

Software Design Description

- [1. Introduction](#)
 - [1.1. Purpose of this document](#)
 - [1.2. Definitions, acronyms, and abbreviations](#)
- [2. Architectural Overview](#)
 - [2.1. Deployment diagrams](#)
 - [2.1.1. Old deployment diagram](#)
 - [2.1.2. Deployment Diagram](#)
 - [2.1.3. Design Decisions related to the deployment diagram](#)
 - [2.2. Back-end package diagram](#)
 - [2.3. Front-end package diagram](#)
- [3. Detailed Design Description](#)
 - [3.1. Design Front-end](#)
 - [3.1.1. Design Class Diagram](#)
 - [3.1.2. Sequence Diagrams](#)
 - [3.1.3. Login User](#)
 - [3.1.4. Is admin check](#)
 - [3.1.5. Code and design patterns](#)
 - [3.1.5.1. GRASP](#)
 - [3.1.5.2. Multi-tier architecture](#)
 - [3.1.5.3. Angular Style Guide](#)
 - [3.1.5.4. SOLID](#)
 - [3.1.6. Design decisions made for the sub-system](#)
 - [3.2. Design Back-end](#)
 - [3.2.1. Design Class Diagram of the domain package](#)
 - [3.2.2. Design Class Diagram of Login](#)
 - [3.2.3. Sequence Diagrams](#)
 - [3.2.3.1. Login](#)
 - [3.2.4. Endpoints](#)
 - [3.2.5. Design decisions made for the back-end](#)
 - [3.3. Database Design](#)
 - [3.3.1. ERD modal \(old\)](#)
 - [3.3.2. ERD modal new](#)
 - [3.3.3. Design decisions related to the database](#)
- [4. List of Resources](#)

1. 1. Introduction

This document details the architectural design. For more information over the scope, see the [Action plan](#) or the [SRS](#).

1.1. 1.1. Purpose of this document

The purpose of this document is to explain the design details of how the EV charging pole works with the Sevcı system in the Microsoft Azure environment and how the Sevcı system is and will be designed. It is aimed to translate software requirements defined in the [SRS](#) document into a representation of software components, interfaces and data to be used later in implementation phase of the project. However, since every software design is open to changes and modifications, it is highly possible to make changes during implementation and update the [SRS](#) and SDD documents accordingly.

1.2. 1.2. Definitions, acronyms, and abbreviations

Term	Description
Sevcı	Sevcı system is the Angular app and the Java REST API
DAO	Data Access Object
DTO	Data Transfer Object
SRS	Software Requirements Specification
SDD	Software Design Description
EV	Electric Vehicle

2. 2. Architectural Overview

In the figures in this chapter you can find the deployment diagrams and the package structure of the system. The deployment diagram paragraph contains two figures. The first figure contains an image of how the deployment looked at the start of the project. The second figure represents how the deployment will look after this project. With these diagrams, the new design choices are documented the diagrams. The package structure is divided into a [front-end](#) and [back-end](#) package structure. The back-end has been divided into three parts, also known as tiers. By doing this, we are making use of the three-tier-pattern. Each tier is responsible for its own thing. The tiers are described above the [back-end package diagram](#).

2.1. 2.1. Deployment diagrams

This chapter contains the deployment diagram of the system. Both old and new are included to give a more clear view on what changed and to help explain why these changes happened. Below these diagrams, a glossary is included to explain each part of the deployment diagram.

2.1.1. 2.1.1. Old deployment diagram

In this figure you can see the old deployment diagram made by the previous OOSE group. The major concerns with this architecture are the communication between the IoT device and the web server, they go over plain text, where it could easily be transformed to something that is more

protocol robust as JSON. Also the particle PHP API that now communicates directly with the database is something that is going to be phased out. This is because we want to create one entry point for the database.

2.1.1.1.1.1. Old Deployment Diagram

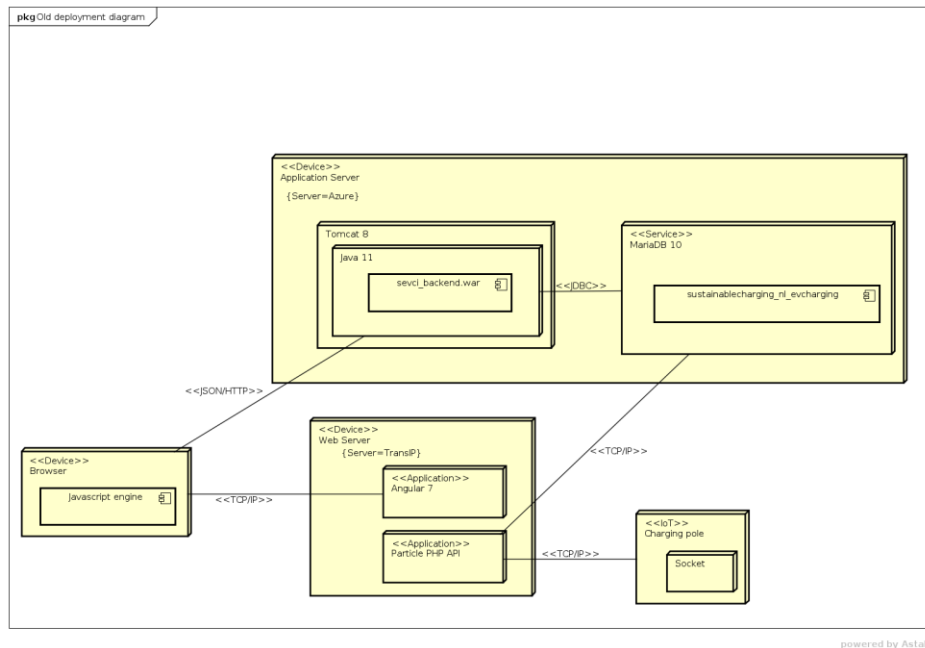


Figure 1: Old Deployment Diagram

2.1.2. 2.1.2. Deployment Diagram

In the figure below you can see the deployment diagram of the system. The IoT device now directly talks to the Java back-end. Another big change is that the Partial PHP API is nowhere to be found in the new diagram. This is because the functionality of this API is moved to the Java REST API. More about these decisions in the next paragraph about [the design decisions related to the deployment diagram](#).

2.1.2.1.1.1. 2.1.2.1.1.1. Deployment Diagram

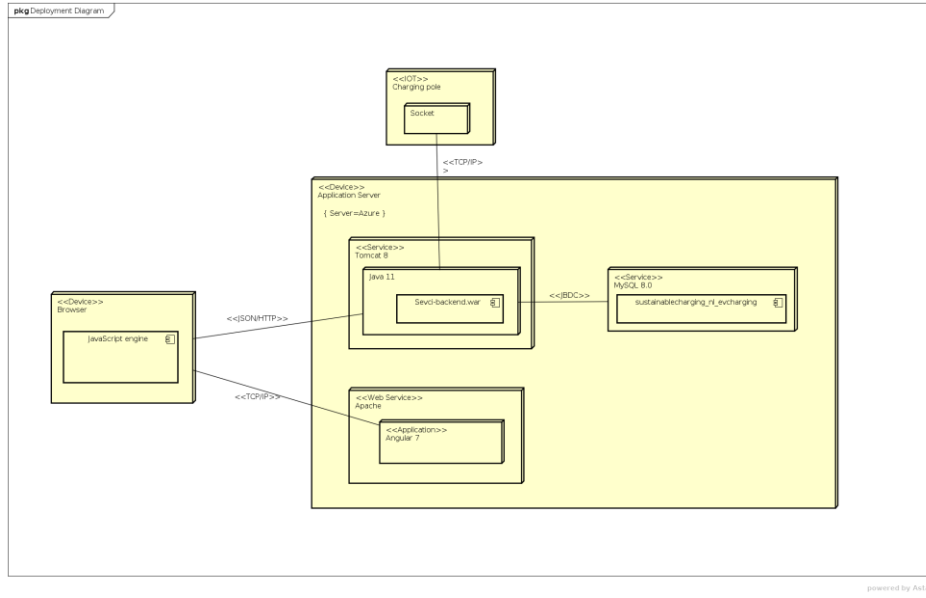


Figure 2: Deployment Diagram

Glossary

Concept	Description
IoT	Internet of Things. Connects the charging pole to the back-end.
Charging Pole	Physical pole that contains two sockets.
Socket	A charging socket is built into a charging pole. Connects to the car.
Application Server	The application runs on Microsoft Azure.
Sevc-Backend.war	The deployed back-end.
sustainablecharging_nl_evcharging	The name of the MySQL database.
Browser	Represent a browser like Firefox or Edge.
Apache	Apache is the web server.
Tomcat	The Java web server.
JSON	JavaScript Object Notation. Uses text to exchange data between the front- and back-end.
JDBC	Java Database Connectivity. Used for the connection between the database and the back-end.

Angular	Angular is a TypeScript-based open-source web application framework.
TCP/IP	Internet Protocol Suite. Provides end-to-end communication. Includes HTTP.

Table 1: Deployment diagram glossary

2.1.3. 2.1.3. Design Decisions related to the deployment diagram

The design decisions that were made while reshaping the architecture of the system, can be found in this paragraph. These also represent the differences that can be found between the deployment diagrams.

Decision	Description
Problem/Issue	The data of the IoT currently goes through a PHP API that connects to the database.
Decision	The team made the decision to by-pass the PHP API and send the information of the IoT directly to the back-end.
Alternatives	Keep the PHP API.
Arguments	<p>It is hard to manage what comes in and what goes out of the database when it is done via several routes. By routing everything via the back-end application there is a total control over the data flow.</p> <p>Another pro is that, when functionality changes or database types change, this only has to be updated in the Java back-end. The PHP API itself is a living example for this argument where it is outdated within half a year.</p>

Table 2: Design decision IoT data flow

Decision	Description
Problem/Issue	What type of database is going to be used?
Decision	It has been decided to continue using MySQL 8.
Alternatives	<p>MariaDB MariaDB supports up to MySQL 5.7, so it isn't compatible with the codebase that was provided.</p> <p>MongoDB Using MongoDB means that the whole database had to be rebuilt and all the functionality that existed had to be refactored.</p>

Decision	Description
Arguments	Since the provided codebase also used MySQL 8 and the learning curve is low.

Table 3: Design decision database type

Decision	Description
Problem/Issue	How will the front-end communicate with the back-end?
Decision	JSON
Alternatives	YAML, XML or CSV
Arguments	<p>JSON is easy to read, use, test, and learn. Also, this was already used in most places of the communication of the already existing codebase.</p> <p>Another argument is that the system communicates via a REST API which is more purpose built for JSON.</p>

Table 4: Design decision communication front- and back-end

Decision	Description
Problem/Issue	Where do we place the Angular application, momentarily it is running separately from the rest.
Decision	Angular is moved to the Azure server instead of running on a separate server.
Alternatives	Keep Angular separately deployed.
Arguments	<p>This decision is made together with the people of the HAN-ICT service. In the past, there was decided to move all parts of the system to a TransIP server. While in the process of moving these parts, another decision was made to move them to Azure. Now parts are scattered across different servers and the HAN-ICT wants to move everything to one place so managing the system will be easier. This is why the choice is made to move the Angular application to Azure as well.</p>

Table 5: Design decision Placement of Angular application

2.2. 2.2. Back-end package diagram

In the figure in this chapter the representation of the back-end structure can be found. This structure is divided into three tiers: The *presentation*, *application* and *data* tier. Design choices regarding the back-end application are included in the design decisions in the chapter [Design Back-end](#).

Presentation tier

The presentation tier is the top level of the API and holds the REST controllers. Within these controllers, the paths for the REST API are set at every class and method accordingly to what they are in the [endpoint table](#). The controllers are responsible for communication with the front-end application, among other things. This is done using Spring boot and by using data transfer objects (DTO's). This layer communicates with the application tier to retrieve or modify data. The presentation layer also holds the CORS package. In this package a filter is included that handles the CORS-headers.

Application tier

This tier is responsible for the logic of the back-end. It basically holds the main functionality of the back-end application. Here the services are located in the service package. The services get the data from the data tier and process them into data transfer objects after doing the required logic. The services communicate with the controllers in the presentation layer through the DTO's. [The decision why this is done can be found here](#). The services communicate with the repositories in the data layer through the domain. This is done for the same reason. The domain package is located in the application tier because the services use the domain objects to execute logic. This tier also holds the database tables representation inside the domain package. This domain package is built using Java persistence API (JPA). JPA is used by Spring Data JPA and makes it easier to retrieve and modify data.

Data tier

Using Spring Data JPA, we can build repositories that hold a couple of basic abstract CRUD functionalities to execute on the database tables. Besides that, it is easy to add a query by adding it to the interface of that specific repository. Using JPA also makes it easier when there is a switch in the database type because queries are not written in the applications source code anymore.

2.2.1.1.1.1. 2.2.1.1.1.1. package diagram back-end

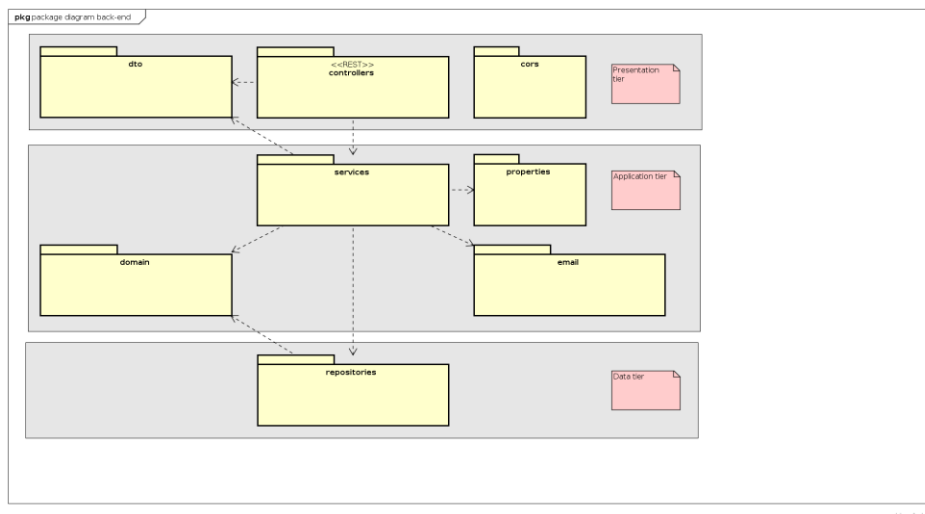


Figure 3: Package diagram back-end

2.3. 2.3. Front-end package diagram

This is a representation of the Front-end structure, this structure is divided into different blocks : Modules, Components, Templates(data binding), Routing and Services. In the following section there is more explanation made for every block.

2.3.1.1.1.1. 2.3.1.1.1.1. package diagram front-end

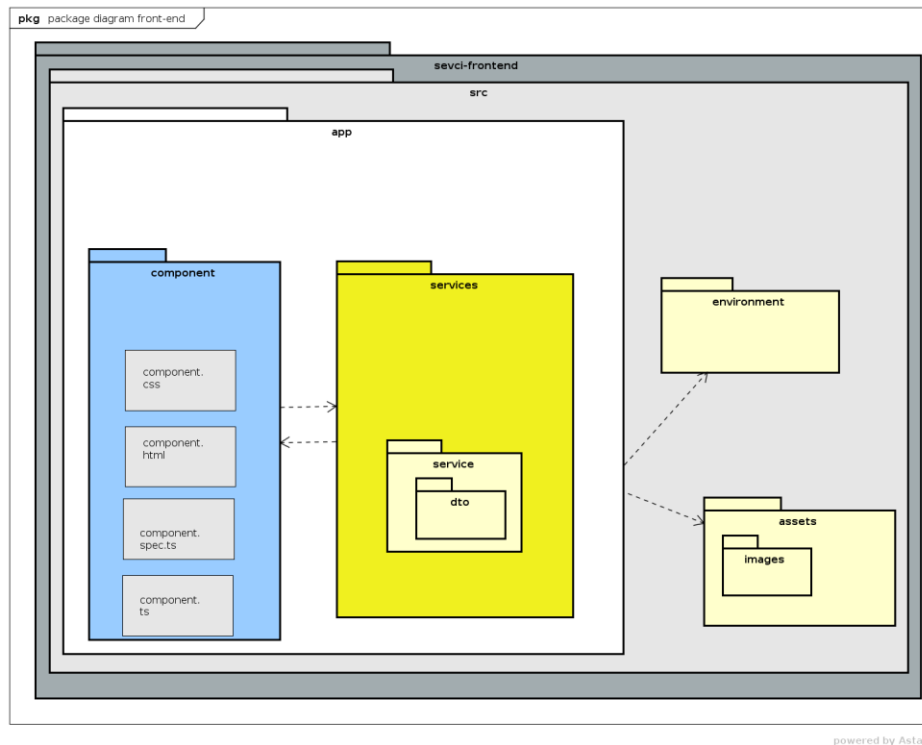


Figure 4: Package diagram front-end

App block

Every Angular app has a *root module*, conventionally named `AppModule`, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

Every Angular application has at least one component. This is the *root component* that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML and CSS *template* that defines a view to be displayed in a target environment.

Services block

A service class definition is immediately preceded by the `@Injectable()` (Angular, 2019a) decorator. The decorator provides the metadata that allows other providers to be injected as dependencies into your class. This is the class where all the calculations and the logic of the front-end application are processed. Each service in this block also contains the files for the DTO used by this service.

Routing

The Angular `Router` (Angular, 2019b) NgModule provides a service that lets you define a navigation path among the different application states and view hierarchies in your app.

Templates(data binding) block

A template combines HTML with Angular markup that can modify HTML elements before they are displayed. Template *directives* provide program logic, and *binding markup* connects your application data and the DOM.

Environments

This folder holds the types of environments. Standard environments are production and development. (production is marked with the ".prod.ts" extension.) Here, multiple parameters can be assigned for different environments.

Assets

Contains images and other asset files to be copied as-is when you build your application.

3. 3. Detailed Design Description

In this chapter you will find three main paragraphs. The first one concerns the design of the front-end application, the second one concerns the back-end application and the last paragraph focusses on the design of the database.

3.1. 3.1. Design Front-end

This chapter describes the front-end application of the system. This application is written in Typescript and is a web application. For the implementation is a class diagram made to show how the web application works. Besides that, useful sequence diagrams are included to show the flow of the system. The last paragraph describes extra design decisions that aren't mentioned in the other paragraphs.

3.1.1. 3.1.1. Design Class Diagram

This class diagram shows the structure of the Angular application. One component has been worked out because of the number of classes and every component

works nearly the same, the only difference is the classes and variables. For more information about what each component does of the following diagram, see the [glossary](#).

Error rendering macro 'astah' : 1

Figure 5: Class diagram front-end homepage.

Glossary

In the following table will be explained what happens in every component.

Concept	Description
Main.ts	The main class is the Angular base class. This class is the application entry point for Angular.
App.components.ts	The app component is the base class for the frontend. This class is the entry point for all the components. Token: The token will be used for verification. (legacy) isAdmin: this string is used to verify if the user is an admin. (legacy)
Approuting.module.ts	The page navigation uses the app-routing module. route: the router object will be used for the page navigation. (legacy)
Homepage.service	The homepage service is used for the homepage logics such as the graph. <ul style="list-style-type: none">• socketRecived: this is unknown• socketRecived\$: this is unknown• httpMsgRecived: this is unknown• HttpMsgRecived\$: this is unknown• pointsRecived: this is unknown• pointsRecived\$: this is unknown (legacy)• Sockets: This is the response of the charging pole sockets (legacy)• httpMsg: This is the HTTP message, it's used for error messages (legacy)
homepage.service.spec	This file contains all the tests for the homepage service.

socket.response.dto	<p>The socket DAO is used as a data model.</p> <ul style="list-style-type: none"> • socketNr: The id of the socket (legacy) • socketsFree: Is the socket free to use (legacy) • carBrand: The brand of the car (legacy) • carModel: The model of the car (legacy) • remainingTimeUntilFree: Time left before the socket will be free to use (legacy) • batteryPercentage: Battery percentage of the car (legacy)
homepage.component.ts	<p>This is the homepage component.</p> <ul style="list-style-type: none"> • token: The token will be used for verification (legacy) • sockets: This is a charging pole (legacy) • lineChartData: This is the chart data for the graph (legacy) • lineChartColors: This is the line color for the chart (legacy) • lineChartLegend: This is the legend for the chart (legacy) • numberOfLabels: We do not know what this is (legacy) • displayTime: We do not know what this is (legacy) • currentTime: We think this is the current time (legacy) • minutesPerStep: We do not know what this is (legacy)
homepage.component.html	This file contains the HTML for the homepage component.
homepage.component.css	This file contains the CSS for the homepage component.
homepage.component.spec.ts	This file contains the HTML for the homepage component.

Table 6: Glossary

3.1.2. 3.1.2. Sequence Diagrams

3.1.3. 3.1.3. Login User

This sequence diagram is a representation for the work flow of the user login process, we tried in this sequence diagram to apply the "Controller pattern" from the GRASP patterns. As you can see that the responsibility is passed to the controller, which is in this case is the 'login.service'. Additionally, it is clear to see that how the flow of the program works:

1. User sends data to the Back-end.
2. Back-end process the data.
3. User gets the data back(in this situation access or no-access).

3.1.3.1.1.1. 3.1.3.1.1.1. Sequence diagrams login

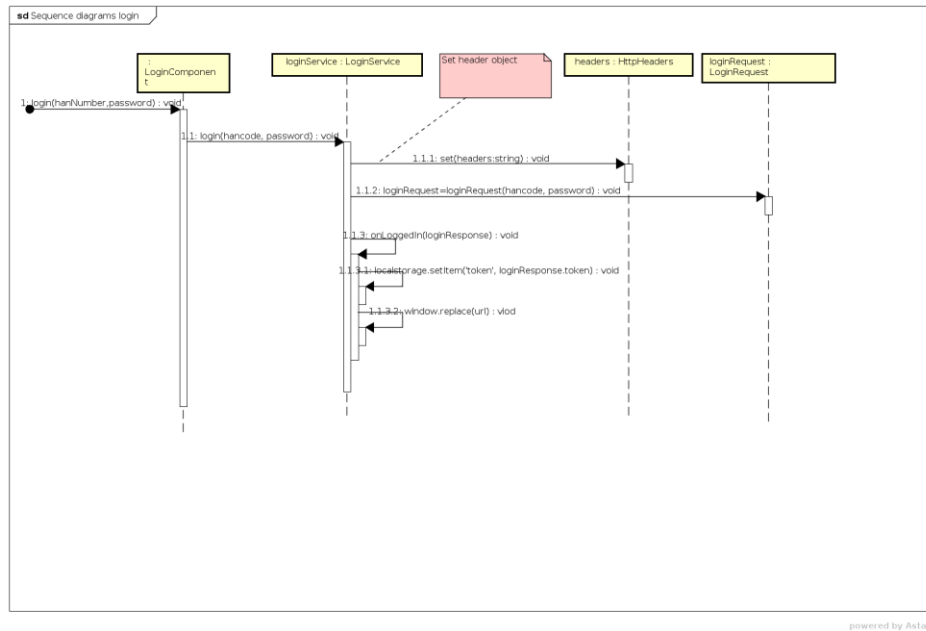


Figure 6: Front-end (login) SD

3.1.4. 3.1.4. Is admin check

This sequence diagram shows the flow for the admin check. The team decided to show a complete flow of the situation for the front-end, in this case, is it the hamburger menu. As you can see that the responsibility is passed to the controller, which is in this case, the 'hamburger.service'. The flow of the program by almost every request is:

1. Build the URL with the token
2. Prepare the request and send it to the server.
3. The server validates if the user is an admin.
4. The system handles the requested data.
5. The system will apply the changes

3.1.4.1.1.1. 3.1.4.1.1.1. isAdminCheck

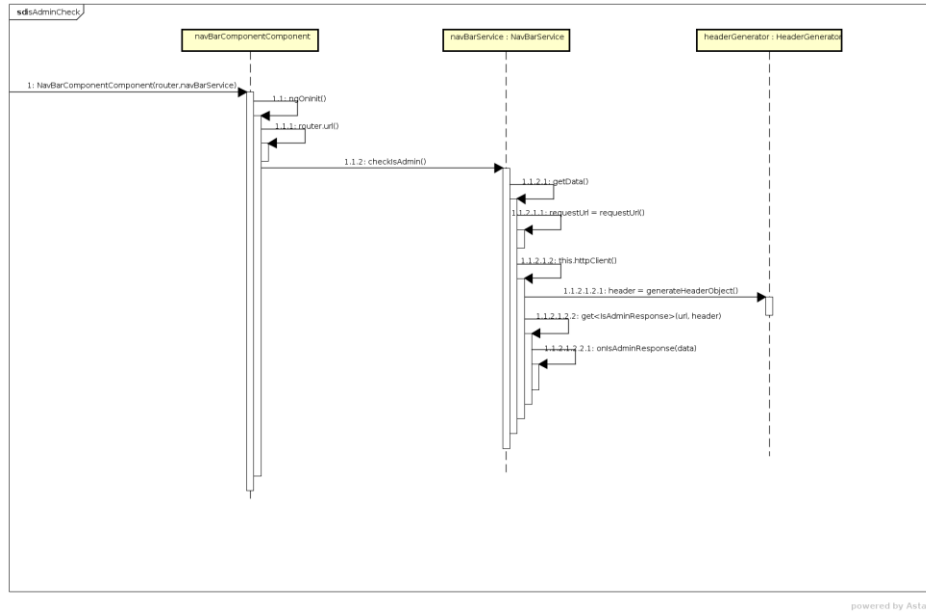


Figure 7: Front-end hamburger menu

3.1.5. 3.1.5. Code and design patterns

Coding patterns are designed for resolving problems. In the front-end, we have the following problems:

- Receiving data from the back-end
- Identify the user
- Verification

By applying the following patterns, it can be assumed that the above-listed problems will be resolved.

- GRASP
- multi-tier architecture
- Angular Style Guide
- SOLID

3.1.5.1. 3.1.5.1. GRASP

The decision to use GRASP is because GRASP contains sub patterns and is used for code structure. The following sub-patterns will be used in the front-end:

- High cohesion
- Information Expert

For easy maintenance, the team uses High cohesion. This has been applied to the HttpHeaders class, this is a separate class with no functionality other than to generate headers. By delegating the code to other classes correctly, the team uses the sub-pattern, Information Expert.

3.1.5.2. 3.1.5.2. Multi-tier architecture

For the structure of the front-end, multi-tier architecture has been applied. These tiers describe the structure:

- View
- Logic
- Dataset

The Angular components use the View tier, this renders the UI. While the Logic tier is used by the Angular services. DTO's make use of the Dataset tier.

3.1.5.3. 3.1.5.3. Angular Style Guide

The Angular Style Guide (ASG) describes the structure of the components and services code. According to the ASG, Angular uses a singleton pattern for rendering the pages. This stops Angular from rendering multiple pages.

3.1.5.4. 3.1.5.4. SOLID

SOLID consists of five different responsibilities. From the five responsibilities, only Single will be used. Single responsibility makes it easier to update the front-end due to the fact that classes only have one functionality (eg. generate a header) so you will only have to update them in one spot. In other words this makes the application easier to maintain and design new features.

3.1.6. 3.1.6. Design decisions made for the sub-system

In this chapter there is a description for all design decisions that were made for the sub-system. Every framework, libraries and other technologies has at least one decision description. Other decisions may be related to software patterns, system-structure, adapted principles or the like.

Decision	Description
Problem/Issue	Building a front-end application with vanilla Javascript costs a lot of time and the code quality will be low because of duplicate code. There are a lot of frameworks for the front-end, every framework has its own solutions for problems.
Decision	Using Angular to reduce duplicate code and increase the productivity of the team with maintainability code.

Decision	Description
Alternatives	<p>ReactJS is a framework created by Facebook. This framework is created for OOP programming, the only disadvantage is that the logic is inside the view.</p> <p>Vanilla JavaScript is plain JavaScript.</p> <p>Lowdash is a commonly used library for JavaScript but not very useful for this project because of the vanilla JavaScript.</p> <p>JQuery is an old framework, that is been used for old web browsers.</p>
Arguments	The previous group used Angular for the front-end, by that is this the fastest way to develop and extend the program with new features. With Angular is it also possible to program OOP with a separate class for the UI and logic.

Table 7: Angular

Decision	Description
Problem/Issue	Building a correct responsive design cost a lot of time and can be very hard. To resolve this problem, a CSS library is the best solution.
Decision	<p>By using a CSS framework increase the productivity of the team and decrease the number of bugs. Most frameworks have a good view on mobile devices.</p> <p>Because the previous team created some CSS, is it to much work to rewrite it, so it is better to go further with the current CSS. By using the current CSS files it reduces a lot of extra code.</p>
Alternatives	MDL, Bone, and Bootstrap are different types of frameworks. The only difference between the frameworks is the design and the class names. The functionality of the frameworks is almost the same.
Arguments	Better design, usability, and less duplicated code. It saves also time by building the design. You only need to assign classes to the HTML tags.

Table 8: Bootstrap

Decision	Description
Problem/Issue	<p>There is a global state in the program with a lot of static strings. The old code uses the local state and depending on the front-end framework for dealing with the data flow.</p> <p>In the chosen framework Angular has no global state to store the values of the variables. When there is no global store application, the data will be lost by switching pages or by refreshing the page.</p>
Decision	<p>There will be no global state in the application, we will use the local and session storage of the web browser. Leaving the page has no effect on the data.</p>
Alternatives	<p>The alternative is to leave it so as it is now. It has no effects on the functionality of the program.</p> <p>Redux is a framework that stores the data into a global store. This storage is only available when the page is not reloaded.</p> <p>Cookies can be used for storing data but they have a maximum body size, which is not ideal.</p>


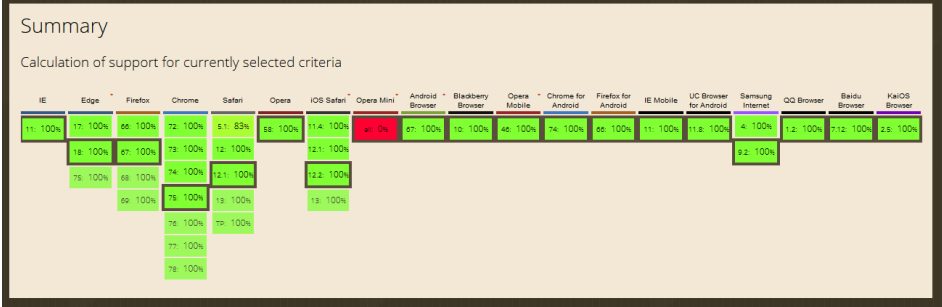
Decision	Description
	<p>Every new webbrowser after 2018 supports the local storage and session storage.</p>  <p>Arguments</p> <p>The overall support of all webbrowsers</p>  <p>This functionality is used in a lot of mobile application because of the supportability.</p> <p>This functionality works natively.</p>

Table 9: Redux

Decision	Description
Problem/Issue	A framework is needed to help to mock and test the functions of the application.
Decision	The decision has been made to use Jasmine for the unit test. Jasmine is a very popular framework for unit testing JavaScript applications. It provides utilities that can be used to run automated tests for both synchronous and asynchronous code.

Decision	Description
Alternatives	Other alternatives are: Unit.j / JEST / MochaJS. The alternatives are almost the same, the biggest differences are the syntax of the tests. Some other frameworks show a nice output in the IDE or command line what can be used for the test report.
Arguments	The argument to use Jasmine is because this framework provides a clean and easy to understand syntax while also having a rich and straightforward API. The previous team used this framework. This framework is easy to learn. This framework is advised by Angular.

Table 10: Jasmine

Decision	Description
Problem/Issue	Classes need to know/be notified when a objects state has been changed.
Decision	The decision has been made to make use of the observer pattern. The observer is a design pattern where an object (known as a subject) maintains a list of an object depending on it (observers). This will automatically notify the observers when the subject's state changes.
Alternatives	Websockets are a solution to the problem. The biggest problem with that is the server resource usage. Long polling is another solution to solve this problem. The problem by long polling is that this is an async solution, which means that it can take a second to a minute before every page is updated or the data is received.
Arguments	The observer pattern used a list of classes, which needs to be updated locally where no internet connection is needed and no server is needed.

Table 11: Observer pattern

3.2. 3.2. Design Back-end

This chapter describes the back-end application of the system. This application is written in Java and consists of an REST API. For the implemented domain layer, a class diagram is made to show what the application works with. Besides that, useful sequence diagrams are included to

show the flow of the system. The next paragraph shows all the endpoints of the application that can be called. The last paragraph describes extra design decisions that aren't mentioned in the other paragraphs.

3.2.1. 3.2.1. Design Class Diagram of the domain package

This class diagram shows the domain package. We have chosen to include this class diagram because it represents the domain we work with in this project. It gives a clear picture of where the logic of the application works with. The domain package represents the database tables as JPA entities. These entities are immutable and all variables are private. Therefore, each private variable has his own getter and setter. In the diagram, getters and setters are not included to improve readability. Below the diagram, a glossary is included to explain each class on its own. The multiplicity between the identities and the class holding that identity is always one-to-one. This is because the identity holds the primary key of the class holding it. The identity on his turn always contains one instance of each class that this identity holds.

3.2.1.1.1.1. 3.2.1.1.1.1. Domain Class Diagram

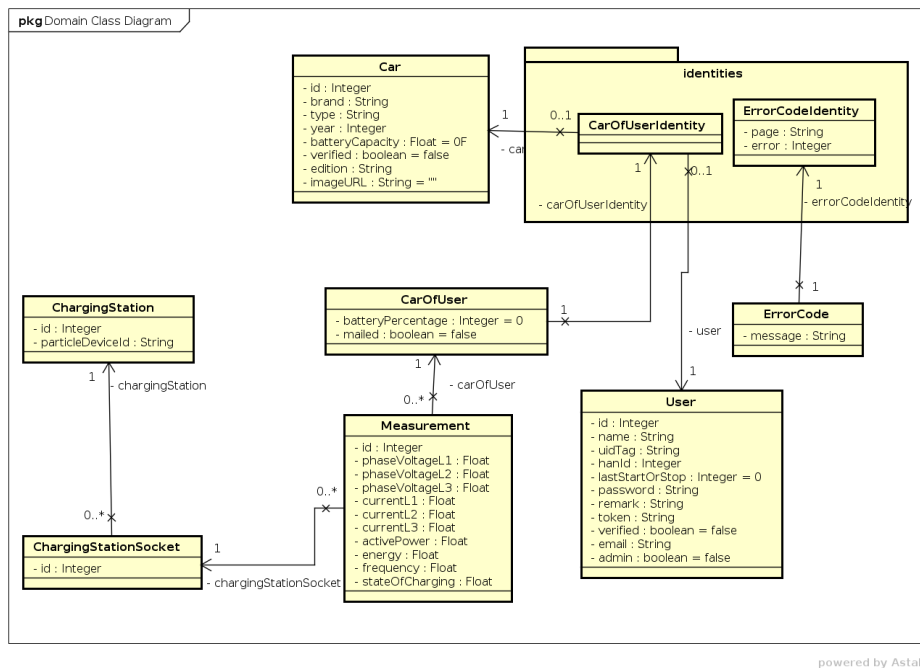


Figure 8: Domain Layer Class Diagram

Glossary

Concept	Description
---------	-------------

Car	id : Identify the car by a number brand : The brand of the car (e.g. Tesla) type : The type of the car (e.g. Model S) year : The build year of the car (e.g. 2016) batteryCapacity : The amount of power a battery can store in kWh (e.g. 85) verified : The car has to be verified by an admin before an user can add it edition : The car is a special edition. (e.g. Cabrio, 75D...) imageURL : URL to an image of the car.
User	id : Identification of the user by a number name : Name of the user uidTag : uidTag is the tag that the NFC chip of the HAN card outputs hanId : The han number of the user lastStartOrStop : When the user charged last or started charging password : Encrypted password of the user remark : Notes on the user token : Unique identification token for the user verified : The user has to be verified by the admin email : Email-address of the user admin : Admin status of the user
CarOfUser	batteryPercentage : The current percentage of battery that the car has when it gets connected, this is used in calculations mailed : The user gets mailed after the car is almost full
Measurement	id : A number to identify the measurement phaseVoltageL1 : Amount of power this phase delivers phaseVoltageL2 : Amount of power this phase delivers phaseVoltageL3 : Amount of power this phase delivers currentL1 : The current that flows through the charger currentL2 : The current that flows through the charger currentL3 : The current that flows through the charger activePower : The amount of power currently added energy : Energy is the total power that has been distributed in a chosen timeframe (Total energy that went through the power meter) frequency : Frequentie is always 50hz, we need to check if it is exactly 50hz all the time (Can fluctuate between 49,9 and 50,1). stateOfCharging : The battery state, when empty this state is 0 and when the battery is full this states 1. createdAt : The date and time of the measurement
ChargingStationSocket	id : A number to identify the socket
ChargingStation	id : A number to identify the ChargingStation particleDeviceId : .

ErrorCode	message: The error message
ErrorCodeIdentity	page: The page the error occurs on error: The error number
CarOfUserIdentity	Does not hold any extra variables other than a User and a Car . This class exists because the domain represents tables from the database

Table 12: Glossary Domain Class Diagram

3.2.2. 3.2.2. Design Class Diagram of Login

This class diagram shows the domain package. The team chose to include this class diagram because it gives an basic idea of the flow of this project. It gives a clear picture about the structure of the application. This class diagram shows the logic of the login functionality. In this class diagram a decision is made to leave the getters and setters because of the readability and to keep every variable private for each class, because of the variable control in the code. Below the diagram is a glossary that included explanation to each variable. The multiplicity between the identities and this class holding that identity is always one-to-one.

3.2.2.1.1.1. 3.2.2.1.1.1. Back-end login

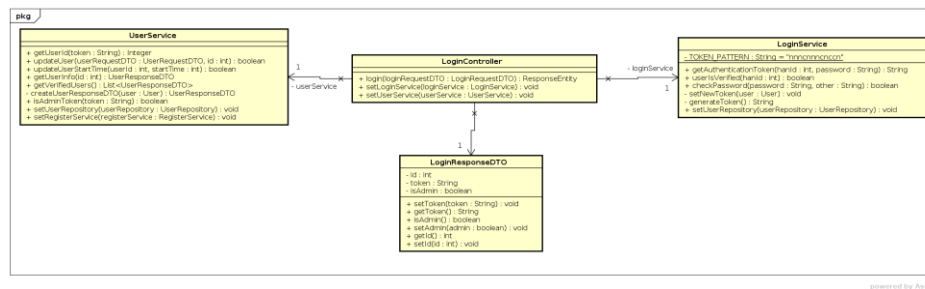


Figure 9: Login class diagram

Concept	Description
UserService	This class has no private or public variable.
LogController	This class has no private or public variable.
LoginService	TOKEN_PATTERN: this pattren will be used for generating the token.
LoginResponseDTO	ID: The id user ID token: The token of the user isAdmin: is the user is a admin or not.

Figure 10: table glossery login

3.2.3. 3.2.3. Sequence Diagrams

This chapter contains sequence diagrams for the more complex flows of the back-end application.

3.2.3.1. 3.2.3.1. Login

The following sequence diagram contains the flow for when a user wants to login. When a request comes in it must contain a LoginRequestDTO, this contains a hanId and password. The request will be handled by the LoginController class. The LoginController will now get an authentication token by using the hanId and password to check if the user has the right combination of id and password for their account. Once the LoginController has acquired a token from the LoginService class, which makes use of the UserRepository and private functions to generate a random String for a token, the class can now get a userId. With the userId, it will check if it is not null, this means that a user has been found. The LoginController will check if the user is verified and return a hanId. After that the LoginController will fill a LoginResponseDTO with the token, userId and hanId.

The wrong credentials will return a HttpStatus.UNAUTHORIZED or a HttpStatus.FORBIDDEN if the user is unverified.

3.2.3.1.1. 3.2.3.1.1. Login

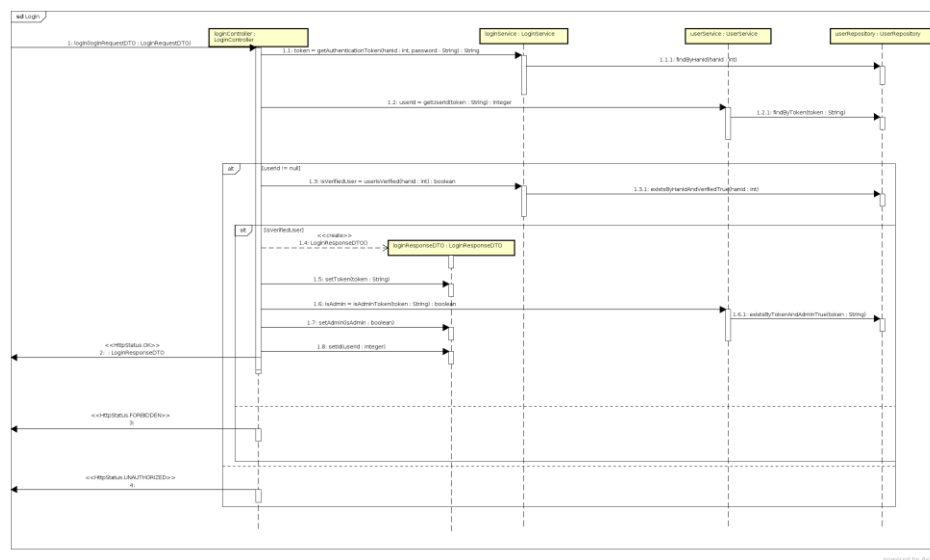


Figure 11: Sequence diagram - Login

3.2.4. 3.2.4. Endpoints

In this chapter, all endpoints of the back-end are included to give a clear picture of every call. The endpoints are grouped per subject, so calls about users are included at "Users" and calls about verifying cars are included at "Verify cars". The consumes column contains the body that is requested by the endpoint. The request parameter holds the query string parameters, it mostly includes the token the requests needs to verify the user. The request method tells you what type of request we are talking about. Produces holds the response of the back-end API. Also, the code location is included so it can easily can be found in the codebase. Most groups contain only one controller.

Endpoint	Consumes	Request parameter	Request method	
/cars	-	Query string: <ul style="list-style-type: none"> token={user_token} 	GET	On HT app ["ht /09] If u HT

/cars	<p>application/json:</p> <pre>{ "id": 184, "brand": "Peugeot", "type": "E-Legend", "buildYear": 2019, "batteryCapacity": 120, "verified": true, "edition": "Turbo" }</pre>	<p>Query string:</p> <ul style="list-style-type: none">token={user_token}	POST	<p>On</p> <p>HT</p> <p>On</p> <p>HT</p> <p>If u</p> <p>HT</p>
/cars/my-cars	-	<p>Query string:</p> <ul style="list-style-type: none">token={user_token}	GET	<p>On</p> <p>HT</p> <p>app</p> <p>[</p> <p>"ht</p> <p>/09</p> <p>]</p> <p>If u</p> <p>HT</p>
/cars/my-cars	<p>application/json:</p> <pre>{ "carId": 27 }</pre>	<p>Query string:</p> <ul style="list-style-type: none">token={user_token}	POST	<p>On</p> <p>HT</p> <p>On</p> <p>HT</p> <p>If u</p> <p>HT</p>

<p>/cars/my-cars/{carId}</p>	<p>-</p>	<p>Query string:</p> <ul style="list-style-type: none">token={user_token}	<p>GET</p>	<p>On HT app { w w w w w w w w w w 20 w 3. w 06- w "ht /09 } On HT If u HT</p>
------------------------------	----------	---	------------	---

/cars/my-cars/{carId}	-	<div>Query string:</div> <ul style="list-style-type: none">token={user_token}	DELETE	<div>On</div> <div>HT</div> <div>On</div> <div>HT</div> <div>If u</div> <div>HT</div>
/cars/non-verified-cars	-	<div>Querystring</div> <ul style="list-style-type: none">token={user_token}	GET	<div>On</div> <div>HT</div> <div>app</div> <div>[</div> <div>.</div> <div>]</div> <div>If f</div> <div>HT</div> <div>If u</div> <div>HT</div>

/cars/verify/{carId}	-	Querystring <ul style="list-style-type: none">token={user_token}	PUT	On HT On HT If f HT If u HT
/cars/verify/{carId}	-	Querystring <ul style="list-style-type: none">token={user_token}	DELETE	On HT On HT If f HT If u HT
U				
Endpoint	Consumes	Request parameter	Request method	

/users/isAdmin	-	<div>Querystring</div> <ul style="list-style-type: none">token={user_token}	GET	<div>On</div> <div>HT</div> <div>app</div> <div>{</div> <div>,</div> <div>}</div> <div>If u</div> <div>HT</div>
/users/{id}	-	<div>Querystring</div> <ul style="list-style-type: none">token={user_token}	GET	<div>On</div> <div>HT</div> <div>app</div> <div>{</div> <div>,</div> <div>,</div> <div>Be</div> <div>,</div> <div>,</div> <div>15</div> <div>}</div> <div>If r</div> <div>HT</div> <div>If u</div> <div>HT</div>

/users/{id}	<p>application/json:</p> <pre>{ "firstName": "Gerrit", "insertion": "van", "lastName": "Beek", "uidTag": "F2 34 3B 87", "hanId": 564231, "lastStartOrStop": 1543454387, "password": "New Secret password", "confirmPassword": "New Secret password", "currentPassword": "Secret password", "email": "G.Beek@han.nl", "description": "This is a admin" }</pre>	<p>Querystring</p> <ul style="list-style-type: none">token={user_token}	PUT	<p>On</p> <p>HT</p> <p>If r</p> <p>HT</p> <p>If u</p> <p>HT</p>
users/to-verify	-	<p>Querystring</p> <ul style="list-style-type: none">token={user_token}	GET	<p>On</p> <p>HT</p> <p>app</p> <pre>{ " " Beek " "G " " "a " 155 }</pre> <p>If f</p> <p>HT</p> <p>If u</p> <p>HT</p>

users/verify/{id}	-	Querystring <ul style="list-style-type: none">token={user_token}	PUT	On HT If r HT If r HT If u HT
users/verify/{id}	-	Querystring <ul style="list-style-type: none">token={user_token}	DELETE	On HT If r HT If r HT If u HT
Endpoint	Consumes	Request parameter	Request method	

Endpoint	Consumes	Request parameter	Request method	
/measurements	application/json: { "phaseVoltageL1": 300, "phaseVoltageL2": 0, "phaseVoltageL3": 0, "currentL1": 15, "currentL2": 0, "currentL3": 0, "power": 150, "energy": 20, "frequency": 50.01, "timestamp": 15623674238, "userId": 123, (optional) "socketId": 2 (optional) }	-	POST	On HT tex Flo stat em On HT
G				
Endpoint	Consumes	Request parameter	Request method	

/graphs/sockets/{id}	-	-	GET	<div>On</div> <div>HT</div> <div>app</div> <div>{</div> <div>,</div> <div>200</div> <div>,</div> <div>50,</div> <div>400</div> <div>,</div> <div>0.2</div> <div>0.4</div> <div>,</div> <div>0,</div> <div>200</div> <div>,</div> <div>0,</div> <div>50,</div> <div>0,</div> <div>0,</div> <div>}</div> <div>If r</div> <div>HT</div>
Sc				
Endpoint	Consumes	Request parameter	Request method	

/sockets/status	-	-	GET	HT app ["v "re "02]]
Http error				
Endpoint	Consumes	Request parameter	Request method	
/http-messages/{page}/code/{code}	application/json: { "page": "admin" "httpCode": 403 }	-	GET	On HT tex E.g If r HT
I				
Endpoint	Consumes	Request parameter	Request method	

/login	application/json: <pre>{ "hancode": 4578, "password": "Psrt213" }</pre>	-	PUT	On HT app { " " "&D } If f HT If u HT
--------	---	---	-----	--

Re				
Endpoint	Consumes	Request parameter	Request method	
/register	application/json: <pre>{ "firstName": "Gerrit", "insertion": "van", "lastName": "Beek", "uidTag": "F2 34 3B 87", "hanId": 564231, "lastStartOrStop": 1543454387, "password": "New Secret password", "confirmPassword": "New Secret password", "currentPassword": "Secret password", "email": "G.Beek@han.nl", "description": "This is a admin" }</pre>	-	POST	On HT On HT On HT

Battery				
Endpoint	Consumes	Request parameter	Request method	Pro

/battery-fill-in/{carId}	<pre> application/json: { "batteryPercentage": 58 } </pre>	Query string: <ul style="list-style-type: none"> token={user_token} 	PUT	On HT On HT If u HT
--------------------------	--	--	-----	------------------------------------

Table 13: Endpoints

3.2.5. 3.2.5. Design decisions made for the back-end

In this paragraph, each design decision made for the back-end is included. These decisions are noted down to explain why the system is build how it is.

Continuation Spring Boot	Description
Problem/Issue	At the start of the project, the application ran on Spring Boot. The question with this is: Can this framework continue being used for further development or is changing to another framework that satisfies the needs of the project better?
Decision	The decision is to continue using Spring Boot as a framework.
Alternatives	None, see argumentation.

Continuation Spring Boot	Description
Arguments	<p>The current controllers are built on Spring Boot and the Services get injected using the @autowired annotation of the framework.</p> <p>Using the autowired annotations makes dependencies easier to maintain. This page tells the basic usage of @autowired for dependency injection.</p> <p>Besides the @autowired annotation, the @RestController and @RequestMapping annotations are used for defining the endpoints.</p> <p>This way, methods and classes can easily be mapped for the REST API. Simple user example:</p> <pre> @RestController @RequestMapping("users") class UserController { @GetMapping("/{id}") public ResponseEntity getUser(@PathVariable("id") int id) { ... } @PostMapping("/{id}") public ResponseEntity updateUser(@RequestBody requestDTO) { ... } } </pre> <p>Here, a simple class is shown where a GET and a POST call on a user is done. More on this in the Spring Boot documentation.</p> <p>There is a possibility to change to other frameworks outside Java but then the whole application would have to be rebuilt. This cannot be done in the given time limit of the project.</p> <p>Same goes for frameworks using Java. By using another framework, too much work is needed to be done to fit inside a project scope of 10 weeks.</p> <p>Using the existing Spring Boot, less work needs to be done refactoring the application.</p>

Table 14: Design decision - Continuation Spring Boot

Continuation Mockito	Description
Problem/Issue	The current application uses Mockito, will the project continue with this way of testing?
Decision	The decision has been made to continue using Mockito as a testing framework.
Alternatives	<p>Spring Test</p> <p>Spring Test is a pretty solid way of testing. The controllers require a "real" request so there for only part of this testing framework will be used. The reason behind this is that is can be implemented in many ways and by the time the testing phase starts there will be a lot of unknown functionalities that could have been useful. Researching a testing framework non-stop will not help progress in this project. The only part that is necessary to know about is the MockMvc which can be learnt about in this MockMvc tutorial (Bodnar, 2019).</p>
Arguments	<p>Spring has excellent integration with Mockito. For the controllers, a MockMvc can be built to do requests to the controller. This way, a "real" request is done instead of a method call. SpringBoot tutorial ("Unit Testing Rest Services with Spring Boot and JUnit," 2017) has a complete tour on how to fully use this functionality.</p> <p>Another argument is that Mockito is taught in the OOSE course and this is an OOSE project. This helps in making tests quicker and easier to review.</p>

Table 15: Design decision - Continuation Mockito

Use of Java Persistence API (JPA)	Description
Problem/Issue	At the beginning of the project, DAO classes were implemented to gather data into Map's and DTO's using queries. This makes it hard to change database types or alter tables.
Decision	The decision is to start using JPA (Millington, 2018) as database data representation in the back-end application.

Use of Java Persistence API (JPA)	Description
Alternatives	<p>Keep using the raw queries and update them when needed</p> <p>As stated, this is part of the problem. Because it is not something that blocks development, it is included as an alternative; keep it as it is. The biggest issue with this is that changing database types causes queries to break. Also, the way it is done now, every time you want to do a call to the database, you need to open, fetch and close the connection. Something a framework can do.</p>
Arguments	<p>First of all is the data persistence. By using JPA, the domain represents the database tables from the database. Instead of doing the mapping yourself all the time, JPA can do this for you. This means more time to concentrate on other development tasks. JPA does the type mapping from database to Java application, otherwise this would have to be done by hand.</p> <p>Second of all, using JPA makes the application database independent. By updating the "<i>Application.properties</i>" file, it is possible to switch easily between database types. This makes it easier with switching between how the application will start; either validate, update or drop-create your database. This means, in a way, the system wouldn't even have to look at the database to use it when setting it to update or drop-create. When setting this property to drop-create, every time the server restarts the database will be deleted (dropped) and created again based on the JPA classes. This also means all data will be lost, so this is a property to use with caution! Update will update the database scheme if possible and spring boot will crash when it can't update. Validate checks if the JPA classes provide a valid representation of the database tables.</p> <p>Another upside is caching of objects. When doing raw calls, data will not get cached, which means that for every relation, you either need to include it in your query (which leaves you with a blob of a query) or you need to do multiple calls. JPA caches the object and optionally 'lazy fetches' the relations.</p> <p>Also, by using JPA there are more options within Spring to further abstract database calls so implementing it won't be necessary to use raw queries.</p>

Table 16: Design decision - Use of JPA

Introduction layer pattern	Description
Problem/Issue	At the start of the project, the structure of the project is kind of messy. In this unstructured shape, development is hard because of all the dependencies going across all packages.
Decision	The decision will be to introduce a layer pattern to make a more clear design and lift dependencies.
Alternatives	<p>Microservices</p> <p>This would be ideal if it were possible to deploy each service separately.</p> <p>The big knock-out here is that one domain is used across many services. This goes in into what microservices are ideal for; services with a small clear bound context with little dependencies. More on microservices, their pros and why they don't work for us: microservices ("Benefits of Microservices - Choreography over Orchestration, Low Coupling and High Cohesion," n.d.).</p> <p>Service based application</p> <p>Another approach would be to let the controller handle most of the tasks. This would then use services to execute logic and nothing else. The main disadvantage of using services this way is that the controller would have to many responsibilities. The appliciation at the start of this project slightly represents this pattern and shows how many dependencies go across the application making the application quite hard to manage.</p>
Arguments	<p>To make this application maintainable for further development, implementing a layer pattern will help achieve this. For a more clear view, more information will be given about this in the package diagram where the tiers are explained.</p> <p>Using the layer pattern, it is easy to distribute responsibilities. Every layer will have it's own kind of responsibility and every package within this layer it's own kind of tasks, like the controllers that receive requests and sends out responses.</p> <p>The layer pattern is something that has been learned in the OOSE semester. Every OOSE group who comes across this project will understand how this project is structured.</p>

Table 17: Design decision - Introduction layer pattern

Use of Spring Data JPA	Description
Problem/Issue	Since it has been decided to use JPA, there still has a decision to be made about how to connect to the database.
Decision	The decision is to use Spring Data JPA (Gierke, Darimont, Strobl, Paluch, & Bryant, 2019) for connecting to the database.
Alternatives	<p>Use plain DAO's with entity managers</p> <p>Using plain DAO's with an entity manager takes longer to develop. On the other hand, it also gives you more control over how to save data and the possibility to add logic to the data layer. Sarcasm is not easily expressed in text, but adding logic to the data layer is not meant as an advantage. What the advantage is of entity managers is the easy use of "flush" and "persist". Standard CrudRepository of Spring Data JPA can't do this, but the JpaRepository class that extends the CrudRepository can preform these functions. Also, entity managers have to be maintained more because they can throw an exception and close their connection. Because of this, the entity needs to be reset and therefore needs to be checked every time for an active connection.</p>

Arguments	<p>Using repositories increases testability because you can use the pre-defined methods when writing tests before even programming the actual methods that call the repository.</p> <p>It also makes the data management even more generic. The only thing that needs to be done is to establish a connection between the database and a service is creating a interface repository. It takes a JPA object and his primary key as Java Generic. It then can do all the basic CRUD operations without adding anything. Also, making more complex calls can be created easily by adding the method to the interface and your done. It not only makes the data layer more abstract, it speeds up development even more. For example:</p> <p>Repository with custom method</p> <pre>public interface UserRepository extends JpaRepository<User, Integer> { List<User> findAllByLastNameOrderByAgeDesc (String lastName); }</pre> <p>Retrieving a User by his primary key and updating his age</p> <pre>@Autowired UserRepository userRepository; int id = 123; User user = userRepository.getOne(id); user.age = 25; userRepository.save(user);</pre> <p>Retrieving all users filtered by last name and ordered by age</p> <pre>@Autowired UserRepository userRepository; List<User> users = userRepository.findAllByLastNameOrderByAgeDesc ("Beek");</pre> <p>The only downside of these repositories is that if you only want to use the abstract way of retrieving data like shown above, you can get very long method names. To counter this, you could make use of the <code>findByExample`</code> method. This method takes an entity and returns all entites that meet the same values as the enity provides to the method. (See example below). More on this: Spring data query example (Baeldung, 2019).</p> <pre>@Autowired UserRepository userRepository;</pre>
-----------	---

Use of Spring Data JPA	Description
	<pre>User user = new User(); user.setLastname("Beek"); Example<User> example = Example.of(user); List<User> users = userRepository.findAll(example);</pre>

Table 18: Design decision - Use of Spring Data JPA

Communication between services and controllers	Description
Problem/Issue	The DTO's are removed from the data layer. Still, there needs to be some kind of communication between the services and the controllers. Some calls require a lot of data sent between them. The question is, how this will be achieved?
Decision	<p>The current DTO's will be used to communicate between the services and the controllers. DTO's holding HTTP status codes will get deprecated to meet the responsibility requirement.</p> <p>Sometimes, Map can still be found, this is legacy code that still needs to be refactored.</p>

Communication between services and controllers	Description
Alternatives	<p>Use a Map to map all return parameters</p> <p>By using a Map, multiple values could easily be returned from a service to a controller.</p> <p>The disadvantage of this is that the service must first map all values and the controller must go through all the values and also officially check whether they are filled. These would then have to be put in a DTO again. This causes duplicate work.</p> <p>Build additional DTO's special for the communication between the services and the controllers.</p> <p>By using additional DTOs, you establish a clear communication between the controller and the service.</p> <p>The downside is the same as the one when using a Map. Values still need to be remapped to the DTO's that will be send.</p> <p>Also, more kind of DTO's means more dependencies and also you would want to store them seperately from the other DTO's. This makes the structure of the application harder to get a GRASP on and harder to maintain.</p>
Arguments	<p>Just as with the additional DTO's, you can establish clear communication between the controllers and the services by using the existing DTO's.</p> <p>The already existing DTO's need to be built up eventually. By doing this in the service, logic can be executed and then set immediately into the DTO. The big advantage here is that you don't have to build up the values to return twice.</p> <p>Some of the existing DTO's hold HTTP status codes. This would be a good argument for why we would not use the existing DTO's. To counter this argument, these DTO's will get deprecated and replaced by DTO's that don't hold any presentation "logic".</p>

Table 19: Design decision - DTO in services

Email design patterns	Description
Problem/Issue	Momentarily, the email package uses two design patterns that cross each other. These are the "strategy design pattern" and the "template method design pattern". There needs to be figured out what will be used and what not.
Decision	The strategy design pattern will be used to build the email package.

Email design patterns	Description
Alternatives	<p>The template method pattern</p> <p>In the template method pattern, a skeleton of logic and steps are defined. These steps can be distributed to sub-classes, so that the output changes. Here is an example of the template class Worker and his sub-classes like Postman.</p> <pre> classDiagram class Worker { +DailyRoutine() +getUp() +eatBreakfast() +goToWork() +work() +returnToHome() +relax() +sleep() } class FireFighter { +work() } class Lumberjack { +work() } class Postman { +work() } class Manager { +work() +relax() } Worker < -- FireFighter Worker < -- Lumberjack Worker < -- Postman Worker < -- Manager </pre> <p>Figure 12: Example Template Method</p> <p>By using this, all logic that now is present in every mail class, can be put into the template class. The only thing that needs to be inside the subclasses of the mail is the actual mail itself.</p> <p>The reason for looking into this was that every class that implements the message interface has his own constructor and sets the name and email in it. Also, the <i>UserVerifiedOrDeclinedMessage</i> holds a boolean for whether the user is accepted or not and the <i>RegistrationConfirmedMessage</i> holds a whole <i>userRequestDTO</i> (SourceMaking.com, 2019a)(SourceMaking.com, 2019b).</p>

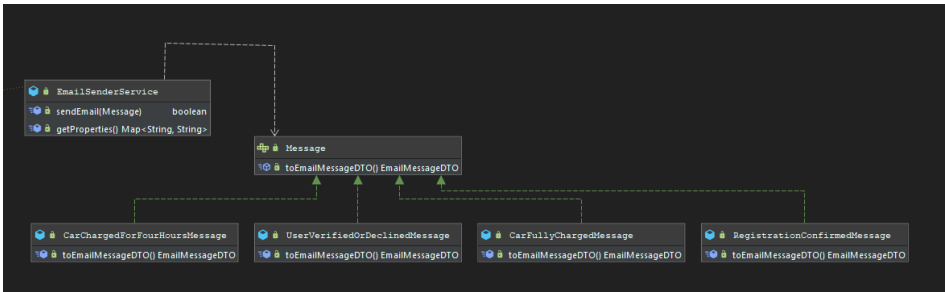
Email design patterns	Description
Arguments	<p>The strategy pattern</p> <p>At the start of the project, the email package has more of a strategy pattern. This is because it uses an interface that holds the method to build emails.</p> <p>Also the <i>EmailSenderService</i> has a method that takes a <i>Message</i> called <i>sendEmail</i>. This method uses the message to send the actual mail. So far, the use of the strategy pattern is good.</p> <p>The main advantage of using this pattern is there only has to be one method to send every type of mail. A visual representation of how this looks in the code is presented below.</p>  <pre> classDiagram class EmailSenderService { +boolean sendEmail(Message) +Map<String, String> getProperties() } class Message { +EmailMessageDTO toEmailMessageDTO() } class CarChargedForFourHoursMessage { +EmailMessageDTO toEmailMessageDTO() } class UserVerifiedOrDeclinedMessage { +EmailMessageDTO toEmailMessageDTO() } class CarFullyChargedMessage { +EmailMessageDTO toEmailMessageDTO() } class RegistrationConfirmedMessage { +EmailMessageDTO toEmailMessageDTO() } EmailSenderService --> Message Message < -- CarChargedForFourHoursMessage Message < -- UserVerifiedOrDeclinedMessage Message < -- CarFullyChargedMessage Message < -- RegistrationConfirmedMessage </pre> <p>Figure 13: Strategy pattern of email package</p> <p>As shown in the above figure, there are four classes that extend the <i>Message</i> interface. The class that calls the <i>EmailSenderService</i> is the context that determines the strategy.</p>

Table 20: Design decision - Email design patterns

BCrypt	
Problem/Issue	The password of an user has to be hashed before it's saved to the database.
Decision	BCrypt
Alternatives	<p>MD5: Not as secure as BCrypt because it generates weaker hashes.</p> <p>SHA: Faster to decode because it is built-in in java, so requests are faster.</p>

Arguments	<p>Java does not have built-in support for BCrypt, thus making a potential attack slower.</p> <p>Example of an implementation of BCrypt.:</p> <p>BCrypt example</p> <pre> public class BcryptHashingExample { public static void main(String[] args) throws NoSuchAlgorithmException { String originalPassword = "password"; String generatedSecuredPasswordHash = BCrypt.hashpw(originalPassword, BCrypt.gensalt(12)); System.out.println(generatedSecuredPasswordHash); boolean matched = BCrypt.checkpw(originalPassword, generatedSecuredPasswordHash); System.out.println(matched); } } </pre> <p>Output:</p> <pre> \$2a\$12\$WXItscQ/FDbLKU4m058jxu3Tx/mueaS8En3M6QOVZIZLaGdWrS.pK true </pre> <p>As seen above, BCrypt can use Salt to improve hashing. It is also very easy to read and implement.</p>
-----------	--

Table 21: Design Decision BCrypt

Request mapping	
Problem/Issue	The question is when to use which request method when it comes to functions and consistency.

Decision	The decision has been made to use the request methods as the defining factor in what a controller will do.		
	See the following table:		
	Request method	Function ends with	Path example
	GET	Returns data	/my-cars/
	POST	Creates a row in the database	/my-cars/
	PUT	Updates a row in the database	/my-cars/
Alternatives	DELETE	Deletes a row in the database	/my-cars/
	Focusing on the URL endpoints to define what the function will do		
	Request method	Function ends with	Path example
	GET	Returns data	/my-cars/
	POST	Creates a row in the database	/my-cars/add
	POST	Updates a row in the database	/my-cars/update
Arguments	POST	Deletes a row in the database	/my-cars/delete
	According to the HTTP/1.1 Method Definition (“HTTP/1.1: Method Definitions,” n.d.), it is true that POST is the most flexible of the four request methods. However, there are a lot trade offs to consider such as it is not cache-able, it isn't safe nor idempotent. Through this way you can also apply the right logic to each request.		

Table 22: Request mapping

3.3. 3.3. Database Design

[The decision](#) has been made to use the existing database. The database in it's old status is shown in the next paragraph. This database can be found in the Microsoft Azure environment and concerns a MariaDB. The reasoning for choosing to continue development on the current database is because it is already connected, so it only has to be further development. Also the current team has knowledge of MySQL. The new, improved structure of the database is shown in the second paragraph. The columns in the tables are explained in a table, both old and new, with their change and it is changed.

3.3.1. 3.3.1. ERD modal (old)

The figure below will show the structure of the database in the start of the project. This figure is included to later on explain the differences and design decisions made concerning the database. As shown below, only a link between cars and users is made through the car_of_user table and between the charging_station_sockets and charging_stations. In the next paragraph, besides the new ERD modal, all fields in the database are explained.

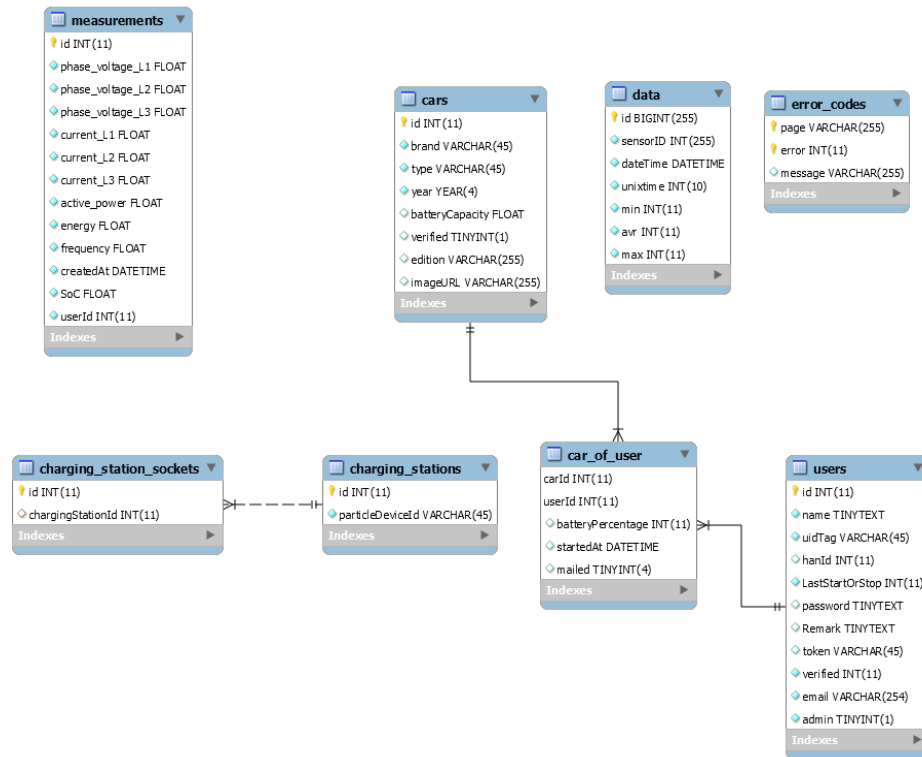


Figure 14: ERD modal old

3.3.2. 3.3.2. ERD modal new

Below, a newly shaped database can be found. As one can clearly see, more relations have been added to the database. The lack of relationships was a major issue when working with the data in the Java back-end application. Data could be added to measurements without a valid user that was currently charging at the charging socket. It could even happen that measurements were included that didn't have a valid socket. Besides relations, also a couple of fields have been altered. These changes can be found in the next last paragraph of this chapter that include the design changes.

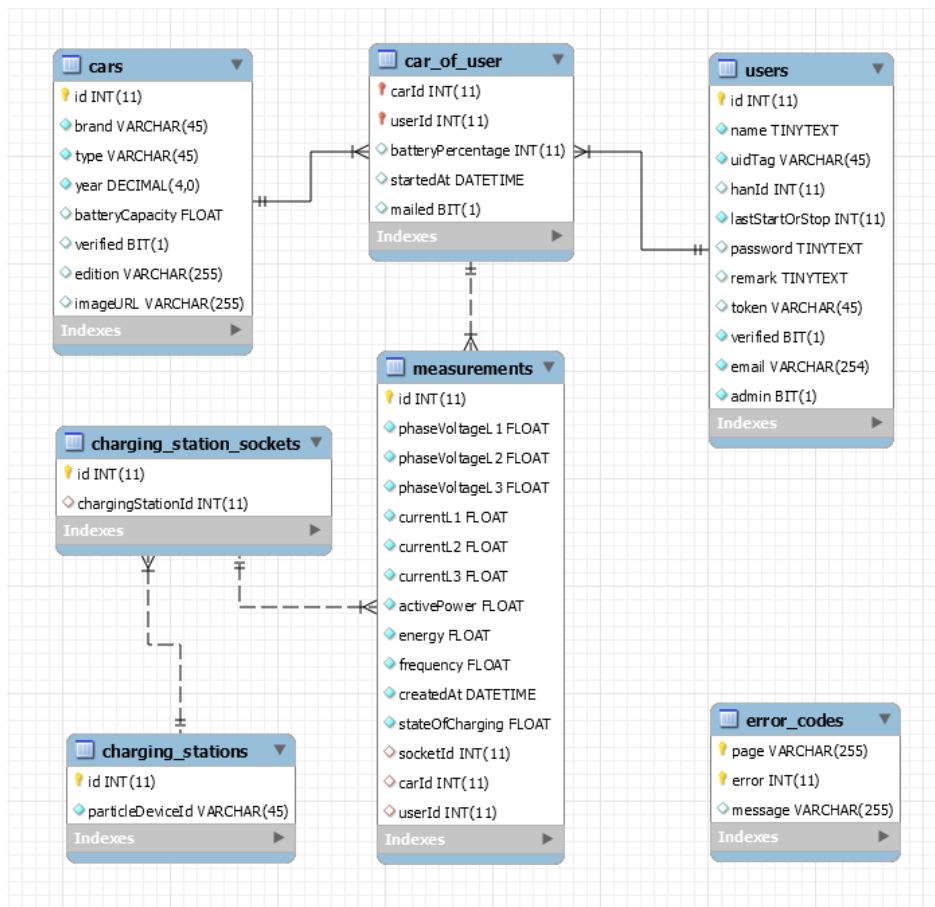


Figure 15: ERD modal new

In the table below, one can find all the tables of the figure shown above. In this table, each column is described, some more obvious than others. There is a column included named "Change". In this column, the change is included between the old and the new database structure. Most of these changes also include a link to the decision related to the change.

Table: cars		
Column	Definition	Change
id	Primary key of the the cars table	
brand	Brand of the car (e.g. Mercedes, Nissan, etc.)	
type	Brand of the car (e.g. A1, Golf, etc.)	
year	Build year of the car	The type of this column is changed from YEAR to DECIMAL

batteryCapacity	Contains the maximum battery capacity of the car	
verified	Cars need to be verified by an admin. This field tells us if it is already verified or not.	The type of this column is changed from TINYINT(1) to BIT
edition	The edition of the car	
imageUrl	When a car gets verified, this field gets filled with an image drawn from google search.	
Table: Users		
Column	Definition	Change
id	Primary key of the users table.	
name	The name of the user.	
uidTag	Each HANcard has an UID tag. This value is stored in here to check when a user wants to start charging.	
hanId	The hanId of the user.	
lastStartOrStop	When the user last started, or stopped charging	Changed LastStartOrStop to lastStartOrStop. Columns should be defined uniform.
password	The password of the user.	
remark	The admin can set a remark on the user	Changed Remark to remark. Columns should be defined uniform.
token	Current authentication token of the user, resets at login.	
verified	Whether the user is verified by an admin or not. After registration the user can be verified.	The type of this column is changed from INT(11) to BIT
email	The email-address of the user.	
admin	Whether the user is a admin or not	The type of this column is changed from TINYINT(1) to BIT
Table: car_of_user		
Column	Definition	Change

carId	Foreign key to table cars.	
userId	Foreign key to table users.	
batteryPercentage	The current battery percentage of the car. Holds a value between 0 and 100	The type of this column is changed from INT(11) to TINYINT(1)
startedAt	When the user did started charging for the last time.	
mailed	A user gets mailed when his car is near done charging.	The type of this column is changed from TINYINT(1) to BIT
Table: error_codes		
Column	Definition	Change
page	On what page the message must be shown. (Primary key)	
error	With what error code the message must be shown. (Primay key)	
message	The message to send back, based on the page and error code (primary key)	
Table: charging_stations		
Column	Definition	Change
id	Primary key of the charging stations table	
particleDeviceId	Holds a reference code to the IoT device on the charging station	
Table: charging_station_sockets		
Column	Definition	Change
id	Primary key of the charging station sockets table	
chargingStationId	Foreign key to the charging station this socket belongs to	
Table: measurements		
Column	Definition	Change
id	Primary key of the measurements table	

phaseVoltageL1	Holds the voltage of the first phase in the three phase power cyclus.	Changed phase_voltage_L1 to phaseVoltageL1. Columns should be defined uniform.
phaseVoltageL2	Holds the voltage of the second phase in the three phase power cyclus.	Same for 2
phaseVoltageL3	Holds the voltage of the third phase in the three phase power cyclus.	Same for 3
currentL1	Holds the current of the first phase in the three phase power cyclus.	Changed current_L1 to currentL1. Columns should be defined uniform.
currentL2	Holds the current of the second phase in the three phase power cyclus.	Same for 2
currentL3	Holds the current of the third phase in the three phase power cyclus.	Same for 3
activePower	The amount of power that currently has been added.	Changed active_power to activePower. Columns should be defined uniform.
energy	Total amount of power delivered since charging/counting	
frequency	Usaly is 50Hz. If not, something must be wrong. It can at most swing to 49,9 or 50,1.	
createdAt	When this measurement was created	
stateOfCharging	The battery state, when empty this state is 0 and when the battery is full this states 1.	Changed it from SoC, because SoC does not show what this column means.
socketId	Foreign key to charging_station_sockets, the socket where this measurements is done.	
carId	Foreign key to car_of_user, holds the car that is currently charging at the socket where this measurement is done.	
userId	Foreign key to car_of_user, holds the user of the car that is being charged.	

Table 23: Database tables

3.3.3.3. Design decisions related to the database

Below, the design decisions are documented made for the database. This includes the reason for the database type and its structure. Furthermore, changes in the database are included such as field type changes.

Database choice	
Problem/Issue	The choice must be made to whether continue using the existing database or to build a new one.
Decision	The choice is made to continue using the existing database.
Alternatives	A new database in MySQL will be build. A new database in another type of database will be build.
Arguments	Since the team got an existing database that works on MySQL 8, the team chose to continue on this platform.

Table 24: Design decision - database choice

YEAR field gives issues on the Java back-end	
Problem/Issue	Database used YEAR for the build year of the car, Hibernate has trouble handling this and in the Java back-end there needs to be a workaround for this issue.
Decision	The choice is made to use DECIMAL(4, 0) instead of YEAR(4)
Alternatives	YEAR could be used. it would be useful if we had a YEAR(2) field. Than a lookup could be done using the year 2010 and the return would be 10, or when inserting 2015, the field would store 15. For YEAR(4), very little arguments can be made other then that it looks easy to read trough the semantic form of YEAR . VARCHAR(4) would also be an option, but this would mean the year could also store the value "ABCD", with DECIMAL we force it to be digits.
Arguments	For YEAR to work, the Java back-end needs a workaround for this field. This YEAR field is not easy to manage. Also, the difference between YEAR(4) and DECIMAL(4, 0) is nil. Both just stores a 4 digit representation of a year.

Table 25: Design decision - year field change

TINYINT for boolean fields does not work properly	
Problem/Issue	The database contains a couple of fields that hold a boolean value (whether is is, or is not). These values are currently TINYINT's and we can't do proper queries in hibernate on whether they are true or false because of this.

Decision	Boolean fields will be changed from TINYINT(m) to BIT.
Alternatives	Use TINYINT(1). When it comes down on this, it does the same, only TINYINT(1) cannot be used that easily by Hibernate.
Arguments	<p>We want to formulate abstract queries in the repositories. Here we can add a rule to only include values true or false.</p> <p>Also, it makes more sense indicating a true or false value with a boolean instead of a integer that holds 1 or 0.</p> <p>Stating</p> <pre>if (verified == 1) // dostuff</pre> <p>Makes less sense then</p> <pre>if (verified) // dostuff</pre>

Table 26: Design decision - boolean field choice

Foreign keys	
Problem/Issue	The database currently contains no hard constraints, only a couple of index keys. This makes it hard to not get corrupted data and keep it trustable.
Decision	Foreign keys will be added between the tables where needed. (Foreign keys are shown in the figure ERD modal (new)).
Alternatives	Keep it as it is, although this isn't really an alternative because of the arguments.
Arguments	<p>When keeping the keys the way they are there surely will be corrupted data among the datasets. It has been proven already in the measurement table where no user or socket id where set. When trying to fetch the data, this causes errors because sometimes, 0 was stored as userId and the system could not find that user.</p> <p>The team works with MySQL, a relational database. A big pro of this are relationship keys between tables. Let's use them when they are available. They are making work easier.</p>

Table 27: Design decision - foreign keys

4. 4. List of Resources

Angular. (2019a). Angular. Retrieved June 12, 2019, from <https://angular.io/api/core/Injectable>

Angular. (2019b). Angular. Retrieved June 12, 2019, from <https://angular.io/guide/router>

Bodnar, J. (2019, January 4). Spring MockMvc tutorial - testing Spring MVC application with MockMvc. Retrieved June 12, 2019, from <http://zetcode.com/spring/mockmvc/>

Unit Testing Rest Services with Spring Boot and JUnit. (2017, January 28). Retrieved June 12, 2019, from <https://www.springboottutorial.com/unit-testing-for-spring-boot-rest-services>

Millington, S. (2018, December 23). The Difference Between JPA, Hibernate and EclipseLink. Retrieved June 12, 2019, from <https://www.baeldung.com/jpa-hibernate-difference>

Benefits of Microservices - Choreography over Orchestration, Low Coupling and High Cohesion. (n.d.). Retrieved June 12, 2019, from <https://specify.io/concepts/microservices>

Gierke, O., Darimont, T., Strobl, C., Paluch, M., & Bryant, J. (2019, May 13). Spring Data JPA - Reference Documentation. Retrieved June 12, 2019, from <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

Baeldung. (2019, January 29). Spring Data JPA Query by Example. Retrieved June 12, 2019, from <https://www.baeldung.com/spring-data-query-by-example>

[SourceMaking.com](https://sourcemaking.com). (2019a). Design Patterns and Refactoring. Retrieved June 12, 2019, from https://sourcemaking.com/design_patterns/template_method

[SourceMaking.com](https://sourcemaking.com). (2019b). Design Patterns and Refactoring. Retrieved June 12, 2019, from https://sourcemaking.com/design_patterns/strategy

HTTP/1.1: Method Definitions. (n.d.). Retrieved June 12, 2019, from <https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>