



ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY
OF CRETE

**TECHNICAL UNIVERSITY OF
CRETE**
**SCHOOL OF ELECTRICAL AND
COMPUTER ENGINEERING**

User Interface development for
the estimation of electric power
demand of plug in electric
vehicles through digital maps and
statistical data of traffic

Stefanos Lintas 2009030004

Thesis committee

Kanellos Fotios (supervisor)
Stavrakakis Georgios
Koutroulis Ephtichios

Χανιά , Οκτώβριος 2021

Acknowledgements

This thesis was written under the supervision of Prof. Fotios Kanellos for whom I am thankful for his guidance and the chance to work on an interesting subject that incorporates my personal interests as well.

Table of contents

CHAPTER: 1 INTRODUCTION	6
1.1 GENERAL:	6
1.2 AIMS AND OBJECTIVES:	7
1.3 RELATED WORK:	8
1.4 THESIS STRUCTURE	9
CHAPTER: 2 BACKGROUND	11
2.1 ELECTRIC VEHICLES	11
2.2 TYPES OF EVs	11
2.3 ADVANTAGES AND DISADVANTAGES	12
2.4 PEV CHARGING STRATEGIES	13
2.5 PEV CHARGING MODES	14
2.5.1 Mode 1: Slow charging (AC)	14
2.5.2 Mode 2: Slow charging (AC) with safety	14
2.5.3 Mode 3: Slow to semi-fast charging (AC)	15
2.5.4 Mode 4: Fast charging (DC)	15
2.5.5 Wireless charging	15
CHAPTER: 3 DATABASE	16
3.1 WE WILL SET THE FUNDAMENTALS OF OUR DATABASES (DBs)	16
3.2 DEFINING OUR DATABASE AND OUR SCHEMA	17
3.3 METHOD TO IMPORT/EXPORT DATA	20
3.4 REPRESENTATIONAL STATE TRANSFER/SIMPLE OBJECT ACCESS PROTOCOL APIs	20
3.5 INITIAL DATA	21
3.6 DB EXTENSION COST	21
CHAPTER: 4 TECHNOLOGIES	22
4.1 OUTLINE OF THE TECHNOLOGIES	22
4.1.1 Apache Tomcat	22
4.1.2 HTML and CSS	23
4.1.3 Rest API	23
4.1.4 Java	24
4.2 SOFTWARE ARCHITECTURE	25
4.2.1 Maven	25
4.2.2 Boundary Layer	26
4.3 UI COMPONENTS	27
4.4 GOOGLE MAPS	27
4.5 MOBILE PHONES VS PERSONAL COMPUTERS (PC)	28
CHAPTER: 5 IMPLEMENTATION	29
5.1 NETBEANS	29
5.2 JAVA PROJECT OBJECT MODEL (POM)	29
5.3 PARENT POM	29
5.4.1 CONTROL LAYER POM	30
5.4.2 CONTROL LAYER – CONTROLLER AND PROPERTIES	30
5.5.1 VIEW AND COMPARTMENTALIZATION	33
5.5.2 ESSENTIAL JAVASCRIPT FUNCTIONS	35
5.6.1 MODEL LAYER POM	41
5.6.2 MODEL LAYER ABSTRACT CLASS	42
5.7.1 PERSISTENCY LAYER POM	44
5.7.2 PERSISTENCY LAYER INTERNAL STRUCTURE	44
5.8 DATA AGGREGATION: PROCESSING AND VISUALIZATION	47
5.8.1 Data gathering and aggregation	47
5.8.2 Aggregated data processing	48

5.8.3 <i>Excel Calculations</i>	51
5.8.4 <i>Chart creation and data visualization</i>	53
5.9 DATA USED TO CREATE OUR DATA SET	55
5.9.1 <i>Probability density for departure time and time spent idle</i>	55
5.10 OPTIMIZATION FOR TRIP CREATION	56
5.10.1 <i>Generating trips.</i>	57
CHAPTER 6 FLOWCHARTS	59
6.1 CODE FLOW DURING TRIP PLANNING	59
6.2 CODE FLOW DURING GRAPH GENERATION	59
6.3 CODE FLOW DURING EXCEL GENERATION	60
CHAPTER 7 CONCLUSIONS AND FUTURE DEVELOPMENT	61
7.1 CONCLUSIONS	61
7.2 FUTURE DEVELOPMENT	61
REFERENCES	63

Figure 1 - Database schema	19
Figure - 2 Rest API example	25
Figure - 3 MVC Architecture	27
Figure 4 - Snapshot from Google Maps UI , depicting a marked/pinned location on the map	29
Figure 5 - Parent .POM	30
Figure 6 - Control Layer POM	31
Figure 7 - Actions.Properties	31
Figure 8 - controller.java	33
Figure 9View Folder-Structure	34
Figure 10 – Head DOM Element: Contains external and internal resources of VIEW	35
Figure 11 - Backbone-Body DOM Element. Made to segment each part of the page. (header.jsp, footer.jsp and the content div where we load segmented views)	36
Figure 12Ajax function in JavaScript	37
Figure 13 - EVModels function in JavaScript	38
Figure 14 - Welcome Page split into header, body, footer	39
Figure 15 – map_controller with its constructors and auxiliary functions	40
Figure 16 - Restricted Autocompleted Destinations	40
Figure 17Calculation results	41
Figure 18 – Additional info for the selected EV	41
Figure 19 - Interactive Data Table Graph	41
Figure 20Model Layer POM	42
Figure 21 - ModelMVC abstract class definition	43
Figure 22 - Post Update Implementaion	44
Figure 23 - Persistency Layer POM	45
Figure 24 - DAOFactory MySQL implementation	46
Figure 25 - UserDAOImpl of signin	47
Figure 26 - UI full filters list	49
Figure 27 - process of DownloadExcel, creating a directory, gathering the filters, contacting the db and writing the file	50
Figure 28 - Dispatch part of DownloadExcel.java configuring the response	51
Figure 29 - Download Prompt for the filtered data in a .xlsx file	51
Figure 30 - Username general information sheet	52
Figure 31 - Aggregate Information sheet	52

Table 1 - 1 Source: Statista (business data provider) Percentage of EVs among all vehicles	6
Table 2 - Example Table of an SQL Data base	16
Table 3 - Two tables of an SQL database with primary/foreign key reference	17
Table 4 - Monetary value of electric power per hour of day example	21
Table 5 - Example of outputs between pure HTML and HTML with css	23
Table 6 - Probability density: Departure Time for each hour of day and type of trip	55
Table 7 - Probability density: Time spent dwelling in minutes for each type of trip	56

Chapter: 1 Introduction

1.1 General:

In recent years there has been a push throughout the world to change from fossil fuels to renewable forms of energy such as wind and solar, due to the finite nature of fossil fuels and their impact on our world and natural environment. This situation has encouraged not only governments but also industries around the world to move towards clean energy, including car manufacturing.

Europe has slowly started to embrace the use of electric vehicles and is trying to incentivize their widespread usage in the context of its broader ambitious plan to reduce the carbon footprint of the whole continent. The set target of the European Union is to achieve a 40% reduction of greenhouse gasses emissions by 2030 compared to 1990. As an integral part of this directive, the plug in vehicles have steadily increased their market share through the recent years.

This table shows the market shares in Europe for the respective types of battery electric vehicles and Plug-in hybrid electric vehicles

	Battery electric Vehicles	Plug-in hybrid electric vehicles
2017	0.7%	0.7%
2018	1%	0.8%
2019	2.1%	1.2%
2020	4.5%	4.7%

Table 1 - 1 Source: Statista (business data provider) Percentage of EVs among all vehicles (Statista, n.d.)

As it can be seen in table 1-1, the market share of electric vehicles has risen from 0% to 9.2% in 4 years and is predicted to reach 28% by 2030. Considering that in Europe the total number of vehicles is circa 312.7 million that predicted target would equate to 87 million electric vehicles. It should also be noted that among those targets of the clean energy directive, there is one that phases out conventional vehicles completely by 2050.

At this point we should clarify that although electric vehicles have zero direct emissions and impact on the environment while being operated, their manufacturing process has. Should the vehicles be fueled solely by clean and renewable energy sources then that is considered to be a worthy tradeoff,

since electric vehicles have reduced “overall” emissions during their lifetimes, leading to a net negative in the greenhouse emissions equation long term.

Despite relying currently on a predominantly fossil fuel based power supply throughout Europe with not enough clean energy production, it is still worth considering an increased usage of electric vehicles, especially in urban areas. This is due to the fact that the generation of harmful gasses is shifted from urban centers with high population density to ones that are more rural. Where it effects only the people working in the power plants, while this is obviously not ideal it is a good stopgap solution for the time being as well as the immediate and obvious improvement of the air quality in urban centers.

Some governments (e.g. Norway, Sweden and France) have established a very aggressive policies to promote the use of electric vehicles and achieve those goals or even surpass them. The Norwegian and French governments have even announced the banning of internal combustion engine (ICE) sales by 2025. United Kingdom is set to follow by 2040 and within a few decades, the rest of the EU will follow suit.

This considerable transition from conventional vehicles to electric powered ones will have a substantial impact on the greenhouse gasses emissions from cars but will also put a lot of strain on the power grids of every country.

1.2 Aims and objectives:

The primary objective of this thesis is to create a tool that can estimate the impact from a large fleet of electric vehicles. This will be done through the development of a UI that uses Google Maps to achieve two objectives.

The primary objective of this thesis is to create a tool that can estimate the kWh spent from a large fleet of electric vehicles. The second objective is to provide the ability to use real world data through an appropriate UI and Google Maps, to enable individuals to see their own expenditure and provide them with useful information such as their total kWh consumption and statistics on their use of EVs.

To make things more transparent, our stated primary objective has been broken down into several individual goals which cumulatively will fulfill our primary goal.

1. Prepare a database that can effectively gather the appropriate data for individuals or families, their vehicles and their usage.

2. Prepare algorithms to receive that data as input and generate an output that can categorize the users or in line with their daily activities and their expenditure.
3. Use the aforementioned outputs to calculate the aggregate impact of a car fleet.
4. Create a UI that will allow for the planning of a trip and deriving its expected power demands. serving thus as a valuable decision making tool on when they should recharge their vehicles
5. The data provided by the UI will be fully compatible with our database and will be able to be used in future analysis, through the algorithms we created at our first sub goal.

1.3 Related Work:

Several theses have been written and undertaken, trying to estimate the impact of plug in electric vehicles on both the greenhouse gasses emissions and kWh spent by their usage, utilizing various methods. What we hope to achieve with this thesis is to make it easy to understand and comprehend the advantages of EVs.

The most commonly used method is an agent based modeling which simulates a fair load distribution by creating agents that represent large blocks in the population distribution. Those agents govern the decision making process. Another method that has been used is one which estimates loads through reinforced learning of model driving patterns and optimal charging policies. Lastly the use of statistical analysis and representation using real world data provided by government agencies such as the NHTS (National Household Travel Survey) or algorithmically generated data through probability density based on experience and the aforementioned data.

In order to calculate those estimations, some assumptions have to be made. A very important one is that people will charge their vehicles at home during the night. However, as recharging stations become more widespread it is expected that this behavior is likely to change creating the need for a different profile to be taken into account in our simulations and future load estimations.

Great considerations should also be given to the fact that while only Grid-to-Vehicle (G2V) charging strategy is currently available for the electric vehicles, this might not be the case in the future and Vehicle-to-Grid (V2G) charging strategies might become just as widespread. Our load estimations would utilize fuzzy logic to decide whether a vehicle should be charged, at

which point it should be recharged as well as to what extent, since a full recharge might not be the optimal decision.

Parameters such as mileage and electric range will be taken from existing databases and incorporated into our database; from there they will be used for the estimations and simulations for the individual user as well as the fleet of vehicles. Once the tool is completed it can be used in a similar fashion for any country as long as the database is updated with the relevant data. In Greece particularly there are few studies that deal with the subject of PEV (Plug-in Electric Vehicle) penetration which on its own is considered to be at an early stage. For our thesis we will attempt to create a framework which will allow for Greece to be closely examined first and set the foundations to easily check each country in the European Union.

Finally, in order to evaluate the large fleet of electric powered vehicles we will assume that the battery model and electric power demands for unidentified vehicles are the mean average of the known. This will allow us to produce a fairly accurate simulation even for those unknown factors.

1.4 Thesis Structure

For easier comprehension we have divided our thesis structure into individual chapters where we describe the key information that is to be presented. More specifically, we will:

- In Chapter 2:
 1. We will provide some background but fundamental information regarding electric vehicles
 2. Describe the current technologies for recharging PEVs (Plug-in electric vehicles) as well as the most widely used charging methods.
- In Chapter 3:
 1. We will set the fundamentals of our database and define our schema.
 2. Furthermore we will correlate it and integrate it with the existing database of NHTS to obtain real world data which then can be used to generate simulated data for similar cases, not directly associated with a specific location
 3. Lastly we must have appropriate documentation on the database in order to allow for future updates to provide Rest or SOAP APIs support.
- In Chapter 4:

1. We will define the methods, assumptions and technologies to be used in order to achieve our goals.
 2. Furthermore design the Software Architecture for our UI and all of its components.
 3. Defining the components and use cases for our UI elements.
 4. Importing the WAZE/Google Maps APIs.
 5. Defining our target simulations output and its purposes. (such as price, needed initial load estimations and recharging decision making)
- In Chapter 5:
 1. Set a goal for the UI and its expected uses.
 2. Examine our use cases and their contribution towards achieving our set goal.
 3. Provide a UI prototype for those use cases.
 4. Develop the UI prototype into a fully working UI.
 5. Arrange the components of our software architecture to calculate in accordance to our inputs and pass them through the appropriate algorithms.
 6. Validation of a use case and present the expected output of our created tool.
 - In Chapter 6:
 1. Propose possible extensions for our tool.
 2. Examine if providing a REST API to import data into our database for further analysis and comparison.
 3. Present conclusions from our tool and its output and propose areas for future development. Such as cost analysis, ease of use in certain locations.

Chapter: 2 Background

2.1 Electric Vehicles

An electric vehicle (hereafter “EV”) can be defined as any vehicle that depends and runs on electric power. That power can be either provided to a small vehicle/car via a battery or through other means such as in the case of electric powered streetcars/ trams and train where the required power is being provided through the surrounding facilities along their predetermined routes.

EVs are not a new concept and the first practical electric vehicles were produced back in the 1880s but advances in the conventional internal combustion engines and their ease of use, left EVs lagging behind their competition. This is due to the disadvantageous (for that time) cost-benefit analysis ratio for researching and developing a competitive engine that uses electric power only. Another major factor is the availability of electric power; while through our modern “common sense” we view electricity as something widely available everywhere it’s challenging to set up a power grid capable of sustaining a large fleet of EVs even with modern technology. Also EVs can’t penetrate easily rural areas. This relates directly to the previous point made for electric grid distribution capable of handling immense loads, but it also makes the point painfully obvious when compared to the availability of internal combustion engines and fuel supply network, in which case simply transporting gasoline is enough to keep conventional cars functional. Lastly the operational convenience for conventional engines is a big factor and that plays a major role in refueling and recharging techniques and times. When a gas tank is empty you could simply refill it in a couple of minutes while recharging a battery capable of running an EV could take from 15 minutes up to several hours, depending on the charging rate.

The reason that EVs have started coming back into the foreground and their advancement is being expedited despite the drawbacks we saw previously, is that EVs have a reduced impact on the earth’s environment (at least in the long run). This incentive for more eco-friendly technology has led to many breakthroughs for the EVs that made them more competitive in the current market such as the super-fast charging capability which could bring the time needed to recharge an EV’s battery even down to 15 minutes.

2.2 Types of EVs

At this stage, it would be prudent and beneficial to distinguish the various types of EV's as well as those of charging strategies and modes, describing in a few words their main differences, advantages and disadvantages

The **Battery Electric Vehicles (BEV)** is a fully-electric vehicle with rechargeable batteries and no gasoline engine. Battery electric vehicles store electricity onboard with high-capacity battery packs. Their battery power is used to run the electric motor and all onboard electronics. BEVs do not emit any harmful emissions like the traditional gasoline-powered vehicles. BEVs are charged with electricity from an external source.

Hybrid Electric Vehicles (HEVs) are powered by both gasoline and electricity. The electric energy is generated by the car's own braking system to recharge the battery. This is called 'regenerative braking', a process where the electric motor helps to slow the vehicle down and recuperates some of the energy normally converted to heat by the brakes. HEVs start off using the electric motor, then the gasoline engine cuts in as load or speed rises. The two motors are controlled by an internal computer, which targets optimum economy for the specific driving conditions.

Plug-in Hybrid Electric Vehicles (PHEVs) can have their battery recharged through both regenerative braking and "plugging in" to an external source of electrical power. While "standard" hybrids can (at low speed) go about 1-2 miles before the gasoline engine turns on, PHEV models can go anywhere from 10-40 miles before their gas engines are required to engage and provide assistance.

2.3 Advantages and Disadvantages

There are quite a few important advantages that EVs have over conventional internal combustion engines. Those advantages are:

1. They are eco-friendly as they do not emit CO₂. If the energy that is used as fuel to recharge their batteries comes from renewable resources, then their use will have truly zero emissions. Even if the electricity they use is produced by fossil fuels, the pollution will be concentrated around the power plants and away from high population density urban areas.
2. Their maintenance costs are lower because ancillary requirements related to ICE such as lubrication of the engines are not needed.
3. Limited noise pollution is another prime benefit and advantage, since electric motors are much quieter. This is especially significant in urban

environments because of the substantial number of vehicles being present there

4. Their driving is considered to be easier. Commercial electric vehicles have an automatic transmission with one long gear. Thus, there is no need for a clutch mechanism and the driver only uses the brake and acceleration pedal enabling him to focus his attention on his surroundings.
5. They can be cost-effective since many governments incentivize the EV ownership through reduced car pricing schemes and registration taxes. They only recharge at special recharge points or at the driver's home from the electricity grid.

Despite the benefits described above, there are many challenges that are delaying the establishment of EVs as the main car type choice:

1. The driving range of EVs is being limited by their battery capacity although this problem is continuously being addressed through latest research and technology. As of now, the most recent EVs with a low price have a range of 80-160 km and some models can reach up to 600km+ range.
2. They still have a higher price than conventional vehicles even when looking at the more affordable brands. This is due to the equipment used and mostly to the batteries which account for a third of the total price. As EV penetration rate rises, the technology used will become mainstream and their price will gradually fall.
3. Their batteries need replacement because of the limited life cycle. Depending on the type and usage, the most recent ones are expected to not degrade below 80% within 8-10 years (100.000 miles).
4. They have longer recharge times compared to the refueling time required for their conventional counterparts (ICE vehicles). Depending on the type of the charging strategy the time required to fully recharge an EV might require a few minutes up to 24 hours.

2.4 PEV charging Strategies

There are three main charging strategies that allow the driver to have varied control of the timing that the charging process starts and one strategy for the driver to sell energy back to the grid:

1. The uncontrolled or “dumb charging”. Due to the low EV penetration there is a scarcity of charging stations and this strategy offers a lot of convenience since it can be done at home without any special preparation or set up. Although the process starts as soon as the EV

is connected to the electric grid and continues until the state of charge (SoC) is fully restored. While convenient this uncontrolled charging puts a lot strain on the network since the bulk of EVs will recharge at the same time and cost more for the owner of the EV since the price of electrical energy fluctuates during the day.

2. The time of use tariff. This strategy divides the day in sections in order to take advantage of the lower price for energy, but again if too many people do this at the same time the load on the power grid will be substantial. Obviously this effect will be exasperated as the EV penetration goes higher.
3. The smart charging strategy. This strategy assumes that a network which will take care of the problems that occur with the “dumb charging” and time-tariff. It can manage to minimize the cost of recharging EVs until they have reached their appropriate SoC that the driver has decided on.
4. The vehicle to grid (V2G). This strategy is an extension of the smart charging in essence. The new assumption made is simply that the EV connected to the grid is also able to send the stored electric powered instead of only receiving. Obviously a network that informs each individual EV just like in the smart charging strategy can do this effectively. This means that V2G will allow owners of EVs to sell electricity back to the grid during peak hours when electricity is at its price peak and buy it back to recharge their own batteries during its lowest price points.

2.5 PEV charging modes

In this section, we'll look at different EV charging modes specified by the International Electro technical Commission (IEC). These four modes are specified in the IEC 61851 standard that deals with electric vehicle conductive charging systems.

2.5.1 Mode 1: Slow charging (AC)

With this mode, the EV is directly connected to a household socket. The maximum current of this mode is 16 A and its voltage should not exceed 250 V with a single-phase system and 480 V in the case of a three-phase network. Mode 1 is the simplest possible charging mode and does not support any communication between the EV and the charge point. This charging mode is prohibited or restricted in many countries.

2.5.2 Mode 2: Slow charging (AC) with safety

Household socket-outlets do not always provide electric power according to the actual standards. Besides, socket-outlets and plugs designed for household applications might not be able to tolerate continuous current draw at the maximum rated value.

That's why connecting an EV to the socket-outlet for a long time with no control and safety functions can increase the risk of electric shock. To solve this problem, specialists developed charging mode 2 that uses a special type of charging cable equipped with an in-cable control and protection device (IC-CPD).

The IC-CPD performs the required control and safety functions. The maximum current of this mode is 32 A and its maximum voltage should not exceed 250 V single-phase or 480 V three-phase. Mode 2 can be used with both household and industrial sockets.

The safety functions of this mode can detect and monitor the protective earth connection. Over-current and over-temperature protection are two other safety functions that mode 2 supports. Moreover, the Electric Vehicle Supply Equipment (EVSE) can perform functional switching as it detects connection to the EV and analyzes its charging power demand.

2.5.3 Mode 3: Slow to semi-fast charging (AC)

This mode utilizes a dedicated Electric Vehicle Supply Equipment (EVSE) along with the EV on-board charger. The AC current from the charging station is applied to the on-board circuitry to charge the battery. Several control and protection functions are employed to guarantee public safety. These include verifying the protective earth connection and the connection between the EVSE and the EV.

Moreover, this mode can adjust the charging current to the maximum current capability of the cable assembly. The maximum current of this charging mode is 250 A with either a 250 V 1-phase or 480 V 3-phase network. It also supports an operational mode compatible with mode 2 where the maximum current is limited to less than 32 A for both 1-phase and 3-phase cases.

2.5.4 Mode 4: Fast charging (DC)

This is the only charging mode that incorporates an off-board charger with a DC output. The DC current is delivered directly to the battery and the on-board charger is bypassed. This mode can provide 600 V DC with a maximum current of 400 A. The high power level involved in this mode mandates a higher level of communication and stricter safety features.

Mode 4 only allows a case C connection, where the charging cable is permanently connected to the charging station.

2.5.5 Wireless charging

Inductive charging works by creating a magnetic-resonance field between a transmitting pad on the ground (which is physically connected to the grid) and a receiving pad on the underside of the vehicle. Wireless signals sent between the vehicle and charging system initiate and stop charging.

High frequencies are used to overcome the air gap and usually the coils from the two sides are tuned to the same resonance frequency for optimal results. The electric power output is about 20kW and the efficiency close to 70%. Some recent research is focusing on integrating wireless chargers on the roads so the vehicle can recharge its battery while moving. While this looks like a very good solution it's fairly wasteful since its effectiveness is 70% and it will have a big impact on the electric grid but will be very convenient for the users if they become widespread.

Chapter: 3 Database

3.1 We will set the fundamentals Fundamentals of the databases (DBs)

There are multiple types of databases such as noSQL, Object-oriented, distributed etc. For the purpose of our thesis we are going to use the standard relational database and more specifically MySQL. The reason for this choice lies to the functionality that we require. Relational databases store data in tables and those tables are made from columns and rows, their intersection is a cell in which data is stored.

COLUMN 1	COLUMN 2
----------	----------

Row 1	Cell 1.1	Cell 1.2
Row 2	Cell 2.1	Cell 2.2

Table 2 - Example Table of an SQL Data base

Each row represents a complete instance of data which is referred to as a tuple, in our example the first tuple would be Row 1 = { Cell 1.1 , Cell 1.2 }. The type of data that is stored in each cell is defined by its column. If for example column 1 is of type integer (int) then the cells 1.1, 2.1 will have an integer, also the columns define the “size” that of data that can be stored. Using int(4) would mean that the cells of that column can hold data up to 4 digits similarly int(11) indicates that data up to 11 digits can be stored etc. There are a lot of types of data that can be stored but the needed knowledge for the next chapter will need the types of:

- Integer: (int) a number without a floating point
- Varchar: can store any sequence of characters
- Timestamp: an instance of type and date (example: '1970-01-01 00:00:01' UTC)
- Float: A number with a floating point (1.001 etc)

Now that we understand how data is stored in a table the next step is to understand how tables connect with each other since there will always be more than one table in a database.

This process is done by using primary keys (PK) and foreign keys (FK). For each table in our database (schema) one or more columns will be the designated primary key. This means that the value of the cells for that column will be unique throughout the table, in case we have more than 1 column as a primary key then their combination must be unique. Foreign keys on the other hand are columns that must always have a correspondence with an existing value of a primary key of another table. This is easier to understand with an example:

Table 1	Column 1 (int 4 PK)		Column 2 (data)
Tuple 1	1		Data 1
Tuple 2	2		Data 2
Tuple 3	3		Data 3
Tuple
Tuple N	N		Data N
Table 2	Column 1 (int 4 PK)	Column 2 (int 4 FK)	Column 3 (data)
Tuple 1	6	1	Data t1
Tuple 2	7	2	Data t2
Tuple 3	8	3	Data t3

Table 3 - Two tables of an SQL database with primary/foreign key reference

As we can see in our example table 2 uses column 2 (FK) to refer to table 1. This means that each row of table 2 will always have a reference to table 1. There are ways to make it possible for table 2 to have a FK and accept null values but that is bad practice that creates problems during the operation of the database. The only null values that should be accepted in any column are those that are not required and even in that case it's common to use a default value instead of null. This is done to avoid issues during data selection since handling null values requires a lot more testing.

Lastly we will refer to the alternatives of data types for future references.

- PK: Primary key
- FK: Foreign Key
- NN: Not null value
- AI: Auto increment. This means that each subsequent insertion in the table will automatically take the last value and add 1
- UN: unsigned value

3.2 Definition of the our database and its schema

The database for the tool we are creating will be done with MySQL which is a relational database (DB). The other contender for our database is PostgreSQL but it will not be selected for our tool. The reasons for selecting MySQL are:

- It is a widespread technology and can be moved to a cloud. Specifically both Google cloud and Amazon WS (Web services) come with support for MySQL.
- It's easy to import data from other sources such as .CSV files.
- Compared to other contenders such as PostgreSQL, MySQL offers great ease of use.
- Its efficiency is better in an "out of the box" set up. Although it should be mentioned that MySQL internally arrays data in B+ tree and is made to only support this specific data structure. While PostgreSQL can support different data structures to increase the efficiency of the database it is an altogether too different subject for the optimizations of the database. Therefore we will use MySQL for the aforementioned reasons since it is more than enough in our case.

At this point for better comprehension it would be prudent to clarify the term *B+ tree*. A *B+ tree* is an *m*-ary tree with a variable but often large number of children per node. *B+ trees* consist of a root internal nodes and leaves. The root may be either a leaf or a node with two or more children. *B+ trees* are categorized by the value of *b* which is the maximum number of children allowed per node while *m* is the number of children of a node.

The tables of our schema will consist of car_model, family, person and trip. The first table car_model will be filled with the relevant data of all existing EVs which we will directly import into our DB. Family will contain data about the family to which each person belongs to, for people who are living alone this will be represented as a one person family. Person and Trip tables have no special circumstances and now we will see each table on each own.

References: (primary key: PK, not null: NN, unsigned: UN, auto increment: AI)

Table: car_model

- id (PK, NN, AI int 11): is the primary key of the table and is the reference point for other tables.
- Manufacturer (NN varchar 45): will represent the name of the manufacturer.
- Model (NN varchar 45): will represent the specific model of the car.
- Range (NN int 11): the integer that represents the range of an EV under ideal conditions in km.
- Efficiency (NN float): energy consumption per km represented in wh/km.
- Charge_type (NN varchar 45): type of charging available for each model
- Batter_load (int 11): the capacity of the battery.

Table: family

- Id (PK, NN, AI int 11): is the primary key of the table and is the reference point for other tables.
- Location (NN varchar 120): the full address of the family.

Table: person

- Family_id (PK, NN int 11): foreign key which references the family each person belongs to.
- Id (PK, NN, AI int 11): is the primary key of the table and is the reference point for other tables.
- Car_id (PK, NN int 11): foreign key which references the car_model each person uses.
- Name (varchar 45): name of the person.
- Surname (varchar 45): surname of the person.
- Age (int 11): age of the person

Table: trip

- id (PK, NN, AI int 11): is the primary key of the table and is the reference point for other tables.

- Person_id (NN int 11): foreign key which references the person.
- From (NN varchar 120): starting location.
- To (NN varchar 120): desired location:
- Starting_time (NN TIMESTAMP)
- Ending_time (NN TIMESTAMP)
- Type (NN varchar 45): Type of trip urban/rural
- Distance (NN int 11): The distance to be traveled measured in kms

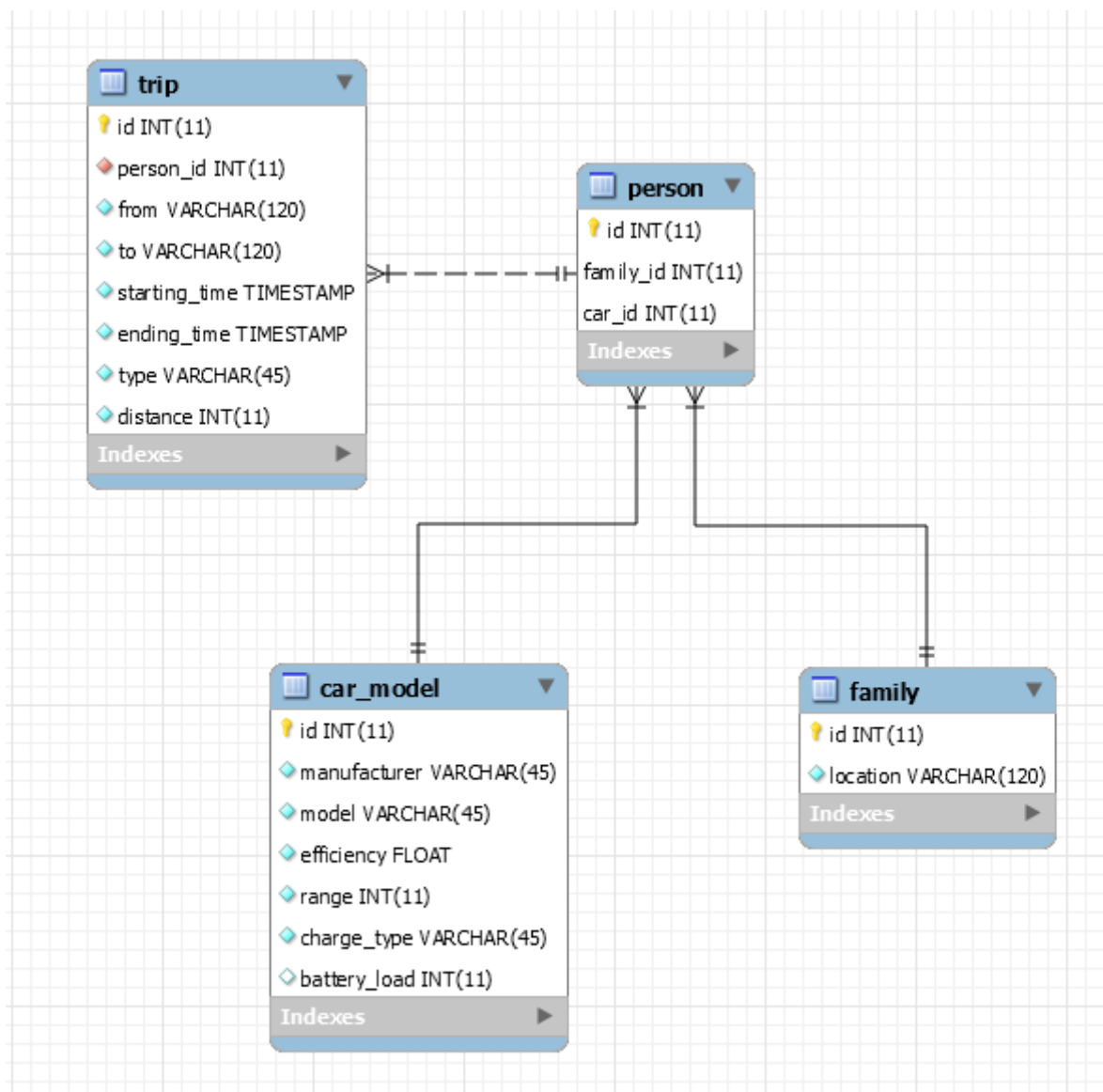


Figure 1 - Database schema

3.3 Method to Import/Export data

There are multiple methods to import data into a MySQL DB, either create a program that reads the .CSV files which will parse them and then import them into the database or use a tool such as HeidiSQL.

The advantage of the 2nd approach lies in the fact that it has a quick and user-friendly interface, making the whole input process more straightforward.

The merits of the 1st approach are not as obvious but it will allow us to set up our own tool in such a way that future import of data would be possible in a remote manner

While an existing tool will provide significant ease, it runs locally. This means that if our database is moved onto a cloud (Google cloud/Amazon WS) it would be no longer usable. The tool we create needs to be able to do it by itself. The same holds true for exporting data as well. While exporting data in a .SQL file can be done through the native libraries of MySQL and then downloaded from the cloud, if in future iterations it needs to export data in other formats then it has to be done in the same fashion as the 1st approach. If it is implemented as described in the 1st approach then we need to set up our server in a way that it can accept files and create the appropriate protocols and UI parts that will allow for its usage, similar to an admin panel of a web page.

3.4 Representational state transfer/Simple Object Access Protocol APIs

Representational state transfer (Rest) and Simple Object Access Protocol (SOAP) are methods through which applications interact with each other. They are not based on the HTTP protocol but are meant for different purposes.

Representational state transfer (Rest) is usually used for web services in order to transfer data mostly to users (business to user). Rest can work with a multitude of formats such as Hypertext Markup Language (HTML), JavaScript Object Notation (JSON), extensible Markup Language (XML) but it is almost always used with the first two. The reason for this is simple; since our UI will display its data and tools to the user in HTML through a web browser and the response of a Rest call can be in HTML, integrating the response into our UI is very simple, a good comparison would be copying pasting a small part of another web application. In the case that data has to be transferred along with the HTML, the data is formatted in JSON. JSON was actually created for this purpose as the preexisting formats (XML) had a bad useful data to useless data ratio and hence reduce efficiency.

Simple Object Access Protocol (SOAP) is a messaging protocol specification for exchanging structured information of web services. SOAP uses the XML format only and is mostly used for exchanging information between businesses. Its advantage over Rest is due to the fact that it provides a solid framework for large data exchanges (business to business). It should be noted that SOAP requires an extensive frame to work and a lot more effort to set up compared to Rest. Each message consists of the envelope (mandatory), header, body (mandatory) and fault. It can be sent and received through both simple message transfer protocol (SMTP) and HTTP but it is typically used with HTTP since HTTP has gained wider acceptance and works well with modern infrastructure.

3.5 Initial Data

Our car_model table in our DB will be initialized with data extracted from <https://ev-database.org/> which is an online database of EVs from the European Union.

In addition should there be a need for it, there is the National Household Travel Survey which can be utilized and is readily available to create profiles of behavioral usage and generate a realistic dataset that we can use to test and show the results of our web application implementation. the National Household Travel Survey (NHTS) is freely available on <http://nhts.orgnl.gov>.

3.6 DB extension cost

In order to be able define the cost of a kW and take it into account when making our tool, it needs to be imported into our database.

The table cost will be created with its primary key set as the country id and a range of hours per day. This will provide us with the framework to create multiple hour-zones for any country and set the appropriate price for a kW. For example:

<i>Id</i>	<i>Hour-zone</i>	<i>cost</i>
GR	00:00-04:00	0.186 €
GR	04:00-06:00	0.182 €

Table 4 - Monetary value of electric power per hour of day example

This example was made with the data from the Hellenic Electricity transmission company. Similar data can be extracted from each country's electric transmission providers as well.

****Note:** Appropriate modifications on the existing schema (Figure 1) will be made as needed for the completion of this thesis.*

Chapter: 4 Technologies

4.1 Outline of the technologies

The creation of our UI will be based on a web application that will make up the full implementation of our tool. The general outline of the technologies that will be used for the first phase are:

- Apache Tomcat
- HTML
- CSS
- Rest API
- Java
- MySQL

Next we will examine the use and the reasons for each of the aforementioned technologies with the exception of MySQL since we reviewed it in the previous chapter.

4.1.1 Apache Tomcat

Apache tomcat will serve as the servlet of our application. A servlet is a piece of software that is responsible for exchanging messages between a client and web applications. Those messages are passed through ports, in our case the port 8080, and read by the servlet and then sent to our application for processing. Servlets also provide other benefits such as thread management. When a messages are received concurrently from clients, our CPU will be under stress since each CPU can support a specific number of threads at the same time. If this is done in a sub optimal manner it can cause the application to crash. Additionally, servlets provide security and are responsible for the advanced secure protocol Hypertext Transfer Protocol Secure (HTTPS) and it is where.

Secure sockets layer (SSL) certificates are stored and checked. HTTPS merely means that the connection is secured; in modern web browsers (such as Firefox) it is denoted by the lock on the left side of each web address. SSL certificates are provided by trusted sources and are used in order to make the messages exchanged unreadable by 3rd parties through cryptography.

The reasons for picking apache tomcat are twofold; firstly it's a free and open source software, secondly it is excellent compared to other free servlets and popular enough to have innate support on cloud based services since it

allows for more personalization such as the ports that will be used and libraries to be included.

As far as security is concerned, the foundations provided by Apache Tomcat will be unused since SSL certificates that are recognized by browsers have to be bought and self-signed SSL certificates are seen as malicious or at the very least suspicious web pages.

4.1.2 HTML and CSS

Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS) will be explained together since they are always used together unless the messages exchanged are strictly text. In order to understand the relation of those two it's easy to think of HTML as the bones of a body and CSS the flesh. This means that HTML is the data that you want to see while CSS is the format in which it is presented.

A simple example of text is presented below:

	Input	Output
Pure HTML	<code> Hello World! </code>	Hello World!
With CSS	<code> Hello World! </code>	<u>Hello World!</u>

Table 5 - Example of outputs between pure HTML and HTML with css

Obviously CSS can do a lot more than simply underlining and making the letters bold. Usually it is done through .css files that include the decorations for all the data we are presenting instead of writing directly on an element of our web page.

Lastly it should be mentioned that there are a lot of ready-made CSS frameworks such as bootstrap which we can use in order to shorten the development phase since creating a truly good CSS framework is work that requires extensive resources and not usually performed by a single person since it would require testing and design on multiple devices (phones, PCs), displays (different resolutions) and environments (Android, Linux, Windows etc). These frameworks take into account the size of each screen and different resolutions on various devices, yielding usually similar or even better quality results in a more efficient and effective manner, compared to what a single source individual developer would.

4.1.3 Rest API

Our UI depends on showing maps and calculating the distance between points on map. This is something far beyond the scope of this thesis and for this reason we will rely on Google Maps. In addition, since tools such Google Maps and Waze currently exist and are readily available, it would be meaningless and wholly inefficient for that matter to create such a tool from scratch. Google Maps will be integrated into our tool through the Rest API that Google provides as a service. Rest API refers to a part of our web application that is imported from an external source. This technology is widely spread and even though we don't realize it, we use it daily. One common example is logging in different web sites through Google/Facebook.

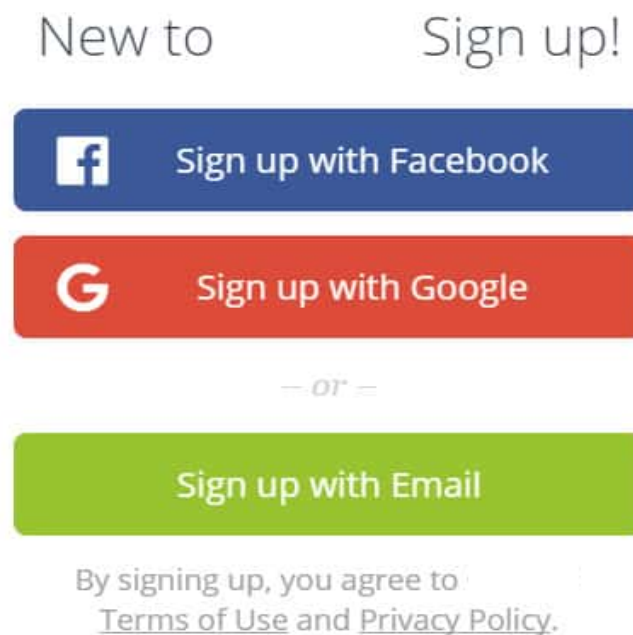


Figure - 2 Rest API example

The first two buttons on this form are essentially messages that will be passed from the application that you want to use to their respective external source. Those sources will then verify you as a user and send a key to the web application which will be used to identify you through it.

A quick Example with Client, Application and Google for better comprehension:

Client A requests to log in at our web application through Google. The Web application calls the relevant Rest API from Google. Once Google answers our application it will pass the message to the client through a dialog window where the client has a direct interaction with Google; the window will then prompt and ask the user if he/she indeed wants to log in. When the Client confirms that, Google will either pass all the relevant data to the web application (to Facebook for example) or a key (minor web applications). At

this point the web application has confirmation and auto creates our account with the data from Google.

4.1.4 Java

Java is software language that is object oriented. That means that it is based around the concept of objects which may contain data, in the form of fields, often known as attributes. The code is mostly written in the form of procedures/methods within objects. The reason for selecting Java is its widespread use which in turn ensures constant support and updates. Another major benefit for using java is the fact that the code can be easily transferred between different terminals and work compared to other languages, something that is actually an exception to the rule.

4.2 Software Architecture

The UI will use Maven (see below) to implement the Model-View-Controller (MVC) architecture and a boundary layer for its connection and exchange of data with the database.

4.2.1 Maven

Maven is a software project management and comprehension tool primarily used in Java projects and it allows for easy and fast deployment and maintenance of the code. Since the MVC architecture (explained in 4.2.2) alone splits our web application up into three separate programs and those programs might have other sub programs, it is a good practice to use a software management tool. Maven contains a .pom file that is essentially an XML format document which states the structure and relations between the multiple programs that it contains. Each program inside the Maven directory has its own .pom files which themselves have data on what programs can “see” each other and their relations. Those .pom files essentially create a tree structure that makes understanding and maintaining the code simple.

4.2.2 Model-View-Controller

The MVC architecture essentially breaks down our web application into 3no. different and distinct parts:

- View
- Controller
- Model

View is responsible for what the client sees, this is the part that has HTML/CSS. The **Controller** receives the requests from the client and then decides which part of the Model can process the request. Lastly, the **Model** is the part that actually does the processing for each request. If data exchange

has to happen in order to process the request of the client, then it uses the boundary layer to read from the database. We will examine how the boundary layer is constructed in section 4.2.2. of this thesis, further below. Once the processing is done, it sends a reply directly through the controller to the client. The other case is that the reply is in simple text in which case View is bypassed.

Furthermore, this division of roles is strict which is not apparent. The controller for example knows only which class in the **Model** is capable of handling a specific request; the controller itself can't process it. The **Model** on the other hand can't receive messages other than those the controller sends, with each part of the model being able to only process a very specific request. This means that for every possible request that the controller can receive the model needs a corresponding java class that can process it.

A simplified picture for the understanding of the MVC architecture:

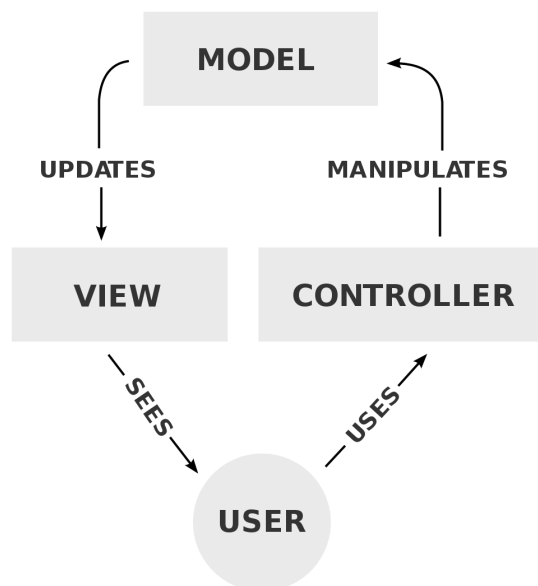


Figure - 3 MVC Architecture

4.2.2 Boundary Layer

The Boundary Layer is an architecture that conceals the database from the web application. Although this sounds at first counter-intuitive, it is a technique which allows to further divide the role of each sub-program in our architecture and simultaneously decouple the processing of data from the managing the database. The reason two main reasons this is worth the extra trouble, firstly it allows us to treat the code as building blocks instead which makes reusing them easy and also makes debugging and understanding the flow of the code easy since you only need to watch the flow of five to six relatively small blocks of code instead of an extremely complicated large one.

As we saw in the MVC architecture, each part has a strict role. The boundary layer provides an interface (the java class type) which allows the Model to receive data from the database without direct access. This is very useful and could be proven beneficial in our case for various reasons but the three important ones are:

1. The database connection can be easily set up through a single object which can be called upon without creating it from scratch every time.
2. If we change the database connection in any way the Model which uses the Boundary layer will not be changed since it can only access the interface of the boundary layer.
3. The boundary layers makes the code clean and easy to maintain and understand.

4.3 UI components

The UI will be composed of a few fundamental components:

1. Header
 - a. Breadcrumbs style directory
 - b. Logging buttons
2. Body
 - a. Imported Frame through the Rest API
 - b. Input fields for the basic data from our users
 - c. Dropdown List with the available EV models
 - d. Divs (HTML component) for displaying the relevant data
3. Footer

The imported frame will be explained in chapter 4.4. The basic data of our users will be auto completed when our users have logged in or manually inserted. The dropdown list will be directly autofilled from the database, although it's much easier to directly put it on the HTML page; this will mean that whenever something is changed on the EV list it will have to be updated on every single page and in the case of importing the list from the database it will only have to be updated once. Lastly the div components are simple boxes for displaying the results of the user's/client's request.

4.4 Google Maps

Google Maps is an application programming interface (API) which allows the importation and usage of the services provided by Google at <https://www.Google.com/maps>, notably: satellite imagery, aerial photography, street maps, 360° interactive panoramic views of streets, real-time traffic conditions, and route planning for traveling by foot, car, air and public transportation.

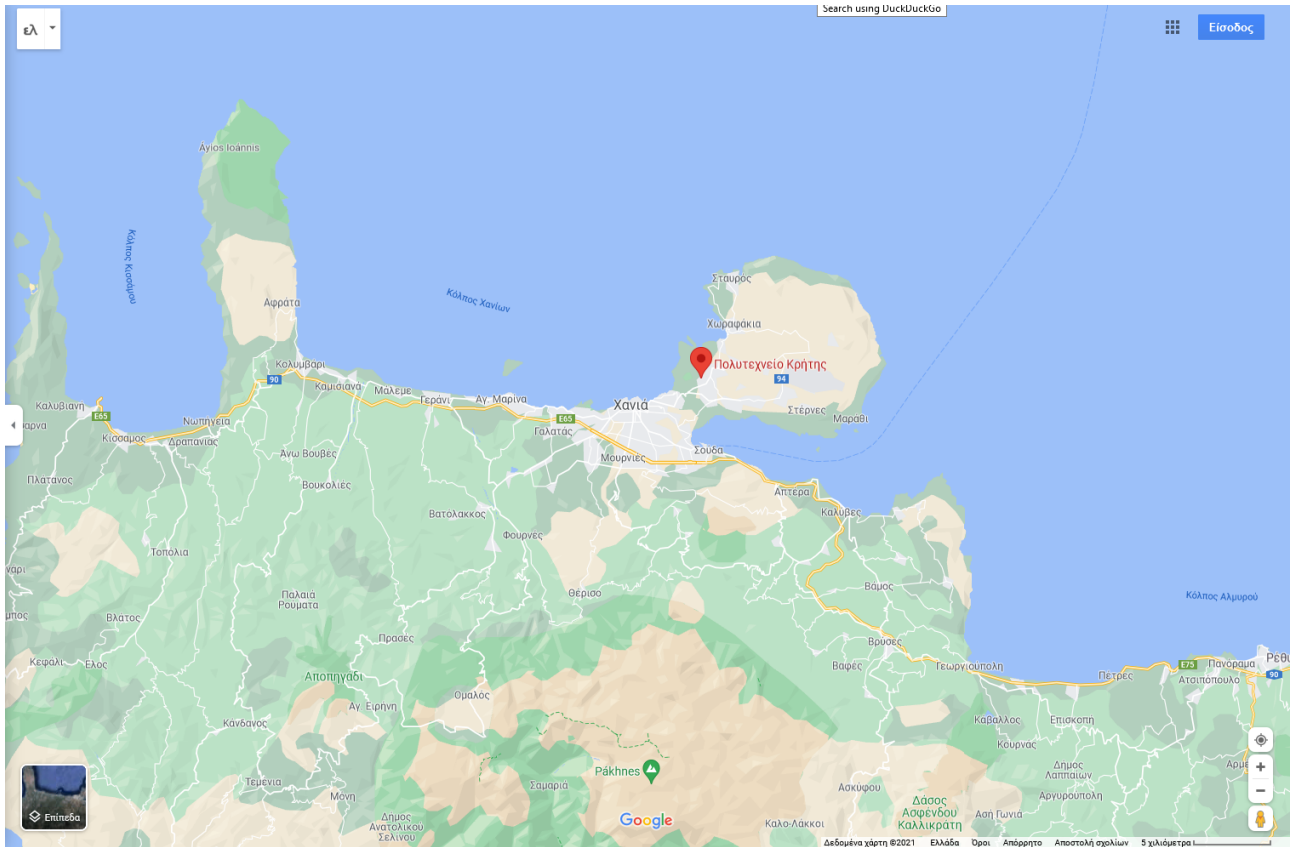


Figure 4 - Snapshot from Google Maps UI , depicting a marked/pinned location on the map

4.5 Mobile Phones vs Personal Computers (PC)

We are going to implement this thesis for a PC environment instead of a mobile application even though the internet use market share has seen constant increase of mobile phones usage since 2001. There are three key points that have led to this decision:

1. Personal computers have greater computational power which will allow our web application to shift some of the processing load on the client

side and make development cheaper. Note that developing and running a web app on a cloud is charged by the number of threads used.

2. Creating the web app in strict accordance to the MVC and boundary layer architecture will make a future development of a mobile app a lot easier since the only part of the web app that needs to change is the **View**.
3. A web application based on a web browser can be used by both PCs and Mobile phones.

Chapter: 5 Implementation

5.1 NetBeans

Our software has been made through the NetBeans integrated development environment (IDE) therefore most of the subsequent screenshots presented below as well as any links for the code, will be viewed through its environment.

5.2 Java Project Object Model (POM)

The anatomy of .POM consists of the below tags:

- Group id
- Artifact id
- Parent id (if it exists)
- Dependencies

The Group ID as the name suggests denotes the group to which the .POM folder belongs to. This grouping allows for easy interdependency and packaging.

The artifact ID is the individual identification for a program within the group.

The Parent ID is the artifact ID of the parent POM

Dependencies are the artifact IDs of the other programs and libraries that need to be packaged with our code in order to produce the executable files.

5.3 Parent POM

As discussed previously within section 4.2, making a large software project can be very challenging and hard to maintain and continuously develop in an efficient manner. Thus a strict structure is required to make the code readable and usable. This is done through the Maven .POM files that declare the sub programs (Modules) that combine to produce our thesis. In our project our parent .POM file is

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6     <groupId>slintas</groupId>
7     <artifactId>Diploma</artifactId>
8     <version>1.0</version>
9     <packaging>pom</packaging>
10    <properties>
11        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
12    </properties>
13    <modules>
14        <module>controlLayer</module>
15        <module>modelLayer</module>
16        <module>persistencyLayer</module>
17        <module>javabeans</module>
18    </modules>
19 </project>

```

Figure 5 - Parent .POM

As seen in the above figure the parent .POM declares that it contains 4 modules (controlLayer, modelLayer, persistencyLayer, javabeans). Note that the parent .POM does not state the relations between its modules but only declares their existence. Their relations are expressed by the .POM of their modules.

5.4.1 Control Layer POM

The control layer project contains the controller, Views and model Layer and it belongs the parent .POM of diploma within its group. This is expressed in its .POM which calls upon the model layer project and denotes its parent's artifact ID.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6     <parent>
7         <groupId>slintas</groupId>
8         <artifactId>Diploma</artifactId>
9         <version>1.0</version>
10    </parent>
11    <artifactId>javabeans</artifactId>
12    <packaging>jar</packaging>
13    <properties>
14        <maven.compiler.source>1.8</maven.compiler.source>
15        <maven.compiler.target>1.8</maven.compiler.target>
16    </properties>
17 </project>

```

Figure 6 - Control Layer POM

5.4.2 Control Layer – Controller and properties

The two main files inside the control Layer are the Actions.properties file and the controller.java. Firstly, the Actions.properties file is in essence a

hash table that has all the acceptable commands for the controller as well as the class that implements them and it looks like this:

```
# Basic welcome page
backbone=slintas.Get.Backbone
home=slintas.Get.Home
# Simple get requests
graphs=slintas.Get.Graphs
# Post requests
update=slintas.Post.Update
postupdate=slintas.Post.PostUpdate
getmodel=slintas.Post.getModel
inserttrip=slintas.Post.insertTrip
# Post json requests
getmodels=slintas.Post.json.getModels
getgraphdata=slintas.Post.json.getGraphData
# Sign in/up handlers
signup=slintas.Sign.SignUp
signin=slintas.Sign.SignIn
logout=slintas.Sign.LogOut
```

Figure 7 - Actions.Properties

For example should the controller receives the command '*GET backbone*' then it will find through the Actions.properties file which class of the model Layer can resolve this request and execute it. This method presents many advantages but the two most important are noted below;

1. It makes it easier to maintain the code since one glance is enough to know which use cases are already implemented and can be used.
2. It makes cross-site scripting/SQL injection a lot harder to do since the controller understands and executes only very specific commands. (Cross site scripting is writing on code on the page/URL directly and then requesting the controller to implement it and SQL injection is the same but instead of attacking the controller it tries to attack the database directly)

Next is the controller.java class which executes the role of the **Controller** in the MVC Architecture.

```

26  @WebServlet(name = "Controller", urlPatterns = {"/Controller/*"})
27  public class Controller extends HttpServlet {
28
29      protected HashMap events = new HashMap();
30
31      @Override
32      public void init() throws ServletException {
33
34          ResourceBundle bundle = ResourceBundle.getBundle("Actions");
35          Enumeration e = bundle.getKeys();
36          while (e.hasMoreElements()) {
37              String key = (String) e.nextElement();
38              String value = bundle.getString(key);
39              events.put(key, value);
40          }
41      }
42
43      @Override
44      public void doGet(HttpServletRequest request, HttpServletResponse response)
45          throws ServletException, IOException {
46          doPost(request, response);
47      }
48
49      @Override
50      public void doPost(HttpServletRequest request, HttpServletResponse response)
51          throws ServletException, IOException {
52
53          String action = request.getPathInfo().substring(1).toLowerCase();
54          String classhandler = (String) events.get(action);
55
56          if (classhandler != null) {
57              ModelMVC model = null;
58
59              try {
60                  model = (ModelMVC) Class.forName(classhandler).newInstance();
61              } catch (ClassNotFoundException | InstantiationException
62                  | IllegalAccessException ex) {
63                  Logger.getLogger(Controller.class.getName()).log(Level.SEVERE, null,
64                      ex);
65              }
66
67              model.process(request, response);
68          } else {
69
70          }
71      }
72
73  }
74
75

```

Figure 8 - controller.java

This java class declares a web servlet with name controller which will capture all requests after the pattern '/Controller/*' (Lines 26). The asterisk (*) is a wildcard which means that any pattern or combination is valid. Next we declare the HashMap type variable named events which is fill from our Actions.Properties file (Lines 31-42). Lastly the two default web servlet functions of doGet and doPost which refer to the HTTP methods. The

important lines are 61 where we used the hash key (command) to load the appropriate class and line 68 where we call the function process of the aforementioned class.

As it can be seen from Figure 8 the controller looks simple and short, but that's precisely how it needs to be since the **Controller** will be called and executed for every single the user takes. As an example the request type of updating the *owned EVs* table at the database will be rarely used therefore even if the code is verbose it is not an issue, but the class implementing the **Controller** will be used hundreds of times more. Therefore it must simply receive the requests finds the corresponding class (Lines 54-55). Furthermore the `Actions.Properties` is put inside the initiation function in order to make sure we call it only once during the start up of our servlet since the **Controller** has to be extremely frugal with the computational actions. Next `ModelMVC` is of the abstract type which allows us to use a single declaration to load and execute all appropriate commands and we will see how it works internally on the next chapter. The remarkable part of this method is that the controller does not know and does not need to know how a request will be resolved it only knows what request it receives and which class is appropriate to resolve it. Additionally because all the implementations of the `ModelMVC` that are responsible for resolving the requests have the `.process` function a single call is enough to proceed.

5.5.1 View and Compartmentalization

Within the controller Layer project there is also the **View** Layer. As previously discussed the View Layer is responsible for presenting data in an appropriate manner. Typically as developers we do not want every view to be directly accessible to the user (client); that is the case for segmented views or for security purposes to disable direct access to Views that contain personal data. Those views are stored in the *WEB-INF* folder which is only accessible through the `controller.java`. The **View** is typically composed of the web pages folder that has by default the sub folders of *META-INF* and *WEB-INF*; in our case we have the addition of a resource folder to further group relevant files.

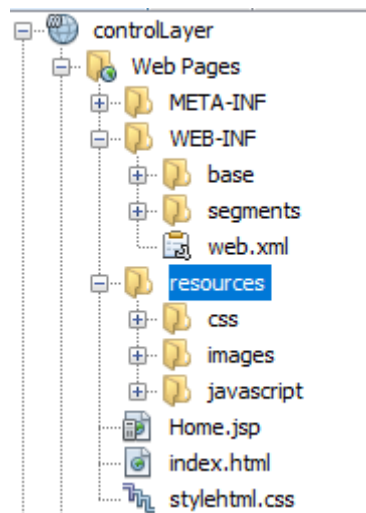


Figure 9 View Folder-Structure

The first file that needs to be understood is *web.xml* which declares some basic information about our project such as the session timer or our welcome page. Typically this file is automatically created when making a new project. Next is *index.html* which is the default welcome page, in our case we have changed the welcome to *Home.jsp*.

The base folder contains the 4 basic parts of our web page and those are:

- Head
- Body
 - Header
 - Body
 - Footer

The head refers to the `<head>` tag of a web page which includes meta-info and the necessary parts for our web-page to work properly. It includes the content type, the relevant cascading style sheets (.css), JavaScript files, Google Maps Embedded Constructors and libraries as well as the title of the page.


```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <link type="text/css" rel="stylesheet" href="../../resources/css/style.css" />

  <script src="../../resources/javascript/ajax_controllers.js"></script>
  <script src="../../resources/javascript/render.js"></script>
  <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>
  <script async defer src="https://maps.googleapis.com/maps/api/js?key=AIzaSyBQwoNgNuemq- z6uhWRNHJUmJ9IrZg&libraries=places"></script>

  <title>ElectricGUIDE </title>
  <link rel = "icon" href = "../../resources/images/logo.png" type = "image/x-icon">

</head>

```

Figure 10 – Head DOM Element: Contains external and internal resources of VIEW

Next is the body, as seen below it contains basic HTML elements as well as a few java commands. The `<%@ include file = "..."%>` are telling the view to load another segmented page and to show it the user as one. This is a good practice since it breaks up the page into parts which can be loaded separately and not reload the whole page after every request. Lastly, the onload trigger event on the body tag at line 11 is calling the initialization function of the JavaScript file to set up and fully load the page. When loading large external libraries (Google Maps in our case) it is good practice not to wait until the whole page has been rendered but instead to do it asynchronously as to provide the user with a smoother experience.

```

7  <%@page contentType="text/html" pageEncoding="UTF-8"%>
8  <!DOCTYPE html>
9  <html>
10     <%@include file="../base/head.jsp" %>
11     <body onload="init()">
12
13         <%@include file="../base/header.jsp" %>
14
15         <div id="blanket" onclick="render.clear();"></div>
16
17         <div id="blanket_info">
18             <div id="blanket_info_header">
19
20             </div>
21             <div id="blanket_info_body">
22
23             </div>
24             <div id="blanket_info_footer">
25
26             </div>
27         </div>
28
29         <div id="content" class="clearfix">
30             Google Maps Is currently loading. If this message persists refresh your browser!
31         </div>
32         <%@include file="../base/footer.jsp" %>
33     </body>
34 </html>

```

Figure 11 - Backbone-Body DOM Element. Made to segment each part of the page. (header.jsp, footer.jsp and the content div where we load segmented views)

Lastly, the segments folder contains segmented Views which can be called through Asynchronous JavaScript And XML (AJAX calls). This means that we will only need to render the whole View once and afterwards simply render the segments we only need to change.

5.5.2 Essential JavaScript functions

There are multiple .js files in our projects but most of them are concerned with View manipulation, error prevention and rendering. The file `ajax_controller.js` though is responsible and acts a pseudo controller on the client side but not in the same sense as the `controller.java` does for our project. `Ajax_controller` is concerned with primarily with three objectives:

1. Communicating with `controller.java`
2. Using client side computational resources instead of the server side
3. Google Maps manipulation

```

269 function ajax() {
270     this.load = function (command, data, format, wait) {
271
272         var xmlhttp = new XMLHttpRequest();
273
274         xmlhttp.onreadystatechange = function () {
275             //alert(xmlhttp.readyState + " " + xmlhttp.status);
276             if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
277                 var info = xmlhttp.responseText;
278                 //alert(info);
279                 if (command === "getModels") {
280                     EVModels.load(info);
281                 } else if ((command === 'signin') || (command === 'signup') || (command === 'logout')) {
282                     window.location.reload();
283                 } else if (command === 'getmodel') {
284                     var temp = document.createElement('div');
285                     temp.id = 'temp';
286                     temp.innerHTML = info;
287                     document.getElementById('currently_owned_EVs').appendChild(temp);
288                     document.getElementById('currently_owned_EVs').appendChild(temp.children[0]);
289                     document.getElementById('currently_owned_EVs').removeChild(temp);
290                 } else if (command === 'postupdate') {
291                     alert(info);
292                 } else {
293                     document.getElementById('content').innerHTML = info;
294                 }
295             }
296         };
297         if (data === '') {
298             xmlhttp.open("GET", command.toString(), !wait);
299             xmlhttp.send();
300         } else {
301             if (format === 'json') {
302                 xmlhttp.open("POST", command.toString(), !wait);
303                 xmlhttp.setRequestHeader("Content-type", "application/json");
304                 xmlhttp.setRequestHeader("Content-length", data.length);
305                 xmlhttp.setRequestHeader("Connection", "close");
306                 xmlhttp.send(data);
307             } else {
308                 xmlhttp.open("POST", command.toString(), !wait);
309                 xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
310                 xmlhttp.setRequestHeader("Content-length", data.length);
311                 xmlhttp.setRequestHeader("Connection", "close");
312                 xmlhttp.send(data);
313             }
314         }
315     };
316 }
317 var ajax = new ajax();

```

Figure 12 Ajax function in JavaScript

As seen above first we declare a function called ajax which handles and sends requests to the controller. The ajax functions parameters are very important:

1. The first parameter is the command which will be sent to the controller
2. the second is the data that will be sent along with the command
3. the third declares the format in which the data will be sent
4. last the wait parameter which decides whether the execution of the ajax called will be synchronously or asynchronously.

Following the execution of the function, the data that has been received will be either further processed by other functions or directly displayed if it is only calling a **View**.

The reason we partly process or store data on the client side instead of the server is to cut down on the processing resources that the server requires. This way even a “low end” PC will be able to provide more computational power than a single server which is divided and used by every user. Next we will see an example of these methods and how they integrate with the rest of the program:

```

319 function EVModels() {
320     var model_list;
321     this.get_model_list = function () {
322         return model_list;
323     };
324     this.load = function (data) {
325         var models = JSON.parse(data);
326         model_list = models;
327
328         var target = document.getElementById('EV_model');
329         for (var i = 0; i < models.length; i++) {
330             var model = models[i];
331             var option = document.createElement('option');
332             option.id = model.id;
333             if (model.variant == "none") {
334                 option.value = model.manufacturer + ", " + model.model;
335             } else {
336                 option.value = model.manufacturer + ", " + model.model + ", " + model.variant;
337             }
338             target.appendChild(option);
339         }
340
341     };
342     this.check = function (element) {...15 lines };
343     this.find_id = function (input) {...12 lines };
344     this.find_id_by_value = function (input) {...13 lines };
345     this.find_model_from_id = function (id) {...14 lines };
346     this.fill_EVModel_info = function (EVModel) {...16 lines };
347     this.clear_EVModel_info = function () {...16 lines };
348     this.mins_to_hours = function (num) {...8 lines };
349 }
350
351 var EVModels = new EVModels();

```

Figure 13 - EVModels function in JavaScript

In order to group up functions that target similar subjects, a variable that will contain the former is made, especially so in the case that storing some data on the client side will make the overall performance increase. For example it can't be expected of our users to be able to remember and know all Variants of every EV and be able to write the model and variant name exactly as stated by the manufacturer. Thusly, once the welcome page loads, we make on the initialization function an ajax call to get the full list of EVs and then fill in an autocomplete list in the fields that require it. Also, if for example our user tries to see a graph and then decides to head back into the main page to see how much he should charge his EV before his trip, the input field will be able to reload the complete list without contacting the server again since the list has been already saved on the client side.

As seen above, the complete *EVModels* function is capable of far more and contains a few auxiliary functions as well such as find the *EVModel* by id and the reverse.

Lastly the third important part of *ajax_controller.js* is the Google Maps Manipulation. This is done in a similar fashion with the function *map_controller* which also is saved as a variable on the client side. Although the differentiation in this case is that it is required by API since it works that way only.

In the following section we will review the welcome Page.

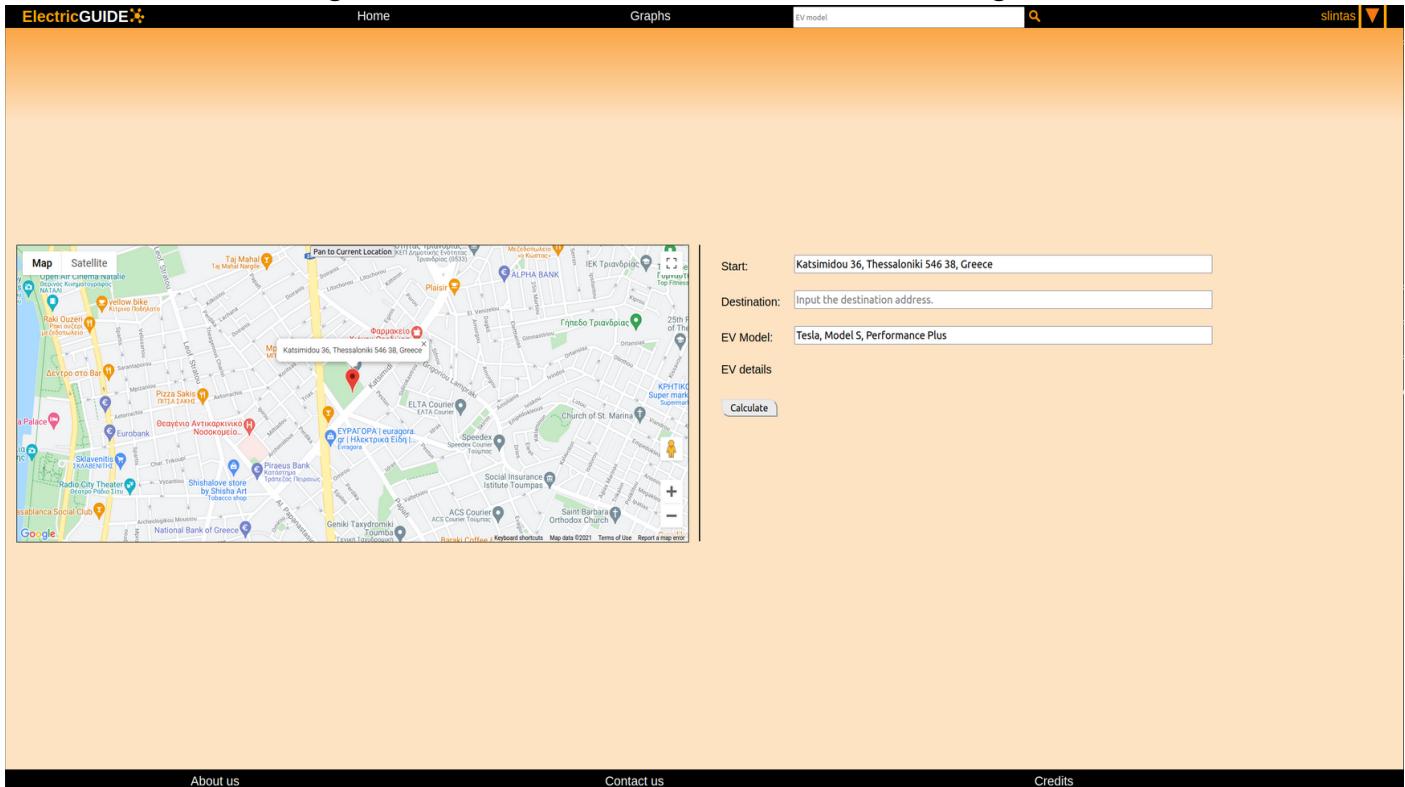


Figure 14 - Welcome Page split into header, body, footer

As soon as our user logs in, *ajax_controller.js* will auto generate the user's EV data and attempt to use the *map_controller* to locate the user. In order to access the location of the user permission is required, if the user does not consent with sharing his location then the **start input** will have to be filled in manually. Additionally both the **start** and **destination** inputs have autocompleted lists of destinations that are asynchronously called. Once our user has filled chosen a destination from the list then he can use the calculate function. The lists of valid inputs are filled in through the Google Maps API and will not constrain the users while ensuring that the input data is valid.

```

9  function map_controller() {
10      let map, infoWindow, geocoder, service, distance, directions, renderer;
11      let curr_latlng, curr_address, curr_dest_latlng, curr_dest_address;
12      this.init = function () {
13          map = new google.maps.Map(document.getElementById("map"), {
14              center: {lat: 35.52999042830537, lng: 24.06861665397316},
15              zoom: 10
16          });
17
18          geocoder = new google.maps.Geocoder();
19          infoWindow = new google.maps.InfoWindow();
20          distance = new google.maps.DistanceMatrixService();
21          directions = new google.maps.DirectionsService();
22          renderer = new google.maps.DirectionsRenderer();
23          renderer.setMap(map);
24
25          this.set_current_location();
26          this.add_current_location_button();
27          google.maps.event.addListener(window, 'load', this.autocomplete_places_destination());
28      };
29
30      this.autocomplete_places_destination = function () {...10 lines};
31      this.on_destination_change = function () {...59 lines};
32      this.draw_line_between_two_points = function () {...18 lines};
33      this.set_current_location = function () {...21 lines};
34      this.add_current_location_button = function () {...10 lines};
35      this.address_from_latlng = function (latlng) {...23 lines};
36  }
37
38  var map_controller = new map_controller();

```

Figure 15 – map_controller with its constructors and auxiliary functions

The names of each sub-function of the function map_controller can provide a pretty clear idea about its purpose. Also, many of them call each other, for example the on_destination_change function will check through the docmplete_places_destination for the validation of inputs. Next we will see the typical use case of a user checking on how much time will be required for recharging the EV in order to complete the trip and cover the desired trip distance, as well as check the details for the EV used on this trip and lastly an interactive data table graph of all EVs in order to see comparisons between them.

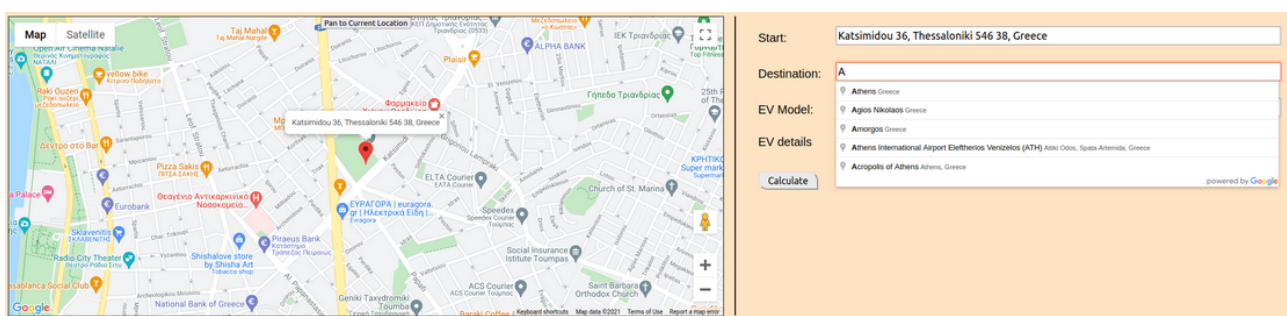


Figure 16 - Restricted Autocompleted Destinations

In order to reduce computational expenses and the API cost, the map_controller will automatically restrict the autocomplete list of destinations within the country of origin. Once the destination is selected, the user can calculate his predicted needed load, duration of trip and distance.

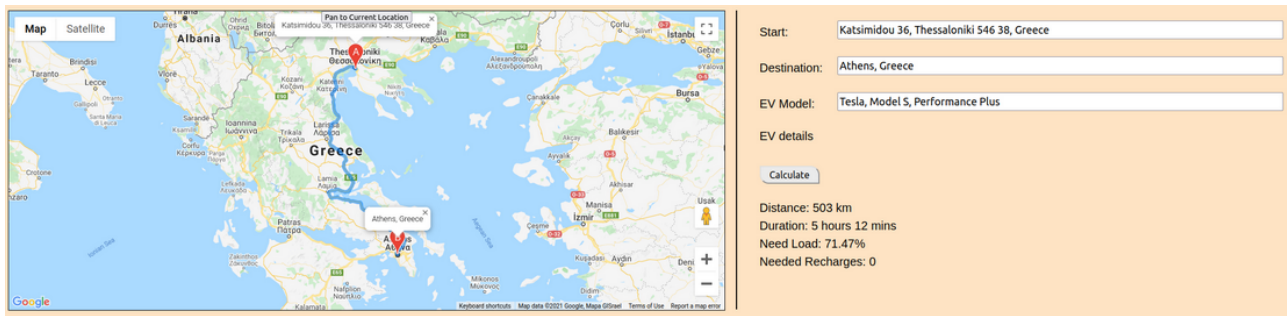


Figure 17 Calculation results

Furthermore, additional details for his/her personal EV can be displayed, note that the **EV Model** input can be changed in order to see a different EV's data and compare:

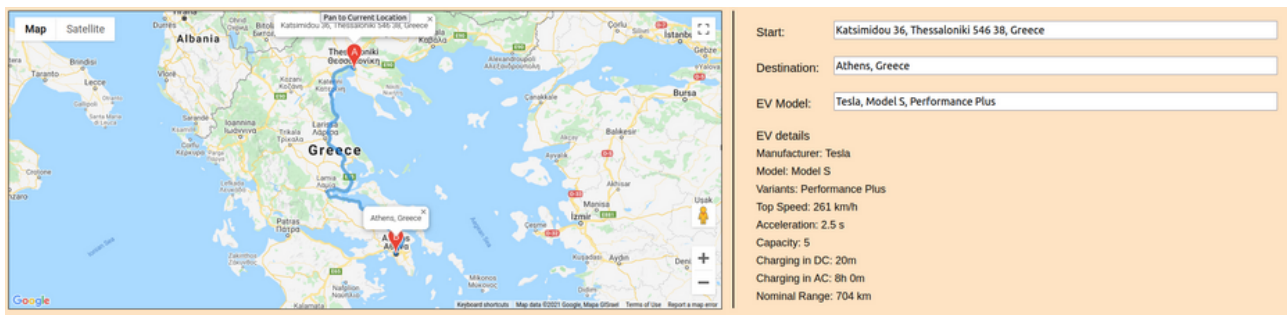


Figure 18 – Additional info for the selected EV

Lastly, the ajax controller will be called once the user calculates his trip and the origin, destination, EV model and user id will be automatically stored in the database. Given enough time and users we can proceed to use this data to produce statistical data which can be used by both us for further research and users that want to get personalized information about their planned trip.

The last important **View** for the user is that of Graphs it provides a way for our users and ourselves to visualize data which will make understanding that data easier. Additionally users can download their personal data in an .xlsx (MS Excel) file and filter it by date, origin, destination, EV Model, distance, duration and username:

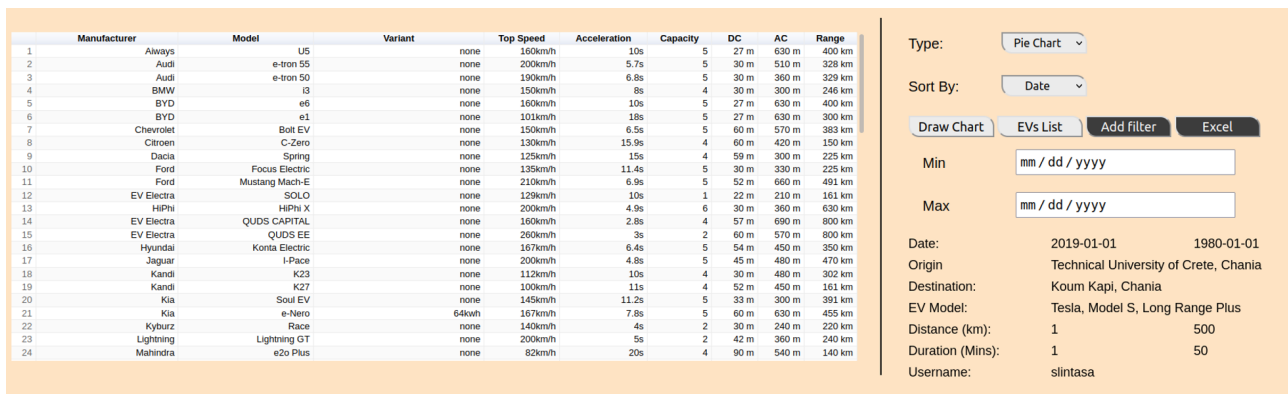


Figure 19 - Interactive Data Table Graph

5.6.1 Model Layer POM

As seen previously in the control Layer, the Model layer's POM contains its parent's ID, group and dependencies. The first dependency *javaee-web-api* is the standard java enterprise edition web API, the fourth dependency *gson* is the standard library to read and convert java classes from and to *JSON*.


```

2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns
3      <modelVersion>4.0.0</modelVersion>
4      <parent>
5          <groupId>slintas</groupId>
6          <artifactId>Diploma</artifactId>
7          <version>1.0</version>
8      </parent>
9      <artifactId>modelLayer</artifactId>
10     <packaging>jar</packaging>
11
12     <dependencies>
13         <dependency>
14             <groupId>javax</groupId>
15             <artifactId>javaee-web-api</artifactId>
16             <version>7.0</version>
17             <type>jar</type>
18         </dependency>
19         <dependency>
20             <groupId>slintas</groupId>
21             <artifactId>persistencyLayer</artifactId>
22             <version>1.0</version>
23         </dependency>
24         <dependency>
25             <groupId>slintas</groupId>
26             <artifactId>javabeans</artifactId>
27             <version>1.0</version>
28         </dependency>
29         <dependency>
30             <groupId>com.google.code.gson</groupId>
31             <artifactId>gson</artifactId>
32             <version>2.8.1</version>
33             <type>jar</type>
34         </dependency>
35     </dependencies>

```

Figure 20 Model Layer POM

As for the second dependency, it is from our project the persistency layer which handles the connection from and to the database. Finally, we have the third dependency JavaBeans which is part of our project and it is used in order to make classes to contain the data that is moved between the projects different components.

5.6.2 Model Layer abstract class

The integral component of the model Layer is the ModelMVC abstract class:

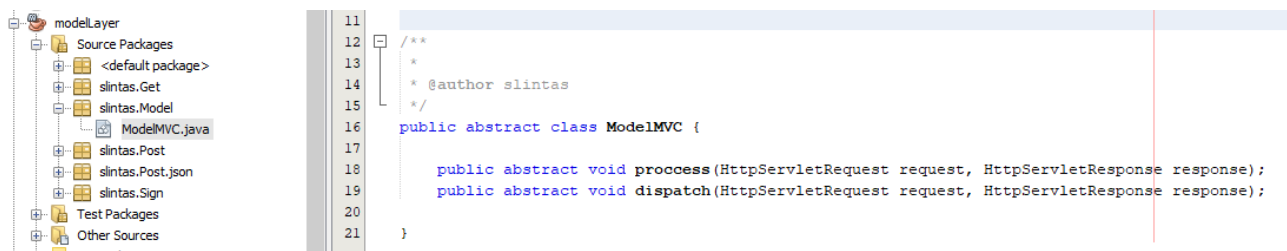


Figure 21 - ModelMVC abstract class definition

This ensures that every class in the model Layer that inherits and is its subclass will also possess and implement, *process* and *dispatch* methods. This restriction in combination with the polymorphism attribute of java classes are precisely what allows our controller to be so simple and elegant.

In the following example we will examine the use case for updating the owned EVs of a user to see the flow of the code. The class name usually consists of the http method followed by a keyword or keywords that make the class's intended use clear. This case will be name *PostUpdate*; *post* means that there is data to read on the server side and update since we are doing an update on the owned EVs of the user. The user's ID is stored in his session and it is the first variable we will have to use. Next, we instantiate the *DAOFactory* interface and get the relevant implementation (details on *DAOFactory* at 6.7). After *DAOFactory* has been used to insert and delete tuples on the database the process function ends and calls the dispatch which will send an answer to the client that the list of EVs has been updated. In the case of updating there is no need to change or manipulate the **View** from the server side thus a confirmation message will suffice but on other cases that dispatch would call upon a **View** to answer the request.

```

25 public class PostUpdate extends ModelMVC {
26
27     @Override
28     public void process(HttpServletRequest request, HttpServletResponse response) {
29         User user = new User();
30         user.setUsername((String) request.getSession().getAttribute("username"));
31         boolean insert, drop = false;
32
33         EVModelsDAO emd = DAOFactory.getMySQLDAOFactory().getEVModelsDAO();
34         List<EVModel> lem = new ArrayList<>();
35
36         int len = Integer.parseInt(request.getParameter("len"));
37         if (len > 0) {
38             for (int i = 0; i < len; i++) {
39                 EVModel em = new EVModel();
40                 String param = request.getParameter(Integer.toString(i));
41                 em.setId(Integer.parseInt(param));
42                 lem.add(em);
43             }
44             insert = emd.insert_models(user, lem);
45         }
46
47         int dlen = Integer.parseInt(request.getParameter("dlen"));
48         if (dlen > 0) {
49             for (int i = 0; i < dlen; i++) {
50                 EVModel em = new EVModel();
51                 String param = request.getParameter("d" + Integer.toString(i));
52                 em.setId(Integer.parseInt(param));
53                 lem.add(em);
54             }
55             drop = emd.drop_models(user, lem);
56         }
57
58         dispatch(request, response);
59     }
60
61     @Override
62     public void dispatch(HttpServletRequest request, HttpServletResponse response) {
63         try {
64             response.setCharacterEncoding("UTF-8");
65             response.getWriter().print("The list of owned EVs has been updated!");
66         } catch (IOException ex) {
67         }
68     }
69
70 }
71

```

Figure 22 - Post Update Implementaion

5.7.1 Persistency Layer POM

As previously seen this .POM has similar components; the only library imported outside of our project is the mysql-connector for java which we use to connect to the database.

```
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>slintas</groupId>
  <artifactId>Diploma</artifactId>
  <version>1.0</version>
</parent>
<artifactId>persistencyLayer</artifactId>
<packaging>jar</packaging>

<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.0.8</version>
    <type>jar</type>
  </dependency>
  <dependency>
    <groupId>slintas</groupId>
    <artifactId>javabeans</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>

<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
</project>
```

Figure 23 - Persistency Layer POM

5.7.2 Persistency Layer Internal structure

There are two integral components in the **persistency Layer**. First the *DAOFactory* an abstract class similar to *ModelMVC*, this time though it has an implementation besides the declaration of the class. This implementation is responsible for connection to the database and it contains the address of the database the possible interfaces that implement the *MySQLDAOFactory* and the function which instantiates a connection.


```

23 public class MySqlDAOFactory extends DAOFactory {
24
25     public EVModelsDAO getEVModelsDAO() {
26         return new EVModelsDAOImpl();
27     }
28
29     public UserDAO getUserDAO() {
30         return new UserDAOImpl();
31     }
32
33
34     public static final String DRIVER = "com.mysql.jdbc.Driver";
35     // public static final String DBURL = "jdbc:mysql://35.195.198.83:3306/diploma";
36     public static final String DBURL = "jdbc:mysql://localhost:3306/diploma";
37
38     public static Connection createConnection() {
39         Connection conn = null;
40         try {
41             Class.forName(DRIVER);
42         } catch (ClassNotFoundException e) {
43             e.printStackTrace();
44             return null;
45         }
46
47         try {
48             conn = DriverManager.getConnection(DBURL, "diploma", "ya9m49$badf4l2NBN");
49         } catch (SQLException e) {
50             e.printStackTrace();
51             return null;
52         }
53
54         return conn;
55     }
56
57     public static DatabaseMetaData GetDatabaseInfo() {
58         Connection connection = MySqlDAOFactory.createConnection();
59         DatabaseMetaData dmd = null;
60         try {
61             dmd = connection.getMetaData();
62         } catch (SQLException ex) {
63             Logger.getLogger(MySqlDAOFactory.class.getName()).log(Level.SEVERE, null, ex);
64         }
65         return dmd;
66     }
67
68 }

```

Figure 24 - DAOFactory MySQL implementation

The benefits for using this roundabout method to implement the database connection are primarily two.

1. First, in case of a data base migration it will not disturb the other sub modules of our project since they won't need to change.
2. Second, it hides what has been done from Model Layer in the same manner that Model Layer hides the implementation from the Control Layer this means that the other modules that call upon this class underneath them will be extremely simplified.

The *EVModelsDAO* and *UserDAO* are interfaces with the advantages as stated previously, the DRIVER is the literal driver used for the connection while the DBURL stands for data base universal resource locator and it's the address of the database.

Next we will examine the example of the sign in process to the database from the *UserDAOImpl*:

```
@Override
public User signin(User user) {
    Connection conn = MySqlDAOFactory.createConnection();
    try {
        User result = new User();
        String query = ""
            + " select users.username, users.password, users_EV.id_EV"
            + " from diploma.users,diploma.users_EV "
            + " where username = ? and password = ? and id_user = username;";

        PreparedStatement ps = conn.prepareStatement(query);
        ps.setString(1, user.getUsername());
        ps.setString(2, user.getPassword());

        ResultSet rs;
        rs = ps.executeQuery();

        if(rs.next()){
            result.setUsername(rs.getString(1));
            result.setPassword(rs.getString(2));
            result.setEv_model_id(rs.getInt(3));
        } else {
            result.setUsername("not found");
        }

        conn.close();
        return result;
    } catch (SQLException ex) {
        return null;
    }
}
```

Figure 25 - UserDAOImpl of signin

As seen in the above code extract, the first action we take in every function of a DAO implementation is to get a connection to the database. Following that, we instantiate the relevant JavaBean that we will use to pass the data between modules. Next, we create the query for the database, grab the results and send them back to the **Model** layer in accordance to the MVC architecture to be dispatched to the **View** and then be presented to our user.

5.8 Data aggregation: processing and visualization

The data that we gather can be processed with various methods on either individual users or individual trips. As our stated objectives at section 1.2 we have:

1. The database that can store and use data for users their EVs and their trips.
2. The UI of the web application can store into the database their planned trips.
3. The users can get recommendations on their predicted expenditure.

In order to fully complete our stated goals the last step is to implement the aggregation of the above. Specifically the ability to filter and draw data from the database not for an individual trip (using an EV to go from point A to point B) but a multitude of trips.

5.8.1 Data gathering and aggregation

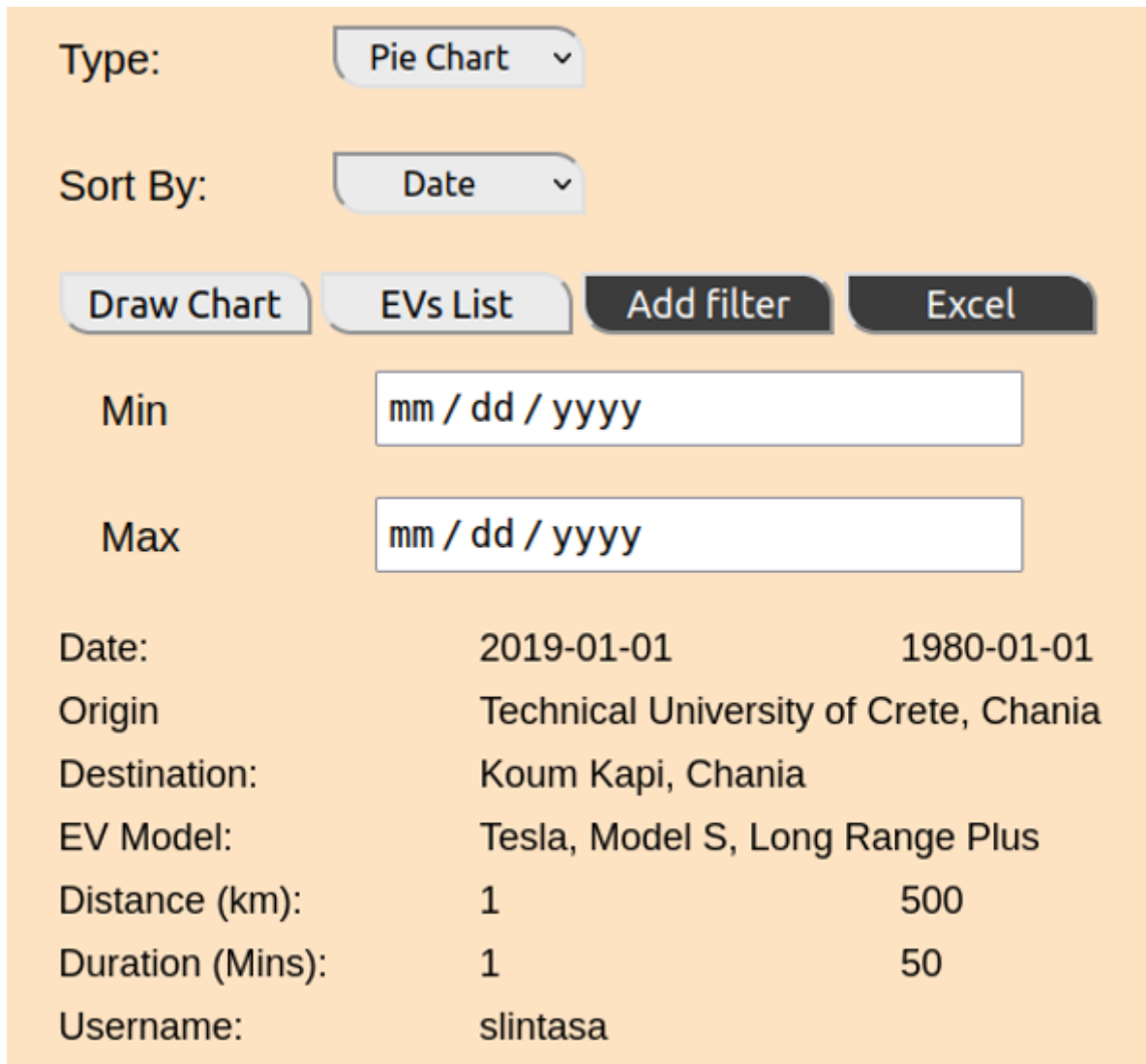
Data aggregation and more specifically gathering large data blocks is simple for our web application since the persistency can handle the gathering of data and packing it in a list in the form of java beans array list to ease the processing further down the code flow. Specifically this is down through the **boundaryLayer** and more specifically the *TripDAOImpl.java* class.

The query to get the data from the database is dynamically constructed using the information from the client side. The client accepts 7 types of filters with min/max values when applicable, those types are :

1. Date (min/max)
2. Destination
3. Origin
4. EV Model
5. Distance (min/max)
6. Duration (min/max)
7. Username

Our user can choose and fill in the values for the filters he/she desires. Note that figure 26 displays an example of a fully filled filter list, the web app when loading the graphs page will have the filter list empty and fill/update its values only after the user adds the filters for visual clarity. Once the filters are

decided *ajax_controller.js* will gather the filters and form them into JSON data that will be sent to the servlet for further processing.



The image shows a web interface for filtering data. It includes two dropdown menus for 'Type' (set to 'Pie Chart') and 'Sort By' (set to 'Date'). Below these are four buttons: 'Draw Chart', 'EVs List', 'Add filter', and 'Excel'. There are two date input fields labeled 'Min' and 'Max', both containing the placeholder 'mm / dd / yyyy'. At the bottom, a table lists various filters and their current values.

Date:	2019-01-01	1980-01-01
Origin	Technical University of Crete, Chania	
Destination:	Koum Kapi, Chania	
EV Model:	Tesla, Model S, Long Range Plus	
Distance (km):	1	500
Duration (Mins):	1	50
Username:	slintasa	

Figure 26 - UI full filters list

Once our user has selected the relevant filters he can either Draw the data or download the results in a .xlsx file.

5.8.2 Aggregated data processing

When the controller receives the request for a filter query he will load the appropriate class from the ModelMVC depending on whether the user wants to display the data (*getGraphData.java*) or download it in a .xlsx file (*DownloadExcel.java*). We will go step by step how the *DownloadExcel.java* works since it contains the *getGraphData.java* with the additional processing for transforming the data into an .xlsx file and sending it back.

```

39  @Override
40  public void process(HttpServletRequest request, HttpServletResponse response) {
41      DirectoryControl dc = new DirectoryControl((String) request.getSession().getAttribute("username") + ".xlsx");
42      try {
43          dc.init();
44          dc.deleteFile();
45      } catch (IOException ex) {
46          Logger.getLogger(DownloadExcel.class.getName()).log(Level.SEVERE, null, ex);
47      }
48
49      EVModelsDAO evmd = DAOFactory.getMySQLDAOFactory().getEVModelsDAO();
50      EVModel evm = evmd.getModel((int) request.getSession().getAttribute("ev_model_id"));
51
52      InputReader ir = new InputReader();
53      String json = ir.inputStreamReader(request);
54
55      Gson gson = new Gson();
56      PreparedStatementConstructor psc = new GsonBuilder().setDateFormat("yyyy-MM-dd hh:mm:ss").create()
57          .fromJson(json, PreparedStatementConstructor.class);
58
59      try {
60          psc.prepareQuery();
61      } catch (ParseException ex) {
62          Logger.getLogger(DownloadExcel.class.getName()).log(Level.SEVERE, null, ex);
63      }
64
65      TripDAO td = DAOFactory.getMySQLDAOFactory().getTripDAO();
66      ArrayList<Trip> alt = td.getTrips(psc);
67
68      if (alt.isEmpty()) {
69          return;
70      }
71
72      alt = td.getTrips(psc);
73      //String file, String username, EVModel evm, ArrayList<Trip> alt, PreparedStatementConstructor psc
74      ExcelWriter ew = new ExcelWriter(dc.getExcelName(), evm, alt, psc);
75      try {
76          ew.full();
77      } catch (IOException ex) {
78          Logger.getLogger(DownloadExcel.class.getName()).log(Level.SEVERE, null, ex);
79      }
80      dispatch(request, response);
81  }

```

Figure 27 - process of DownloadExcel, creating a directory, gathering the filters, contacting the db and writing the file

Once the *DownloadExcel.java* is called we first create a directory. The directory's path is the one that is currently running our code which will be the location of the servlet (Apache tomcat section 4.1.1). The file's name will be made from the user's username which we can access through the session. Once created we do a check if the file already exists, if it does then we delete it. This is done to avoid issues of having multiple files that simply waste space and trying to write over a preexisting file and causing an *IOException*. Next we read the filters sent by the users and transform them from JSON data to javabeans using the *Gson* class. Note that the JSON data are directly converted to the *PreparedStatementConstructor* class which takes the filters sent by the user and by calling the *.prepareQuery()* method it construct the query needed. Once our query is ready we can contact the database through the data access objects (DAO), in this case *TripDAO* and store the results in an array list of trips. The next step is checking whether the list is empty, since

if it is, it will cause errors by trying to print the nonexistent. Finally we call the `ExcelWriter` class to print the results in a `.xlsx` file and proceed to dispatch the request.

```
83  @Override
84  public void dispatch(HttpServletRequest request, HttpServletResponse response) {
85      DirectoryControl dc = new DirectoryControl((String) request.getSession().getAttribute("username") + ".xlsx");
86      try {
87          dc.init();
88      } catch (IOException ex) {
89          Logger.getLogger(DownloadExcel.class.getName()).log(Level.SEVERE, null, ex);
90      }
91      String fileName = dc.getExcelName();
92      String contentType = "application/ms-excel";
93
94      byte[] file = null;
95      try {
96          file = Files.readAllBytes(Paths.get(fileName));
97      } catch (IOException ex) {}
98      Logger.getLogger(DownloadExcel.class.getName()).log(Level.SEVERE, null, ex);
99
100
101      response.setHeader("Content-disposition", "attachment;filename=" + fileName);
102      response.setHeader("charset", "iso-8859-1");
103      response.setContentType(contentType);
104      response.setContentLength(file.length);
105      response.setStatus(HttpServletResponse.SC_OK);
106
107      OutputStream outputStream = null;
108      try {
109          outputStream = response.getOutputStream();
110          outputStream.write(file, 0, file.length);
111          outputStream.flush();
112          outputStream.close();
113          response.flushBuffer();
114      } catch (IOException e) {
115          throw new RuntimeException(e);
116      }
117  }
```

Figure 28 - Dispatch part of `DownloadExcel.java` configuring the response

Once we start the dispatch we call upon the `DirectoryControl` class again to target the file we just created from the dispatch side. Next we configure the response, since we are sending a file which can be rather large and not simple text we need to transform our file into an array of bytes. After the conversion into a byte array we set the necessary attributes such as charster and content type and proceed to send it back to the user. When sent `ajax_controller.js` will read the byte array transform it back to an `.xlsx` file and prompt the user to download it

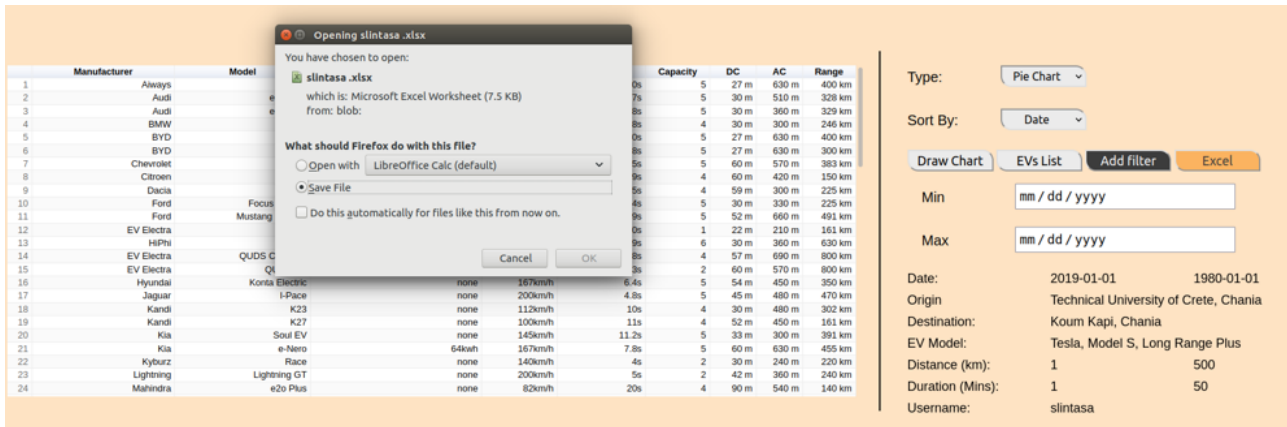


Figure 29 - Download Prompt for the filtered data in a .xlsx file

In the case of `getGraphData.java` we would dispatch and display the data after getting them from the database.

5.8.3 Excel Calculations

The excel file is split into 3 separate sheets. This was done for clarity since the sheet of the trips alone will contain many rows of data and ease of use. The first sheet contains the user's info and EV model details (Username General information), the second sheet contains the aggregate information and the last sheet has the trips in details.

1	Username:	slintasa
2	Manufacturer:	Stevens
3	Model:	ZeCar
4	Variants:	none
5	Top Seed (km):	90
6	Acceleration (100km/h):	20
7	Capacity:	5
8	Charging DC (mins):	30
9	Charging AC (mins):	180
10	Nominal Range (km):	80

Figure 30 - Username general information sheet

1	Starting Date:	2021-01-01 00:00:00.0
2	Ending Date:	2022-02-01 00:00:00.0
3	Total Distance(km):	583
4	Total Time spent driving:	9 Hours 43 Minutes
5	Total iddle Time:	194 Hours 38 Minutes
6	Total kwh used:	116,6
7	Total optimal load need (%):	728.75%
8	Total optimal recharges:	7
9	Total realistic load need (%):	910.94%
10	Total realistic recharges:	9

Figure 31 - Aggregate Information sheet

The displayed information has the starting and end date so that the user can see the timeframe that corresponds with the data. Next the sum of the distance he has covered in the trips within that timeframe as well as the time spent driving. The total idle time refers to the time spend not driving after leaving home (work, shopping, socializing). An estimation of the kWh spent which is done by using either averages or specific EV model data. Lastly we see the charge that was needed to cover the distance sum. Optimal load refers to an optimistic calculation that uses the nominal range as stated by the manufacturers at face value, while the realistic approach estimates that the real nominal range of an EV is about 80% of the optimal.

1	Origin	Destination	Datetime	Distance	Duration	Optimal Needed Load	Optimal Recharge Needed	Realistic Needed Load	Realistic Recharge Needed	Trip type
2	Profti Ila 110, Chania	Police Station, Leoforos Soudas, Souda, Chania	2022-01-01 17:00:11.0	6	11 1.5%			0 1.88%		0 Shopping, Social
3	Police Station, Leoforos Soudas, Souda, Chania	Police Station, Leoforos Soudas, Souda, Chania	2022-01-01 18:00:11.0	0	0 0.25%			0 0.31%		0 Work
4	Police Station, Leoforos Soudas, Souda, Chania	Profti Ila 110, Chania	2022-01-02 02:00:22.0	6	11 1.5%			0 1.88%		0 Home
5	Profti Ila 110, Chania	Church of Evangelistria, Chalepas, Chania	2022-01-02 08:00:02.0	1	2 0.25%			0 0.31%		0 Shopping, Social
6	Church of Evangelistria, Chalepas, Chania	East Moat Theater Parking, Kiprou, Chania	2022-01-02 09:00:09.0	2	7 0.5%			0 0.63%		0 Shopping, Social
7	East Moat Theater Parking, Kiprou, Chania	Police Station, Leoforos Soudas, Souda, Chania	2022-01-02 10:00:21.0	5	12 1.25%			0 1.56%		0 Shopping, Social
8	Police Station, Leoforos Soudas, Souda, Chania	Chania International Airport Ioannis Daskalogiannis	2022-01-02 13:00:37.0	13	16 3.25%			0 4.06%		0 Shopping, Social
9	Chania International Airport Ioannis Daskalogiannis	Church of Evangelistria, Chalepas, Chania	2022-01-02 14:00:56.0	11	13 2.75%			0 3.44%		0 Work
10	Church of Evangelistria, Chalepas, Chania	Profti Ila 110, Chania	2022-01-03 00:00:51.0	1	1 0.25%			0 0.31%		0 Home
11	Profti Ila 110, Chania	Cruise terminal, Agiou Nikolaou, Souda, Chania	2022-01-03 07:00:11.0	6	11 1.5%			0 1.88%		0 Work
12	Cruise terminal, Agiou Nikolaou, Souda, Chania	Cruise terminal, Agiou Nikolaou, Souda, Chania	2022-01-03 12:00:11.0	0	0 0.25%			0 0.31%		0 Shopping, Social
13	Cruise terminal, Agiou Nikolaou, Souda, Chania	Cruise terminal, Agiou Nikolaou, Souda, Chania	2022-01-03 13:00:11.0	0	0 0.25%			0 0.31%		0 Work
14	Cruise terminal, Agiou Nikolaou, Souda, Chania	Profti Ila 110, Chania	2022-01-03 22:00:21.0	6	10 1.5%			0 1.88%		0 Home
15	Profti Ila 110, Chania	SeaStudios, Mesologgiou, Chania	2022-01-04 18:00:06.0	2	6 0.5%			0 0.63%		0 Shopping, Social
16	SeaStudios, Mesologgiou, Chania	Profti Ila 110, Chania	2022-01-04 19:00:13.0	3	7 0.75%			0 0.94%		0 Home
17	Profti Ila 110, Chania	Chania International Airport Ioannis Daskalogiannis	2022-01-05 07:00:11.0	10	11 2.5%			0 3.13%		0 Shopping, Social
18	Chania International Airport Ioannis Daskalogiannis	Profti Ila 110, Chania	2022-01-05 10:00:22.0	9	11 2.25%			0 2.81%		0 Home
19	Profti Ila 110, Chania	Loutraki Beach, Akrotiri	2022-01-06 07:00:17.0	12	17 3.0%			0 3.75%		0 Shopping, Social
20	Loutraki Beach, Akrotiri	Loutraki Beach, Akrotiri	2022-01-06 12:00:17.0	9	0 0.25%			0 0.31%		0 Work
21	Loutraki Beach, Akrotiri	Profti Ila 110, Chania	2022-01-06 17:00:34.0	12	17 3.0%			0 3.75%		0 Home
22	Profti Ila 110, Chania	Nea Hora Chania Marina, Chania	2022-01-07 06:00:14.0	5	14 1.25%			0 1.56%		0 Work
23	Nea Hora Chania Marina, Chania	old market chania	2022-01-07 16:00:21.0	1	7 0.25%			0 0.31%		0 Shopping, Social
24	old market chania	old market chania	2022-01-07 21:00:21.0	0	0 0.25%			0 0.31%		0 Work
25	old market chania	Elafonissi Beach, Kissamos	2022-01-08 01:01:50.0	73	89 18.25%			0 22.81%		0 Shopping, Social
26	Elafonissi Beach, Kissamos	Profti Ila 110, Chania	2022-01-08 05:03:29.0	83	99 20.75%			0 25.94%		0 Home
27	Profti Ila 110, Chania	East Moat Theater Parking, Kiprou, Chania	2022-01-08 16:00:09.0	3	9 0.75%			0 0.94%		0 Shopping, Social
28	East Moat Theater Parking, Kiprou, Chania	East Moat Theater Parking, Kiprou, Chania	2022-01-08 17:00:09.0	0	0 0.25%			0 0.31%		0 Work
29	East Moat Theater Parking, Kiprou, Chania	Profti Ila 110, Chania	2022-01-08 20:00:17.0	3	8 0.75%			0 0.94%		0 Home
30	Profti Ila 110, Chania	Chania International Airport Ioannis Daskalogiannis	2022-01-09 07:00:11.0	10	11 2.5%			0 3.13%		0 Work
31	Chania International Airport Ioannis Daskalogiannis	Profti Ila 110, Chania	2022-01-09 16:00:22.0	9	11 2.25%			0 2.81%		0 Home
32	Profti Ila 110, Chania	Nea Hora Chania Marina, Chania	2022-01-01 08:00:14.0	5	14 1.25%			0 1.56%		0 Shopping, Social
33	Nea Hora Chania Marina, Chania	Nea Hora Chania Marina, Chania	2022-01-01 11:00:14.0	0	0 0.25%			0 0.31%		0 Work
34	Nea Hora Chania Marina, Chania	Elafonissi Beach, Kissamos	2022-01-01 12:01:43.0	73	89 18.25%			0 22.81%		0 Shopping, Social
35	Elafonissi Beach, Kissamos	Koum Kapi, Chania	2022-01-01 13:03:17.0	73	94 18.25%			0 22.81%		0 Shopping, Social
36	Koum Kapi, Chania	Profti Ila 110, Chania	2022-01-01 15:03:22.0	2	5 0.5%			0 0.63%		0 Home
37	Profti Ila 110, Chania	SeaStudios, Mesologgiou, Chania	2022-01-02 05:00:06.0	2	6 0.5%			0 0.63%		0 Shopping, Social
38	SeaStudios, Mesologgiou, Chania	Profti Ila 110, Chania	2022-01-02 06:00:13.0	3	7 0.75%			0 0.94%		0 Home
39	Profti Ila 110, Chania	Splantzia Square, Chania	2022-01-03 07:00:09.0	3	9 0.75%			0 0.94%		0 Work
40	Splantzia Square, Chania	Profti Ila 110, Chania	2022-01-03 16:00:20.0	4	11 1.0%			0 1.25%		0 Home
41	Profti Ila 110, Chania	Chania National Stadium, Andrea Papandreou, Chania	2022-01-04 07:00:07.0	3	7 0.75%			0 0.94%		0 Shopping, Social
42	Chania National Stadium, Andrea Papandreou, Chania	Profti Ila 110, Chania	2022-01-04 08:00:14.0	3	7 0.75%			0 0.94%		0 Home
43	Profti Ila 110, Chania	Seitan Limani, Akrotiri	2022-01-05 07:00:24.0	18	24 4.5%			0 5.63%		0 Shopping, Social
44	Seitan Limani, Akrotiri	Seitan Limani, Akrotiri	2022-01-05 10:00:24.0	0	0 0.25%			0 0.31%		0 Work
45	Seitan Limani, Akrotiri	Profti Ila 110, Chania	2022-01-05 11:00:48.0	18	24 4.5%			0 5.63%		0 Home
46	Profti Ila 110, Chania	Chania International Airport Ioannis Daskalogiannis	2022-01-06 04:00:11.0	10	11 2.5%			0 3.13%		0 Shopping, Social
47	Chania International Airport Ioannis Daskalogiannis	Profti Ila 110, Chania	2022-01-06 06:00:22.0	9	11 2.25%			0 2.81%		0 Home
48	Profti Ila 110, Chania	Venizelos Graves, Agorastaki, Chania	2022-01-07 08:00:03.0	2	3 0.5%			0 0.63%		0 Work
49	Venizelos Graves, Agorastaki, Chania	beach pavillion near kalathas	2022-01-07 09:00:18.0	8	15 2.0%			0 2.5%		0 Shopping, Social
50	beach pavillion near kalathas	beach pavillion near kalathas	2022-01-07 10:00:18.0	0	0 0.25%			0 0.31%		0 Work
51	beach pavillion near kalathas	Profti Ila 110, Chania	2022-01-07 12:00:33.0	9	15 2.25%			0 2.81%		0 Home
52	Profti Ila 110, Chania	Chania National Stadium, Andrea Papandreou, Chania	2022-01-08 07:00:07.0	3	7 0.75%			0 0.94%		0 Work
53	Chania National Stadium, Andrea Papandreou, Chania	Cruise terminal, Agiou Nikolaou, Souda, Chania	2022-01-08 13:00:17.0	5	10 1.25%			0 1.56%		0 Shopping, Social
54	Cruise terminal, Agiou Nikolaou, Souda, Chania	Cruise terminal, Agiou Nikolaou, Souda, Chania	2022-01-08 14:00:17.0	0	0 0.25%			0 0.31%		0 Work
55	Cruise terminal, Agiou Nikolaou, Souda, Chania	Police Station, Leoforos Soudas, Souda, Chania	2022-01-08 15:00:19.0	1	2 0.25%			0 0.31%		0 Shopping, Social
56	Police Station, Leoforos Soudas, Souda, Chania	Profti Ila 110, Chania	2022-01-08 16:00:39.0	4	11 1.0%			0 1.25%		0 Home

Figure 32 - Trips List

The labels for each columns in the trip list are:

- Origin
- Destination
- Date, time
- Distance
- Duration
- Optimal Needed Load
- Optimal Recharge Needed
- Realistic Needed Load
- Realistic Recharge Needed
- Type

The optimal needed load is calculated by dividing the distance with nominal range, while the realistic needed load is calculated the same way but with 80% of the nominal range. Lastly idle time (sheet no.2) is calculated trip by trip by using the time difference between the trips.

Trip A,B (arrival time, duration)

$$idleTime = arrivalTimeB - arrivalTimeA - durationB$$

1	Num of trips0:	start date time	end date time	distance sum (km)	duration sum (mins)	idle time (mins)	Optimal Needed Load	Realistic Needed Load	spent kW (ideal)	spent kW
2	2	2022-01-01 17:00:11.0	2022-01-02 02:00:22.0	12	22	539 3.0%	3.76%	3.76%	2,4	3,36
3	4	2022-01-02 09:00:09.0	2022-01-02 14:00:50.0	31	48	299 7.75%	9.69%	9.69%	6,2	8,68
4	2	2022-01-03 07:00:11.0	2022-01-03 22:00:21.0	12	21	899 3.0%	3.76%	3.76%	2,4	3,36
5	2	2022-01-04 18:00:06.0	2022-01-04 19:00:13.0	5	13	59 1.25%	1.5699999999999998%	1.5699999999999998%	1	1,4
6	2	2022-01-05 07:00:11.0	2022-01-05 10:00:22.0	19	22	179 4.75%	5.9399999999999995%	5.9399999999999995%	3,8	5,32
7	2	2022-01-06 07:00:17.0	2022-01-06 17:00:34.0	24	34	599 6.0%	7.5%	7.5%	4,8	6,72
8	3	2022-01-07 06:00:14.0	2022-01-08 20:00:17.0	11	31	2279 2.75%	3.44%	3.44%	2,2	3,08
9	2	2022-01-09 07:00:11.0	2022-01-09 16:00:22.0	19	22	539 4.75%	5.9399999999999995%	5.9399999999999995%	3,8	5,32
10	2	2022-01-01 08:00:14.0	2022-01-01 15:03:22.0	7	19	422 1.75%	2.19%	2.19%	1,4	1,96
11	2	2022-01-02 05:00:06.0	2022-01-02 06:00:13.0	5	13	59 1.25%	1.5699999999999998%	1.5699999999999998%	1	1,4
12	2	2022-01-03 07:00:09.0	2022-01-03 16:00:20.0	7	20	539 1.75%	2.19%	2.19%	1,4	1,96
13	2	2022-01-04 07:00:07.0	2022-01-04 08:00:14.0	6	14	59 1.5%	1.88%	1.88%	1,2	1,68
14	2	2022-01-05 07:00:24.0	2022-01-05 11:00:48.0	36	48	239 9.0%	11.26%	11.26%	7,2	10,08
15	2	2022-01-06 04:00:11.0	2022-01-06 06:00:22.0	19	22	119 4.75%	5.9399999999999995%	5.9399999999999995%	3,8	5,32
16	3	2022-01-07 08:00:03.0	2022-01-07 12:00:33.0	19	33	239 4.75%	5.9399999999999995%	5.9399999999999995%	3,8	5,32
17	3	2022-01-08 07:00:07.0	2022-01-08 16:00:30.0	14	28	539 3.5%	4.38%	4.38%	2,8	3,92
18	2	2022-01-09 05:00:06.0	2022-01-09 09:00:13.0	5	13	239 1.25%	1.5699999999999998%	1.5699999999999998%	1	1,4
19	4	2022-01-09 11:00:19.0	2022-01-09 20:01:14.0	26	61	539 6.5%	8.139999999999999%	8.139999999999999%	5,2	7,28

Figure 33 - Completed Trips defined as leaving and returning Home

As seen in figure 32 it is difficult for a human to understand the raw data therefore we created the pseudo trip defined as a collection of trips that start since the moment the user has left his Home and end when he/she returns. Although this list is easier to understand it might still be difficult for many users.

5.8.4 Chart creation and data visualization

Last but not least our users can directly use the filters they like to make charts and see the data in real time. The next 4 examples are graphical representations of an aggregated data set.

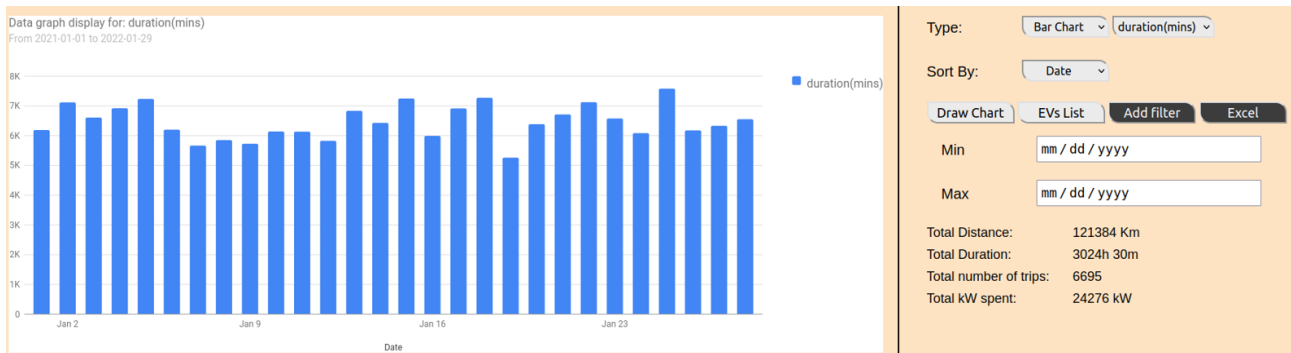


Figure 34 - Chart screenshot displaying the total duration of driving per day (in mins) from date 01/01 to 01/28

In this example (figure 34) we display the sum of time spent driving EVs during each day for a range of dates (01/01 – 01/28).

Additionally the total distance, total driving time(duration), numbers of trips and total kW spent for our date range are printed on the bottom right of the screen.

The dropdown list in the top right corner of the screenshot once used will change the chart to display the new relevant information in detail for the aforementioned calculations.

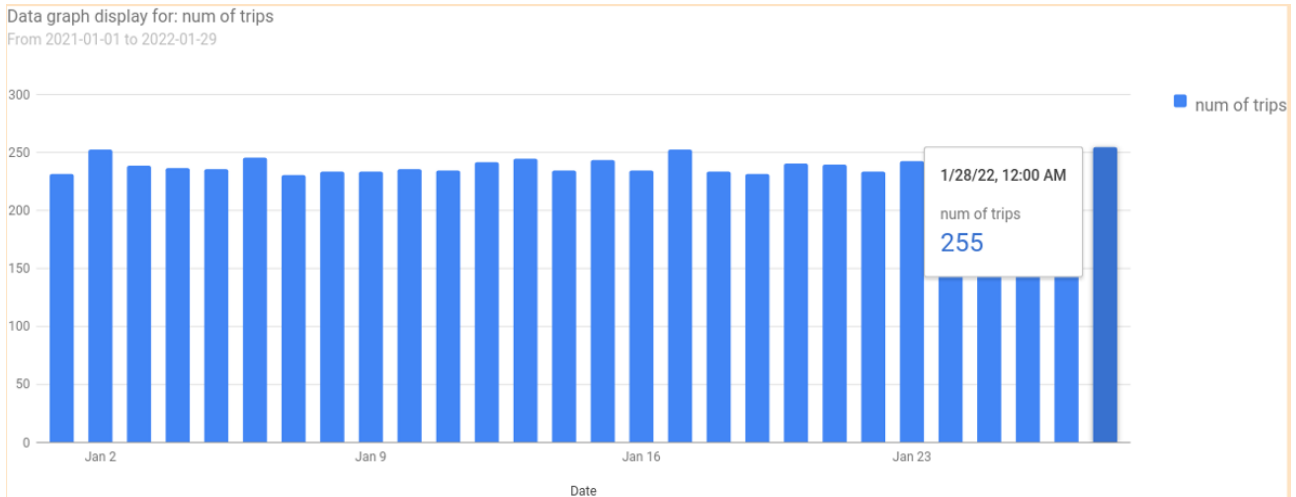


Figure 35 - Chart display the number of trips in each day

As displayed in figure 35 the chart displays the number of trips per day and for additional clarification the user can simply use his mouse to see the info represented by the data bar in detail.



Figure 36 - Chart Displaying Distance covered in km

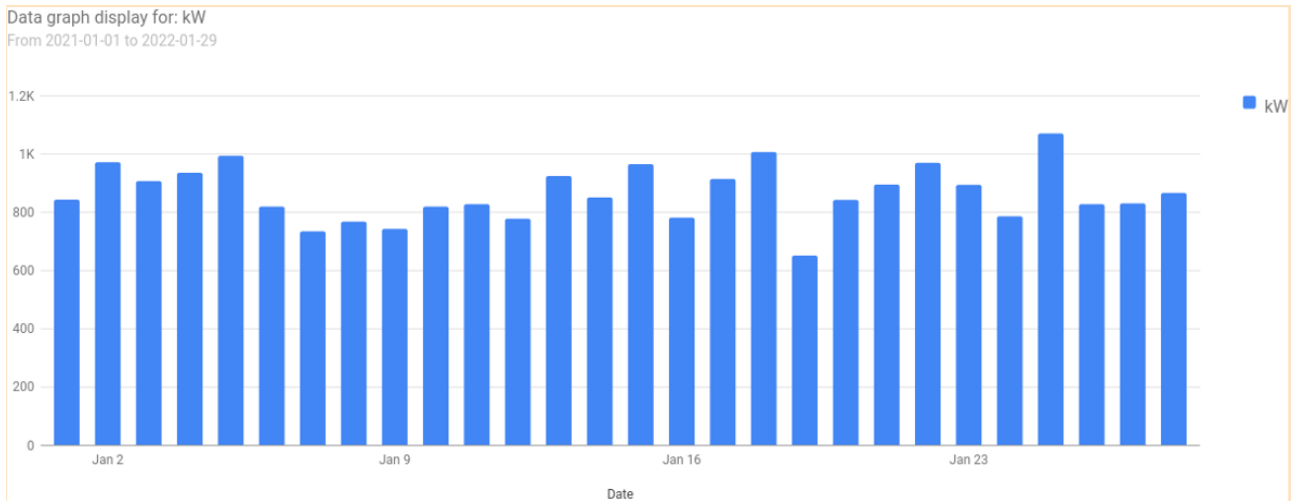


Figure 37 – kws by EVs for our time range

5.9 Data used to create our data set

The data set that we used for the implementation of this thesis has been created by using the data from the [National Household Travel Survey \(NHTS\)](#). [NHTS](#) is a governmental agency from the US. Using their data as a basis we have access to the probability densities of departure time and time spent idle (idle EV) for each trip.

5.9.1 Probability density for departure time and time spent idle

The two following arrays contain the probability of an EV departing and its dwelling time on its destination. The probabilities for departure change by the time of day for each type of trip while the probabilities for dwelling time change by type of trip. This data is imported and used by our web application through the *ExcelReader.java* class.

Time (Hour)	Probability density: Departure Time		
	Home	Shopping, Social	Work
1	0,002539645	0,002102653	0,000338
2	0,001777952	0,001424255	0,0017808
3	0,000836537	0,001783512	0,0023155
4	0,000929415	0,001696862	0,0098972
5	0,001796834	0,005870751	0,0409441
6	0,005971318	0,011079306	0,1092366
7	0,012049346	0,020879644	0,2215048
8	0,017120482	0,041741603	0,2010203
9	0,018686449	0,068144992	0,0927608
10	0,026487605	0,092743305	0,050821
11	0,040749545	0,09403961	0,0372432
12	0,052283648	0,086530946	0,040547

13	0,051574667	0,084833912	0,0535365
14	0,060379781	0,085553721	0,0398391
15	0,088594954	0,086021487	0,0308463
16	0,121180194	0,090677751	0,024371
17	0,144419655	0,090632454	0,0170565
18	0,090427256	0,064820261	0,0100092
19	0,074110724	0,036181876	0,0044899
20	0,0671996	0,018310989	0,0023366
21	0,054506258	0,008945268	0,0024516
22	0,031424761	0,003929825	0,0029605
23	0,024450724	0,001544306	0,0013622
24	0,01050265	0,00051071	0,0023314

Table 6 - Probability density: Departure Time for each hour of day and type of trip

Time (minutes)	Probability density: EV dwelling time		
	Home	Shopping, Social	Work
0	0	0	0
60	0	0,567109	0,085704
120	0	0,216752	0,075198
180	0,016238	0,099668	0,079044
240	0,136169	0,049388	0,097816
300	0,132222	0,029642	0,071127
360	0,094092	0,018527	0,049591
420	0,089135	0,008448	0,05331
480	0,156414	0,007311	0,129888
540	0,200023	0,003156	0,196163
600	0,099929	0	0,097692
660	0,042098	0	0,040912
720	0,018042	0	0,017977
780	0,012028	0	0,005579
840	0,003608	0	0
900	0	0	0
960	0	0	0
1020	0	0	0
1080	0	0	0
1140	0	0	0
1200	0	0	0
1260	0	0	0
1320	0	0	0
1380	0	0	0

1440	0	0	0
------	---	---	---

Table 7 - Probability density: Time spent dwelling in minutes for each type of trip

5.10 Optimization for trip creation

Next we will see how the data set was generated while using the probability density arrays seen at 5.9.1 and Google Maps. Firstly we created a list of destinations within the city Chania, Greece, then paired those destinations in every combination possible and used Google maps to find the distance and duration of the trip between those two destinations. Note that destination A towards destination B does not necessarily have the same duration and distance as destination B towards Destination A. After the processing we had a list of origin, destination, distance, duration which we saved on our database. This was done because during trip generation we will need to reuse those trip routes several times and making a new HTTP request for each of them will be slow and expensive, therefore we effectively pre-load all possible results we will need from Google Maps during our trip generation. Once this procedure is complete we are ready to use the *TripGenerator.java* class.

5.10.1 Generating trips.

Initially during the trip generation we instantiate all the needed classes and get the data we will need from the database. As seen on the figure (38) below the variable *altue* contains a list with all the users we have generated and the variable *alt* contains the list with the routes we will use to generate trips while the variable *odi* contains the probability density arrays. Then we proceed for each generated user to generate trips that will span over a month. The trips generated for each user will be stored into a sub-list (*subList*) of trips which is added to the total list of trips(*alt*).

```

public static void main(String[] args) throws IOException, ParseException, InvalidFormatException {

    DistanceMatrixApi dma = new DistanceMatrixApi();

    UserDao ud = DAOFactory.getMySQLDAOFactory().getUserDAO();
    ArrayList<TripUserEV> altue = new ArrayList<>();

    altue = ud.getUsersWithEvs();

    TripDAO td = DAOFactory.getMySQLDAOFactory().getTripDAO();
    ArrayList<Trip> alt = new ArrayList<>();
    altds = td.getTripsDataSet();

    OriginDeparturesInit odi = new OriginDeparturesInit();
    odi.init("data.xlsx");

    for (int u = 0; u < 100; u++) {
        TripUserEV tue = altue.get(u);
        TripSeed ts = new TripSeed();
        ts.setHomeLocation(odi.getRandomOrigin());
        ts.setWorkLocation(odi.getRandomDestination("", ts.getHomeLocation(), ""));

        for (int i = 0; i < 28; i++) {
            ArrayList<Trip> subList = new ArrayList<>();
            Date date = new Date();
            date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse("2022-01-" + String.format("%02d", i + 1) + " 00:00:00");
            Timestamp timestamp = new Timestamp(date.getTime());

            subList = totalTrip(odi, tue, "", timestamp, subList, ts);
            alt.addAll(subList);
            subList.clear();
        }
        System.out.println("\n\nUser No." + u + " out of " + altue.size() + " done!");
    }
    td.insertTrips(alt);
    alt.clear();
}

```

Figure 38 - Trip Generator main function

We define a complete trip as a list of trips that start from the moment he left his/her home and end when the user returns. Additionally each trip might contain multiple instances of the user going to work, shopping, socializing more than once in a day based on probability densities we saw. Therefore we have a recursive function that generates trips with the end condition of returning home. We should also note that we consider each user to have a single home and work locations therefore we use the *TripSeed* type variable to pass that information along inside the recursion.

```

private static ArrayList<Trip> totalTrip(OriginDeparturesInit odi, TripUserEV tue, String type, Timestamp datetime, ArrayList<Trip> altRecursion, TripSeed ts)
{
    if (type.equals("Home")) {
        return altRecursion;
    }
    altRecursion.add(createAndCalculateTrip(odi, tue, type, datetime, altRecursion, ts));
    return totalTrip(odi, tue, altRecursion.get(altRecursion.size() - 1).getType(), datetime, altRecursion, ts);
}

```

Figure 39 - Recursion function to generate Trips for each user

Lastly we have the trip individual trip generation function split into two parts. The first part has the assignment of type, date-time, origin, destination, distance and duration for each trip.

```

private static Trip createAndCalculateTrip(OriginDeparturesInit odi, TripUserEV tue, String type, Timestamp datetime, ArrayList<Trip> prevTrips, TripSeed ts)

    DistanceMatrixApi dma = new DistanceMatrixApi();

    Trip trip = new Trip();
    trip.setUser_id(tue.getUsername());
    trip.setEv_id(tue.getEVid());

    if (prevTrips.isEmpty()) {
        trip.setType(rollForType(type));
        trip.setDate(odi.departTime(trip.getType(), datetime));
        trip.setOrigin_location(ts.getHomeLocation());
        if (trip.getType().equals("Work")) {
            trip.setDestination(ts.getWorkLocation());
        } else {
            trip.setDestination(odi.getRandomDestination(ts.getHomeLocation(), ts.getHomeLocation(), ts.getWorkLocation()));
        }
    } else {
        Trip prevTrip = prevTrips.get(prevTrips.size() - 1);

        trip.setDate(odi.departTime(type, prevTrip.getDate()));
        trip.setType(rollForType(prevTrip.getType(), trip.getDate()));
        switch (trip.getType()) {
            case "Home":
                trip.setOrigin_location(prevTrip.getDestination());
                trip.setDestination(ts.getHomeLocation());
                break;
            case "Work":
                trip.setOrigin_location(prevTrip.getDestination());
                trip.setDestination(ts.getWorkLocation());
                break;
            default:
                trip.setOrigin_location(prevTrip.getDestination());
                trip.setDestination(odi.getRandomDestination(trip.getOrigin_location(), ts.getHomeLocation(), ts.getWorkLocation()));
                break;
        }
    }

    TripDataSet tds = new TripDataSet();
    tds = getTripDataSet(trip.getOrigin_location(), trip.getDestination());

    trip.setDistance(tds.getDistance());
    trip.setDuration(tds.getDuration());

```

Figure 40 - Trip generation. Assigning type, date-time, origin and destination

The *TripDataSet* *tds* variable contains the distance and duration of the trip from the dataset we had preloaded into the database. If this was not done and we wanted to recalculate the distance and duration through Google Maps we would instead run the code in green.

```

trip.setDistance(tds.getDistance());
trip.setDuration(tds.getDuration());

double otemp = (double) (tds.getDistance() == 0 ? 1 : tds.getDistance() / tue.getNominalRange());
double rtemp = (double) (tds.getDistance() == 0 ? 1 : tds.getDistance() / ((double) tue.getNominalRange() * 0.8d));
trip.setDate(new Timestamp(trip.getDate().getTime() + (long) tds.getDuration() * 1000));

JsonNodeReader jnr = dma.findDistanceDuration(trip.getOrigin_location(), trip.getDestination(), trip.getDate());

trip.setDistance(jnr.getDistanceInKMeters());
trip.setDuration(jnr.getDurationInMins());

double otemp = (double) (jnr.getDistanceInKMeters() == 0 ? 1 : jnr.getDistanceInKMeters() / tue.getNominalRange());
double rtemp = (double) (jnr.getDistanceInKMeters() == 0 ? 1 : jnr.getDistanceInKMeters() / ((double) tue.getNominalRange() * 0.8d));

trip.setDate(new Timestamp(trip.getDate().getTime() + (long) jnr.getDurationInMins() * 1000));
trip.setOload(otemp * 100);
trip.setOrecharge(otemp > 1 ? calcNumOfRecharges(otemp) : 0);

trip.setRload(rtemp * 100);
trip.setRrecharge(rtemp > 1 ? calcNumOfRecharges(rtemp) : 0);
return trip;
}

```

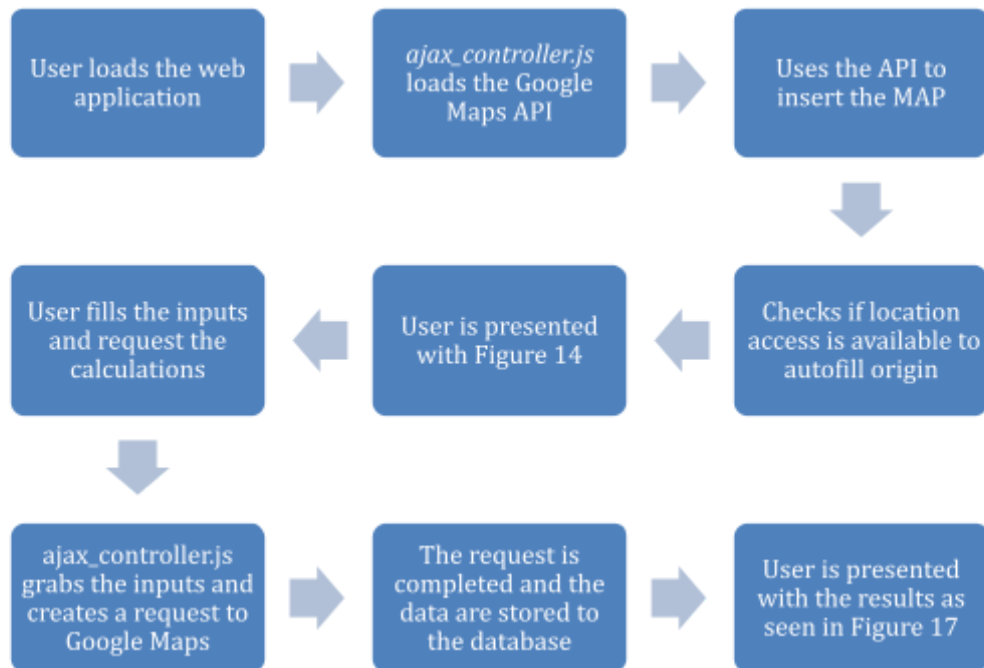
Figure 41 - Calculating date-time after trip, percentage of battery left and recharging if needed

With the values of distance and the nominal range we can proceed to calculate the battery load we will need to complete the trip as well as check if recharging is required to complete the trip.

Chapter 6 Flowcharts

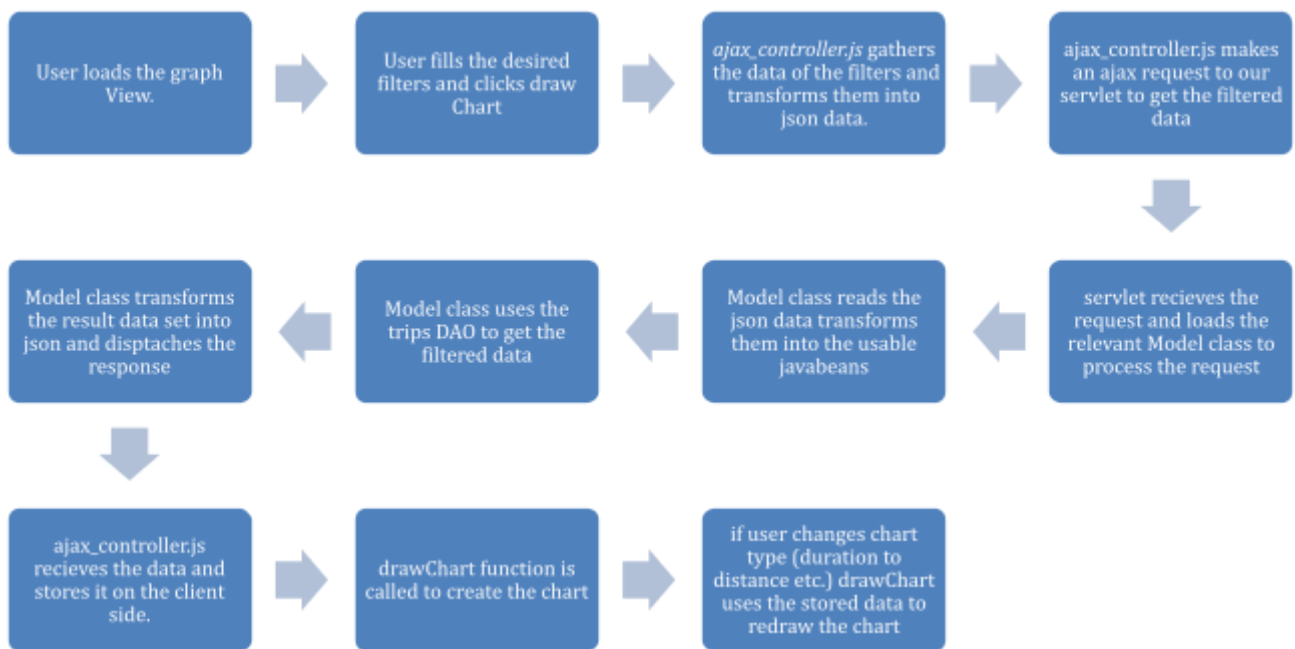
6.1 Code flow during trip planning

We will see a flowcharts of everything that happens during the trip planning use case. Example screenshots of what the user sees during this flowcharts are in section 5.5.2: Figure 14, Figure 17.



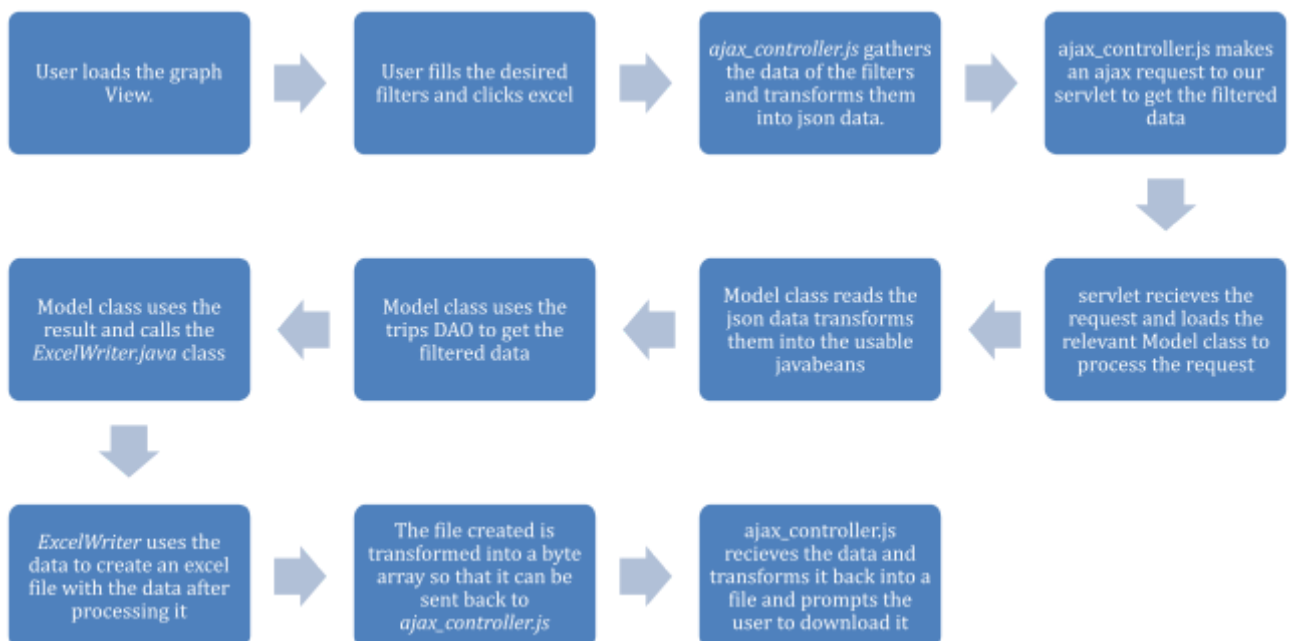
6.2 Code flow during graph generation

This flowchart will show a step by step example of generating charts by the user. Example screenshots of what this process looks like are in section 5.8.4: Figure 34, Figure 35, Figure 36, Figure 37 as well as information on how this process is implemented in section 5.8.



6.3 Code flow during excel generation

This flowchart will show a step by step example of generating results in an excel by the user. Example screenshots of what this process looks like are in section 5.8.4: Figure 34, Figure 35, Figure 36, Figure 37 as well as information on how this process is implemented in section 5.8.



Chapter 7 Conclusions and Future Development

7.1 Conclusions

EVs are the future for a more efficient and clean “mode” of transportation, their development and study are essential as well as tools that can serve to complement their use such as the tool we created in this thesis. As of 2021 there is no infrastructure in Greece where this web application was developed which constrains our ability to further develop more use cases for our web app such as planning for recharge stations. Even though we had those constraints we succeeded in implementing and completing all of our stated goals. We have set up a fully-fledged web application capable of storing information about planned trips with all the accompanying data that each trip needs, functionality for our users and us the developers to use the data and calculate its aggregates in order to predict the kWh spent by a fleet of EVs.

Besides the theoretical research we had decided to complete this thesis in the form of a web application instead of other alternatives in order to make it more accessible and have prospects of reuse and further development. While using conventional software tools would have been faster to complete the “fleet of EVs” objective it would be harder to use in the future and maintained, since only the developer and those with access, knowledge and understanding of the source code would be able to reuse it. This is not the case for our thesis since a web application can be accessed and used easily by anyone and be publicly available. The main tools used for the implementation were Java, JS, Maven and Apache Tomcat, the first 3 ensure that our web application is easily transferable between terminals/servers while Apache tomcat is the servlet we used for demonstration and development.

The results we had in this attempt were very satisfactory since achieving the data aggregation and processing in times that are comparable to locally run functions (excluding the download time). Furthermore the data can be directly downloaded in a .xlsx file which makes reuse simple. Lastly our results and predicted values are very reliable since the input we use comes directly from Google Maps instead of estimations or auto-generated data sets.

7.2 Future Development

Lastly we are fully aware that the modern trends for this type of web application is to go towards mobile phones. This was the main reason we chose to implement the project in strict accordance to the MVC architecture. Right now the only thing needed for an IOs or Android app to work is an interface since the backend is implemented in java and has been tested on cloud services. The major advantage of a web application based on PCs instead of mobile phones is computational power. Creating graphs and statistics on the data the way we've implemented, it shifts part of the weight on the client side. If the same is to be done for mobile phones it has to shift most of the computational weight back on the server side which will be costly since Google APIs are not free and outside the scope of a thesis.

In the case that a web app is created using our web app's backend and further popularized, real time large data gathering will be available. While it already is with our current implementation, users that prefer mobile phones will not be easily tempted to participate in our web application without a specific app.

Additionally besides the technical improvements that can be made we can also implement an extra functionality on the existing web application. The next step would be calculating the impact that such a fleet of EVs would have on the electric grid, predicting their recharging times and estimating whether the electric grid is capable of handling the extra load. The only missing components to add this functionality is the probability density function for the time of recharging EVs, the respective function for the charging decision and the decision making process for thw drivers to access to closest available charging point.

References

- Apache Software Foundation. (n.d.). *Apache Maven*. Retrieved from Apache Maven Project: <https://maven.apache.org/>
- Apache Software Foundation. (n.d.). *Apache Tomcat*. Retrieved from Apache Tomcat: <http://tomcat.apache.org/>
- Apache Software Foundation. (n.d.). *Netbeans*. Retrieved from <https://netbeans.apache.org/>
- energy, U. d. (n.d.). *The history of the electric car*. Retrieved from <https://www.energy.gov/articles/history-electric-car>
- Frank Buschmann, K. H. (n.d.). *Pattern-oriented software architecture*.
- Google INC. Google Maps API. <https://developers.google.com/maps/apis-by-platform>
- M. Ehsani, Y. G. (2018). *Modern Electric, Hybrid Electric and Fuel Cell Vehicles: Fundamentals, Theory and Design*. CRC Press .
- MySQL. (n.d.). *MySQL*. Retrieved from <https://www.mysql.com/>
- Oracle. (n.d.). *Java SDK*. Retrieved from <https://www.oracle.com/java/technologies/downloads/>
- Sruvey, N. H. (n.d.). *NHTS*. Retrieved from <https://nhts.ornl.gov/>
- Statista. (n.d.). Retrieved from <https://www.statista.com/statistics/1010938/share-of-ev-on-total-number-of-registered-cars-in-european-countries/>
- Ulalah, A. (n.d.). *Model-View-Controller (MVC)*. Retrieved from <https://tinyurl.com/yuhf8ake>