directory, and the 'Application´ class is able to load them, same idea goes for the model and view layer.

With inspiration from the MVC pattern and usage of the framework, the website was then structured as seen in Figure 6.4. The idea of the figure is to represent the structure, where we abstract from concrete entity, view, and controller names, and instead denote these with the names 'Entity$_j$´, 'EntityService$_j$´, 'ConcreteController$_i$´, 'View$_u$´. The structure is split into several parts, each of which is examined in turn.

Model Layer

If you take a look at the model layer, it is split into three parts, Entity$_j$, EntityService$_j$ and iService. Entity$_j$ is a wrapper class, which is used to encapsulate a row from a table in the database. EntityService$_j$ is then a service, implementing the CRUD methods of iService, read is exluded from the interface because the parameters needed for read varies. EntityService$_j$ is what part of the model that then establishes the connection to the database, and enables the controller to work on the model, via calling the CRUD methods of the service, when a read method is called from a controller, an entity or a list of entities representing the row(s) is then what is returned. The reason we differentiate between Entity$_j$ and EntityService$_j$ is that we found it makes sense to differentiate between the data and the methods working on the data, this is due to the EntityService sometimes requiring multiple types of Entities. Furthermore it provides a separation of responsibility from the Entities and the EntityServices, given that Entities only provide a means of representing a row in the database and that EntityServices provide the behaviour for those representations.

Controller Layer

The controller layer consists of a number of controllers ConcreteController$_i$, each of which inherits from a base Controller class, which provides functionality to load the model as

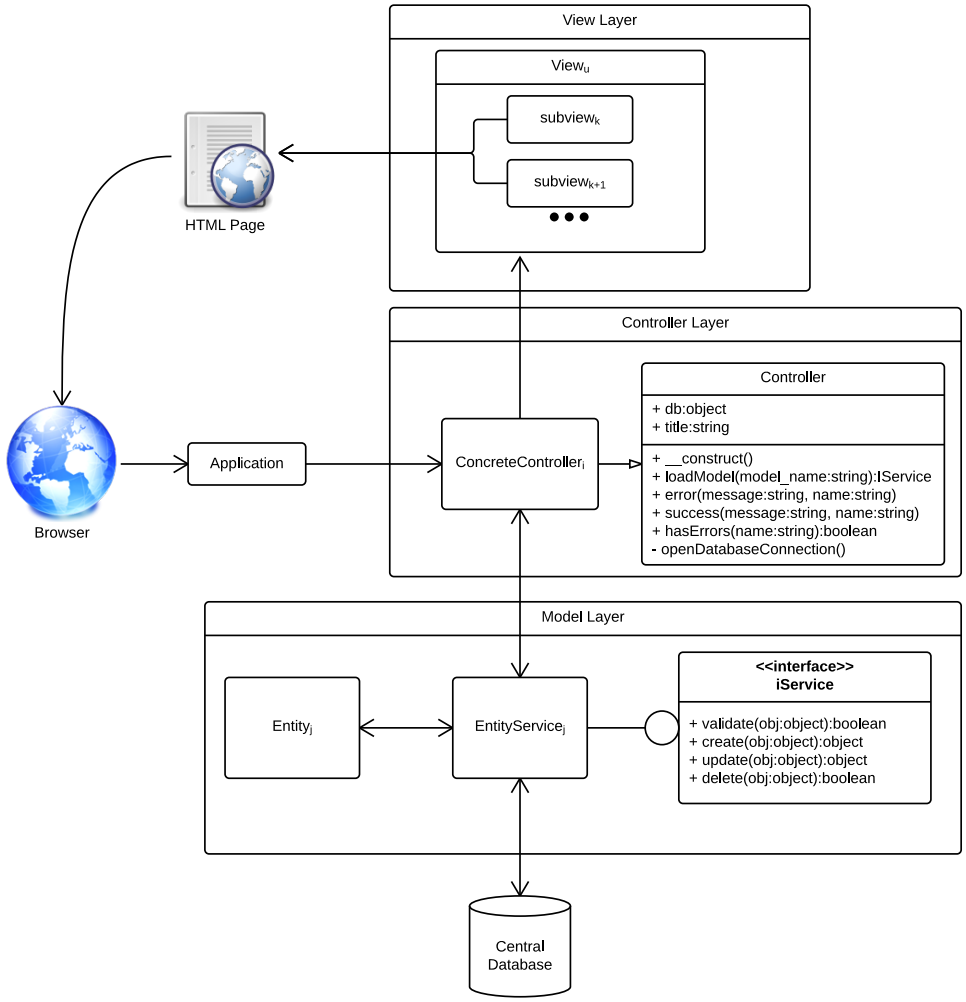


Figure 6.4: Website Structure

well as registering successes and errors to be displayed to the user by the view(s). Each concrete controller then contains a number of functions, corresponding to different sites, these functions are also called 'actions´. The responsibility of the concrete controller is then to work on the model and include views.

View Layer

The view layer takes care of the HTML generation part of the website, for each view included by the controllers, it consists of a number of subviews. Complete views should be thought as a combination of subviews, where some subviews occur in different views. Different classes of subviews exist, normal subviews and templates that are used almost universally throughout the site, for example the header and footer subviews. The combination of these subviews, the complete view, then takes care of representing a HTML page to be provided to the browser.

Application

When the browser visits the website, rewrite rules has been set in the *.htaccess* file, to ensure that the relative path part of the url is given as an argument to the default index file. This index file then loads the required files, such that the instantiated Application object has access to the configuration files needed. The GET variable url, which was set from the .htaccess file is then used by the Application object to navigate to the correct controller.

For a more detailed look at the model and controller layer, see Appendix A.1 and Appendix A.2.

6.3 Database

To support the necessary information storage, a database is implemented in accordance with the ER diagram described in Section 5.2. The implementation of the ER diagram leads to the database schema that can be seen in Figure 6.5, however, there is more than what the ER diagram describes. In this figure, the connections between the tables represents foreign key constraints.

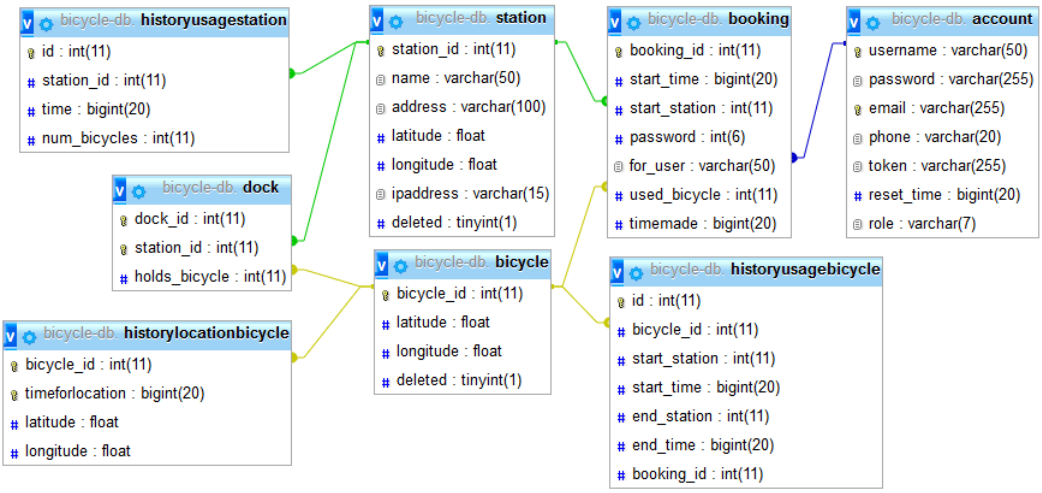


Figure 6.5: Database tables

The additional tables (historyusagestation, historyusagebicycle, and historylocationbicycle) are added to store the information needed for the features described in Section 5.3 and which implementation is discussed in Section 6.9. The three additional tables serves as a log, in order to store information about usage of the system, to be used by the administration part of the system to analyse on the usage of the system.

Because of how logging is used is different from the general usage of the system because it has to use all historical data. That means that for the purposes of the tables mentioned above it is important that for the station and bicycle tables that rows are not deleted, but rather just flagged as deleted, why is the cause of the deleted attribute in those tables. For regular use of the system it changes little but administratively it allows for using historical data.

In the translation from the ER diagram to the schema presented, the relations between the various entities could in all cases be represented as an attribute on one of the involved entities. This attribute is a foreign key. This was all that was needed to convert the ER diagram into the database schema, however, with the introduction of the administrator features and their associated data tables, more was needed.

When the administrator features was added into the database, in addition to adding three new tables to contain their information. The administrator features also required addition of additional attributes to some of the already existing tables, an example of this being *role* attribute on the account table, used to distinguish regular users from the ones with administrator privileges.

This was implemented in MySQL, primarily because it was packaged with the webserver that we used to develop the system with. There is no reason why MySQL was chosen instead of competing DBMS such as postgresql or MSSQL other than convenience, and little reason why the schema could not be adapted to work on others.

6.4 Interfaces

As is described in the design, some communication between the stations and the database has to be established, to retrieve bookings etc. This communication is established through the three interfaces gpsregister, websitetostationnotifier and stationtodbregister. These interfaces being from website to station, station to database, and bicycle to database. Each of these interfaces are described below.

6.4.1 Website to Station

This section is about the websitetostationnotifier interface from Figure 5.7. The website has to establish contact to a station when a booking or un-booking has been performed. It is evident that this is important, as the stations need to be notified to keep track of bookings involving themselves as station.

This contact is established through TCP/IP, and is used in implemented notification methods. These being `notifyStationDockChanged`, `notifyStationBooking` and `notifyStationUnbooking`. The call to the notification methods is added in the *BookingService*, as each time a booking is created, updated, or deleted, it is called through BookingService, and as of such we ensure the notification method is always called.

In general, the notification methods work in the following way:

- Construct JSON encoded message to be sent to the station.
- Create the TCP socket.
- Connect the socket to the station involved with the booking.
- Send the data to the station.
- Close the socket.

The reason the message is JSON encoded, is to have a standard way of encoding data, which can be easily decoded by the station. It is then up to the station to parse this information and perform the action specified in the message. How the JSON encoded message looks like, as well as how the station handles this notification is elaborated in Section 6.10.2.

6.4.2 Station to Database

This section is about the stationtodbregister interface from Figure 5.7. The station to database interface is important in order to register when a bicycle has been taken at a station or returned to a station, in order to give correct information on the website status page. Additionally the interface is used to retrieve information about bookings registered in the large database, in case of power-up of a station.

The interface has been implemented in a SOAP encoding by help of the NuSOAP library[33]. This library makes you able to write regular PHP functions and then register these with the NuSOAP library, in order to have a webservice generated. Bear in mind that all methods registered with NuSOAP needs to have a return value and as such a dummy boolean value is used where no other return value is needed.

An example of the implementation of an update and read operation on the database is presented in List 6.1.

```
1  $server->register('BicycleReturnedToDockAtStation',
2    array('bicycle_id' => 'xsd:int',
3    'station_id' => 'xsd:int',
4    'dock_id'    => 'xsd:int'),
5    array('return' => 'xsd:boolean'),
6    $SERVICE_NAMESPACE,
7    $SERVICE_NAMESPACE . '#soapaction',
8    'rpc',
9    'literal',
10   'Registers that a given bicycle has arrived at a given dock at a given
         station'
11 );
12 function BicycleReturnedToDockAtStation($bicycle_id, $station_id,
       $dock_id)
13 {
14   global $db;
15   $stmt = $db->prepare("UPDATE dock SET holds_bicycle = ? WHERE
         station_id = ? AND dock_id = ?");
16   $stmt->bind_param("iii", $bicycle_id, $station_id, $dock_id);
17   $stmt->execute();
18   $stmt->close();
19
20   [...]
21   return true;
22 }
```

List 6.1: Method for registering a bicycle as been returned to a dock at a given station.

If you take a look at List 6.1, the code is split into two parts, line 1-11 and line 12-21, which handles different parts of providing a method to register that a bicycle has returned to a dock at a given station.

Line 1-11 takes care of registering the PHP function, such that it can be included in the auto-generation of the SOAP encoded webservice. As can be seen from this specification, we tell the NuSOAP library the name of the function to register on line 1. Then line 2-4 specifies the input parameters, line 5 specifies the return value, and line 6-11 is less interesting parts which deals with how the method should be represented in SOAP.

Line 12-21 is the actual method for registering that a bicycle has been return to a dock at a given station. This is performed with use of prepared statements, as can be seen in line 15-18, which is done to prevent SQL injection. As you can see, the actual statement expresses an update on the dock, such that the dock, where the bicycle has been placed, gets a reference to that bicycle.

```
1 $server->register(
2   'GetAllBookingsForStation',
3   array('station_id' => 'xsd:int'),
4   array('return' => 'tns:BookingObjectArray'),
5   $SERVICE_NAMESPACE,
6   $SERVICE_NAMESPACE . '#soapaction',
7   'rpc',
8   'encoded',
9   'Get all bookings for station'
10 );
11 //in case you want to read everything.
12 function GetAllBookingsForStation($station_id)
13 {
14   //returns all bookings from database from a given station, as a json
       encoded array.
15 }
```

List 6.2: Method for reading all bookings for a given station

An example of a method that reads from the database is the *GetAllBookingsForStation* function, seen in List 6.2. Such a method is for example useful when first booting a station where its local database is empty or that the local database data is outdated, e.g. it has been offline for some time.

Taking a look at line 1-10, you see the registration of the PHP function for integration in the SOAP specification. As can be seen, it utilised a custom type called *BookingObjectArray*, which is an array type used to contain multiple JSON encoded strings that represents bookings. The result is a JSON encoded array and can be used by the given station to traverse the array returned and decode each string to get its corresponding booking information.

6.4.3 Bicycle to Database

FiXme Warning: Se comments i source her.

This section is about the gpsregister interface from Figure 5.7. The interface from bicycle to database, is an interface that is needed to register the location of a bicycle, as GPS tracking is decided to be implemented, due to the meeting held with Aalborg Kommune. The idea is that each bicycle use the interface to inform the system where it is located, according to the coordinates received from GPS.

The interface is implemented in the same fashion as the interface from station to database. As such, it is implemented as a SOAP web-service, using the NuSOAP library to gain the an encoding that makes the interface easy to call. There exists one method, the *RegisterGPS* method, which takes three arguments, the bicycle-id, latitude, and longitude. The way this interface is constructed is similar to the other SOAP encoded interface, but where the bicycle location is updated instead.

6.5 Model & Model Services

FiXme Warning: D
redegørende

This section describe the implementation of the model layer. We will illustrate this by giving an example model, showing what it looks like and what it does. Furthermore, a diagram of the model layer can be seen in Appendix A.2.

The Bicycle entity class can be seen in List 6.3. And as can be seen, what it does is to capture the attributes of the Bicycle entity from the database.

```
1  <?php
2  class Bicycle
3  {
4      public $bicycle_id = null;
5      public $longitude = null;
6      public $latitude = null;
7
8      function __construct($bicycle_id, $latitude, $longitude){
9          $this->bicycle_id = $bicycle_id;
10         $this->longitude = $longitude;
11         $this->latitude = $latitude;
12     }
13 }
14 ?>
```

List 6.3: Bicycle Class

Every entity model have a corresponding service that will contain the manipulation and handling of objects of the same type as the corresponding entity model. Each model service will implement an interface enforcing the class to implent methods such as `validate`, `create`, `update`, and `delete`, for an overview see Figure 5.7. Enforcing these methods to exist in every service dictates the responsibility of the service, for example enforcing a `validate` method dictates that it is the responsibility of the service to ensure a correct format of the model entity before making any changes to the database. Furthermore it dictates that the service should handle all contact with the database concerning its specific entity.

It is also the responsibility of the model service to handle all reads from the database, although it is not included in the implemented interface. This is because it may not always make sense to have a method that can read an entity from the database, and thereby this would possibly never be used. The reason for this is because it is more often that you would like to **read** multiple rows from the database than e.g. **delete** or **update**.

With the model layer described, we take a look at how the controllers are constructed, and how they use the model layer.

6.6 Controller

This section shows how the controller layer works. We illustrate this by giving an example controller, showing what it looks like and what it does.

In List 6.4 part of the Home Controller can be seen. It defines an 'action', this action is what is interpreted from the request sent by the client. For space saving purposes all actions but the index action have been omitted. The home controller uses two services, StationService,

used to retrieve an array of all stations, line 8-9, and BookingService, used to retrieve an array of all active bookings for the currently logged in user, line 11-14. This illustrates the general idea behind the separation of the entities (the models) and the services associated with those entities. It then uses the information loaded by 'including' views, and these views then use this information for displaying the content read from the services.

```
1  <?php
2  class Home extends Controller
3  {
4      public function index()
5      {
6          $this->title = "Home";
7          $currentPage = substr($_SERVER["REQUEST_URI"], 1);
8          $stationService = new StationService($this->db);
9          $stations = $stationService->readAllStations();
10
11         if (Tools::isLoggedIn()) {
12             $bookingService = new BookingService($this->db);
13             $activeBookings = $bookingService->getActiveBookings(
                   $_SESSION["login_user"]);
14         }
15
16         require 'application/views/_templates/header.php';
17         require 'application/views/home/index.php';
18         require 'application/views/_templates/footer.php';
19     }
20     // [...]
21 }
```

List 6.4: Home Controller Class

As can be seen on line 16-18 three views is included into the function. Because of the way include works in PHP the included content is in the same scope as the rest of the function, which means we can use the variables declared in the function, inside the included content.

The controllers and their 'actions' can be seen in Appendix A.1.

6.7 Views

This section describes the view layer and what it does, illustrating by example.

In List 6.5 a truncated version of the home index view can be seen. Most of it is normal HTML, but the important part is how it utilizes variables set in the controller, because a view is included by a controller, it gives access to the variables of the controller. This allows it to use them to generate more HTML as can be seen on line 6-10 where it uses the array of stations set in the controller to output option elements in a select element.

```
1  [...]
2  <form action="/Home/Book/" method="post">
3      [...]
4      <select name="station" id="stations" style="width: 243px;" onchange=
           "UpdateMarker()">
5      <option value="0" disabled selected>- Select Station -</option>
```

```
6       <?php
7       foreach($stations as $station){
8           echo '<option value="'.$station->station_id.'">'.$station->name
                .'</option>';
9       }
10      ?>
11      </select><br />
12      [...]
13      <?php
14          if (Tools::isLoggedIn()){
15              echo '<input type="submit" value="Book" />';
16          } else {
17              echo '<a href="/User/Login/">Login</a>';
18          }
19      ?>
20      [...]
21  </form>
22  [...]
```

List 6.5: Home Index View

6.8 Google Maps API

The Google Maps API [34] is used to show the map on both the front page and the administrator page. However, both of these pages use the API differently, the front page uses the API to show the map with the stations that are currently in the system, whereas the administrator page visualises where all the bicycles are located.

To use the API, the construction of the map is done first, which can be seen in List 6.6. Map options are chosen here, these options are e.g. the center of the map, which zoom level the map should have, and the style of the map.

```
1 var mapOptions = {
2   zoom: 13,
3   center: aalborg,
4   panControl: false,
5     zoomControlOptions: {
6     style: google.maps.ZoomControlStyle.LARGE,
7     position: google.maps.ControlPosition.RIGHT_TOP,
8   }
9 };
```

List 6.6: Construction of the map

After the map is constructed the map is filled with either the stations or the bicycles, depending on the page.

For the front page the stations are inserted into the map by use of AJAX. For each station, the work-flow of this is as follows:

Gather data
Use the stationservice to read data to be displayed on map.

Create info window

The info window is a window that appears whenever the user clicks on a station marker. The info window shows information about the station, which is the name, amount of free bicycles, and the amount of free docks.

Marker creation

Create a marker, which are the bulletpoints indicating the station on the map. The creation of the marker contains position, title, icon.

Assign listeners

Assign listeners to the marker clicks, in order to show the infowindow for the given marker when the marker is clicked.

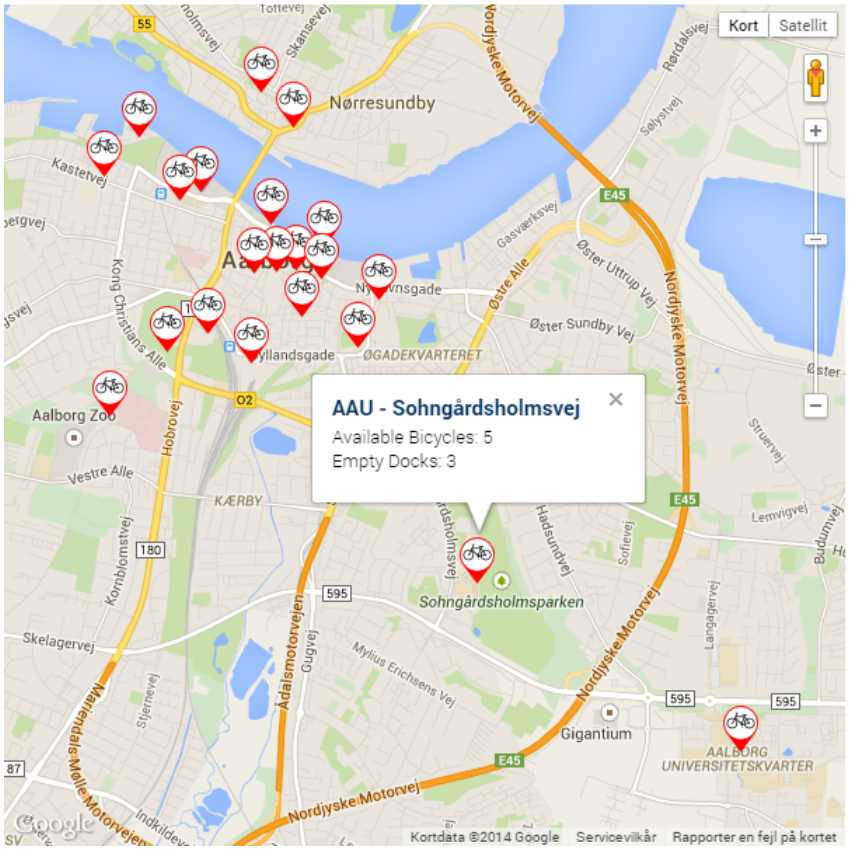


Figure 6.6: Google map with station markers and info window shown

An example of the final google map with markers placed can be seen in Figure 6.6.

For the administrator page the markers are bicycles, which are updated at a set frequency using AJAX, and is created in a similar fashion, just by other objects placed on the map. The main purpose of the administrator tracking page is to be able to see where the different bicycles are at a given time, which is the reason why the markers are updated once in a while. For the purpose of this project it updates every tenth second, to be able to simulate how it could look

like, however, in the real system every tenth second might be too often, depending on how often the GPS's of the bicycle send data to the database. For the update of the marker position AJAX is used to get all the positions of the different bicycles, which allows the makers to move without refreshing the page.

6.9 Administration Tools

In Section 5.3 a list of questions were made and explained, setting out requirements or guidelines for the features of the administration tools, in that an administrator should be able to answer the questions using these tools. In this section we implement the database schema designed in Section 5.3 and features using the data to visualise it in a manner making the administrator able to answer these questions. To answer the question about *which routes are used* we implemented a feature to show route history, which can be seen in Section 6.9.1. *Hotspot detection* was not implemented, but GPS history was implemented in Section 6.9.2. The *traffic of bicycles during some period* can be seen using the chord diagram in Section 6.9.3 The *current amount of bicycles at a given station* can be seen using the line diagram in Section 6.9.3. How the *amount of bicycles at a station change over time* can be seen in the same line diagram.

6.9.1 Route History

This part of the administration site handles showing a map of the historical locations of a bicycle. The administrator provides one or more bicycles, a start, and end date. Then if there are historical GPS coordinates for the bicycle(s), they will be shown on the map.

The purpose of the route history page is to give the administrators an idea of which roads the bicycles are traveling a lot on. This can then be used as a decision-making tool to decide where to put new stations, in that if the administrator can see that a lot of bicycles are traveling to a specific point it might be a good idea to put a station there. It could also be used to see if a specific bicycle is being 'misused' in the sense that it would be visible if the bicycle is only being used to travel to one spot and nowhere else.

See Figure 6.7 for a figure of what the page looks like.

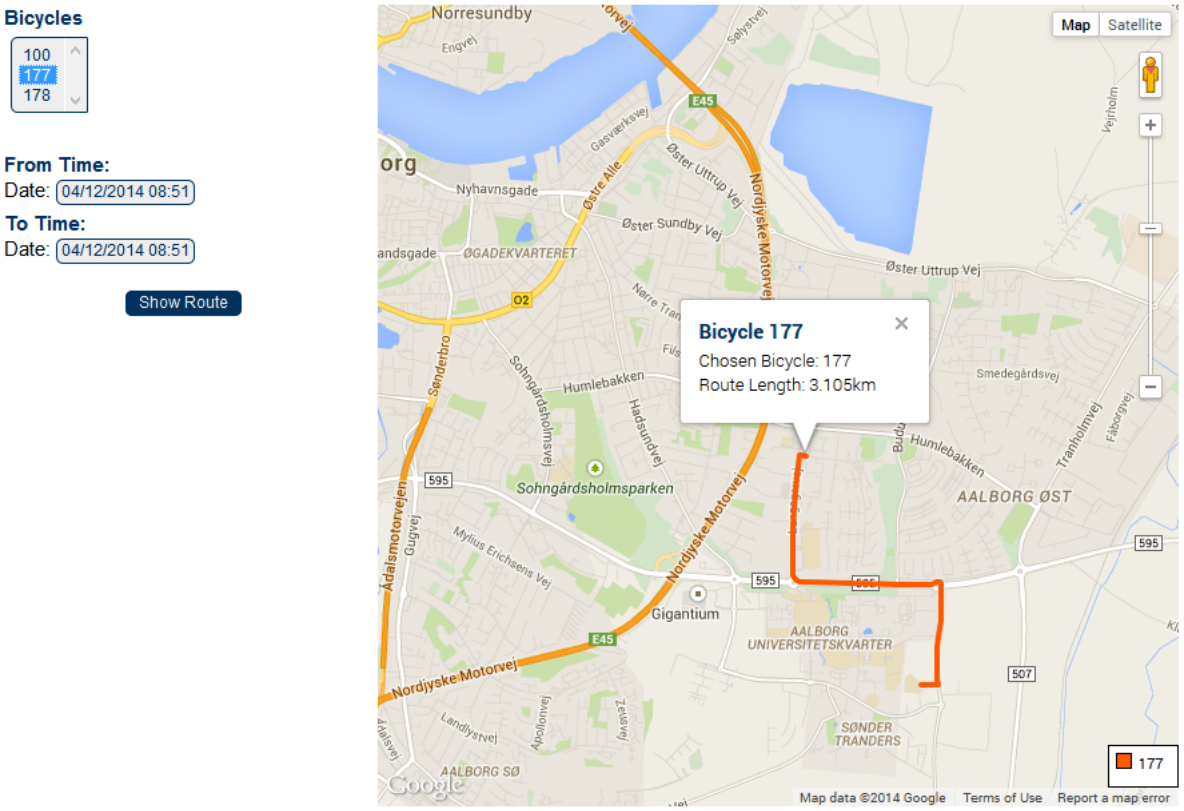


Figure 6.7: Picture of Route History

From the figure, the presentation of routes are shown for the bicycles with id 36 and 95.

The reason multiple bicycles can be selected is to enable the administrator to compare routes on the same map, this includes the route polyline, chosen bicycle and length,

This feature will make use of the historyusagelocation table designed and implemented, showing the coordinates from the table in a map. The table is inserted into by the gpsregister interface described in Section 5.4. It is implemented by finding all the bicycles that have associated historical GPS data. The administrator then chooses which bicycles to map, and these bicycles then have their routes loaded which are then given to Google Maps API to display as a polyline.

6.9.2 GPS History

From the previous administration feature a historyusagelocation table was implemented to store the positions of a bicycle. This can also be used to find the last known location of the bicycle. This would allow the administrator to determine where missing bicycles are, which in turn would allow for easier recovery of bicycles.

6.9.3 Bicycle Usage History

Besides recording the exact location of each bicycle with GPS coordinates, the system also records the stations a given bicycle visits. This information is stored in the historyusagebicycle table implemented from Section 5.3. Information is inserted into this table based on events happening at the stations. Every time a bicycle is taken from a dock at a station the stationtodbregister interface is contacted, which takes care of inserting the information about the start station of the trip. Furthermore every time a bicycle is returned to a station the interface takes care of updating the trip with an end station. Keeping such information about a bicycle trip can be used to give an overview of the bicycle traffic between stations.

The amount of docked bicycles is tracked for each station, giving an overview of which stations that have high and low activity. This is implemented through the historyusagestation table described in Section 5.3, storing the amount of bicycles at each station. The insertion is event-based and happens every time a bicycle is either returned to or taken from a station, again through the stationtodbregister interface described in Section 5.4.

For illustration the current amount of bicycles at a station and how the amount changes over time, a line diagram is generated showing time on the x-axis and the amount of bicycles on the y-axis, see Figure 6.8.

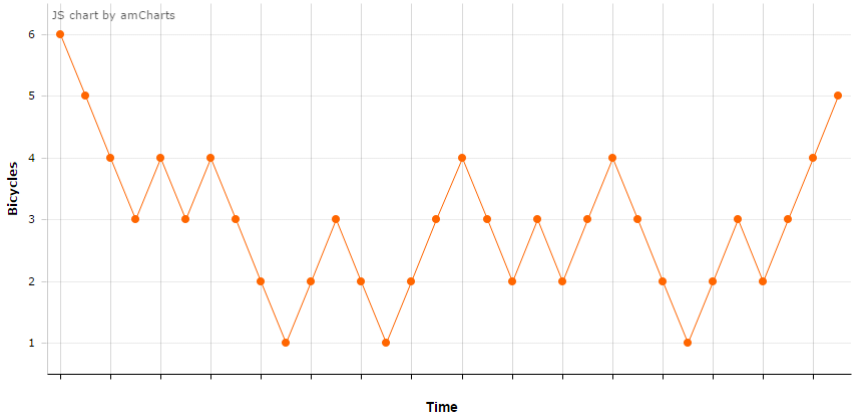


Figure 6.8: Line diagram showing the amount of bicycles.

To illustrate the traffic between stations we use a chord diagram. An example of this diagram can be seen in Figure 6.9. The input to the diagram is a list of station names and a $n \times n$ matrix where n is the number of stations and where each entry $e_{i,j}$ is a number of bicycles travelling from station i to station j. Each block at the edge of the diagram represents a station. The wider the block, the more bicycles have left that station. Each arc of the same colour as the block it is connected to, indicates the number of bicycles leaving the station and the destination of the bicycles. The width of the arc at each end shows how many bicycles have travelled in each direction between the two stations. For example the red one in the top between the two blue blocks shows that bicycles leaving have travelled to two different destinations. As seen from the table on the right, one bicycle went to Karolinelund, and one to Strandvejen. Since the arc is narrow at the destination block it means that no bicycle went the other way. In some cases the arc destination is somewhere between blocks. This means that the destination station does not

have any bicycles leaving in the selected time period, as only stations with bicycles leaving is shown as a block.

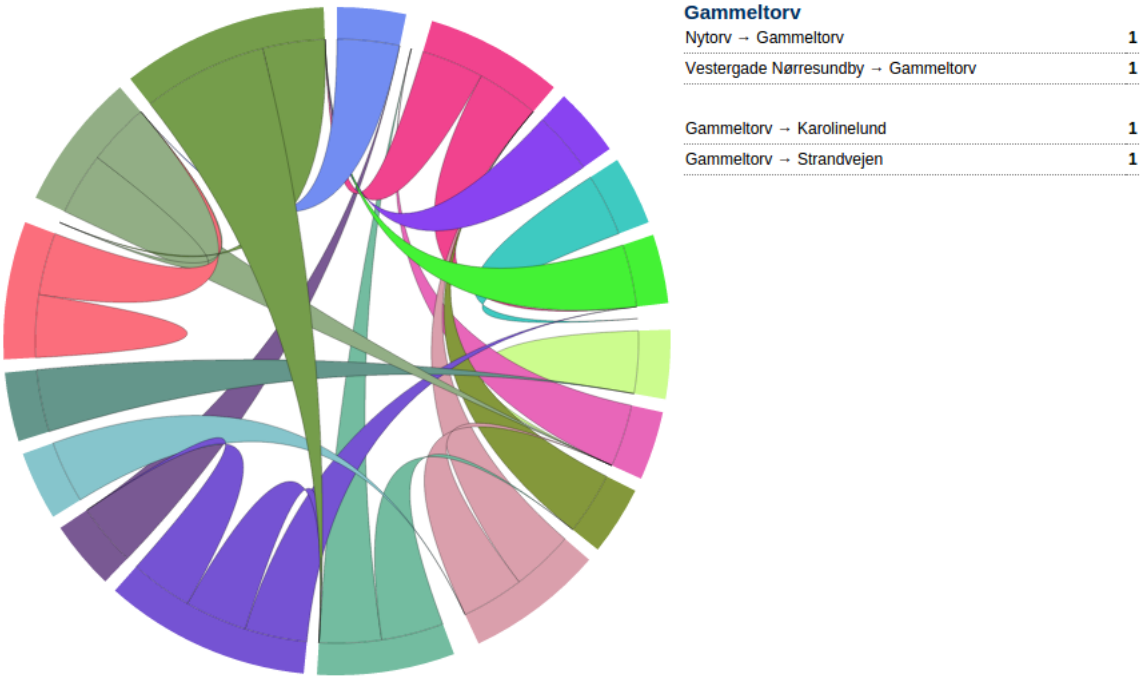


Figure 6.9: Chord diagram showing traffic of bicycles.

The timestamp of the event is generated just before query execution, which means that some delay can occur from the actual event firing time to the insertion time, resulting in an imprecise trip duration. This is, however, considered a minor deviation in most cases, because the communication delay is short (seconds maybe even less). In case of a bad connection, the generated timestamp will cause useless data in the light of statistical usage, if time is an important aspect of the analysis. For showing the traffic between stations the timestamp is used for filtering on a specified time interval and thereby allowing up to an hour of imprecision. This time interval is an assumed granularity and would have to be specified by the administrator. The graph showing the amount of bicycles docked at a station shows the timestamp on the x-axis, however, in this case it is considered acceptable with some imprecision since this data will not be used for statistics but for a visual overview.

6.10 Bicycle Station

The station software package contains two aspects, software intended to be run on the stations that are placed at each bicycle location throughout the city, and software to simulate the hardware we do not have access to, used as proof of concept. For an overview of the station structure, see Figure 5.8. However, to use these two aspects a graphical user interface is needed, and is described hereafter.

6.10.1 Graphical User Interface

The Main UI window has two parts, which can be seen in Figure 6.10, part one is the station software, and part two is for the hardware simulation, which will be explained later in this section.

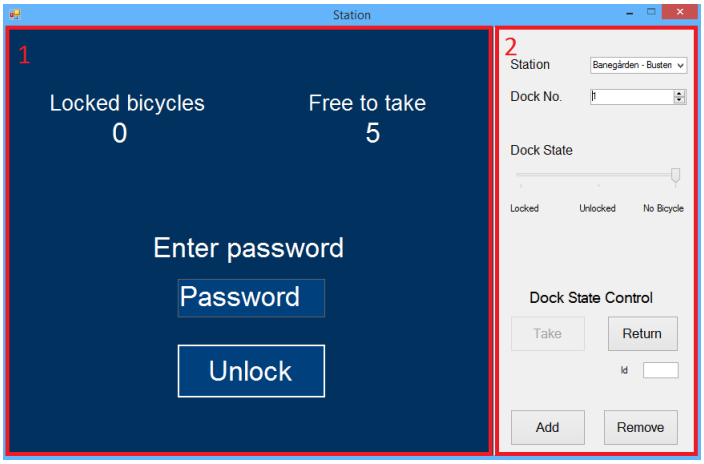


Figure 6.10: UI for the station software

The station software UI, part 1 in Figure 6.10, has been designed to be simple and understandable by users of the system with only the necessary content. The blue colour was chosen as this is the colour of the bicycles and the Aalborg Bycykel homepage. The station software UI is divided into two pages, the first one can be seen in the figure. It has a field to input a password for a booking to unlock the booked bicycle. In addition to this it also displays how many bicycles on the station that are locked and how many that are free to take. When a valid booking password is input, the station UI changes, which can be seen in Figure 6.11. The user is told at which dock the bicycle for him/her has just been unlocked. There is also a button to quickly return to the main window, which otherwise happens after 10 seconds.

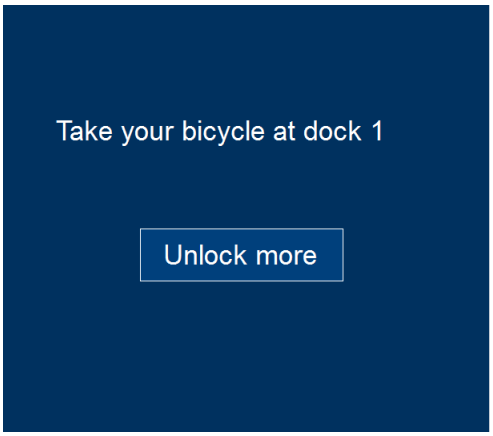


Figure 6.11: UI for unlocked bicycle

6.10.2 Station Backend

The station backend contains three parts as listed below.

- A listener that listens for signals from the global database interface called websitetostationnotifer, see Section 6.4.1, signalling that a booking has been created or removed.
- A lock manager that is responsible for locking a dock when a booking is close to its start time, and unlocking if a booking has expired.
- Communication with the global database through a SOAP service interface described in Section 6.4.2.

In the following each of these three parts will be described further.

Listener

The listener is made as a TCP Listener based heavily on code found on Microsofts Developer Network [35]. The listener listens for messages, which is done on port 10000, reporting changes in bookings, either addition or removal of one. The messages are JSON encoded strings of different forms depending on the content, two such examples can be seen in List 6.7 and List 6.8.

```
1 {
2  "action":"unbooking",
3  "start_station":5,
4  "booking_id":2
5 }
```

List 6.7: Example of an unbooking message

```
1 {
2  "action":"booking",
3  "start_station":5,
4  "booking_id":2,
5  "start_time":1414135298,
6  "password":483923
7  }
```

List 6.8: Example of a booking message

The information contained in the received message is then used to either remove a booking from the database, or add a booking to the database at the specified station, which action to perform is decided based on the action parameter.

SOAP Service

The SOAP Service is used to report changes in the local data of the station to the global database, which is implemented in the stationtodbregister interface, see Section 6.4.2. The methods in use along with at description of when they are used is listed below.

BicycleWithBookingUnlocked
Used when a booking has been used by the user and when an booking has not yet been used but has expired.

BicycleTaken
Used when a bicycle has been taken at a dock possible by means of a booking.

BicycleReturnedToDockAtStation
Used when a bicycle has been returned to a dock.

SyncDockStatus
Used when a dock has been added or removed from a station. It will synchronise the status of the docks at that station.

At the start-up or reboot of a station all bookings are synchronised, in order to get updated information for that station, where all bookings are first removed from the station database, and then all bookings are requested from the global database through the interface, using the method GetAllBookingsForStation. This is done in order to have up to date information and catch bookings performed while the station was offline.

With the communication between the station software and the global database, there is a risk of losing data during the communication or not having a connection at all. However, this connection loss is taken care of by starting threads that will attempt to send the data to the global database every second. This do give another problem which is when the threads are executed, which does matter because if a bicycle is returned and taken shortly after, the threads can be executed in a different order which makes the system believe that the bicycle is at the station. In order to solve this, a queue of function calls are made, so that the function calls are called in the right order, and thereby making this part of the program thread safe.

Lock Manager

The *LockManager* runs in its own thread, with the sole responsibility of locking and unlocking docks based on booking start times. Currently the time constraints is set to locking a bicycle to a dock one hour before booking start time, and unlocking again one hour after start time, see Section 5.4.1. As this functionality requires constant monitoring, the function is run in an infinite thread, however, since we do not want to waste our processor time by executing this constantly, it sleeps for a short period of time after each iteration.

FiXme Warning: Skal referere til implementation problems?

The loop starts out by finding all the expired bookings, removing the lock they had on a bicycle, telling the global database that the bookings have expired, and removing the bookings from the station database. The loop then finds all bookings that start within the next hour and locks a bicycle for the booking, if any of these bookings were not locked in the last iteration of the loop. It is important to note that if more than one booking start at the same time, the order of docks being locked is given by when the booking was created.

At the end of the execution of the loop the UI is updated to reflect any changes performed to the database.

6.10.3 Simulation of Hardware

The part of the UI representing the simulated hardware is the second part seen in Figure 6.10. The simulated hardware represents what would normally be observed at the physical station. We simulate the ID chips on the bicycles and the lock on the dock. The ID chips are simulated when they would normally have been read at the docks, which is when the bicycles are returned to it after use. In this case, when a bicycle is returned, we find a random ID from among those not currently in a dock, and selects this as the ID returned. Random ID is sufficient at this time, as we have no way of predicting where a bicycle in use would be returned to. Alternatively a bicycle ID can be specified from the GUI, giving more control for illustration purposes. But

for a real station, this can be performed with an RFID chip. If we want to register a specific id, this can also be specified in this proof of concept software. The lock is simulated by a boolean representation, stating if it is locked or not.

In the simulation part of the UI, there is a dropdown list where it is possible to select which station a user is standing at, along with a numeric selector allowing selection of a specific dock at the current station. In addition, there is also a slide bar showing the state of the selected dock. The possible states being locked, unlocked, and no bicycle. At the bottom of the UI there are four buttons that simulate the user actions of returning and removing a bicycle, a field giving the option of specifying which bicycle is returned, and buttons to add and remove docks at the selected station.

Additionally there are buttons for adding and removing docks for the selected station. This is handled station-side and not centrally, because otherwise it would require the administrator to add the dock to the system on the website and it could create situations where the administrator adds a dock that does not exist at the station. This is of course a design choice that could work in either direction, but due to the reasons given station-side seemed more natural.

6.11 Synchronisation

As the system is split into stations and a central system, it is necessary to have communication between them, and tackle the synchronisation of data. This communication data flow have been illustrated in Figure 5.7, additionally see Section 5.5 has discussed this problem previously as well.

Bicycle locking/unlocking not synchronised
: What is important for the central part of the system to know is at what docks at stations bicycles are placed. However, it is not important for the central system to keep track on what bicycles are locked and unlocked at a station, as this information is of no use for the central system. The information is on the other hand important for the station, as this is the part of the system that ensures that bicycles are locked in time. For that reason, the information is kept at the station. Furthermore, the central system can calculate how many bicycles are locked at a given station, as it can use the bookings for this.

Removal/Insertion of bicycles
: On the other hand it is important for the central system to know when a bicycle has been removed or inserted at a station. This is due to the website providing status information to the user, such that they can know where free bicycles are located. The communication to ensure this, as previously mentioned, is then performed by use of SOAP. In order to ensure that this communication goes through, the station has a seperate thread to communicate this information, if the station is unable to communicate that a bicycle has been removed/inserted, it will wait a bit and then try again until connection has been established.

Booking/Unbooking
: The booking and unbooking of bicycles are performed on the website. The central database is then updated accordingly, and in order to ensure that the station affected is notified, a TCP package is sent, which contains the JSON encoded action, as described earlier. However, problems may arise if the connection to the station is down. In such a case, you can risk the station not registering the booking, and as of such, one of the following actions have to be performed.

- Flag to indicate new bookings
- Constantly ping central database (unreliable)
- Call local service to continuously try and connect until notification has been achieved.
- Give the user an error message, telling him to try and book/unbook again later.

The idea behind having a flag to indicate new bookings is that a flag will be set for the station that failed to be connected to. On the station side, a thread will then run and check every x amount of seconds for a flag that may have been sent. If such a flag is registered, the station then knows that it should update its bookings to correspond to those of the central database.

Another more simple, but non reliable, idea is to constantly ping the central database. If you for one or several of your pings then fail to connect to the central system, you know that the next time you get connection, you have to update the bookings. One problem that makes us not use this approach, though, is in the timespan of two pings, if the connection fails there, the station will not get notified of a registered booking in that said time-period.

Those two ideas involves giving the station the responsibility of having up to date booking information. However, another perspective can also be taken. One such approach is to develop a service to run on the central part of the system. The responsibility of this service is then to ensure that the station will get notified. The idea is that the service holds queues of notifications to be sent out, one such queue for each station. The service can then get notified to enqueue a notification, but the service will then continuously try and deliver the notifications from the queues to the stations. This is found to be a good approach, since it ensures that when connection is established to a station again, the notifications it needs gets delivered.

The last approach is to give the user an error message telling him to try and book/unbook again later. This approach is not desirable as we want the system to be as responsive as possible. The approach with a flag or the additional notification service is desirable. However, due to lack of time, it has been chosen as the solution, it was deemed a necessary compromise, but with more time, the flag or the separate service solution would be strived for.

Station boot/reboot

When the station boots/reboots it may have missed some booking information. For that reason, when the station boots/reboots it reloads the booking information. However, with the notification securing set in place, this should not be necessary. This part of the synchronisation was developed before the other part, though, and is kept in case the other synchronisation techniques for some reason fails. Additionally, as this is a simulation, we run the station software on different computer, and as of such it is advantageous for testing to keep this updating of booking information in place, such that the various computers gets the booking information updated.

Dock insertion/removal

When a dock gets inserted or removed from a station, the station has a separate thread that keeps trying to call the web-service set in place on the central system part, to register that a dock has been inserted/removed at the given station, this is repeated until the web-service has been correctly used. This is achieved with specific functions set in place for the 'stationtodbregister' service.

Hardware versus software simulation

For the system developed, software simulation of hardware has been used. However, if you were to implement real docking hardware, a synchronisation between the stations and its docks would have to be considered. This is somewhat achieved with the local database set for each station, however, with the sensors active, you would need to setup some communication protocol between the station and its docks, to ensure that the local database is up to date. Additionally, to go the other way around and send signals to the docks, notifying them if they should lock or unlock some bicycle.

This ends the general synchronisation implementation details, with a brief documentation of our testing following hereafter.

7 Test

Testing is an important aspect of ensuring that a system works correctly and as intended without unknown side effects. That is why it is important to verify that the model and interface layer works as intended, this can be done using unit testing, which is why we perform unit tests on those layers.

While unit testing works for behavioural code and verifying that everything works correctly there, we have an architecture that separates the behavioural aspects of the system away from the visual and experience aspects, specifically the controller and view layers. These layers also need to be tested, and unit testing does not fit well with them and as such a different way to test them is required, which is where usability testing comes into the picture because while they mostly reveal usability problems they can also reveal critical errors where the system does not behave as expected. As such usability tests are also performed.

7.1 Unit Testing

Unit testing is testing individual components of the code systematically and thoroughly, to determine if the overall system works as intended. The process of a unit test follows these steps:

Set Up
The objects, and/or data needed is set up.

Action
Some action are performed on the objects and the data. Some times not necessary.

Test The output of the actions are tested if they match the expectations through assertions.

Tear Down/Clean Up
The objects and data are cleaned up. This is specifically important for us because we insert data into a database and it needs to be removed afterwards in order for the tests to be repeatable and to remove test data that might be shown on the website if not removed. Sometimes not necessary.

The unit tests are performed on the model layer of the website along with the SOAP interface used to send messages to the station software. As the tested parts are written in PHP, the PHPUnit testing framework is used for running the tests because it provides a consistent approach to writing and running them.

```
1 public function testRead()
2 {
3   $stationService = new StationService($this->db);
4   $station = new Station(10000, "test station name", "test address",
        57.1, 9.2);
5
6   $stationService->create($station);
7   $result = $stationService->read($station->station_id);
```

```
8   $this->AssertEquals(10000, $station->station_id);
9   $this->AssertEquals("test station name", $station->name);
10  $this->AssertEquals("test address", $station->address);
11  $this->AssertEquals(57.1, $station->latitude);
12  $this->AssertEquals(9.2, $station->longitude);
13
14  $stationService->deleteForTest($station);
15  $result = $stationService->read($station->station_id);
16  $this->AssertEquals(null, $result);
17 }
```

List 7.1: Example of a unit test

For an example of a unit test, see List 7.1. As can be seen it sets up the data on lines 3-6, acts on line 7 and 15, tests on lines 8-12 and on line 16, and cleans up on lines 14.

We currently have 44 tests with 181 assertions and these cover every model and their associated modelservices. The benefit from unit testing was great as it showed errors that were not immediately obvious, for example multiple places when binding parameters to mysqli statements incorrect types were used and as another example methods were called incorrectly. As such the unit testing led to a lot of errors being resolved. Additionally, unit testing was good to register erroneous changes made to the program, as you could run your tests and the erroneous parts of the program would be located.

7.2 Usability Test

This section covers the usability tests performed on the system. Tests have been performed on the public part of the website. The administrator page was not tested because no potential users were available to do so.

These are the tasks:

1. Establish an overview
2. Status for bicycle
3. Booking - Login
4. Booking - Time and booking
5. Cancel planned booking
6. Examine the booking history

For a full description of the tasks see Appendix C.

The usability problems were uncovered using Instant Data Analysis [36].

The procedure for IDA is as follows:

- Test monitor introduces test subject to the evaluation procedure. After which the test monitor gives the first task to the test subject.

- The test subject attempts to solve the task until feel they have accomplished it, they have accomplished it, or until the test monitor gives them a new task.
- Test subjects talk about what they are doing, explaining what they are doing, and how they feel they understand the overall system. Test monitor does not help.
- The test subject is interviewed and talk about their experience.

After the test, there is a meeting with the user where problems are discussed and which they felt were significant. The result of Instant Data Analysis is a set of usability problems categorised into one of three categories: Critical, Serious, and Cosmetic. We used IDA because it catches most of the usability problems a more rigorous approach would have caught, in much less time.

7.2.1 Test Subjects

The tests were performed on four test subjects, whose demographics are as follows.

1. Gender: Female, Age: 29, Profession: Lawyer, Technological abilities: Average
2. Gender: Male, Age: 28, Profession: Lawyer, Technological abilities: Above average
3. Gender: Female, Age: 56, Profession: Office clerk, Technological abilities: Average
4. Gender: Male, Age: 58, Profession: Electrician, Technological abilities: Above average

7.2.2 Usability Problems

As a result of the tests, the following usability issues were uncovered.

#1 Fields reset
All subjects experienced a problem with the fields resetting if they tried to book with incorrect information. This is for test subject 3 particularly critical, as she double booked a bicycle.

#2 Error message understandability
Test subject 4 had a problem with understanding the error message if location is not selected. The error message is not descriptive enough, as it says "Please fill in all fields" instead of "please choose a location".

#3 Difficulty finding history
Test subject 1 and 3 both experienced some problems finding the booking history button, with subject 3 having it being a bit more severe, as it took her half a minute to find the button, whereas for subject 1, it took about 15 seconds. The problem for both of them being that they did not notice the button the first time they navigated to the profile part of the site.

#4 Booking/Unbooking confirmation
Subject 2 became a bit confused on whether he booked/unbooked something, as no confirmation is shown to the user. Additionally, he requested a confirmation box, to prevent a person with clumpsy fingers to on accident book/unbook a bicycle.

Issue Description	Test Subject ID			
	1	2	3	4
#1 Fields reset	S		Crit	S
#2 Error message understandability				Cos
#3 Difficulty finding history	Cos		S	
#4 Booking/Unbooking confirmation		Cos		

Table 7.1: Usability issues overview.

An overview of the issues that each test subject experienced can be seen in Table 7.1. The issues are ranked for each test subject, such that Cos means a cosmetic issue, S means a serious issue, and Crit means a critical issue.

As can be seen, the amount of usability issues found is very low. This may be because the website is simple and easy to use, however, it is probably also connected to the size of the usability tasks document. Four issues was, however, found, and for each of these issues, we go through what might be changed in order to fix these issues.

#1 Fields reset
The issue with fields resetting when you click on book/unbook is tied to not using AJAX for that part of the website. The issue we find to be able to solve in two ways. One is to restructure that part of the website to use AJAX. Another way is to save the information of the fields in sessions, in order to be loaded when the page gets reloaded.

#2 Error message understandability
This issue is tied to checking if any of the fields are empty. Instead, a more specific error message could be given to indicate which field has a missing entry.

#3 Difficulty finding history
In order to solve this issue, the button size could be increased. However, as it is a matter of overlooking the button, it is likely an issue that will not be present when the user have located the button at least once before.

#4 Booking/Unbooking confirmation
The booking/unbooking issue can be solved in two ways. One way is to present some status text when the booking/unbooking action has been performed. Another way is to enforce a confirmation box for the user to agree with the action initiated.

We find the first way better, for the booking action, and the confirmation box better for the unbooking action. This is due to booking a bicycle by accident is not as severe an action as the unbooking, since if you booked something by accident, you can unbook it after and no harm is done. On the other hand, if you by accident unbook something, it is a more severe action, since you might not be guaranteed to have a bicycle booked again, if for instance another person books the last bicycle in the meantime.

An effort to correct the problems was made, and these are the changes that were made: More descriptive error messages, confirmation dialog boxes added for unbooking, and the history button was made easier to locate by making the font size larger. At the same time corrections for fields being reset was not done because of time constraints.

8 Discussion

While developing the system, various problems were found, which may affect the functionality of the system. Several of the problems were solved, but some problems still exist that may affect the functionality of the system in the Aalborg Bycykel domain. For that reason, it is necessary to touch upon these issues and discuss what can be done differently, or why the chosen solution is sufficient.

In relation to this, implementation decisions made are discussed. Additionally, the system is compared to the existing systems discussed in Section 2.2, in order to determine the pros and cons of the developed system. Moreover decisions about what should be implemented in the future to support or improve the system is discussed in Section 8.2.

Booking Static

A concern of Aalborg Kommune, mentioned in Section 2.1.1, is that the booking system might be too static. By that meaning that too many bicycles are locked at docks instead of actively being used around Aalborg. The problem with the system is that as the booking part of the website is used more, the system becomes more static. Ways to ensure that the system stays a bit dynamic are one or more of the following, each of which are discussed in turn:

Lock late
The idea of this solution is to delay the locking of bicycles, reducing the static time each booking affects the system. However, the risk with this approach is that as you reduce the amount of time a bicycle is locked, you increase the risk of the bicycle not being available for the planned booking. This is something to consider, but for the moment, the simulation of a station has a lock time of one hour before a bicycle is planned to be obtained. Alone, this solution is not desirable, as you have no way of detecting how many bicycles are available in the future. To compensate this, the lock time could vary if you were to integrate the locking mechanism with GPS tracking, and is related to the latter point about prediction.

Subset of bicycles for booking
The idea is to only allow a subset of the amount of bicycles on a station to be booked, this could for instance be that at all times at most half of the bicycles on a station can be booked. There are a few different approaches to how this could be handled. One option is to simply exclude some of the docks from the system, so that at each station some of the docks physically placed at the station will not be in the system. Another option is that the system itself keeps track of how many bicycles are currently at each station, and then it cannot lock anymore than a predefined percentage of these. The problem with the first option is that the system cannot accurately tell how many bicycles are available at a station, if it does not have access to read from all the docks. The problem with the second approach is that it is difficult for users of the website to know when a booking is possible as the amount of lockable bicycles at each station is dynamically updating. This leads to a third option, which is a combination of the first two. This option is that each station has a set amount of bicycles that it can lock. As this a fixed amount, it is possible to predict if a bicycle can be locked at the given time and due to this, it is possible to inform users at

the time of booking creation, whether the booking was successful or not, but also give a status of how many bicycles are available for booking and how many are free to take.

Prediction

The idea is that if you are able to predict when bicycles are returning to the station, you can unlock bicycles on that basis. An example of this is that you have a booking in 15 minutes, there is one bicycle left at the station. Then since you know that a bicycle returns to the station in about 5 minutes, the last remaining bicycle can be unlocked for use. How you are then able to predict this could be with use of GPS tracking and substantial statistical analysis. This would allow for a solution to the unreliability of the system, as the prediction could be integrated into how locking of bicycles are performed. However, to implement this is a project on its own, but is worth considering for further development. In such a further development, different machine intelligence methods could be looked at, in the area of classification. This is due to you having previous labelled which is previous bicycle routes with their end stations, by enough samples of this, it could be used to predict whether one or more bicycles will arrive at the station in some timespan or not.

Website Designed for PC

At the current stage, the website part of the system have been designed with a PC in mind. However, it is evident that there is a tendency to more people using smartphones and tablets [37]. For that reason, it would be a good idea to design the website such that it is also easy to use for such devices. There are several ways to tackle this, each of which are discussed in turn.

Dynamic scaling

The main idea of this approach is to make the website more dynamic. This can be achieved with CSS to consider your screen size, and then transform the site such that each element of the site can be seen clearly, where to obtain all information, you then have to use scrolling. This is better than the standard webpage, as the website at the moment for small screens, known from smartphones, is near unusable unless you zoom in. However, it does not touch upon the type of device used for the website, an example is the touch gestures known from tablets and smarthpones.

Detection of device type

The idea is to read the device type, and then have separate website layouts for each type of device. Contrary to the previous method, this makes it easier to design the layout for the different type of devices. Furthermore, it can change some elements to be better suitable for touch devices, an example is a list of items, for smartphones the list can then be automatically zoomed on to select a specific item. However, it still does not give the best feel for smartphone devices, as you are constrained to a browser. The pro is that you can reach many devices this way, on the other hand, applications could gain a more specialised feel, and is described hereafter,

Separate Application

The idea of this approach is to develop applications for some regularly used smart-phone and tablet devices. The advantage of this approach is that you can specialise the interaction with the system for a specific device, ensuring the best interaction overall. The disadvantage is that it increases the workload a lot, as you have to create and maintain each such application for changes performed to the system.

This ends the discussion of website designed for PC versus a more dynamic website. While it would definitely increase the accessibility of the system if the website became more dynamic, it would at the same time require additional resources. Our target for the project has been to develop the website for PC usage, and that being optimal. However, in the future with more time available, the other options are definitely worth considering.

Hardware vs. Simulation

One aspect of the system implemented that would have to be changed in the situation that the system was put into actual use, would be the simulation of the hardware. Specifically, the station, docks, and bicycles are all simulated and in the real world these would need to be changed to be real hardware. The simulation is done by having multiple programs behaving somewhat similar to how the real world would interact with the system. An example of this is the bicycle, the bicycle should upload its GPS coordinates to the system every now and then. A program for this was created to upload GPS coordinates that are loaded from a file instead of a real bicycle reporting the actual coordinates.

A big difference between the simulation and the hardware going to be used, is how the simulation works. For example the simulation of all the stations is implemented in a single program that handles all interaction. This would, however, not be the case in the real world where each station would have its own software. However, this problem was thought of in the development of the station software, such that each station has its own ip address, and can therefore work even if multiple computers are running the station software. As the central server requests the right IP address when sending a booking/unbooking to the station.

The simulation is done such that the development of this system does not require real station hardware. Furthermore, this project is a software development project, and therefore hardware is not the focus of this project. For this purpose of this project the simulation is considered sufficient because it allows the system to be interacted with as it is supposed to be in the real world without making it needlessly complex, for example by having to run multiple stations at the same time on several different computers.

8.1 Perspective

This section provides a perspective comparing our implemented system with existing systems, see Section 2.2, and the existing system in Aalborg, see Section 2.1.

In the Copenhagen Gobike system, the system uses expensive hardware which could make it difficult to purchase new bicycles because every bicycle needs a tablet before they can be integrated with the system. Because of this Copenhagen Gobike costs money to use, as do the other existing systems (Cibi and Alta Bicycle Share). This is something we wanted to avoid, we wanted a system that could be used for free, or at least only be used on a deposit basis where you get the money back at the end of using the bicycle ride. We did manage to do this, though it is unknown what kind of expenditures would be associated with the stations, docks, and GPS reporters on the bicycles.

One of the bigger issues, especially for the administrators of Aalborg Bycyklen, is that there is no tracking capability. With Copenhagen Gobike this is not a problem as they provide GPS coordinates for their positions. However, for Cibi and Alta the only kind of tracking of bicycles is that the stations report how many bicycles are at the dock at any given moment. This leaves much to be desired in that it does not provide any kind of information about the bicycles once

they leave the stations, which of course makes them difficult to locate if they get lost. Though they try to prevent people from stealing or otherwise not returning the bicycles by penalizing the users with extra fees, loss of deposits or fines. For our system, there is a mechanism for finding lost or stolen bicycles through GPS reporting, which does seem to be the simplest solution to the problem of not being able to recover bicycles. However, the only penalty for not returning a bicycle would be the loss of the deposit, but the original deposit is so low as to be fairly irrelevant. This is not something we considered changing, and our impression of Aalborg Kommune's desire for the system is that it should be easily accessible and not something you have to pay to use.

For existing systems, the unlocking process requires some kind of identification which is something we wanted to replicate, as such we ended up with a booking system that lets the user book bicycles and then provides the user with a code that unlocks the bicycle at the specified station. This is probably one aspect of our system that is somewhat worse than solutions other systems have used, but it does provide a means of reserving and ensuring that bicycles are available when you want to use it, which is something other systems do not do. It should, however, be said that the system is kept open and that you are not required to use bookings to use it. So while the implementation letting the user to go online and booking bicycles could be better, it does come with benefits and the user is not required to use it. One consideration made, to provide a more easily accessible booking and unlock mechanism, would be to have a mobile booking application providing a simplified version of the website implemented, see Section 8.2.1.

We attempted to provide an updated version of Aalborg Bycykel that could challenge other existing systems, which we feel we managed to do. However, it did come at the potential cost of compromising the ideas behind the original system to some extent, for example through locking of bicycles through bookings instead of leaving them unlocked and available to anyone. These changes are not made lightly and makes the system more predictable and reliable, because as Aalborg Kommune said they often see that stations remain empty at all times making it difficult to even use the system, see Section 2.1.1. The changes, however, could bring new problems that were not predicted and as such the system of course would have to be corrected and fixed once these come up.

These corrections would be part of the further development, which is discussed hereafter.

8.2 Further Development

This section covers various features that could be implemented in future iterations of the system. Specifically a mobile booking application, closest available station information, hotspot detection, usability testing for the administrator page, and sending unlock code over SMS.

FiXme Warning: Mangler der noget?

8.2.1 Mobile Booking Application

During early analysis, a mobile platform was suggested for implementation, but in the end the target platform chosen was the web. However, a simplified mobile application could be implemented, simplified in the sense that it would only provide login and booking functionality and not much else. It would, however, also be an ideal platform for another further development feature, see Section 8.2.2, given that GPS and internet connection are pervasive in modern mobile platforms. The fact that you can take smart-phones with you makes them much more practical in the sense that you can determine at any time where the nearest station is.

8.2.2 Closest Available Station

During usability testing, one of the test subjects brought up an idea. Specifically a feature that could be implemented on the website or as a mobile application is suggestion of closest available station.

The website could provide a suggestion of which station to go to based on distance where there is a bicycle free to use, where as the mobile application could suggest both the closest station with an available bicycle but also a station with an empty dock. The closest station with an available bicycle can be used to see where the user would be able to get a bicycle, where as the available dock could be used when using a bicycle so you know where to go to deliver the bicycle to the station.

8.2.3 Hotspot Detection

In Chapter 5.3 a feature for administrators was suggested for showing 'hotspots' for bicycle activity. It was thought that it would allow administrators to more easily make decisions on where to place new stations, because if a hotspot showed areas with a lot of activity not close to any existing station, it would be logical to put a new station there.

However, questions about how the location data should be used were raised. This meant more analysis on how to do it properly, and because of resource constraints this meant that no more development time would be dedicated to this feature. Specifically, should only location data where the bicycle has been at the same position for a long time be used? Should all location data be used, and would this not mean that popular routes would be hotspots as well and therefore be misleading?

There are many different ways of detecting hotspots, or clusters.

The simplest method that might work is using a third party premade library for Google Maps API called MarkerClusterer. The algorithm works in a grid based manner [38]. For each zoom level of the map, the clustering is performed. In a brief description, the algorithm traverses the markers to be clustered and adds a marker into the closest cluster it is nearest if it is within the minimum square bounds, else it is added to a new cluster. An example of the result of the clustering can be seen in Figure 8.1. Other clustering libraries for google maps exist and would have to be compared before a specific algorithm were to be chosen, a good place to start would be a list of various Google Maps clustering algorithm APIs [38].

8.2.4 Usability Testing on Administrator Part

We had originally wanted to usability test both the public and administrative parts of the website. Usability tests were performed on the public part, and problems were found and resolved. The administrative users were, however, not available for usability testing. This usability could have given more features to be implemented on the administrator page, which had not been thought of. It could also have resolved potential problems that we were not previously aware of it. Furthermore, the usability test would be sufficient to make sure that the administrator could use the page.

The administrator usability test should be done before the website is released, since it might reveal some problems with the page.

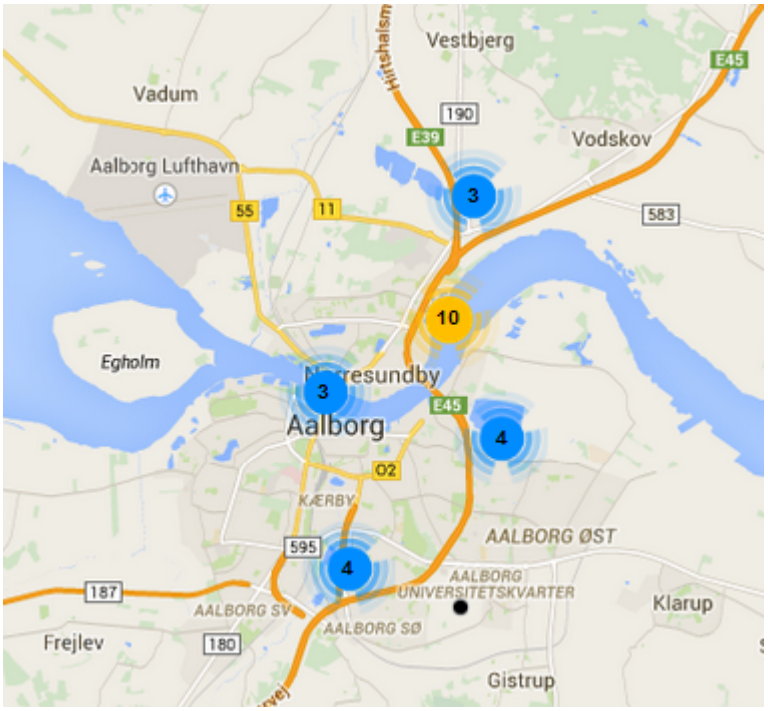


Figure 8.1: A result of the MarkerClusterer library

8.2.5 Sending Unlock Code over SMS

Originally, it had been intended that the unlock code for a booking would be sent over SMS. This seemed out of scope for the project and it would have cost money to do so, as such it was decided to provide the unlock code on the front page. This, however, is something that could be added in the future.

9 Conclusion

Bibliography

[1] Regeringen. Sundere liv for alle - nationale mål for danskernes sundhed de næste 10 år. URL http://sum.dk/~/media/Filer%20-%20Publikationer_i_pdf/2014/Nationale-maal/Nationale-Maal-2.ashx.

[2] Energi Styrelsen. Dansk klima- og energipolitik. URL http://www.ens.dk/politik/dansk-klima-energipolitik.

[3] Impact evaluation of a public bicycle share program on cycling, owner = Lasse, timestamp = 2014.12.04, url = http://ajph.aphapublications.org/doi/abs/10.2105/AJPH.2012.300917?journalCode=ajph, .

[4] AFA JCDecaux. Cibi by AFA JCDecaux. Web, . URL http://cibi.dk/.

[5] Gobike. Gobike. Web. URL http://gobike.com/.

[6] AltaBicycleShare. Web. URL http://www.altabicycleshare.com/.

[7] Aalborg Havn Aalborg Komune. Bycyklen. Web, . URL http://www.aalborgbycyklen.dk/.

[8] aml-teknik@aalborg.dk Anne Marie Lautrup Nielsen, Civilingeniør. Integreret cykelplanlægning i Aalborg for pendlere, turister og børn. Technical Report ISSN 1603-9696, Aalborg University, 2012. URL http://www.trafikdage.dk/papers_2012/65_AnneMarieLautrupNielsen.pdf.

[9] Aalborg Havn Aalborg Komune. Det er let at finde en bycykel. Web, . URL http://www.aalborgbycyklen.dk/default.aspx?m=2&i=37.

[10] Aalborg Havn Aalborg Komune. Det er simpelt at låne en bycykel. Web, . URL http://www.aalborgbycyklen.dk/default.aspx?m=2&i=36.

[11] Aalborg Havn Aalborg Komune. Bycyklen er din anden cykel. Web, . URL http://www.aalborgbycyklen.dk/default.aspx?m=2&i=38.

[12] AFA JCDecaux. Cibi by AFA JCDecaux. Web, . URL http://cibi.dk/omcibi/.

[13] Robert Love. Why Does GPS Use More Battery Than Any Other Antenna Or Sensor In A Smartphone? Web. URL http://www.forbes.com/sites/quora/2013/08/06/why-does-gps-use-more-battery-than-any-other-antenna-or-sensor-in-a-smartphone/.

[14] Danmarks Statistik. Anvendelse af internet på mobiltelefonen $16 - 74år$ efter type og formål. Web. URL http://www.statistikbanken.dk/BEBRIT15.

[15] GPSPrimer.net. How GPS Works FAQ. Web. URL http://www.gpsprimer.net/how-gps-works-faq/freqently-asked-questions-about-how-gps-works/#axzz3EP9JxegH.

[16] US English dictionary. Internet of things. Web. URL http://www.oxforddictionaries.com/definition/english/Internet-of-things.

[17] Marcelo Ballve. The 6 Basic Building Blocks Of The Things In The 'Internet Of Things'. Web. URL http://www.businessinsider.com/defining-the-the-internet-of-things-2013-12.

[18] Holler & Tsiatsis & Mulligan & Avesand & Karnouskos & Boyle. From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence , 1st Edition. Academic Press, 10 Apr 2014.

[19] http://postscapes.com. An Internet of Things. Web. URL http://postscapes.com/internet-of-things-examples/.

[20] Smart Grid. What is the Smart Grid? Web. URL https://www.smartgrid.gov/the_smart_grid.

[21] Gartner. Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. Web. URL http://www.gartner.com/newsroom/id/2636073.

[22] ABIResearch. More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020. Web. URL https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne0-billion-devices-will-wirelessly-conne.

[23] ARIN American Registry for Internet Numbers. IPv4 and IPv6. Web, . URL https://www.nro.net/wp-content/uploads/2011/02/ipv4_ipv6.pdf.

[24] Kevin Bonsor and Wesley Fenlon. How RFID Works. Web. URL http://electronics.howstuffworks.com/gadgets/high-tech-gadgets/rfid.htm.

[25] H. Dagan, A Shapira, A Teman, A Mordakhay, S. Jameson, E. Pikhay, V. Dayan, Y. Roizin, E. Socher, and A Fish. A Low-Power Low-Cost 24 GHz RFID Tag With a C-Flash Based Embedded Memory. Solid-State Circuits, IEEE Journal of, 49(9):1942–1957, Sept 2014. ISSN 0018-9200. doi: 10.1109/JSSC.2014.2323352.

[26] ARIN American Registry for Internet Numbers. Location matters: Spatial standards for the Internet of Things. Web, . URL http://www.itu.int/dms_pub/itu-t/oth/23/01/T23010000210001PDFE.pdf.

[27] Stephen E. Deering and Robert M. Hinden. Internet Protocol, Version 6 (IPv6) Specification, month = December, 1998. URL http://tools.ietf.org/html/rfc2460.

[28] Internet Movie Database. Web, . URL www.imdb.com.

[29] w3schools. Soap introduction. URL http://www.w3schools.com/webservices/ws_soap_intro.asp.

[30] John Mueller. Understanding soap and rest basics, January 2013. URL http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/.

[31] Mike Rozlog. Rest and soap: When should i use each (or both)?, April 2010. URL http://www.infoq.com/articles/rest-soap-when-to-use-each.

[32] Chris Panique. Php-mvc. Web. URL https://github.com/panique/php-mvc.

[33] Nusoap - soap toolkit for php, . URL http://sourceforge.net/projects/nusoap/.

[34] Google Inc. Google Maps Javascript API v3. Web. URL https://developers.google.com/maps/documentation/javascript/.

[35] Tcp listener source. Microsoft Developer Network, . URL http://msdn.microsoft.com/en-us/library/system.net.sockets.tcplistener(v=vs.110).aspx.

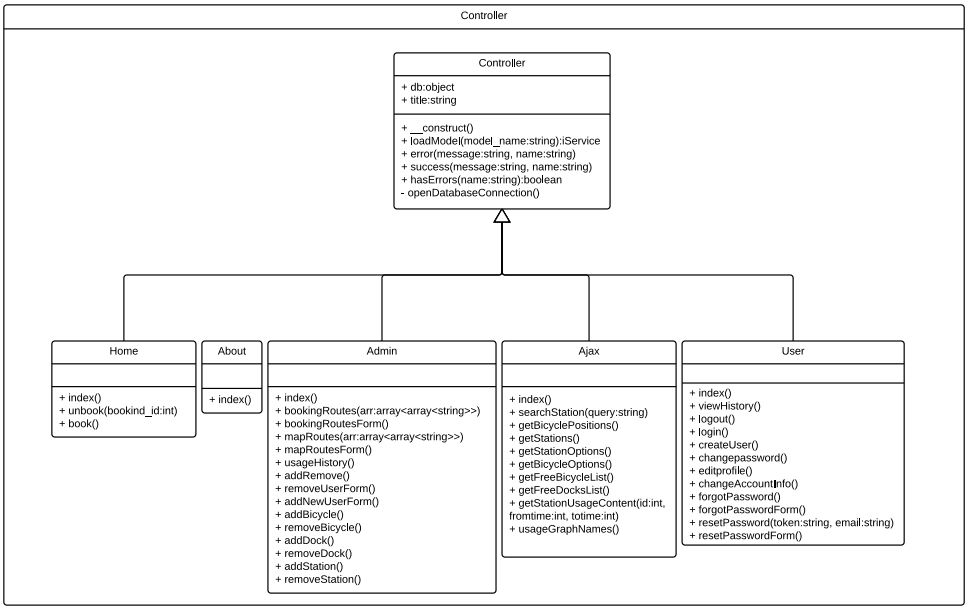
[36] Mikael B. Skov Jesper Kjeldskov and Jan Stage. Instant data analysis: Conducting usability evaluations in a day. URL http://people.cs.aau.dk/~jesper/pdf/conferences/Kjeldskov-C23.pdf.

[37] John Heggestuen. One in every 5 people in the world own a smartphone, one in every 17 own a tablet [chart]. Business Insider, December 2013. URL http://www.businessinsider.com/smartphone-and-tablet-penetration-2013-10.

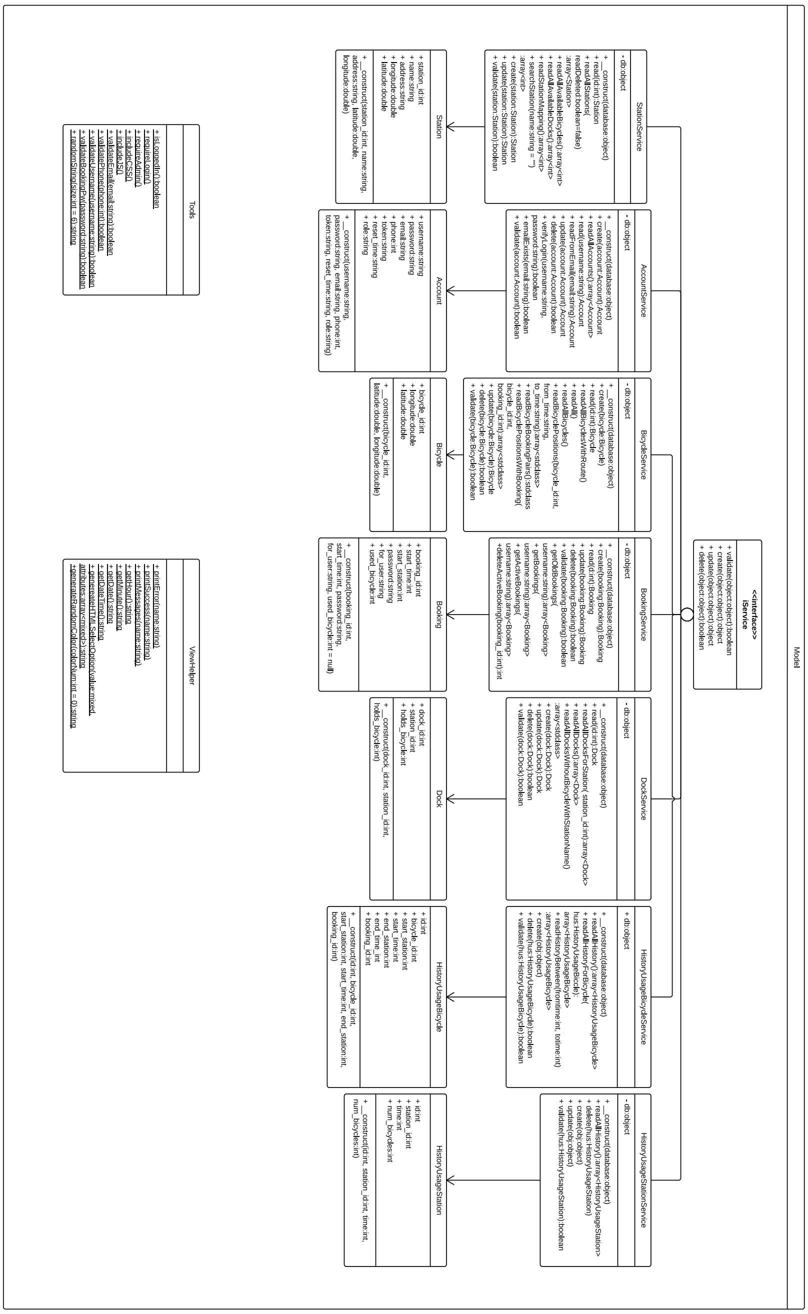
[38] Luke Mahe and Google Geo APIs Team Chris Broadfoot. Too many markers!, December 2010. URL https://developers.google.com/maps/articles/toomanymarkers.

A Website Architecture

A.1 Controllers

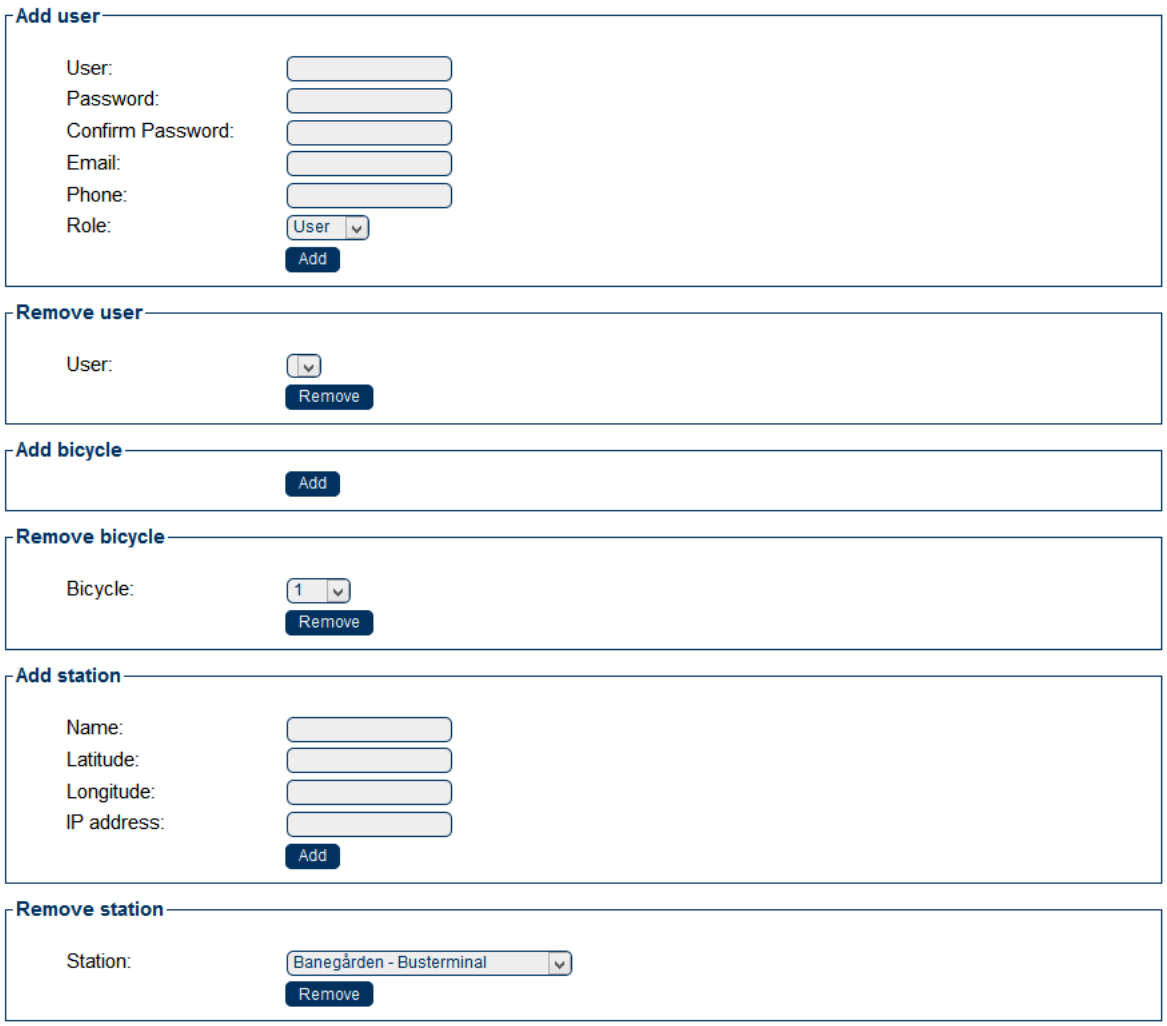


A.2 Model


Model
<<interface>>
IService
StationService
AccountService
BicycleService
BookingService
DockService
HistoryUsageBicycleService
HistoryUsageStationService
Station
Account
Bicycle
Booking
Dock
HistoryUsageBicycle
HistoryUsageStation
Tools
ViewHelper

B Administrator Site

B.1 Add-Remove



C Usability Test

The website you are about to test is a site that enables you to book Aalborg bicycles. You will go through different parts of the website, involving booking and status of Aalborg bicycles.

Establish an overview
Take your time to get an overview of the website.

Status for bicycle
You are considering to borrow a bicycle from Aalborg Banegård, but you are uncertain whether more bicycles are available at the station or not.

Find the amount of available bicycle at the station: "Banegården Busterminal".

Booking - Login
In order to gain access to the booking part of the website, you have to be logged in. Luckily you already have a user with the following login information:

Username: testuser

Adgangskode: testpassword

Use the above information to login.

Booking - Time and booking
Now that you are logged in you are ready to book a bicycle.

Book a bicycle for tomorrow at 11:25 a.m.

Cancel planned booking
You notice there is another booking planned for tomorrow but at 11:45 a.m. at Kunsten. This booking is not necessary and thus you want it cancelled.

Cancel the booking tomorrow 11:45 a.m. at Kunsten.

Examine the booking history
You have now booked a bicycle for tomorrow and cancelled another booking. However, you are interested in getting an overview of all your bookings, those active and previously used ones.

Find and read the booking history for your profile.