Smartphone application in PhoneGap

M2C's electric vehicle smart charger

M2C's smart charger för elbilar

Daniel Bönström
Abstract

The purpose of this project is to create a smartphone application in PhoneGap that can be used in combination with an electric vehicle smart charger. The application is to be multiplatform, be able to interact with the charger and display information to the user.

The application in this project has been created on request from the OSNA group at NUI Galway and M2C Smart Charge in Dublin.

The tools used in the application are HTML, JavaScript and css. As a compliment to these tools are Ajax, Json and JQuery. The server side of the system is written in PHP and a MySql database is used as a mockup for the charger and car.

The application handles the user inputs and the view. The server processes the inputs and returns a result. The database contains the information about the cars battery and the chargers status. When the application sends a request to the server it is handled by the controller. The controller decides where in the business model to process the information. If the request requires information from the Database the model will handle the connection. When the model has created a result it will be returned to the controller which returns a response to the Application. The Application will then display the Response in an appropriate way to the user.
# Table of Contents

1. Introduction .................................................................................................................. 1  
   1.1. Dissertation Layout ................................................................................................. 3  
2. Background ..................................................................................................................... 4  
   2.1. Open Sensor Network Authentication group (OSNA) .............................................. 4  
   2.2. MC2 ......................................................................................................................... 4  
   2.3. The need for a smart charger .................................................................................... 4  
   2.4. M2C Smart Charger ............................................................................................... 7  
   2.5. The application ...................................................................................................... 8  
   2.6. PhoneGap ................................................................................................................ 11  
3. Project Design and Implementation .............................................................................. 13  
   3.1. Model View Controller (MVC) ............................................................................... 13  
   3.2. Design ..................................................................................................................... 14  
      3.2.1 Application screen layout ................................................................................... 15  
      3.2.2 Application start screen ...................................................................................... 16  
      3.2.3 Charging Status design ..................................................................................... 17  
      3.2.4 Charging Statistics design .................................................................................. 18  
      3.2.5 Setting the charging type ................................................................................... 19  
      3.2.6 Setting a manual charge ................................................................................... 20  
      3.2.7 Setting a timed charge ..................................................................................... 22  
      3.2.8 Functionality ..................................................................................................... 24  
   3.3. Implementation tools ............................................................................................... 24  
      3.3.1 Jquery .................................................................................................................. 24  
      3.3.2 Ajax (Asynchronous JavaScript and XML) .......................................................... 25  
      3.3.3 JSON .................................................................................................................. 25  
   3.4. Application Code ..................................................................................................... 27  
      3.4.1 Application File Structure .................................................................................. 27  
      3.4.2 Application flow ................................................................................................. 28
3.4.3 Application Initiation ................................................................. 29
3.4.4 Information display to the user ................................................... 31
3.4.5 Setting a manual charge .................................................................. 33
3.4.6 Setting a timed charge .................................................................... 35
3.4.7 ESBN Charge ................................................................................. 36
3.4.8 User inputs ..................................................................................... 36
3.5 Server Code ....................................................................................... 39
  3.5.1 Controller ....................................................................................... 39
  3.5.2 Status of the charger and the car ................................................... 40
  3.5.3 Response ....................................................................................... 40
  3.5.4 Set a manual charge ...................................................................... 42
  3.5.5 Set a timed charge ......................................................................... 43
  3.5.6 Database interface .......................................................................... 44
  3.5.7 Conversions ................................................................................... 46
  3.5.8 Get an estimated price ................................................................... 47
3.6 Database ............................................................................................... 50
  3.6.1 Car mockup ................................................................................... 50
  3.6.2 Charger mockup ........................................................................... 50
  3.6.3 Tariff for the electricity price ......................................................... 51
4. Results/Evaluation .................................................................................. 52
  4.1 Project Goals ..................................................................................... 52
  4.2 Project Results ................................................................................... 52
    4.2.1 User Feedback ............................................................................ 54
5. Conclusion ............................................................................................... 55
  5.1 Views and esthetics ............................................................................ 55
  5.2 Error messages .................................................................................. 55
  5.3 Real-time system ............................................................................... 55
  5.4 Optimizisation ................................................................................. 56
5.5. Final Comment ................................................................. 56
References ........................................................................ 57
Appendix A: Project specification....................................... 59
Appendix B: Setting up the work environment ..................... 63
Appendix C: Activity Diagram ............................................ 67
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>System Overview</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Mobile Device Screen Example</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>M2C Smart Charger</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Connection scheme between ESBN and M2C Smart Charger</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Application interface mockup</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Price of electricity from SEMO for the period 2014-04-01 to 2014-04-01</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>PhoneGap interaction with a mobile device’s operating system</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>PhoneGap Build</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>Model View Controller illustration</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td>Design used for the whole system</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>Application screen break down</td>
<td>15</td>
</tr>
<tr>
<td>12</td>
<td>Start screen</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>The design of the charging status</td>
<td>17</td>
</tr>
<tr>
<td>14</td>
<td>Charging Statistics</td>
<td>18</td>
</tr>
<tr>
<td>15</td>
<td>Selecting a charging type</td>
<td>19</td>
</tr>
<tr>
<td>16</td>
<td>Selecting the manual charge</td>
<td>20</td>
</tr>
<tr>
<td>17</td>
<td>Setting a manual charge</td>
<td>21</td>
</tr>
<tr>
<td>18</td>
<td>Selecting the timed charge</td>
<td>22</td>
</tr>
<tr>
<td>19</td>
<td>Setting a timed charge</td>
<td>23</td>
</tr>
<tr>
<td>20</td>
<td>Jquery code example</td>
<td>24</td>
</tr>
<tr>
<td>21</td>
<td>Using Ajax</td>
<td>25</td>
</tr>
<tr>
<td>22</td>
<td>Json example to encode, decode and read the object</td>
<td>26</td>
</tr>
<tr>
<td>23</td>
<td>Application file overview</td>
<td>27</td>
</tr>
<tr>
<td>24</td>
<td>Application flow diagram</td>
<td>28</td>
</tr>
<tr>
<td>25</td>
<td>Index.html</td>
<td>29</td>
</tr>
<tr>
<td>26</td>
<td>The JavaScript for index.html</td>
<td>30</td>
</tr>
<tr>
<td>27</td>
<td>Charger status html code</td>
<td>31</td>
</tr>
<tr>
<td>28</td>
<td>Charger status JavaScript code</td>
<td>32</td>
</tr>
<tr>
<td>29</td>
<td>JavaScript to set a manual charge</td>
<td>34</td>
</tr>
<tr>
<td>30</td>
<td>Set the timed charge</td>
<td>35</td>
</tr>
<tr>
<td>31</td>
<td>Start screen buttons</td>
<td>36</td>
</tr>
</tbody>
</table>
Figure 32 JavaScript for buttons ................................................................. 37
Figure 33 The css associated with the statistics label .................................. 38
Figure 34 Application Interaction: Time display, slider and button .................. 38
Figure 35 The controller on the server side .................................................. 39
Figure 36 Get charging status php code ....................................................... 40
Figure 37 The answer class used by the response function ............................ 40
Figure 38 Creating a response .................................................................... 41
Figure 39 Php code for setting a manual charge .......................................... 42
Figure 40 Php code for setting a timed charge ............................................. 43
Figure 41 Php code for interactions with the MySQL database .................... 44
Figure 42 Inserting data to the database ..................................................... 45
Figure 43 Some of the conversion functions used by the server ....................... 46
Figure 44 Algorithm to get the estimated price .......................................... 47
Figure 45 Accessing the database ............................................................... 49
Figure 46 Database car table ...................................................................... 50
Figure 47 Database charger table ............................................................... 51
Figure 48 Database tariff table .................................................................... 51
Figure 49 Comparison between the mockup and the implementation ............ 53
1. Introduction

The purpose of this project is to create a smartphone application in PhoneGap [11] that can be used in combination with an electric vehicle smart charger. The system design can be seen in Figure 1.

![Figure 1 System Overview](image)

PhoneGap is implemented using HTML [21], CSS [23] and JavaScript [22]. The specification concerning the application is to be multiplatform, be able to interact with the charger and display information to the user. To interact with the charger and create information concerning the charger, the application connects to a PHP-server. See Appendix A for the complete specification.
Figure 2 shows the start screen of a mobile device for the finished system. It shows at the top the logo of the company that is creating the Smart Charger [25]. Below the logo the information display shows that the charger is charging. The charging mode is set to manual, the current battery charge is 21% and the vehicle connected to the charger is a Volvo with the registration number as shown. Below the information display are the possible inputs available to the user.

![Figure 2 Mobile Device Screen Example](image.png)
1.1. **Dissertation Layout**

In chapter 2 this report will cover what a smart charger is and why there is a need for it and also give a brief explanation of PhoneGap. Chapter 3 will cover the tools used in this project, the design and the implementation of the system. The tools used are Ajax [16], Json [17] and Jquery [15]. The design, which is the layout of the screen, will give the reader a comprehension of how the application looks and acts without having to use the application. The implementation will cover in detail how the system works and is mainly aimed towards readers with a basic understanding of HTML [21] (hypertext markup language), JavaScript [22], CSS [23] (cascading style sheet) and PHP [24]. Chapter 4 will compare the specifications made in chapter 2 with the finished system to explain what have been implemented and have not. Chapter 5 will cover the problems in the current system and what can be done to improve it.
2. Background

The background will describe the actors involved, the need for the product and a short introduction to PhoneGap. The target demographic for the product is Ireland and therefore most of the information in this report will be centered on data concerning Ireland.

2.1. Open Sensor Network Authentication group (OSNA)

The OSNA group is a cyber-security research group located at the Discipline of IT at the National University of Ireland. The OSNA group deals with security concerning all forms of control systems and sensor networks, for example real-time embedded systems and distributed systems [1]. In this project the OSNA group will be in charge of data transfer security. The application that this report will cover has been written for Dr. Hugh Melvin, whom is a member of the OSNA group and made the request for a smartphone application that can be used on multiple operating systems.

2.2. MC2

M2C is Ireland’s leading manufacturer of electric vehicle chargers, according to their webpage [2], and located at the Dublin City University campus in Dublin. As of March 2014 ten people are currently working at M2C.

In April 2013, M2C announced in a press release that they had won a contract with the Electric Supply Board (ESB) to provide the ESB with intelligent electric car chargers. The ESB is a corporate body established in 1927 to control and develop Ireland's electricity network. The charger in question is a device designed to monitor, control and log charging activity for an electric vehicle. For more information about the charger, see section 2.4. According to the agreement, ESB ecars Ireland will install the charger for free on the first 2000 electric cars purchased in Ireland [3].

2.3. The need for a smart charger

Many industrialized countries are trying to increase the generation of clean and renewable energy, such as wind power. According to the Irish Wind Energy Association (IWEA) [4], Ireland has currently a potential of generating 2632 MW from wind power. This accounts for 18% of Ireland’s total electricity need in 2013 [4]. While wind power has the benefit of being
a clean energy source, the drawback is that wind power is variable and weather dependent. According to the specification from M2C, the variability in wind poses a problem for companies such as ESB Networks and EirGrid. ESB Networks is responsible for the coordination and distribution of electricity within the Irish electrical grid and EirGrid is the state-owned electric power transmission operator in Ireland [5].

To allow more wind generated electricity on the grid, the variability of the wind has to be considered and managed so as to maintain adequate system frequency control. This is one of EirGrid’s main purposes, to manage the frequency of the alternating current on the grid in real-time, so it always is close to 50 Hertz (Hz). According to Eirgrid’s Frequency Control Workstream report [8], the normal operating limits for frequency in Ireland are from 49.9Hz to 50.1Hz, which varies according to the energy generated by the source. When it produces excessive energy, the frequency on the grid increases. While when less energy is produced, the frequency decreases. To handle this, there are generators, frequency reserves or regulating reserve [6], running on non-environmental dependent fuel. These are often signaled automatically to adjust to the change in frequency within 10 seconds [7]. More information about this can be found in the EirGrid's frequency control report [8].

The other issue and the cause of frequency variations is the variability of the supply from wind energy where on Ireland the electricity generated from wind was 18% 2013. Common strategies to cope with the variability of the supply from wind energy are to reserve generation capacity to use during periods of low wind, also known as spinning reserve. Spinning reserve means that there are other sources of energy, like coal, connected to the grid that can replace the expected energy output of the wind farm in the case of no or little wind. While the frequency reserve is a spinning reserve and they work in a similar fashion, they are often defined as; a frequency reserve shall respond within 10 seconds and a spinning reserve that is dealing with energy shortage shall respond within 10 minutes [7]. Furthermore, the Frequency reserve is automated while a spinning reserve is usually assigned by an operator. Since the frequency reserve has to respond within 10 seconds, it is not possible for a human being to constantly monitor and make the necessary changes. Therefore, the frequency on the grid is managed by a computer while the reserve, that is supposed to take care of the demand, is controlled by a human being.
The use of spinning reserve is one way to deal with energy shortage but, while it works, it costs money to have facilities running only to deal with the issue of an increased demand during peak hours. Another approach that would be cheaper, as it would ease the demand on the spinning reserve, is to dynamically change the load on the grid to match the wind generation capacity. This is known as energy demand management and means that any electric devices that do not have to be charged or used during peak hour will be scheduled to run when the demand for energy is lower.

Energy demand management could play an important role in the future as, according to M2Cs press release [9], the Irish government expects electric cars to make up 10% of the total car fleet in Ireland by 2020. This equals approximately 230 000 cars which all have to be charged using the electric grid. Furthermore, recent studies show that electric vehicles could make up as much as 60% of the passenger cars on Ireland by 2050 which would have an even greater impact on the energy demand.

One way of countering this is to dynamically alter the demand during peak hours. In this case, ESB electricity network would control and redirect energy demanding devices, like electric vehicles, to use the electric grid during a time when the wind is abundant or fewer devices are using it. That would ease the load during the most critical times of the day. One implementation of this would be, according to the specification from M2C, to ask the user when the vehicle is to be used again. Then, if there is sufficient time and the vehicle is able to be charged at different rates, a lower charging rate (energy per time unit) will be selected. From the M2C report, see appendix A, a few key elements are required to use a dynamic charging rate.

1. ESB can only vary the charge rate if it will not impact on availability of the vehicle.
2. The ‘variable load’ data should be anonymized by data aggregators and presented to the ESB as summary data on a control screen.

The first statement means that ESB cannot choose to charge the car at the slowest rate if it means that the car will not have time to reach the target charge before the user has stated that the car will be used again. Statement 2 refers to user privacy and that ESB cannot identify the
users that are connected to the mainframe server.

### 2.4. **M2C Smart Charger**

To allow the user to change how a device is charged, the outlet has to be equipped to handle user input. The M2C smart charger, Figure 3, is a charger for electric vehicles that can relay charging data to the user via a wireless local area network (Wi-Fi), Ethernet or the global system for mobile communications (GSM).

![M2C Smart Charger](image)

**Figure 3 M2C Smart Charger**

The data the charger can relay to the user is charge time, length of charge, charging status and faults in charging, such as the plug is not connected. Another significant feature is that the timer allows the user to set when to charge. With this, it is possible to start charging during a period when the electricity is cheaper, such as during the night. This feature is expanded further in the application section 2.5.
One of the main purposes with the Smart Charger was to allow ESBN to choose when to charge the device. One proposed solution for connecting to the ESB mainframe requires the charger to be connected to the local network which in turn has to be connected to the internet. To signal the ESB mainframe computer that it can obtain control over the charger the user needs a smartphone application that is paired with said charger. The connection scheme can be seen in Figure 4 below.

![Connection scheme between ESBN and M2C Smart Charger](image)

### Figure 4 Connection scheme between ESBN and M2C Smart Charger

#### 2.5. The application

In January 2014, Open Sensor Network Authentication group (OSNA) and M2C had a meeting where M2C requested a mobile phone application to compliment the M2C’s Electric Vehicle Smart Charger. At the same meeting, a high level specification was created describing the functionality of the application. The following is from the M2C Mobile App specification and can be seen in appendix A:

- Visualize various statistics relating to the charger (power usage, car charge rate, etc.)
- Set charging type
- Set the time when the vehicle will be fully charged
- Calculate a price for charging the vehicle [20]
The application should present statistics about the charger and current status to the user using a smartphone. If something is wrong, the user should be notified about this. The design of this will be covered in chapter 3 and the mockup from the specification can be seen below in Figure 5.

![Application Interface Mockup](image)

**Figure 5 Application interface mockup**

The application should also present an interface to set a charging type. The charging types are, after consulting with Dr. Hugh Melvin the application owner; manual, timed and ESBN.

Manual means that the user decides how fast it will charge and to what battery percentage. If possible, the application will display an estimation of when the target charge will be reached.
Timed means that the user enters the time when the vehicle is required to be used again. When a time has been set, the user will be notified of how much it will cost to charge the car. That cost will be calculated based on the current tariff from the electricity provider, where the price changes every half hour as can be seen in the tariff in Figure 6. The x-axis shows the date and time. The left y-axis shows the load on the system and the right y-axis is the single market price (SMP) in euro per megawatt hour. This is the price set by the electricity supplier [10].

![Figure 6 Price of electricity from SEMO for the period 2014-04-01 to 2014-04-01](image)

ESBN means that ESBN mainframe computer decides how and when to charge and has been covered earlier in this report.

The reason for this functionality is mainly to complement the ESBN charge type. As mentioned earlier, the user can let ESBN set the charge rate as long as it does not interfere with the usage of the car. The problem with this is that most people probably do not care about peak hours or feel the need to allow ESBN to change their charge rate (in Kw/hr). To encourage vehicle charging during hours outside the peak, when the demand for energy is lower, the user will be presented with the lowest possible price within that time span. This will give ESBN indirect control over the charger by setting the price lower or higher, depending on the demand which will affect how the application chooses to charge the vehicle.
2.6. **PhoneGap**

The application will, as according to the specification (appendix A), be using PhoneGap.

PhoneGap is an open source solution to develop mobile applications for multiple platforms with the common web technologies HTML, JavaScript and CSS [11]. PhoneGap was donated in October 2011 to the Apache Software Foundation under the name Apache Cordova. According to PhoneGap.com [12], Cordova is used interchangeably with PhoneGap in many instances but it should be noted that PhoneGap is a distribution of Cordova. Currently, the only difference between PhoneGap and Cordova is the name [13], which may or may not change in the future.

With PhoneGap, the developer creates a web application and, if there is a need to interact with the device specific functions, utilizes the tools provided by PhoneGap. To run the application on a mobile device, PhoneGap creates a package that is built by each system's proprietary tool. Since each mobile device implements system calls differently, these calls have to be specified for each system. With PhoneGap that is not something the developers have to think about, since the PhoneGap interface will handle the call to the native device API [14]. The whole process can be seen in Figure 7.

![Figure 7 PhoneGap interaction with a mobile device's operating system](image)

With PhoneGap, the developer also has the option of using PhoneGap Build which acts as a bridge between the different platforms such as iOS for iPhone or Android. Without it, the
developer would have to compile the application for each platform individually with the proprietary tools available. PhoneGap Build is a web service where the developer uploads the source code to, in a zip-file, and receives either a QR-code to be scanned by a smartphone or a downloadable package for each operating system. In a project with many participants, a version control system repository can be used instead of zip-files. The developer in that case links the PhoneGap Builder to the repository of the project and fetches the latest code from the master branch.

Figure 8 PhoneGap Build

When compiling the application, the developer has the option to turn on a function called hydrating. This allows the user to update their applications to the latest version whenever a new compilation has been made as opposed to downloading the whole application again.
3. Project Design and Implementation

This chapter is divided into two main parts; the design and implementation. The design part will cover the user interface and an overview of the interactions between the parts in the system. This will be complimented with activity diagrams, which are a way of following how an action is handled.

The implementation part will cover how the functionality in the system has been implemented in detail to give a thorough understanding of how it works. This is illustrated with pictures of the relevant parts of the code.

3.1. Model View Controller (MVC)

Model-view-controller is a design pattern used to create web applications [19], see Figure 9 for a graphical representation.

The model is the part that handles the business logic, like calculating results or interacting with the database.

The controller handles input and navigation which means that all the server calls goes through the controller.

The view is the user interface.

![Figure 9 Model View Controller illustration](image)
3.2. **Design**

The system contains three parts: Application, Server and the Database, see Figure 10.

![Diagram showing the system's design](image)

**Figure 10 Design used for the whole system**

The frontend of the system is the application that runs on a smartphone. Its purpose is to act as an interface that communicates with the server and also presents the response from the server in a view for the user. Between the database and application is the server.

The server contains two parts; the controller, called `server.php`, and the model. The controller is a PHP-file that is used for every request made by the application and determines where in the model to send the request for processing. It is also the controller that returns the results from the model.

In the model is a collection of PHP-files that performs specialized tasks, such as sending queries to the database. This has the benefit of removing all computation from the client to the server while also making the implementation of new functionality easier as it only requires the programmer to update the server-files. Another benefit that this brings is that adding new functionality or updating existing functions may not require the application to update its software.

The database acts as a mockup for the charger and the car. A mockup is a representation of a real device that does not do anything outside the specific testing it is made for. The database will be a part of the full system but at present the car’s status (battery capacity or battery charge) is set to fixed values that require manual editing in the database. In the future, the plan is to allow the charger to update those values.
3.2.1 Application screen layout

The application layout is divided into five display areas, see Figure 11, where area 1 is static, it will not change, and areas 2-5 are dynamic and will change according to

![Application screen layout](image)

**Figure 11 Application screen breakdown**

Area 1 is the first one to load and acts as a container for the other areas. The only content in area 1 is the logo.

Area 2 contains the information display which will, most of the time, display the current charger status. It is also used to display messages from the server such as confirmations or errors.

Area 3 is mainly used for user input and the detailed statistics about the charger and car. The reason for putting the detailed information in this area was to avoid reprinting the information from area 2 and instead complement it with additional information. This way they user is able to see all the available information at the same time.

Area 4 is the back button which was allocated its own area because of the frequent occurrence
of a back button. The back button will bring the user back to the start screen, see Figure 12.

Area 5 is not used at present, but can be used for commercial or any other information that the product owner would like to display. Area 5 is shown here because it was mentioned in the specification for the layout but no further information was specified and is therefore a blank area.

3.2.2 Application start screen

The application start screen is the first view that is presented to the user when the application starts, see Figure 12. It is from this screen the user will navigate to all the possible inputs that can be made and it is also to this screen the back button will bring the user. It consists of a charging status display, see 0 Charging Status design, and the four buttons seen in Figure 12.

Figure 12 Start screen
Charging Status design

The charging status is displayed in area 2 and shows the type of charging, the current level of battery charge as a percentage and which vehicle is connected the charger. In Figure 13, a Volvo with the registration number 131-G-123456 is connected with 42% of the battery charged and the charging type set to Manual.

![Figure 13 The design of the charging status](image)

The topmost text in Figure 13 shows the charging status, and may have three background colors and texts.

1. If the battery is charging, the background will be orange and the text will say “Charging”.
2. If the user has set a charge that is not full, i.e. not 100%, but the charger has the reached the target level, the text will say “Finished” on a yellow background.
3. If the battery is fully charged. Then color will change to light green and the text will say “Full”.
3.2.3 Charging Statistics design
Clicking the Charger Statistics button will bring up a detailed view of what the charger is currently being instructed to do. In Figure 14, the charger is charging the battery to 71\% at the fastest rate and it is estimated to be finished in 1 hour and 46 minutes.

Figure 14 Charging Statistics
3.2.4 Setting the charging type

When the user selects “Set Charging Type”, area 3 will display the available types. These are; Manual Charge, Timed Charge and ESBN Charge and can be seen in Figure 15.

![Figure 15 Selecting a charging type](image-url)
3.2.5 Setting a manual charge

If manual charge is selected, the user will be presented with a screen asking for the necessary details regarding charging of the car, see the right hand screen in Figure 16.

![Figure 16 Selecting the manual charge](image)

The user then has to determine to which level to charge the battery and at what rate to charge. The percentage can only be set to a value between the current charge, which is 21% in Figure 16, and 100%. The rates for how fast to charge the battery are, from fastest to slowest; fast, medium and trickle.
When the Set Charge button is pressed, the information screen at the top will change to a confirmation screen, or an error screen depending on what is received from the server. A confirmation can be seen on the right hand side in Figure 17. The screen currently displays almost the same information as the user receives by pressing the Charger Statistics button. The exception is that this screen also shows the mode in area 2.

Figure 17 Setting a manual charge
3.2.6 Setting a timed charge

The second option, when selecting a charge type, is a timed charge, see Figure 18.

When selecting the timed charge, the user is required to select at what time the vehicle is estimated to be used again and to what percentage to charge the battery. The available values for the battery charge are between the current charge plus one and 100%, since it would not make sense to set a zero difference. When pressing the “Get Price” button, a request is sent to the server to calculate the lowest price available within that time span. The price should be calculated with the daily tariff provided by the electricity company but is currently only using the tariff from 2014-04-18. If the server cannot calculate the price due to the time span being
too short, the response will be “Not Possible”; otherwise it will display the price in euro. If the user is satisfied, the charge has to be confirmed with the Confirm Charge button. Otherwise, a new time or charge level can be selected and a new price will be calculated. By extending the time, the price will most likely be cheaper or stay the same. This is because the price changes every half hour and the price calculating algorithm on the server side chooses those half hours which are cheapest until enough energy to meet the energy needs are satisfied.

When the “Confirm Charge” button has been pressed the information display, area 2, will change to a confirmation display from the server, see Figure 19, if the input is correct.

Figure 19 Setting a timed charge
3.2.7 Functionality

In appendix C, an activity diagram is presented, describing how the activity to set charge type will be handled. There are three charge types: Manual, timed and ESBN. The latter will not be implemented in this project since it requires specific knowledge concerning security and access to the ESBN mainframe computer.

3.3. Implementation tools

This subchapter will give a brief introduction to the tools used in project and how they are used.

3.3.1 Jquery

Jquery is a free, open source, JavaScript library designed to simplify HTML document traversal, manipulation and event handling [15]. In this project main use of Jquery is to find elements and their values in the HTML files used by the application and insert values to the elements.

To call a Jquery function, the global keyword Jquery is used but for simplicity the dollar sign, $, has been bound to the same keyword. To retrieve an element from the HTML page the element needs an id or a class tag. To retrieve an element with the class name, the hash symbol, #, is used and to find by the class name, a period is used, see code in Figure 20. In the first line of the example, a function will be run when the button has been clicked and in the second line of the example, the attribute value, which is the name of the button, will be set to “Save”. In short, Jquery consists of two parts. The left part of the period finds the element, and the right part defines what to do with the element.

```
$('#backButton').click(function());
Jquery('.backButton').attr('value', 'Save ');
```

Figure 20 Jquery code example
3.3.2 Ajax (Asynchronous JavaScript and XML)

Ajax is a way of sending and retrieving data from a server asynchronously, which means it is being done in the background, without interfering with the current HTML page [16]. It can load data in the background and display it without reloading the page.

In this project, Ajax requires the programmer to specify the parameters: type, data, url and timeout. See Figure 21.

```
function setCharge() {
    $.ajax({
        type: 'POST',
        'request': 'setManualCharge',
        'targetCharge': $('#targetChargeSlider').val(),
        'chargingRate': $('#chargingRate').val(),
        url: 'http://www.example.com/project/server.php',
        timeout: 3000,
        success: function(response){
            printResponse($.parseJSON(response));
        },
        error: function(){
            printResponse("Failed to connect to server");
        }
    });
}
```

Figure 21 Using Ajax

*Type* defines the request type and can be set to either POST or GET. This project uses POST as it allows for more data to be sent and is considered safer but either one would most likely suffice in this project.

*Data* creates a named array that is sent to the server. Named array means that, instead of an ordinary array that finds data with the help of index numbers, a named array finds the position of the data with a name.

*Url* specifies to which address the application shall connect. In this project the address is the same for every call to the server, independent of who makes the call.

*Timeout* specifies how long the application will wait for a response from the server. If a timeout occurs, the function following “error”, see figure 16, will be executed.

3.3.3 JSON

JSON is an open standard format for transmitting text-data between a server and web client [17]. In this project, the response from the server, unless it is an error, is always an object containing the relevant information. Errors are returned as strings.
To return the object from the server, only one function, `json_encode()`, is used, see Figure 22.

To decode the json object on the application side there is also only one function used, `jQuery.parseJSON()`.

Accessing a decoded object is simply the object followed by a period and the name of the parameter, see Figure 22.

```json
json_encode($response);
jQuery.parseJSON(response);
response.chargingType;
```

Figure 22 Json example to encode, decode and read the object.
3.4. **Application Code**

3.4.1 Application File Structure

In the applications root directory, see Figure 23, are three folders; css (cascading style sheet), img (image) and js (JavaScript) and the html-files. This is a very common way of categorizing the files in PhoneGap and is done automatically if the project setup is done as in appendix B.

![Application file overview](image)

Figure 23 Application file overview

The css folder (Cascading Style Sheets) contains one file called styles.css that handles the way similar elements, like buttons, look. The img folder contains an image of the logo seen at the top of the application, area 1 in Figure 11, and an image of the splash screen that has not been implemented.

In the js folder are the JavaScript files that handle the dynamic activities, such as loading new
content when a button has been pressed. These files have the same name as the HTML page they correspond to and are often loaded by that HTML page when that is loaded.

The HTML-files in the root-folder could be moved to an HTML subdirectory if the developer prefers. If this is the case, for the application to work, the index.html and config.xml must remain in the root-folder. This is by design in PhoneGap and thus cannot be avoided.

### 3.4.2 Application flow

When the application is launched, the index, which is the main container, holds all the other views. On initiation, the charger status will be loaded in to area 2 and the main buttons in area 3, see Figure 24 which shows what is on the screen when navigating the application. Arrows indicate an input.

![Application flow diagram](image)

Figure 24 Application flow diagram.

Clicking on one of the buttons will bring the user to level 1 where the possible inputs are to
set a charge or click the back button that will bring up the start screen again. If the user decides to set a charge, the application will move to level 2 that differ only in area 3 and allows to user to set the parameters regarding the charge. When a charge has been set, area 2 will change to a confirmation message instead of a charger status. On clicking the “back button”, it will change back to the Charger Status and the user will again be at the start screen.

3.4.3 Application Initiation

When the application is launched, the index.html is the first page to be loaded and acts as a container for the different views of the application, see Figure 25.

```html
<header class="heading"> <img src="img/m2c-logo.png" /> </header> <br>
<div id="infolabel"> Loading... </div>
<div id="content" class="content"> Loading... </div>
<div class="content"> <button class="mainbutton" type="button" id="backbutton"> Back </button> </div>
<div id="serverResponse"> </div>
<script type="text/javascript" charset="utf-8" src="js/index.js"></script>
```

Figure 25 Index.html

The header and each of the divider tags in Figure 25 correspond, in the same order, to the areas in Figure 11. infoLabel is area 2; content is area 3; backButton is area 4 and serverResponse is area 5.

Area 5 was mentioned earlier to be an area created for commercial purpose but is currently used as a container for developers for testing purpose. By calling the function testServerResponse() anywhere in the application with the answer from the server as a parameter, it will be printed in area 5.
When index.js is loaded, the first line in Figure 26 will run the function `initiateApplication()`.

```javascript
$(window).load(initiateApplication());

function initiateApplication()
{
  loadStatus();
  loadMainMenu();
  removeBackButton();
}

function loadMainMenu()
{
  $('#content').load("menu.html");
}

function loadStatus()
{
  $('#infolabel').load("status.html");
}

function showBackButton()
{
  $('#backButton').show();
}

function removeBackButton()
{
  $('#backButton').hide();
}

$('#backButton').click(function()
{
  loadStatus();
  loadMainMenu();
  removeBackButton();
});
```

Figure 26 The JavaScript for index.html

This function is only run once as `index.html` is only loaded once, which is when launching the application. The function, `initiateApplication()`, will then execute `loadStatus()`, `loadMainMenu()` and `removeBackButton()`.

`loadStatus()` will load the html-page status.html in to the divider named `infoLabel`, the divider that makes up area 2, which will display what the charger is currently doing.

`loadMainMenu()` will load menu.html in to the divider named `content`, the divider that makes up area 3, which will show the possible user inputs.

`removeBackButton()` will hide the button that was created when index.html loaded, see Figure 25, as there is no “back” from the start screen. The function name `removeBackButton()` may therefore be confusing as it does not actually remove it but hide it.

The bottommost function in Figure 26, concerning the `backButton`, is the Jquery way of finding an element by id and then stating what happens when it is clicked. In this case, the
function to be executed is declared inside the parenthesis, which is a very common way of declaring a function in JavaScript. When the back button is pressed anywhere in the application, this function will be called, the button will remove itself and call the exact same functions as the `initiateApplication()` function.

### 3.4.4 Information display to the user

Displaying information views, for example charger status or error messages, to the user is done by loading the html-page specifically created for the view the user is trying to access. In Figure 27, an example of how the charger status is displayed.

```html
<div class="statisticsLabel">
  Mode: <span id="type" class=""/>
  Charge Rate: <span id="rate" class=""/>
  Target Charge: <span id="targetCharge" class=""/>
  Estimated Time Left: <span id="timeLeft" class=""/>
</div>
<script>loadStats();</script>
```

**Figure 27 Charger status html code**

The “estimated time left” seen in Figure 27 is calculated at the server and is therefore not updated in real-time, since there is no continuous stream of data between server and application. This means that, as long as the user remains on the “charger status”, display the “Estimated Time Left” will not change. A future implementation could use the value in “Estimated Time Left” and calculate locally, on the application, the time left as it would count down towards zero.
When the user launches the application or returns to the start screen, the function `loadStats()` will be executed and display the chargers status to the user in area 2. This is done when the status.html is loaded, which was covered in 3.4.3 Application Initiation. `loadStats()` can be seen in Figure 28 and can be found in `chargerStats.js`.

```javascript
function loadStats()
{
    $.ajax({
        type: 'POST',
        data: {'request': 'statistics'},
        url: 'http://danu6.it.nuigalway.ie/bonstrom/project/server.php',
        timeout: 3000,
        success: function(response){
            jsonResponse(jQuery.parseJSON(response));
        },
        error: function(){
            printResponse("Failed to connect to server");
        }
    });
}

function printResponse(response){
    $('#type').text(response.chargingType);
    $('#rate').text(response.chargingRate);
    $('#targetCharge').text(response.targetCharge);
    $('#timeLeft').text(convertTime(response.timeLeft));
}

function convertTime(timeLeft){
    var hours = Math.floor(timeLeft);
    var minutes = Math.round(60*(timeLeft-hours));
    if(hours>=1)
        return hours + "h " + minutes + "min";
    else
        return minutes + "min";  
}
```

Figure 28 Charger status JavaScript code

`loadStats()` will query the server for what the chargers current status is with Ajax. In this example, the `type` is set to POST, which determines how the data is sent to the server. The other option is GET and is just as valid to use as POST in this project.

The `request` is `statistics` which tells the server where to send the request internally. For a complete list of possible request see the implementation of the server in 3.5.1 Controller.

The `data`-field will, when setting a charge or querying the server for an estimated charging price, also contain information about charging time and target charge.
The *url* will always be the one seen in Figure 28, independent of which part of the application that makes the call to the server, which is the whole point of the MVC-design pattern, see 3.1 Model View Controller. This is why the request must be set.

If the server does not respond within 3000 milliseconds, the algorithm will run *error* which sends an error message to the *printResponse()* function. As of writing the report, error messages from the server have not been implemented. See 5.2 Error Messages for more details concerning that. If a connection is successful, *printResponse()* will receive a json-object which is parsed before it is handled by the function.

The *printResponse()* function will, with the help of Jquery, find the appropriate HTML-element and insert the string that is inside the object received from the server. All of the strings are inserted as they arrive except the time left. The time left created on the server is on the format 1.35 hours and not very appealing for most users. To avoid doing any formatting on the server side, as the time format is a matter of taste, it is sent like that and formatted by the application. On this project the format 1 h 35 min is used.

**3.4.5 Setting a manual charge**

When the user sets a manual charge, the required parameters are a charge level (in percentage) and the rate to charge the battery (watt hours per hour), see Figure 17 for a visual representation. The JavaScript to handle the communication with the server is similar to the *loadStats()* function in Figure 28, with the difference that the data field contains the previously mentioned parameters. The server will then try to set the charge specified and return a response with the details concerning the charge.

The first three lines of the response are *type* and *rate* which is echoing what was sent to the server.

The *type* in this case is *Manual*.

The *rates* can be; trickle (slowest), medium and fast.

The *target charge* is a numerical value between the current charge plus one, which is set as the minimum value on the slider in Figure 17, and 100%. The current charge is extracted from the information screen, area 2 in Figure 11, and not the actual value from the charger.

The *time left* is calculated by the server and arrives as a decimal value in hours. The hours are
then converted by the convertTime() function, can be found in chargerStats.js, to hours and minutes.

The HTML-page responsible for displaying the response, chargingResponse.html, will be loaded in to the infoLabel divider (area 2). The difference in this printResponse() function and the others is that populating the elements is done inside a function in the load function, see Figure 29.

```javascript
function printResponse(response){
$("#infolabel").load("chargingResponse.html", function() {
   $('#type').text(response.chargingType);
   $('#rate').text(response.chargingRate);
   $('#targetCharge').text(response.targetCharge);
   $('#timeLeft').text(convertTime(response.timeLeft));
});
}
function loadCharge(){
   $('#minLabel').text(getCurrentCharge() + "%");
   $('#targetChargeSlider').attr("min", getCurrentCharge());
}
function getCurrentCharge(){
   return $('#currentCharge').text();
}
function updateRangeLabel(){
   $('#rangeLabel').text($('#targetChargeSlider').val() + "%");}
```

Figure 29 JavaScript to set a manual charge

This is called a callback function [18] which means that it will be executed after chargingResponse.html has been fully loaded. Otherwise, it will try to populate the elements before the page has been loaded, since that is executed much quicker, and will fail to display anything. Most of the other print functions will load a new HTML-page that has its own JavaScript-file that will populate the elements on the page. In short, this printer function is trying to populate an HTML-page that is not already loaded, that is why it is different.
3.4.6 Setting a timed charge

When the user tries to set a timed charge, the application will first ask the server for a price. To calculate the price, the server requires the target battery charge (as a percentage) and an estimated time the vehicle will be used again. If it is possible to charge the battery according to the desired input, a price will be returned from the server and displayed on the screen (Figure 19) and will allow the user to confirm the charge. If the server has returned “Not Possible” or that no price request was made before pressing the confirmation button, an error message will be displayed in area 2, see Figure 11. The test can be seen in Figure 30.

```javascript
function getPrice()
{
    $.ajax({
        type:'POST',
        data:{
            'request': 'getEstimatedPrice',
            'targetCharge': $('#targetChargeslider').val(),
            'targetTime': $('#targetTime').val()
        },
        url:'http://dnuG.it.nuigalway.ie/bonstrom/project/server.php',
        timeout:3000,
        success: function(response){
            printEstimatedPrice(response);
        },
        error: function(){
            printResponse("Failed to connect to server");
        }
    });
}

function setTimedCharge()
{
    if($('#estimatedPrice').val()=="" || $('#estimatedPrice').val()=="Not Possible"){
        $('#infoLabel').load("error.html", function(){
            $('#error').text("Error: Not possible to set charge");
        });
    }
    else{
        $.ajax({
            type:'POST',
            data:{
                'request': 'setTimedCharge',
                'targetCharge': $('#targetChargelslider').val(),
                'targetTime': $('#targetTime').val()
            },
            url:'http://dnuG.it.nuigalway.ie/bonstrom/project/server.php',
            timeout:3000,
            success: function(response){
                printResponse(response);
            },
            error: function(){
                printError("Failed to connect to server");
            }
        });
    }
}
```

Figure 30 Set the timed charge
3.4.7 ESBN Charge

ESBN Charge has not been implemented as that would require collaboration with ESBN to access their servers. In the future, it might also be named something else, such as Utilities, since, according to the developers of the charger, it should be possible to use the option with electricity providers other than ESBN.

3.4.8 User inputs

The buttons on the start screen and any other buttons are loaded into the content divider just like any other HTML-page. The buttons HTML-page is called MainMenuButtons.html and can be seen in Figure 31.

```html
<button class="mainButton" type="button" id="statsButton"> Charger Statistics </button>
<button class="mainButton" type="button" id="chargeButton"> Set Charging Type </button>
<button class="mainButton" type="button" id="test"> Test Server </button>
<button class="mainButton" type="button" id="connectServer"> Connect to server </button>
<script type="text/javascript" charset="utf-8" src="js/menu.js"></script>
```

Figure 31 Start screen buttons

In this project, only the two first buttons in Figure 31 are part of the specification. The Test Server button is there for the developer to see the response from the server, which will be printed in area 5 just like the testServerResponse() that was discussed earlier. The Connect to server button was created when it was discovered that if the application on launch could not access the server, the only way to get the charger status in area 2 was to restart.

The back button, that has its own area, was originally in the same file and area as the other buttons but was later changed, since that would require it to be loaded every time the content changed. Now the button only has to be disabled or enabled when you leave or return to the start screen.
When a button is clicked there are two ways to handle the event. The first, and most common among beginners, is to add an `onClick` function in the tag on the HTML-page. In this project, all the click functions are handled by JavaScript as seen in Figure 32.

```
$$('#statsButton').click(function() {
    showBackButton();
    $('#content').load('chargerStats.html');
});
```

**Figure 32 JavaScript for buttons**

The function contains the buttons id, `statsButton` in this example, a click action, which is tied to the action of clicking made by the user, and what function to call. All the buttons in this project have the function created inside the click function.
All the buttons in the application share the same class `mainButton` which is a way of making them look uniform as the class is tied to the CSS-file `styles.css`. If an element in this application has a class name associated with it, that element’s appearance will be changed by a CSS function in `styles.css`. An example of the CSS handling the statistics information can be seen in Figure 33.

```css
.statistics_label {
  font-weight: bold;
  font-size: 20px;
  background-color: lightblue;
  text-align: center;
  margin-left: 5%;
  margin-right: 5%;
  border-radius: 15px;
  border-color: black;
  border-style: solid;
  border-width: 3px;
}
```

**Figure 33** The css associated with the statistics label

Except for buttons, the application implements two other input methods; sliders and a set time display: see Figure 34 for all the possible interactions. All three input methods are standard HTML elements that do not require the programmer to further customize them to work. The button and time display has been tied to a CSS-style which is the only customization that has been done. The slider has been given a minimum value that equals the current charge plus one. The slider is also surrounded by labels showing the current charge, minimum and maximum values that are updated as the slider changes.

**Figure 34** Application Interaction: Time display, slider and button.
3.5. **Server Code**

3.5.1 Controller

When the application makes a request to the server side, it is handled by a controller called `server.php`. The controller’s purpose is to be the central point of the server side where all the data passes through. This is part of the MVC designed pattern mentioned earlier in this report: see Figure 35 for the code and Figure 10 for an illustration of the controller’s purpose.

```php
if(isset($_POST['request'])) {
    $request = escape($_POST['request']);

    switch ($request) {
        case "status":  $response = getStatus();
                        break;
        case "statistics":       $response = getStatistics();
                        break;
        case "setManualCharge":          $response = setManualCharge();
                        break;
        case "getEstimatedPrice":   $response = getEstimatedPrice();
                        break;
        case "setTimedCharge":       $response = setTimedCharge();
                        break;
        case "setESBNcharge":       $response = setESBNcharge();
                        break;
        default: $response = "Server";
    }
    else
        $response = "Server: No request set";

    sendResponse($response);
}
```

**Figure 35** The controller on the server side

The controller implemented on this server (Figure 35) will look at the POST variable named `request` to determine where the data should be sent for processing. All of the data needed for the request is inside the post variable, which is a named array, and is globally available inside the server. This means that the functions do not take any parameters, as they can access the data themselves. The functions will then generate a response that is sent to the `sendResponse()` function where it will be encoded in to a json-object and sent back to the application.
3.5.2 Status of the charger and the car

The status of the car and the charger is fetched when the server receives a request that says \textit{status}. It will then execute the function \texttt{getStatus()}, see Figure 36, that will use the interface in \texttt{database.php} to get the status about the car to check if it is connected. If the car is connected, a default response message is created from the \texttt{response.php} file. If in the future a more specialized message is required, this is the function to be updated.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure36.png}
\caption{Get charging status php code}
\end{figure}

3.5.3 Response

Every response sent from the server is a class called \textit{Answer} and contains all the data the server has access to. See Figure 37 for all of the possible data that can be returned.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure37.png}
\caption{The answer class used by the response function}
\end{figure}
The function, when called, will create a new answer object and set each and every variable in it to a value. This is done no matter which function is calling it and leaves it to the application to extract the relevant data it finds useful to display. The code seen in Figure 38 shows how to set the object’s variable.

```php
$charger = getChargerData($db);
$car = getCarData($db);

$response = new Answer;
$response->chargingType = $charger['type'];
$response->chargingRate = $charger['chargingRate'];
$response->targetCharge = $charger['targetCharge'];
$response->batteryChargePercent = $charger['batteryChargePercent'];
$response->currentCharge = $car['batteryChargePercent'];
$response->batteryCapacity = $car['batteryCapacity'];
$response->timeLeft = $car['timeLeft'];
$response->model = $car['model'];
$response->regNumber = $car['regNumber'];
$response->connected = $car['connected'];
$response->targetTime = $charger['targetTime'];
$response->activity = $charger['activity'];
// $response->price = getPrice();
return $response;
```

Figure 38 Creating a response
3.5.4 Set a manual charge

When the server sets a timed charge, it needs the data about the car to be able to calculate to which value the user wants to charge the car’s battery. This is done with the `batteryChargeWh()` which will convert the target battery percentage to the actual watt hours that correspond to with the battery in the car. After setting the variables seen in Figure 39, they are sent to the database interface to be inserted to the MySQL database used in this project; see database interface for further information of how that works. The database interface will then return the value one if the insertion was successful and a zero if not. This is then tested to create an appropriate response message.

```php
function setManualCharge(){
    $db = getDatabase();
    $car = getCarData($db);

    $targetCharge = batteryChargeWh($_POST['targetCharge'], $car['batteryCapacity']);
    $chargingRate = chargingRateToValue($_POST['chargingRate']);
    $chargeType = "Manual";
    //response is a true or false type.
    $response = setChargerData($db, $targetCharge, $chargingRate, $chargeType, $activity);
    if($response)
        return createResponse($db);
    else
        return "Could not set requested charge";
}
```

Figure 39 Php code for setting a manual charge
3.5.5 Set a timed charge

Setting a timed charge requires two steps on the application, that the user receives a valid price and then confirms it. On the server side, those two functions are independent of each other, since the charger does not care what the price is, and therefore other parts of the application could ask for the price of charging if that is of interest in future implementations.

```php
function setTimedCharge(){
    $startTime = date('H:i:s');
    $endTime = $_POST['targetTime'];
    $db = getDatabase();
    $car = getCarData($db);
    $charger = getChargerData($db);
    $energyReq = getEnergyRequired($car['batteryCharge'], batteryChargeOn($_POST['targetCharge'], $car['batteryCapacity']));
    $tariff = getTariff($db, $startTime, $endTime);
    $chargingTimes = getChargingTimes($tariff, $energyReq);
    setChargerData($chargingTimes);
    return "Server: Not implemented";
}
```

Figure 40 Php code for setting a timed charge
3.5.6 Database interface

Connecting to the database is mainly performed by functions on the server and is therefore implemented by an interface. This also collects the most sensitive part of the system in one place. To allow a function to query the database a connection must be established. This is done with the `getDatabase()` function, see Figure 41. If the connection is successful, a database object is returned that can be used to query the database. The database object is then passed to query functions such as `getCarData()` and `getChargerData()`, seen in Figure 41. The function `getCarData()` will create a query to the database asking for all the data in the Car table. The data will then be saved in the $response variable as an array with the indexes as names instead of numbers. This is called an associative array and is done with the `fetch_array(MYSQLI_ASSOC)`.

```php
function getDatabase(){
    $database = "mydb144B";
    $usr = "mydb144B";
    $password = "ha/gp";
    $mysqli = new mysqli("localhost", $usr, $password, $database);
    if ($mysqli->connect_errno){
        echo "Failed to connect to database : (" . $mysqli->connect_errno . ")" . $mysqli->connect_error;
    }
    return $mysqli;
}

function getCarData($db){
    $sql = "SELECT connected, batteryCapacity, batteryCharge, model, regNumber FROM Car";
    $response = $db->query($sql);
    return $response->fetch_array(MYSQLI_ASSOC);
}

function getChargerData($db){
    $sql = "SELECT chargingRate, targetCharge, type, targetTime, activity FROM Charger";
    $response = $db->query($sql);
    return $response->fetch_array(MYSQLI_ASSOC);
}
```

Figure 41 Php code for interactions with the MySQL database
To insert data to the database, the server implements the prepare function, see Figure 42. This will take a MySQL statement and replace the variables with question marks. The question marks will then have to be bound to variables with the `bind_param()` function. This is a way to sanitize inputs from malicious code done by PHP. The “iiss” in that function is telling the function what type the variables are. In this case they are: integer, integer, string and string. After the parameters have been bound the statement is executed and a response from the database is stored in the `$response` variable.

```
function setChargerData($db, $targetCharge, $chargingRate, $type, $activity)
{
    $sql = "UPDATE Charger SET targetCharge = ?, chargingRate = ?, type = ?, activity = ?";
    $stmt = $db->prepare($sql);
    $stmt->bind_param("iiss", $targetCharge, $chargingRate, $type, $activity);
    $response = $stmt->execute();
    $stmt->close();
    return $response;
}
```

Figure 42 Inserting data to the database
3.5.7 Conversions

Many of the functions on the server need to convert the data between what the user wants to see and what is actually useful. In Figure 43 are some of the commonly used functions; `batteryChargePercent()`, `chargingRateToken()` and `chargingRateToValue()`.

`batteryChargePercent()` is used to convert the current energy of the battery to a percentage of the total capacity. In the database, the current battery charge is stored in Watt-hours, which is not very useful to display for a user. Instead, the current charge is divided with the capacity of the battery to return the ratio between them.

`chargingRateToken()` and `chargingRateToValue()` both perform a similar operation, but opposite direction. The first converts the current charging rate, which is Wh/h (Watt hours per hour), into a representative token while the second takes a token and converts it into a charging rate. The use for them is therefore; the first is going to the user while the second is going to the database.

```javascript
function batteryChargePercent($targetCharge, $batteryCapacity)
  {
    $charge = 100 * $targetCharge / $batteryCapacity;
    return round($charge);
  }

function chargingRateToken($chargingRate)
  {
    if($chargingRate==4000)
      $rate = "Fast";
    else if($chargingRate==2000)
      $rate = "Medium";
    else if($chargingRate==1000)
      $rate = "Slow";
    else
      $rate = "Incorrect conversion";
    return $rate;
  }

function chargingRateToValue($chargingRate)
  {
    // rate is in Wh / h
    if($chargingRate=="Fast")
      $rate=4000;
    else if($chargingRate=="Medium")
      $rate=2000;
    else
      $rate=1000;
    return $rate;
  }
```

Figure 43 Some of the conversion functions used by the server
### 3.5.8 Get an estimated price

If the user tries to set a timed charge a price will first be calculated with the `getEstimatedPrice()` function, see Figure 44.

```php
function getEstimatedPrice()
{
    $startTime = date('H:i');
    $endTime = $_POST['targetTime'];
    $db = getDatabase();
    $car = getCarData($db);
    $charger = getChargerData($db);
    $energyReq = getEnergyRequired($car['batteryCharge'],
            $car['batteryChargeWh'] * $_POST['targetCharge'],
            $car['batteryCapacity']));
    $tariff = getTariff($db, $startTime, $endTime);
    $price = getLowestPrice($tariff, $energyReq);
    return $price;
}

function getLowestPrice($tariff, $energy Req)
{
    // Fast charge rate is 4.000 Wh/h
    $chargeRate = chargingRateToValue("Fast");
    // How long each charge is. This is currently 30 minutes as that is the length of time
    // the price is valid. Note that this isn't completely true as a charge might start
    // in the middle thus only 15 minutes is available at that price rate.
    $chargeInterval = 0.5;
    $price = 0;
    $i = 0;
    while($i < sizeof($tariff->price) and $energyReq > 0)
    {
        if($energyReq > $chargeRate)
            $price += $chargeRate * $chargeInterval * $tariff->price[$i];
        else
            $price += $energyReq * $tariff->price[$i];
        $i++;
        $energyReq -= $chargeRate * $chargeInterval;
    }
    if($energyReq > 0)
        $price = 0;
    return round($price, 2);
}
```

Figure 44 Algorithm to get the estimated price
getEstimatedPrice() will set the start time as the server’s current time and the end time from the post variable targetTime which is set by the user. This determines the interval that the charger can charge the car. The algorithm will then use the specialized functions designed to handle the database, see Figure 45, to get data from the mockup that is the car and charger. The difference between current battery charge and the target battery is the energy required for the charging session. It is then possible with the energy required and a tariff in the interval specified earlier it is possible to calculate a price which is done with the getLowestPrice() function.

getLowestPrice() will always use the fastest charge rate, which is defined as 4000 Watt hours per hour (Wh/h), and multiply it with the charging interval (hours), which is half an hour as that is the interval that the price is valid. A unit conversion of this gives: (Wh/h) * h which is the same as Wh, another way of writing energy (joule). For a price, the energy will have to be multiplied with the cost of when it is being extracted.

The price or tariff is requested from the database and is an array sorted in ascending order: from cheapest half hour price to most expensive. The algorithm will then draw the amount of energy that equals a fast charge rate, currently set to 4000 Wh/h, every half hour from the array as long as the energy required is a positive value or there are no more half hours to draw from the array. If that happens, the price returned is set to zero, which will be interpreted by the application and generate a message informing the user that the requested charge is not possible.

The price is calculated by multiplying the tariff cost, which is euro per Watt hour, with the energy extracted that half hour.
To avoid having to rewrite frequently recurring code, that also is prone to security bugs, the database connections are collected in one file called `database.php`. The connection implements mysqli, which is a collection of predefined functions controlling access to MySQL database, and gives the programmer a greater degree of security with regard to database access and control.

```php
function getDatabase() {
    $database = "mydb1448";
    $usr = 'mydb1448bd';
    $password = 'ha/gup';
    $mysqli = new mysqli("localhost", $usr, $password, $database);
    if (!$mysqli->connect_errno) {
        echo "Failed to connect to database : ( " . $mysqli->connect_errno . " ) " . $mysqli->connect_error;
    }
    return $mysqli;
}

function getCarData($db)
{
    $mySQL="SELECT connected, batteryCapacity, batteryCharge, model, regNumber FROM Car";
    $response = $db->query($mySQL);
    return $response->fetchAll(MYSQL_ASSOC);
}

function getChargerData($db)
{
    $mySQL="SELECT chargingRate, targetCharge, type, targetTime, activity FROM Charger";
    $response = $db->query($mySQL);
    return $response->fetchAll(MYSQL_ASSOC);
}
```

Figure 45 Accessing the database
3.6. **Database**

The database contains a mockup of the actual charger and car while also storing the tariff used to calculate the price for charging the car.

3.6.1 Car mockup

The car mockup contains five fields; `connected`, `batteryCapacity`, `batteryCharge`, `model` and `regNumber`. `Connected` is a Boolean used for testing how the application should react if the car is not connected. There is currently no implementation of a non-connected car. `batteryCapacity` is the energy in watt hours (Wh) that the battery is theoretically capable of storing. This is used in combination with `batteryCharge`, the current charge in Wh, to display the current charge in percentage of full charge. `batteryCharge` is used frequently in the code whenever the current charge is needed. `Model` and `regNumber` is used to see which car is currently connected to the charger. How this will be implemented in a real scenario has not been discussed.

```
+----------------+----------+---------+--------+----------+
<table>
<thead>
<tr>
<th>connected</th>
<th>batteryCapacity</th>
<th>batteryCharge</th>
<th>model</th>
<th>regNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24000</td>
<td>5000</td>
<td>Volvo</td>
<td>131-G-123456</td>
</tr>
</tbody>
</table>
```

Figure 46 Database car table

3.6.2 Charger mockup

The charger mockup contains the five fields; `chargingRate`, `targetCharge`, `type`, `targetTime` and `activity`. `chargingRate` is the rate the battery will be charged and is given in the format 4000 Wh/h. To further extend on this: it charges the battery with 4000 Wh every hour which means that a complete loading process (24 000 Wh) would take 6 hours (24000/4000). `TargetCharge` is to what level the user wants to charge the battery, therefore when `targetCharge` and `batteryCharge` are equal to one another, the charging should stop as it is finished. Type is which mode the user has chosen. There are currently two types available; 1) `manual`, the user sets charge rate (Wh/h) and 2) `timed charge`, the user sets the target charge level and estimated time to use the vehicle again. `TargetTime` is when the charger should stop doing what it is currently doing. In Figure 47 there is only one entry as the type of charge is
set to *manual* which will charge until it reaches the target charge level, independent of the
time it will be reached. If the charge is set to timed, the charger will have to switch on and off
depending on the price of electricity. In that case, the database will contain a field for each
half hour the charger will be running. *Activity* will display to the user what the charger is
currently doing, if it is charging, waiting or turned off.

<table>
<thead>
<tr>
<th>chargingRate</th>
<th>targetCharge</th>
<th>type</th>
<th>targetTime</th>
<th>activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>14640</td>
<td>Manual</td>
<td>00:00:00</td>
<td>Charging</td>
</tr>
</tbody>
</table>

Figure 47 Database charger table

### 3.6.3 Tariff for the electricity price

Figure 48 contains a small part of the tariff table downloaded from SEMO. A similar tariff can
be seen as a graph in Figure 6. When the server calculates a price for the charging of the
vehicle, the database will be queried with a time span and will return the delivery time and
SMP, the price in euro per megawatt hour, from this table sorted in ascending order on SMP.

<table>
<thead>
<tr>
<th>Trade_Date</th>
<th>Run_Type</th>
<th>CURRENCY_FLAG</th>
<th>Delivery_Date</th>
<th>DELIVERY_TIME</th>
<th>SMP</th>
<th>SYSTEM_LOAD</th>
<th>LAMBDAA</th>
</tr>
</thead>
<tbody>
<tr>
<td>17/04/2014</td>
<td>EA</td>
<td>EUR</td>
<td>19/04/2014</td>
<td>00:00:00</td>
<td>41.170</td>
<td>3497.002</td>
<td>41.170</td>
</tr>
<tr>
<td>17/04/2014</td>
<td>EA</td>
<td>EUR</td>
<td>19/04/2014</td>
<td>21:00:00</td>
<td>39.140</td>
<td>3451.723</td>
<td>39.140</td>
</tr>
<tr>
<td>17/04/2014</td>
<td>EA</td>
<td>EUR</td>
<td>21/04/2014</td>
<td>01:00:00</td>
<td>39.030</td>
<td>3312.725</td>
<td>39.030</td>
</tr>
<tr>
<td>17/04/2014</td>
<td>EA</td>
<td>EUR</td>
<td>21/04/2014</td>
<td>01:30:00</td>
<td>35.150</td>
<td>3167.982</td>
<td>35.150</td>
</tr>
<tr>
<td>17/04/2014</td>
<td>EA</td>
<td>EUR</td>
<td>21/04/2014</td>
<td>02:00:00</td>
<td>35.150</td>
<td>3051.963</td>
<td>35.150</td>
</tr>
<tr>
<td>17/04/2014</td>
<td>EA</td>
<td>EUR</td>
<td>21/04/2014</td>
<td>02:30:00</td>
<td>33.960</td>
<td>2922.654</td>
<td>33.960</td>
</tr>
<tr>
<td>17/04/2014</td>
<td>EA</td>
<td>EUR</td>
<td>21/04/2014</td>
<td>03:00:00</td>
<td>33.960</td>
<td>2840.418</td>
<td>33.960</td>
</tr>
<tr>
<td>17/04/2014</td>
<td>EA</td>
<td>EUR</td>
<td>21/04/2014</td>
<td>03:30:00</td>
<td>33.960</td>
<td>2791.676</td>
<td>33.960</td>
</tr>
</tbody>
</table>

Figure 48 Database tariff table
4. Results/Evaluation

This chapter will present a comparison between the final application (the implemented prototype) and the requested application from the original specification.

4.1. Project Goals

When the project started, the goal was to create an application that would run on any smartphone and be able to interact with a database. The application should be able to visualize various statistics relating to the charger, set charging type, set the time when the vehicle will be fully charged and calculate a price for charging the vehicle.

4.2. Project Results

Currently the application has only been tested on android, since there is a license fee associated with developing for iPhone devices, and the author did not have any of the other operating systems available. The application has been developed in PhoneGap which means that, in theory, the application should run on the other systems but no testing has been done and therefore it cannot be assured that it works.

The application can visualize: the charge rate, the chargers current activity, the charging mode set by the user, the current charge in percentage, the vehicle connected, target charge in percentage and the estimated time left. See Figure 14 for a screenshot of the available statistics.

The user can set two charging types; manual and timed charge, which were the two types requested by Dr. Hugh Melvin. Both have been implemented and work but are lacking certain features such as error reporting.

Timed charge implements the feature of setting the time when the vehicle will be charged to the level specified by the user. This is a slight change to the original statement which says the vehicle should be fully charged, proposed by Dr. Hugh Melvin.

The price information is also a feature available when using the timed charge option and has been implemented and tested.
The tariff for calculating the price is not updated automatically, which means that the price is always tied to the date when the tariff was downloaded from the SEMO webpage. There is a problem concerning the implementation of live calculations of the price but this is discussed in Chapter 5 Conclusion.

Figure 49 shows a comparison between the mock up provided in the specification and the same screen in the application. The layout with the logo at the top, an information display followed by the buttons is the same. The information display is similar with the added feature that it shows what kind of charge is currently set. The buttons have changed a lot from the mockup as the charger statistics is the only one that was implemented. The other three seen in the mockup were not covered in the specifications or discussed in the meetings and therefore not implemented.

Figure 49 Comparison between the mockup and the implementation

To summarize, the project has fulfilled all the requirements that were established but requires more work to be considered an actual useable application.
4.2.1 User Feedback

The application has been tested by a few people that are not associated with the project in order to receive user feedback. The general opinion is that the application’s information display is clear and easy to interpret. The colors are helpful and the overall color scheme of the application is not aggressive to the eye, i.e. the colors are comfortable to look at. The size of the buttons is good and has been well received by people with both big and small hands.

The application is easy to navigate and very fast. The overall experience was very positive but it should be noted that the test environment is not the real environment and more user tests should be conducted as the system grows.
5. Conclusion

This chapter will cover: the remaining issues in the applications, future implementations and what could or should have been done differently.

5.1. Views and esthetics

The views in the application should have been implemented differently. Instead of loading a previously created html page into a divider when new content is accessed, the JavaScript should create the view when needed and load it in to the appropriate divider of the index.html. By creating the view in JavaScript, the number of files in the project would effectively be halved and it is also a very common way of designing html pages. The reason for not doing this was mainly because the author of this thesis did not have the necessary experience prior to the project and that there were few HTML pages in the beginning that rapidly grew towards the end of the project. At the end, it became much harder to navigate to the correct view then it would have been by implementing said solution. It was by then too late to re-implement the views, which are going to be reimplemented this summer.

The coloring and design of the application should be reviewed by someone with an artistic background or changed to use Bootstrap or any other similar framework for css. The css currently used is mostly written by hand and not as pleasing to the eye as it could be.

5.2. Error messages

Error messages from the server were not implemented because the author of this report did not know how best to handle errors and that the application’s views currently cannot display them in a proper way. This is closely tied to the previously mentioned view problem. There are of course ways to implement this in the current view but it would have led to the creation of more complicated code and was therefore not done at all. This is to be implemented this summer.

5.3. Real-time system

The tariff used to calculate the price is downloaded manually as a csv-file from SEMO’s webpage. This file contains a date format that does not match the date format used in MySQL. Because of this, there is currently no way of telling MySQL that the column contains dates
and are therefore treated as a string of character. This creates an unwanted side effect with querying the database for all the data with a specific date impossible. This occurs when there is a need for data after 23:59, as that is a new day. To ask for data after 23:59, the date would have to be specified as to not confuse it with the data from the current day.

Updating the current battery charge on the application side. The current battery charge is only updated when the application makes a request to the server which is dependent on when the user makes a request to the server. This could be very rarely which means that the charge will stay the same for a long time. One proposed solution would be to use JavaScript and let the application calculate at what charge the battery should be with the current charging rate. Then when the application actually gets updated the displayed battery charge will be modified to the actual charge and the process will begin again with the updated information. This would of course be a problem with the timed charge as it does not charge constantly but would still be an improvement over not updating at all.

5.4. Optimisation

The application will connect to a centralized server which will handle possibly thousands of connections. As the user base grows, the bandwidth used by the server will grow with it. Therefore, optimization of the response function might require consideration. As was mentioned in Chapter 3.2, the implementation of the server, the response object contains the same kind of variables independent of who is calling the createResponse() function. This means that sometimes the variables sent will contain unnecessary information that will use the bandwidth. To deal with this, it would be ideal to create a customized response message depending on what the request was.

5.5. Final Comment

This project has fulfilled the specifications provided when the project began, but the system created is still far from being a complete system. Creating a complete system was never the goal of the project but rather to create a system that could be further developed into a commercial application. With the help of this report, the next developer should receive a complete understanding of the system and guidelines of how to proceed.
References


20. Private communication with Dr. Hugh Melvin


Appendix A: Project specification

M2C Mobile App Development High Level Spec

At a meeting between OSNA and M2C (Mick Berry) in January 2014, M2C highlighted the need for a mobile phone app to compliment M2C’s Electric Vehicle Smart Charger.

During the meeting the idea was further fleshed out and the following is a high level spec of the desired app functionality as understood by OSNA. This spec will form the basis for the deliverables on the app development project.

Background
ESB Networks (ESBN) and Eirgrid are responsible for the coordination and distribution of Electricity within the Irish electrical grid. In order for the ESBN to allow greater wind generation capacity to be connected to the grid, it needs to deal effectively with wind variability and wind generation predictability so as to maintain adequate system frequency control. Strategies include; reserve generation capacity during periods of low wind (termed spinning reserve), and preferably, the ability to dynamically change the load to match wind generation capacity.

Conventional power plants operate most efficiently and effectively at rated capacity so running with reserve under a ‘spinning reserve’ strategy costs money. There is however a potential to exploit load control. If ESBN could control large loads within the home in real time, across a large percentage of the grid, then they could shed/gain load to match the variability in wind generation. As electric vehicles become more prevalent, the demand they place on the grid will increase. This increase in demand however can be used to the ESBN’s advantage if the loading can be varied and time shifted to suit generation capacity.

A system could be developed whereby the chargers could use a statistical model to switch on/off based on the frequency of the supply. A statistical model element is required to prevent clusters of chargers switching in/out in a localised area at the same time, putting strain on the local LV (low-voltage) network. Such a model would require extensive testing, and while it would provide a solution, the ESBN would have no visibility or direct control of the ‘switchable load’.

The figure overleaf outlines the architecture for a proposed solution which would provide ESBN with both visibility and near real-time control of M2C Chargers. This architecture could be further expanded to include additional devices within the home, including domestic heating and water heaters.
The App
The mobile app is a key element within the proposed architecture. The mobile app allows the user to visualize various statistics relating to the charger (power usage, car charge rate, etc). More importantly the app also allows a user to set the desired ‘Fully Charged Time’ for their vehicle. This feature is particularly important as it will determine if the charger can vary the charge rate while the vehicle is attached.

App Operation
When a user first opens the app it will need to pair with the charger. This will need to be completed over an ad-hoc wireless link in order to configure the charger with the SSID and password of the user’s home router.

Once this step has been completed the app can re-connect with the home Wi-Fi and communicate with the charger over the home network. When re-connected the app will need to identify the charger and pair with it.

The app is then ready to download the data required; current charge status and any available stored statistics. At this point the first feature of the app is completed.

The app will then detect the set-up of the charger. The charger will have a default ‘vehicle connected’ behaviour which the user can then vary.

The user can choose between; available charge rates, delayed available charge rates and dynamic.
- Available Charge Rates: If multiple charge rates are available (e.g., fast, medium, trickle charge) the user can select one as a default.
- Delayed Available Charge Rates: as with above but start after a fixed period or at a fixed time of day (e.g.: off-peak)
- Dynamic: This is the advanced option. This would allow the user to configure a timetable with 'Fully Charged Times' settings. Once configured, the charger can vary the charge rate to provide a full charge within the given timeframe (i.e.: Vehicle Connected Time -> Fully Charged Time).

As these are default settings the app should allow the user to do 'once off' actions also, i.e.: the user could select an immediate fast charge if they needed the vehicle outside their usual usage pattern.
**Dynamic Rate – Why?**
The dynamic setting described above underpins the availability of variable load within the grid, controllable by ESBN. If the charger has three charge rates and the vehicle can be fully charged in the period between Vehicle Connected Time and Fully Charged Time on all three, then ideally the charger would start charging at the mid rate. This would allow ESBN to increase or decrease the charge rate thereby increasing and decreasing grid load according to supply availability.

A number of key elements are required to support adoption of the ‘dynamic’ rate.

- ESBN can only vary the charge rate if it will not impact on availability of the vehicle.
- The ‘variable load’ data should be anonymized by data aggregators and presented to the ESB as summary data on a dashboard.

**Things which will improve the adoption rate of the technology / ‘dynamic’ rate.**

- Incentivised charges
- An expandable platform allowing users to control more devices.

While this solution is dependent on the end user providing broadband access, this is unlikely to hinder the project. Areas where load variability will have the greatest impact are within built urban areas where broadband penetration/reliability is higher. The customer most likely to make use of such technology is also likely to be more technology aware and therefore also have broadband access.

**App Appearance**
The core of the app will consist of a main frame with a status bar and a dynamic sub-frame.

The contents of the sub frame will vary as the user navigates through the app.

See overleaf for a mock up of how the app could look.
Appendix B: Setting up the work environment

There are many ways to set up your PhoneGap project. This will show how it can be done with Google chrome and Eclipse.
In Eclipse I use a plugin called Aptana to write HTML, CSS etc.

Install Aptana
1. Open Eclipse
2. Menu -> Help -> Install Now Software
4. Follow the instructions and restart Eclipse.

Alt. Download the standalone version of Aptana

Add Chrome to Aptana
1. Open Eclipse
2. Menu -> Run -> Run Configurations
3. In the left menu "Web Browser", [left click] -> Add New
4. Browser executable: [find chrome.exe]
   My location: C:\Program Files (x86)\Google\Chrome\Application\chrome.exe
5. Arguments: --allow-file-access-from-files
To run PhoneGap applications you can use an extension for Google chrome called Riddle. This is available at the Google chrome store. With Riddle it is possible to emulate different kind of phone settings in your browser.
Create a helloWorld application (Option 1)

1. Make sure node.js is installed
   Available: http://nodejs.org/

2. Open the command line interface and enter: “npm install -g phonegap”

3. Navigate to your project folder

4. phonegap create helloWorld
   Name your project what you want. I choose helloWorld

5. Eclipse -> import -> Existing Folder as New Project

6. Find the folder and finish

Create a helloWorld application (Option 2)

I do not recommend this option. I have not tried it myself yet.

1. Open Eclipse

2. Create new web project
If you are not in the "web" view chose new project and find the web project option
Option 1 do the following step automatically.
3. Create index.html and config.xml in your root
   These files might have to be in a folder named "www". Not sure
4. Create a JavaScript (js) and css folder
5. Put cordova.js in the JavaScript folder.
   Cordova does not work with Ripple for me so ignore it for now. You can find this file in phonegap-x.x.x/lib/android.

6. Paste the code at the end of this document to index.html and config.xml
   You might have to tweak the index to find cordova.js, and any other JavaScript files

**Running helloWorld**

1. Open Eclipse
2. Chose the project in the left menu
3. Run as: Chrome
   "Run as" is the green and white play button at the top. Press the arrow down and select Chrome
4. Chrome should open. Click the Riddle icon at the top right of your screen
5. Click "Enable"

   The emulator should start and you should see something similar to the picture above
Appendix C: Activity Diagram