MyCar.com

Vehicle owner information center as SaaS

Mika Lifflander

Master's thesis
April 2017
Master's Degree
Information Technology

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Master's Degree in Information Technology

Mika Lifflander:
MyCar.com
Vehicle owner information center as SaaS

Master's thesis 53 pages, appendices 14 pages
April 2017

Rapid expansion of public cloud has introduced multitude of software-as-a-service, or SaaS, solutions that brings information to users just when they need it. MyCar.com is a business idea of such service, that would bring the relevant information related to personal vehicles to everyone.

My thesis addresses the questions about architectural design of such service from software architecture of single component all the way up to the SaaS model itself. I explain different phases of modern development practices and demonstrate why proof of concepts are so beneficial. There is also detailed view on monetizing strategies with reasons making Freemium business model the most promising for this kind of service.

Key words: software-as-a-service, proof of concept, public cloud, software

CONTENTS

ABBREVIATIONS AND TERMS

API	Application Programming Interface
AWS	Amazon Web Services
B2C	Business to Consumer
BPaaS	Business Process as a Service
CLI	Command Line Interface
CLV	Customer Lifetime Value
CRM	Customer Relationship Management
CSS	Cascading Style Sheets
eCPM	Effective Cost Per Mille
ERP	Enterprise Resourcing Planning
IAM	Identity and Access Management
IaaS	Infrastructure as a Service
IPM	(Advertisement) Impressions Per Minute
MVP	Minimum Viable Product
ORM	Object Relational Mapping
OS	Operating System
POC	Proof of Concept
REST	Representational State Transfer
ROI	Return on Investment
SaaS	Software as a Service
SOAP	Simple Object Access Protocol
TAMK	Tampere University of Applied Sciences
UI	User Interface

1 INTRODUCTION

In todays connected world there are many applications and services for people to find information on timely basis. One can order food, pay bills or get movie tickets with only few clicks wherever they are, 24 hours a day. But when it comes to your personal transportation you are out of options.

MyCar.com is a comprehensive solution to collect all the data of personal transportation to one service and give users of this solution a clear view of what is going on with the vehicle. Solution would provide access to real-time price information of fuel, repairs, insurances etc. as well as access to full maintenance and ownership history of the vehicle.

Current offering for these needs is scattered to many independent applications. For example, fuel price information has dedicated application while maintenance history is only available in manufacturers system. While all the data is technically available these applications have their limitations like price, availability or unreliable data source. These limitations create a unique opportunity for MyCar.com to provide reliable real-time access to data that is not available to everyone.

This research focuses on real-time price information service and aims to find out what is needed for such service, how it would be developed and made profitable service that attracts lots of users and remains beneficial to all stakeholders.

2 CONCEPTS

2.1 Software as a Service

Term Software as a Service, or SaaS, is used to describe a software service that can be accessed over the internet by its users (Singleton, 2011). Actual application is hosted in the cloud instead of locally on users' device. Well known SaaS examples of SaaS are Google, Facebook and Twitter while there are many others too.

According to Gartner research (Gartner, 2016) total market for all cloud services were forecasted to be roughly 204 billion U.S dollars. SaaS is seen as third largest segment after advertising and BPaaS with over 37 billion U.S dollars, bigger than the rest of the three segments combined.

History of SaaS as a centralized computing dates back to 1960s when mainframe providers, i.e. IBM offered centralized solutions that were referred to as time-sharing computing. These services offered data warehousing and computing power hosted in the providers' data centres.

After the rapid expansion of the Internet in the late 20th century new breed of centralized computing emerged. Application Service Providers, or ASPs, provided businesses service of hosting and managing third party business applications. While there are many similarities between ASP and SaaS, there are also clear difference in operational model between these two (table 1).

TABLE 1. Comparison of operational models

	ASP	**SaaS**
Application	3rd party application	Developed by provider
Architecture	Traditional Client-Server	On the Web
Tenancy	Separate instance per customer	Multitenant

SaaS does not require physical distribution channel as the application can be deployed almost instantly and is available for user right after the deployment. The need for distribution channel emerges if and when SaaS implementation is expanded with the connected application. Even in these cases there are rarely a need for physical distribution

as OS vendors usually have their own software distribution platform, like Google Play Store for Android or App Store for Apple iOS and macOS.

2.1.1 SaaS Architecture Options

There are two options for the architecture of SaaS, Single-Tenant and Multi-Tenant. In Single-Tenant architecture each customer, or tenant, has dedicated instance of the application while in Multi-Tenant architecture one application instance is used to serve multiple customers (Manish, 2016). One might argue that Single-Tenant architecture is closer to ASP or hosted model than true SaaS but whether Multi-Tenancy is required component for SaaS remains controversy topic.

Like in any other case, there are pros and cons for each option. Single-Tenant architecture enforces maximum privacy and minimum interference as no code is shared amongst the customers but it comes with higher price tag. Multi-Tenant solution optimises costs, use of power and hardware and maintenance efforts but limits customization options (Eisenhower, 2015).

2.1.2 SaaS Integration

There are two ways for SaaS integration, horizontal and vertical. Vertical SaaS is software that addresses the needs for specific industry while horizontal SaaS focuses on software category (Polous, 2015).

In other terms, horizontal integration means providing one level of service for multiple businesses and customers, like in agriculture horizontally operating company would provide one type of raw material for other industries, food, animal food, chemical industry etc. For horizontal integrated company to be successful they must cover as many customers as they can.

Vertical integrated company would look to own more than part of the process while focusing on narrow set of customer types. Again, analogy to agriculture would be that company provides full service chain from collecting raw materials, processing and manufacturing for customer types of their selection, i.e. chemical industry.

In software world, great example of horizontal SaaS is Salesforce, who provide vast range of applications like CRM, ERP and ecommerce. Example of vertical SaaS is OpenTable which provides fine-tuned service for restaurants. While it might be technically possible, OpenTable has not expanded horizontally to for example hospital bed reservations.

2.1.3 SaaS Characteristics

While each SaaS implementation have their unique features, there are also common characteristics that most of SaaS implementations share.

SaaS often provides customers a set of customizing options, that are usually limited to colours and maybe company logos. Basic layout and feature set are shared with all users unless application is deliberately built with that kind of customizing options

SaaS applications commonly have much frequent update schedule than traditionally distributed software. SaaS applications are commonly updated on weekly or monthly basis with new features and patches.

Other commonly shared aspect of SaaS is open API that provides accessible endpoint for connected applications and possibly other SaaS implementations over wide area network.

Many of the SaaS implementations also have social and collaborative features built in to them, be that integration to Facebook, internal discussion forum, voting, document sharing or something similar.

2.2 Deliverables

During the life-cycle of software project there can be many different deliverables. Technically all documents, plans etc. can be considered to be deliverables but in the scope of this topic I will cover different software deliverables only.

2.2.1 Proof of Concept

Proof of Concept: “Evidence, typically deriving from an experiment or pilot project, which demonstrates that a design concept, business proposal, etc. is feasible.” (English Oxford dictionary)

Proof of Concept, or POC, is a small implementation to test feasibility of new idea. It’s mainly used to test and demonstrate that concept or theory will work. It’s common for POCs not to be complete solutions but only cover a small portion of it (Swallow, 2016).

In software development, there are quite often demos for commercial software. POC is quite different even though it can be used to demonstrate functionality. Usually a demo is part of the actual solution and users are granted a limited access, either time or features, for the demo usage. POCs are rarely used as a part of actual solution but only to show or verify feasibility of technical theory.

2.2.2 Minimum Viable Product

When talking about start-ups, software development and/or agile methods, term Minimum Viable Product or MVP is often used. Term is used to describe a product that has just enough features to gather real feedback from real users to validate an idea for the application and to fuel further development (Swallow, 2016; Chang, 2016).

Building an application can, and full blown SaaS solution even more so, be very expensive and time consuming effort. Worst thing that could happen for newly emerged start-up is that they invest heavily to develop something that nobody actually wants. This is where MVP comes into play.

Unlike the POC, the minimum viable product is a version of the actual product. It has least number of features still being useful to users (figure 1). MVPs main goal is to solve real problem by incorporating simplest possible solution (Brainhub 2016).

FIGURE 1. Illustration of Minimum Viable Product from Hackernoon.com

Building a MVP instead of full solution immediately provides some significant benefits. When investing time and money, it's all about ROI and risk. The goal is to maximize the former while minimizing the latter. Creating as small and simple solution as possible as fast as possible have unbeatable ROI / Risk –ratio. MVP will also speed up the development of the final product as the idea and significant features are verified with customer feedback.

With SaaS knowing your customer is of high importance. To be successful the solution must answer to customers' needs and no presentation, questionnaire or survey can outperform getting feedback from actual product. Getting to know customers also gives a clearer vision of what features the final product should contain.

Fourth major benefit of MVP is that it can be used to find the early adopters (figure 2). Early adopters are those users who are first in line to take new application in to daily use. They are so eager to get their problem solved that they are willing to use application even with limited feature set. These early adopters are the ones can and will validate any predictions or assumptions about a product.

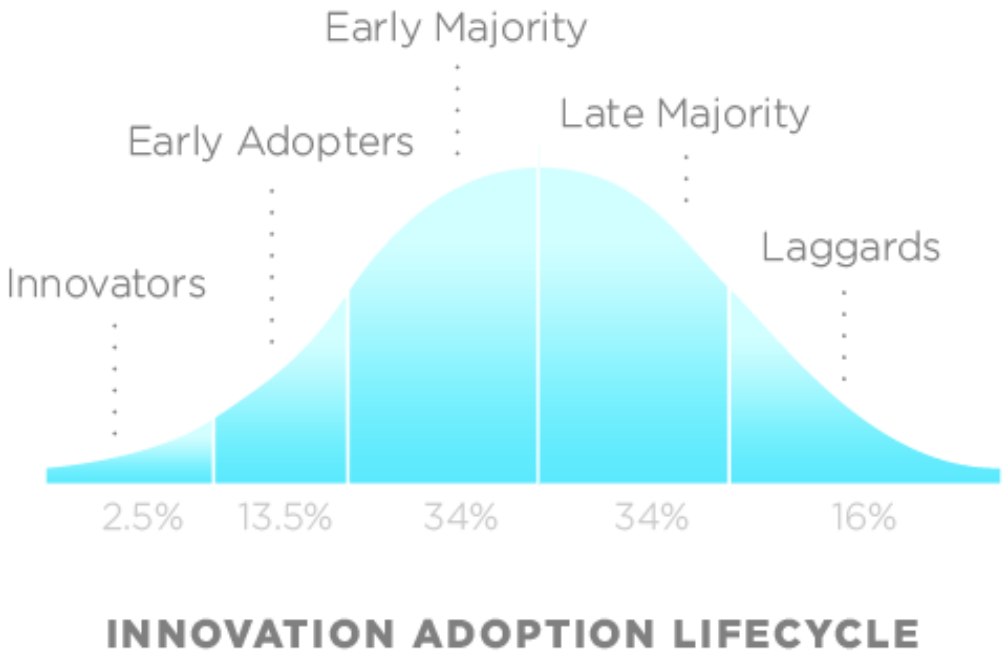

FIGURE 2. Innovation Adoption Lifecycle from Wikipedia

2.2.3 Web Application

Term web application is usually used to describe application used in web browser. While this is true it only covers the visible UI part. Modern web applications also require server side implementation that communicates with database, provides authentication and authorization and so forth.

For this topic, term web application includes all three layers of application architecture (database, server, UI) thus making it easy to confuse it with SaaS. Yes, web application can be a SaaS on its own but SaaS is not limited to web application but it can have multiple connected clients communicating with the server.

2.2.4 Connected Application

Connected application is platform specific, native application that can connect to SaaS server to retrieve and post data. In mobile world, two major platforms are Android and iOS and in computers Windows and macOS.

To be categorised as connected application it must access the server through some API. OS or mobile / computer division does not matter. To enable seamless access modern APIs are commonly designed with REST principles but SOAP is still used, especially in older products that do not have server side implementation rewritten.

2.3 Cloud

Like noted in the chapter 2.1, SaaS requires cloud where the application is hosted. There are multitude of options when considering cloud. If the application is simple enough it can be hosted from simple web hotel. If the security and data privacy is of high importance one can have on-premises servers or dedicated part of the larger server farm.

The rapid growth of IaaS markets in the last years gives the clear signal that instead of web hotels or on-premises servers, applications are hosted from the public cloud more and more. The growth continues as reasonable prediction for the year 2017 is double-digit growth of IaaS markets (1reddrop 2017).

Public cloud has important benefits over the other solutions. When compared to web hotels there are virtually no restrictions of used technologies while web hotels usually set tight restrictions what technologies you can use. Cost effectiveness through scale is also a major contributor for public cloud success (Savvas, 2014). Bigger cloud providers have automated monitoring, load balancing and scaling options available for everyone, features which are mandatory for all publicly available SaaS solutions.

While there are many IaaS providers, two major contesters stand out. Leader in the market is Amazon with their Amazon Web Services, or AWS. Second in the race is Microsoft with its cloud platform Azure, but the margin is substantial between number one and two. Amazon holds over 30% market share and Microsoft little over 10%. IBM and google hold less than 10% market share. Next 20 biggest providers have roughly 25% market share combined, well below that of AWS (Ramel 2016).

Considering growth of these platforms Google comes first with considerable margin to Azure. Market leader AWS is also showing healthy growth and while the numbers are smaller compared to Google or Microsoft they are likely to maintain their market leader status for some time.

3 PROOF OF CONCEPT

The POC for this project has four separate components; database, server application, web application and mobile application. Selection process for tools and technologies for each component was based on MVP mind-set, meaning maximizing the result while minimising the effort. Synergies to other projects led to final technology (table 2).

TABLE 2. POC Technology stack

Component	**Technology**
Database	MySQL Community Edition
Server	PHP Slim Framework 3
Web Client	Angular 2, Bootstrap 4 Alpha
Mobile Client	Android API 19+

MySQL is the leading open source database engine currently available (Solid It, 2017). While the referenced study does not directly calculate the real usage of database engines the results still give a very good prediction of what are the most popular choices. In general, MySQL is enterprise grade database engine that has good tools available for development.

PHP had its peak in early 2000 when Mark Zuckerberg decided to build Facebook with PHP. While the usage of the language has declined since, PHP is still relevant player in the market (BuiltWith.com, 2017). In fact, when comparing the usage statistics, PHP is a leading platform (figure 3). In this project, backend serves only a REST API which lead to a decision to use Slim framework 3.0. Slim framework is lightweight, easy to use and fast to setup.

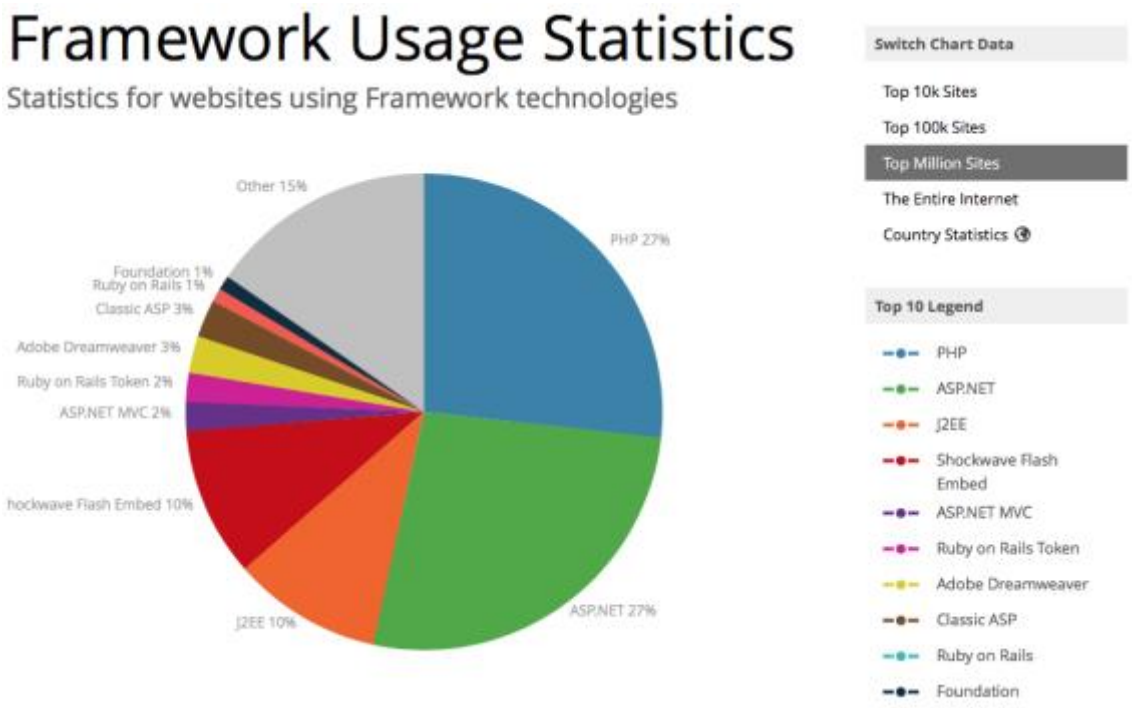

FIGURE 3. Framework Usage Statistics from BuiltWith.com

While other selected technologies are in the "go for the biggest" –category, the web client is totally different. Decision for using Angular 2 was based on the need to evaluate Angular 2 as technology and it was seen as a great synergy for this project. Despite its name, Angular 2 is not a successor for Angular 1 but a new product all together. It uses Typescript as a language and introduces modular structure with modules and components built in. Angular 2 was accompanied with new and still in alpha state version of popular CSS style library Bootstrap.

Mobile client technology was also selected from the biggest –category. Android has the lead by far, having more than 80% market share (Statista.com, 2017). Global operating system market shares are shown on the next page (figure 4).

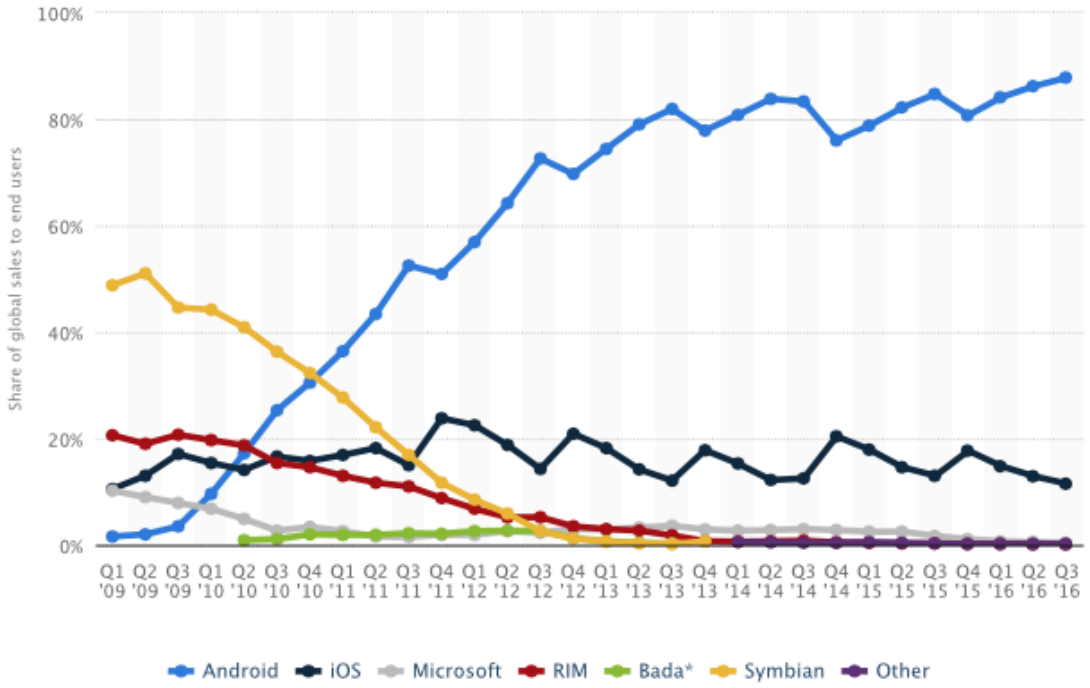

FIGURE 4. Mobile OS market share comparison from statista.com

With Android, one must also select which version and API level to support. Google provides statistics on how popular each Android version is. Selection for application API level is essentially a compromise with features and possible users. As mobile client for project does not need all the latest features, the selection was made to go with the lowest API version that has significant user base. At decision time, it was API 19 or Android 4.4 KitKat which has over 20% individual and 86% cumulative distributions (figure 5).

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	1.0%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.0%
4.1.x	Jelly Bean	16	4.0%
4.2.x		17	5.7%
4.3		18	1.6%
4.4	KitKat	19	21.9%
5.0	Lollipop	21	9.8%
5.1		22	23.1%
6.0	Marshmallow	23	30.7%
7.0	Nougat	24	0.9%
7.1		25	0.3%

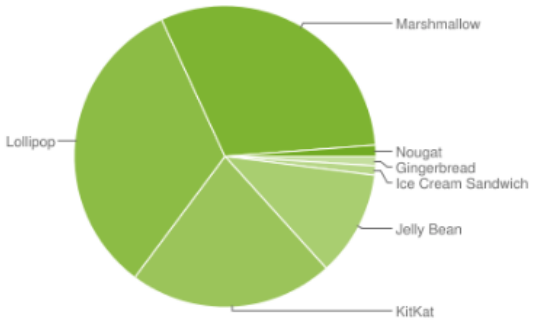

FIGURE 5. Android version distribution from Android developer portal

3.1 Database

MyCar database is made of four components; attributes, stations, identity and access management and user data. In the context of this database, term component refers to logical isolation of tables and does not include any physical isolation schemes. All tables are located in same database and instance.

Each component defines set of tables dedicated for one purpose (table 3).

TABLE 3. Database component explanations

Component	**Responsibility**
Attributes	Holds service level attributes that are used to enumerate selectable values
Identity & Access Management	Holds registered user info and their respective access tokens
Stations	Holds data about stations, fuel prices, ads etc. All data that is related to fuel stations.
User Data	Holds data entered by individual users. Vehicle and fuelling data are currently supported

Final database for POC is well normalized and does not hold many-to-many relationships (figure 6). Use of composite identifiers have been avoided, however there are few cases where composite ids are natural and therefore used. For easy database replication, an initialization script was also created (Appendix 1.).

On the side note, composite ids can be challenging for ORM libraries and present limitations for those. Even though this project does not use any ORM library in the server, it's still valuable consideration to limit the usage of composite ids making possible migration of server implementation with ORM easier.

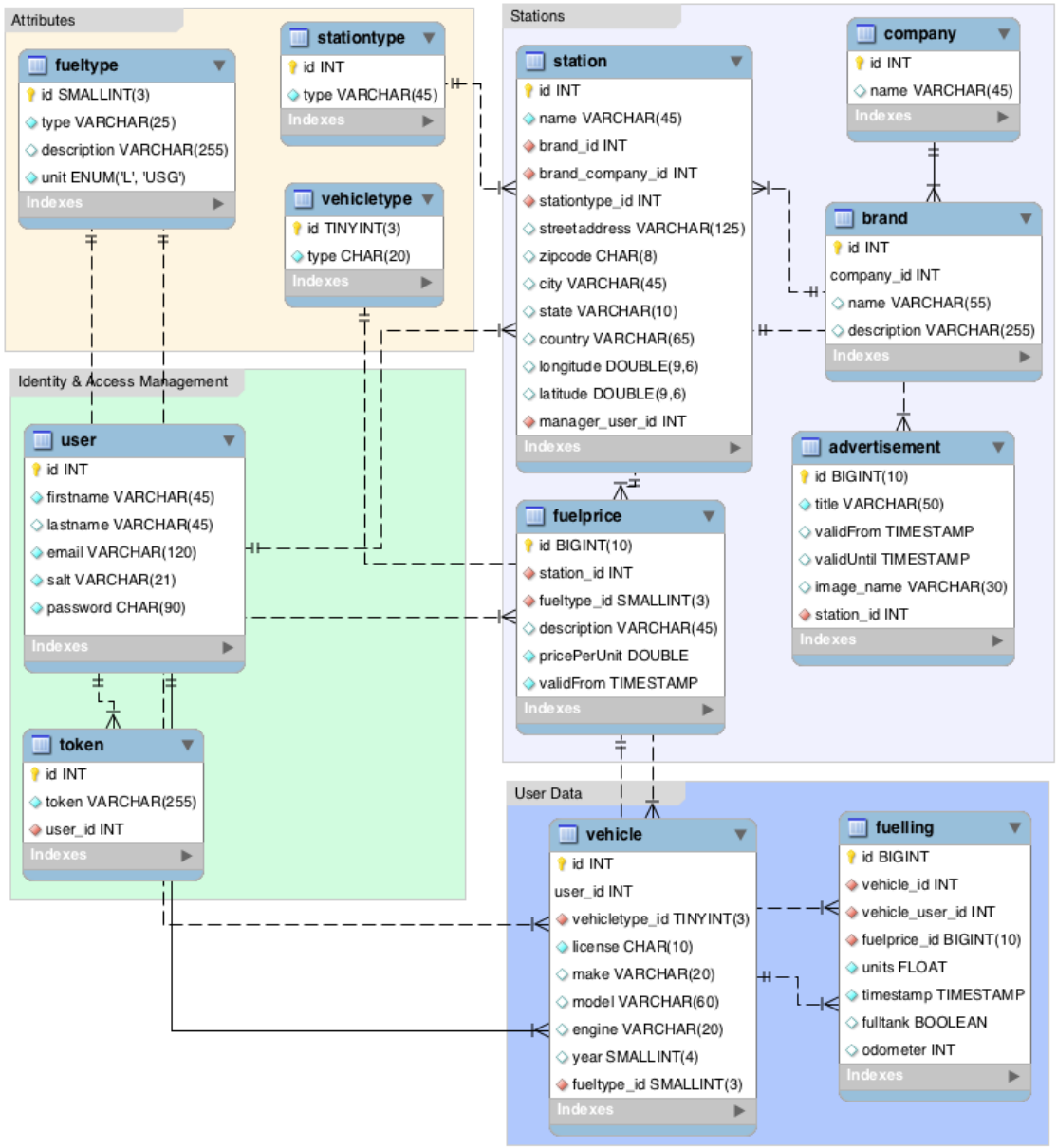

FIGURE 6. MyCar Database ER Model

3.2 Server

API is designed to conform RESTful principles. In practice this means proper use of HTTP methods – Get, Put, Post, Delete and one URL per resource design pattern. Proper use of HTTP status codes is advised, even though they are not mandatory by any means.

REST is stateless architecture, meaning that server side does not know anything about the client or its state. Because of this approach, client must always send all the details that are needed to process the request to server. Most obvious example of this is user login information. In session based architectures the server knows who has logged in but with REST client must provide authorization information with every request.

Modern convention is to use token that carry information about the user performing an action. There are multitude of options for tokens but standardized implementation is JSON Web Tokens, or JWT (IETF, 2015).

The leading design principle for server is modularity. Each logical entity is designed as dedicated API, providing necessary methods and endpoints that are relevant to that entity alone (table 4).

TABLE 4. Server API descriptions

API	**Description**	**Methods**
Identity and access management (IAM)	API for creating, updating and authorizing users	Create user Update user Login Logout
Stations	API for accessing fuel station data, like station names and locations, creating and updating station information	Create station Update station List stations
Prices	API for accessing, creating and maintaining fuel price information	Create price information List current prices List price history
Advertising	API for creating and maintaining station specific advertisements	Create ad Publish ads Remove ads

The implementation target for the server in this project was to create a proof of concept, therefore IAM API only implements logging in. One user was added to system when database was created for demonstration purposes. User creation, update, logging out, token based authorization and advertising API were left out.

3.2.1 Implementation

API is implemented in PHP. Basis for the project is Slim Framework v3 that gives solid baseline for REST API development.

Implementation is divided to three components; Common, Controllers and Models (figure 7). The Models part was only experimental and seemed only to complicate things rather than solve anything and after couple of resources models pattern was discarded. Using models would be more beneficial when used with ORM library. As this was not the case, models did not provide any meaningful benefits to the implementation.

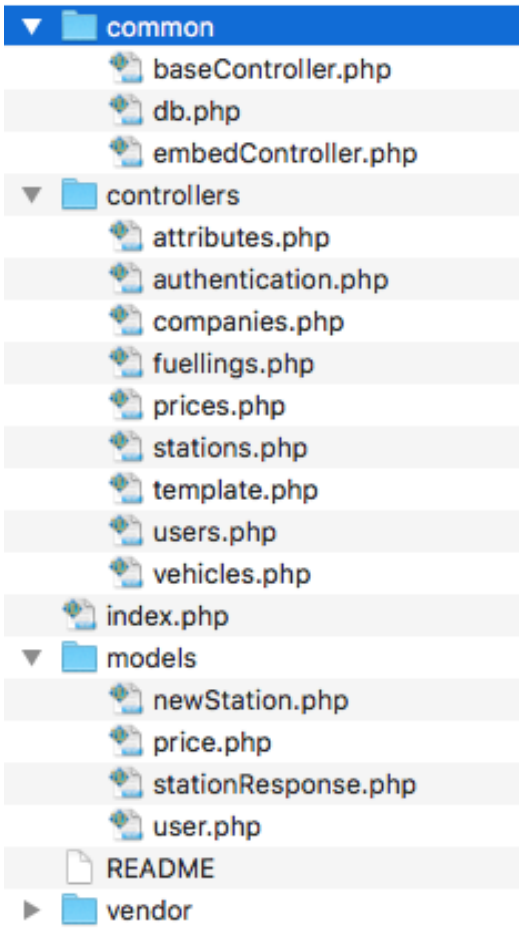

FIGURE 7. Directory and file structure of server implementation

Common parts contain database functionality as well as base classes for actual controllers that handle the resource requests. There are two base classes, BaseController and EmbedController. Base controller automates some tasks, for example JSON decoding of request bodies. When there is a need for finer control for the request handling, embed controller comes to play.

Controllers are where the magic happens. Requests are transformed to SQL queries which are sent to underlying database and returned results are transformed to JSON before sending them as response.

Routing for controllers is implemented in file called “index.php”, which is located in root directory of the API implementation (Appendix 2.).

3.3 Web Client

While the web client could serve both administrative users, those being the ones that add and update public information to system, and regular users alike, main purpose for web client POC is to cover the needs of the former group.

Technology stack for web client was chosen to be Angular 2.0 release that was made public late 2061 and Bootstrap 4.0 Alpha 4. Even though it is in alpha state, library turned out to be in good enough shape to allow POC implementation for the web client.

Web client design follows the Angular 2 architecture with modules, components, services et cetera. Client architecture consists of three modules, excluding the main module that is used to launch the application. Each application view is designed as module, meaning welcome page, login view and station manager view. Navigation bar that is shared across all modules should be a component.

Each module is divided to logical components. Basically, one logical group of elements on the UI is considered as component. Each module also should implement its own services based on the needs for that module. This way, amount of global code is as small as possible which in turns affects positively on the debugging and implementation efforts.

3.3.1 Implementation

Project structure and required tools were obtained with open source command line tool called Angular CLI. There are only two additional dependencies that do not come with the Angular 2; module for Google Maps and UI Router that replaces the Angular built in router.

In case of module for Google Maps, the motive for using that dependency was purely on effort. It was much faster to get maps integrated to client with readymade module than it would have been by writing it from the scratch. For the UI Router decision was based on the familiarity. UI Router has been de facto routing solution for Angular 1 making the change fairly trivial for Angular 2.

Implementation follows design principles closely. There are only two global providers, also known as services, one for attributes and one for user related operations as login and logout (Appendix 3.).

The relation between visible UI elements and Angular components can be seen in the screenshots of Angular application (figure 8). The price list on the left side is logical group elements, effectively showing the fuel price of various fuel types. That part of the source code has its code isolated to dedicated component (figure 9).

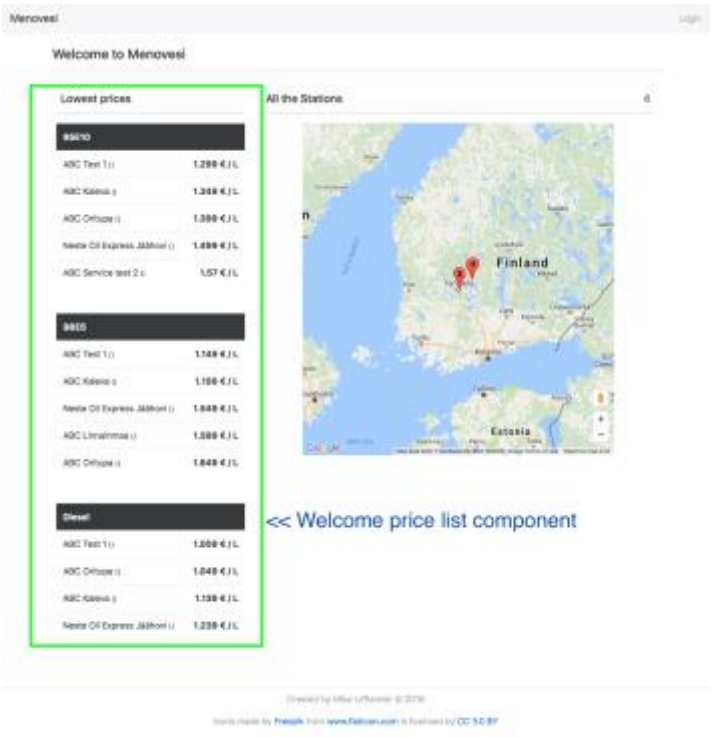

FIGURE 8. Web client welcome page

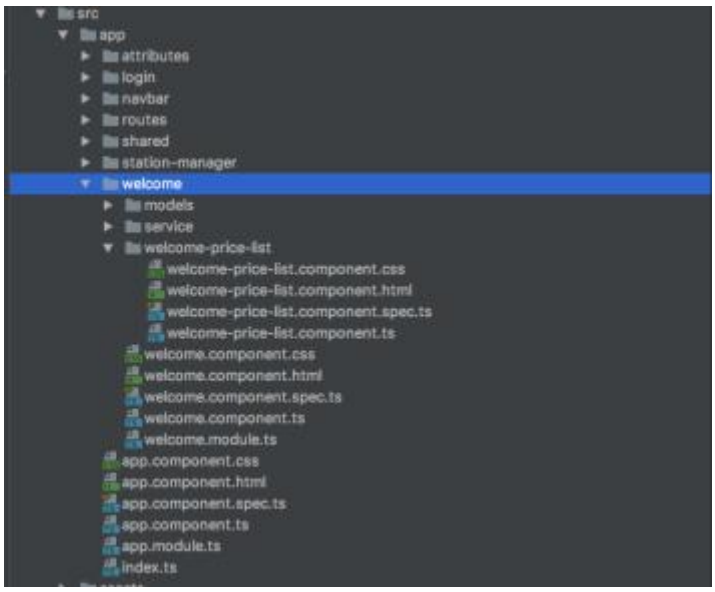

FIGURE 9. Web client welcome module components

Each component is made of four files, each with special purpose (table 5). For the POC implementation the automated tests were left out.

TABLE 5. Angular 2 component file purposes

File	**Purpose**
*.component.css	Component specific CSS styles
*.component.html	Component HTML template
*.component.spec.ts	Automated tests
*.component.ts	Controller implementation

In terms of implemented features, the web client proof of concept has very basic authentication with login screen, a welcome page showing the prices on list and map and simple manager module to maintain information about fuel stations and prices.

3.4 Mobile Client

Goal for mobile client is to cover the needs of regular users of the service. Regular users that use the service to find the information and possibly storing their personal data to the system.

Like mentioned in chapter 3, technology stack for the mobile client was chosen to be Android API 19 or more commonly Android 4.4 KitKat. Required features from the

platform are well covered by that version and has very good distribution amongst the Android devices.

Mobile application is designed to have three views for user to interact with. At start, user will be shown a map with his/her location highlighted. Map should show markers for fuel stations which have coordinates set and indicate which station has the lowest price. Second view for the application is list view which shows stations, fuel price and distance to each station. List should be ordered ascending by distance.

Third view is reserved for settings. Application should provide at least selection for preferred fuel type as well as refresh settings.

3.4.1 Implementation

Implementation of mobile client was pretty straightforward with Android Studio. Project structure and necessary resources are automatically generated when new project is created. As all other Android applications, also this one is written in Java and enforces best practices of that language; inheritance, packages and so forth.

The source code is divided to packages based on their intended use and role in the application. There are total of 6 packages (figure 10). Maps, station list and settings contain user interface implementation, station package holds data model and utility functions for ordering the lists, networking implements JSON data download functions and base package provides abstraction for activities that require menu bar or need to access Google API services.

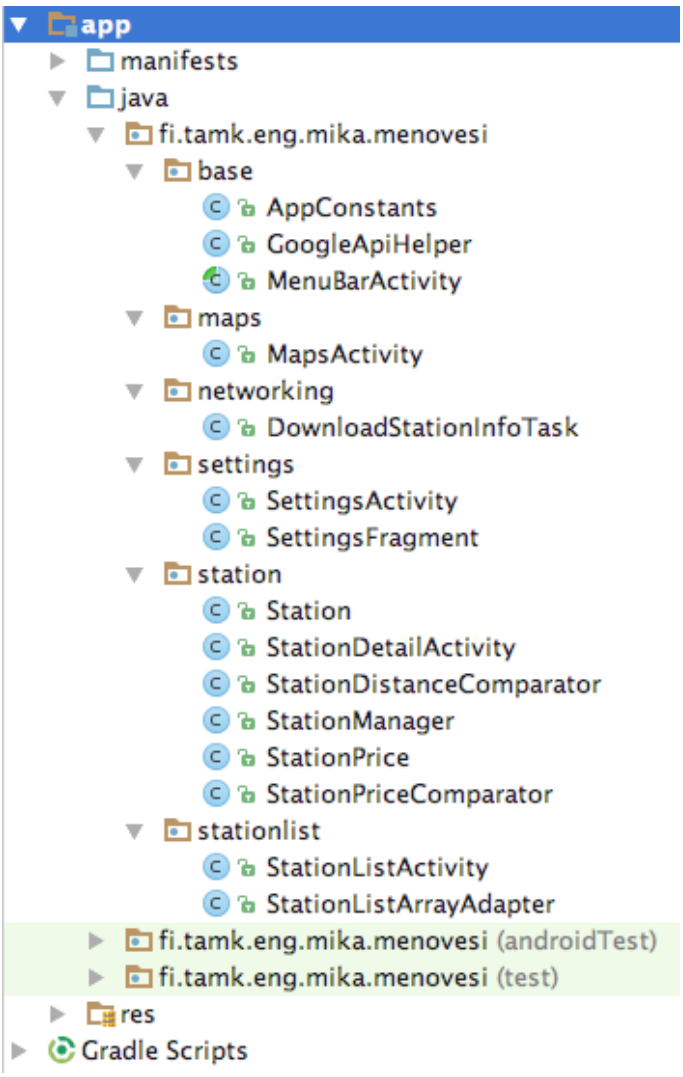

FIGURE 10. Mobile client package structure

By default, UI classes known as Activities, are reloaded every time they are navigated to by the user. With shared menu bar and possibility to navigate back and forth of multiple views whit presents a problem. For example, if user opens view 1, navigates to view 2 and then navigates to view 1 without pressing the back button, UI stack will have all three views.

Solution to this problem is to load views only on the need basis. To do so, platform must be instructed to reorder the view back in front of the UI stack instead of reloading it completely (Appendix 4.).

Two of the three views available to user shares same dataset. Map and list views present the same data only in different format and order. While it would be technically possible to make both views to download the data they need, a much better approach for this is to

use singleton object that will be available to all views needing it. This is where class Station Manager comes into play (Appendix 5.).

3.4.2 Application views

On application start, the map view is loaded by default and is used to show users current location as well as fuel stations identified by default map markers. Cheapest option(s) has green marker (figure 11). If needed, user can launch a default navigation application by tapping the desired marker and tapping Navigate from the bottom of the screen (figure 12).

In order to change to other views, user needs to tap either List-icon (List view) or Menu-icon (Settings) in the right side of the navigation bar.

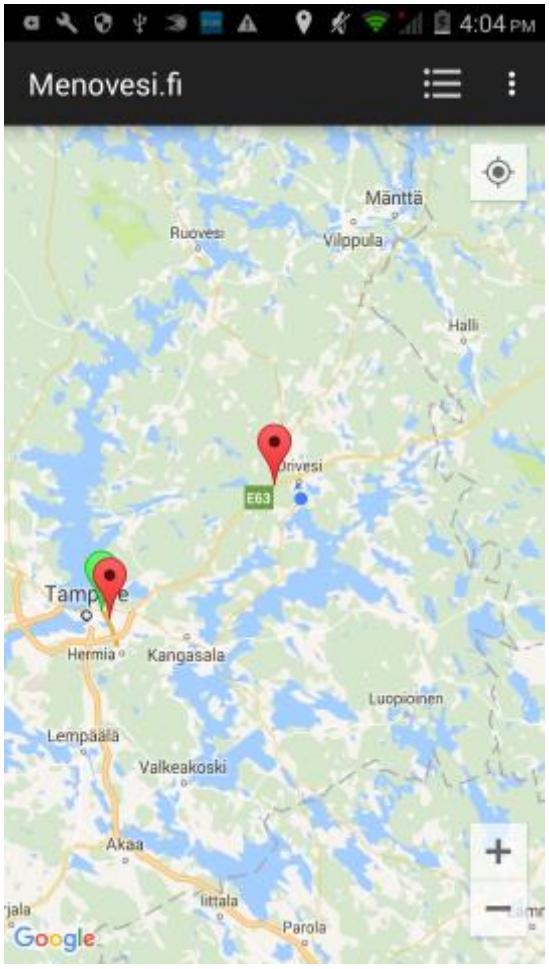

FIGURE 11. Default map view

FIGURE 12. Map view with selected marker

The other option for station information is the list view. In this mode, stations are listed in ascending distance order, meaning the closest station to users' current position is listed at the top (figure 13).

In order to change views, user can either tap Map-icon or phone Back-button to return to the map view or Menu-icon to go to the settings.

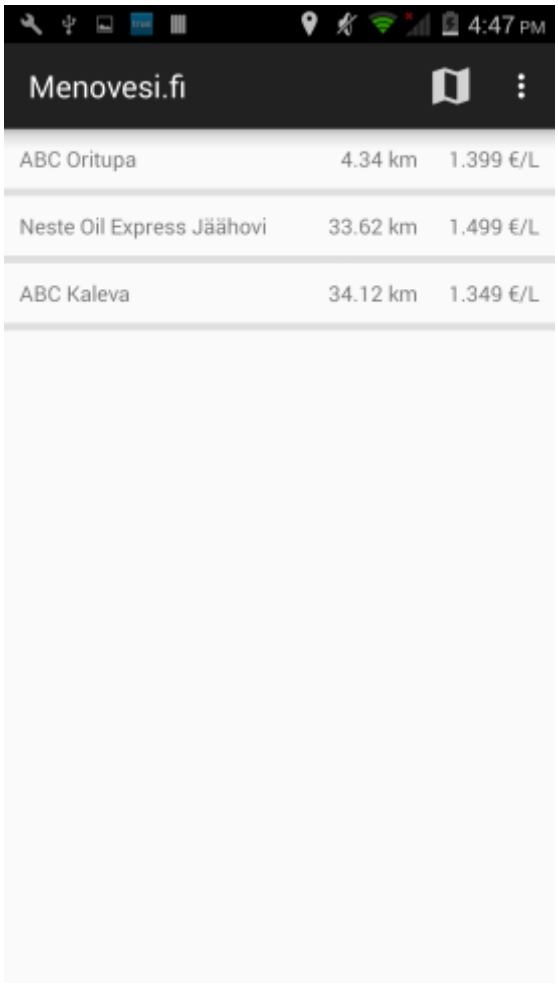

FIGURE 13. Station list view

Third view is settings view (figure 14), from which user can change the behaviour of the application. User can change the preferred fuel type, how often price data is reloaded and how often distance info should be updated.

In order to change the view back to previous one, user must use phones Back button.

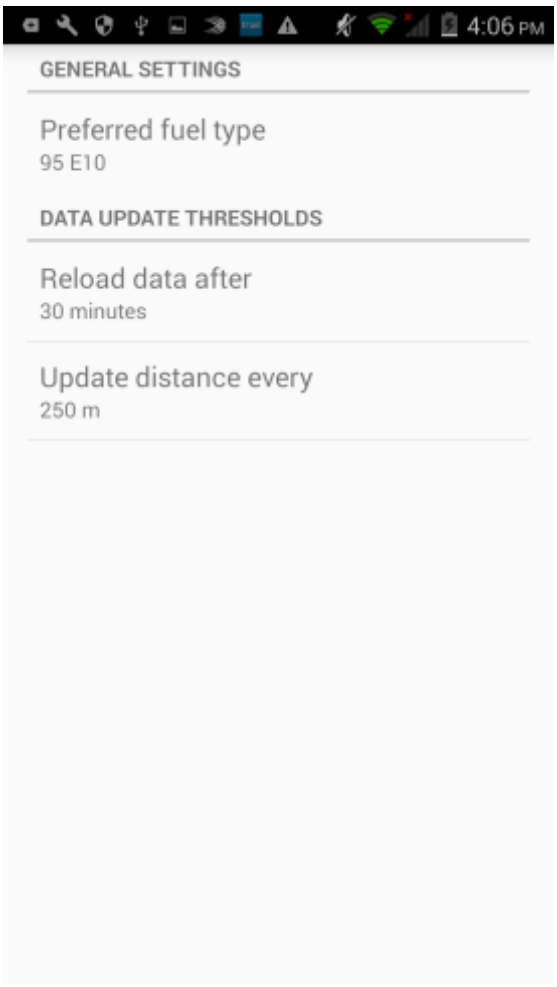

FIGURE 14. Settings view

3.5 Deployment

To evaluate POC and feasibility of the service, all parts of the service must be made publicly available. As described in chapter 2.1, a web application does not need a physical distribution channel. With mobile client on the other hand, some form of distribution channel is required.

3.5.1 Database and Web Application

For database and web application, there are multiple options how it can be distributed, however purchasing and maintaining dedicated server for the purpose is not viable option. Purchasing a service from web hotel would be a better option, but when costs are taken into account with the limited time frame the POC is supposed to be run, this option soon runs out of any competitive edge.

This means that only one relevant option remains, public cloud. Selection for the service provider comes down to three biggest options; Amazon, Microsoft and Google. All the providers have similar possibilities available and are priced quite similarly. While evaluating these three options, fourth contester presented itself for the task – TAMK.

TAMK provides hosted network platform for students to test and deploy their applications in very similar way with web hotels. They also provide MySQL instances with multiple database that can be used for the same purpose. Both web server and MySQL can be accessed over the internet thus making it a feasible for the POC. As this option is free of charge for the students, it clearly has the best cost-benefit ratio.

All the factors taken into account, serving the web application and database from the TAMK server was seen the best option.

3.5.2 Mobile Client

Android application can be distributed in three ways. Either through the Google Play Store, sharing the readymade installation package of the application or sharing the source codes for the application which can be compiled and deployed to phone.

Option one, Google Play Store is not viable as the application will not be production ready or even MVP at the time of testing. While it might be possible to publish it in play store it does not provide any benefits to do so, mainly because gathering user feedback would be nearly impossible with anything else than controlled test group.

From the remaining options, both were seen as good candidates. Source code was made publicly available with Bitbucket collaboration solution. Any interested party can get the codes, compile and run the application for themselves. Installation packages we also provided per individual requests for evaluation purposes.

4 MONETIZING THE SERVICE

As the goal for the project is to create financially profitable SaaS solution, the money has to come somewhere. In the current scope, there are three parties to the system that can provide financial input for the service provider; users who create and share the data, users who use the data and advertisers. This creates more options for monetizing than common provider-user structure.

There are some best practices and proven revenue models with SaaS (Bishop, 2014; Vogel, 2015). Options range from free for everyone to paid applications and subscriptions. Somewhere in the middle are the free trial and freemium revenue models (figure 15).

	Paid	Free Trial	Freemium	Free Forever
Type:	Pay for first use	30-days free	Free for individual, upsell for business	Free but monetize by ads, adjacent products
Type of Company:	Workday	Zendesk	Box	Spiceworks, Wave Acct
Benefits:	Highest value	Lower cost sale	Viral adoption	Easy adoption
Challenges:	Requires heavy marketing and sales	May get "looky-loos"	Free features vs. paid features	Need high usage to monetize
Works Best:	Heavy integrations or up front implementations	Considered purchase	Tipping point - usage, manageability, features	Ability to monetize through other products
Comments:	Big SaaS companies	Must be easy to use	Requires large number of users	Fewer companies go this route

FIGURE 15. SaaS revenue models from Scale VP blog by Stacey Bishop

Last option, free forever, together with the requirement for financial profitability limits the revenue options to advertising and other paid services and/or products. First three options on the other hand provide additional options. The question is who should pay for the service, when and how much.

4.1 Paid Applications

The classical model for monetizing software is selling the software one by one. Basic principle is that user pays the fee before he/she is granted access to the software and user

can use the application as long as he/she wants. This is still very common business model with standalone desktop and mobile applications.

Paid applications provide high value as the revenue is not affected by how long the user stays with the product. When dealing with heavy up front implementation costs this revenue model can be well desired. Users who have made the purchase are also good candidates to be engaged with the product due to the financial investment they made to acquire the application.

To be successful, paid application revenue model requires heavy sales and marketing effort. Uses are generally less likely to pay for the application they can't try for free. Also, the application should provide wide range of features to be compelling option in the current market.

Adoption and revenues for paid applications have great variance on different platforms. According to the report (Dilger, 2016) Google Play Store gets double of downloads compared to the App Store, while latter creates double the revenue (figure 16).

FIGURE 16. Comparison of app downloads per platform from Apple Insider article

This indicates that for high adoption rate Android should be prioritized but as Android users are less likely to pay for the application this can also cripple the revenue.

To be viable service, MyCar needs to have application for both platforms. As paid applications require wide range of features and heavy marketing efforts, this revenue

model also requires heavy upfront investment to both development and marketing. In addition to turn-by-turn navigation and fine-tuned user experience for each platform, the mobile applications do not necessarily provide any added value to the user making them less likely to pay for the application.

In the context of MyCar.com, paid application would also give access to all background services and web client. After purchase user would be allowed to create user account and use the service as he/she sees fit.

4.2 Free Trial

Free trial applications differ from the paid applications for the short period of time. Users are given a short period when they can try the software for free and if they want to continue to use the application after that period they are required to pay for it. It's quite common that trial period is between 14 and 30 days.

Free trials tend to ease the effort for customer acquisition and may lower the cost for initial marketing efforts while the need for upfront development investment remains roughly the same. Customer retention, meaning the ratio for purchase per trial, still requires considerable marketing investment.

There are technical limitations with free trial applications. For example, Apple App Store does not currently offer any means to offer free trial for paid applications. This limitation combined with the fact that iOS and App Store are generating most of the revenue as seen in figure 16, reduces the benefits of free trial revenue model considerably.

4.3 Freemium

In freemium model parts of the service are given free of charge while other parts of it are gated behind purchase (Munir, 2014). Freemium model offers variety of different strategies which can be applied to MyCar.com. In general, choosing the freemium strategy is matter of choosing which part of the service is free and which require payment.

First and very common strategy is gated features. In this strategy, some of the application or service features are free while others and commonly more advanced ones require

payment. Usually, unlocking gated feature requires one-time payment. In the context of MyCar.com free features could be related to reading the information and paid features could allow storing the data of owned vehicles.

Other possible strategy is gated content, which is also called Paywall. Unlike in gated features strategy, all features are free to use but some of the content requires payment. Good example of this strategy is Financial Times. It's quite common that unlocking gated content is based on continuous payments as active subscription, however it's also possible to make it similar to gated features so that users will pay one-time fee to access specific content. In the context of MyCar.com gated content could be for example real time price information. In practice this would mean that paying customers have access to real time data while others have access to still relevant but slightly delayed data.

Third feasible strategy for freemium model is division of personal users and business users. Anyone using the service for personal gain has free access to all data and features. Company users who use the application for financial gain are required to pay for their usage. Applying this strategy to MyCar.com would be fairly straightforward as the division of business and personal users is very clear.

As discussed earlier, customer acquisition can be easier when they don't have to pay for the service. They are also more likely to continue to use the service for the longer period of time as they can do so without investing any money. The challenge in this model is finding the right balance of free and paid features. Too few free features and customer churn will be high, too many free features and nobody wants to pay for additional ones.

Important metric with freemium model is conversion ratio. In this context, it means how many of the service users are paying for the service. While one might easily think that higher the better, it's not necessarily so. Very high conversion rate could also mean that users don't find the free content interesting enough and causing new customer acquisition more difficult. According to the Harvard Business Review article, good target for conversion rate is somewhere around 2% to 5% (Kumar, 2014).

4.4 Free Forever

Service can also be offered free of charge - forever. While free product is the most effective model for new customer acquisition, it also presents challenges for profitability. The money has to come somewhere and while there are couple of different options for that, they all require wide adoption rate to be successful.

Advertising is obvious answer to how to gather revenue with free software. Ads can be integrated to all clients and can be targeted to specific groups. With large user base, advertisements can create steady and decent income, but it's highly unlikely that ads alone would be sufficient to make business profitable. Therefore, it's crucial to have other revenue options available.

Free software can also be used to introduce other services and/or applications which are not free. Revenues of those services could be used to cover costs of free offering. For example, LinkedIn offers their service for free, and they are selling advertisement, premium services and recruiting tools. When the product itself has natural options for additional paid services this can be profitable model.

However, there are also a lot of unknowns when it comes to advertisement making revenue calculations and estimations very tricky matter. Unlike in paid business models, the number of actual downloads is not the metric that matters most, but the time one user actively uses the application is, metric that can and will be very hard to predict.

Mobile advertising revenue is measured by revenue per thousand ad impressions, where one impression means one ad fetched from the server and showed to user. To calculate the revenue, we need to first figure out some key variables (table 6). Values it the table are estimates based on the statistical data (Chaffey, 2017; Adtapsy.com, 2017).

TABLE 6. Revenue Calculation variables

Variable	**Description**	**Value**
Conversion rate	Percentage of users who will continue to use application after first try	10 %
Network Attached	Percentage of users who have access internet at any given time	100 %
IPM	Ad impressions per minute	2
Fill Rate	Percentage of delivered ads per requested ads	75%
eCPM	Revenue per thousand ads shown	4.25$

Let's calculate possible revenue per 100 000 downloads. Since we are very positive about our service, we predict a conversion rate of 10%, which is similar to mobile gaming. This gives us two figures; 100 000 * 0.1 = 10 000 users who continue to use our application after first day and 100 000 – 10 000 = 90 000 users who use it only for a day.

Next thing is to estimate how many of these users have network connection. Because application is not usable without internet connection, we can safely assume this variable to be 100%.

The next factor is how many ads can be shown per minute, or ad impressions per minute. For this calculation, we assume that one ad is shown for 30 seconds which gives as a impressions per minute value 2.

Fill rate describes how many ads are actually shown versus how many have been requested. There will always be network issues like errors and timeouts and user can leave the app before the ad is shown. All of these factors lower the fill rate which we predict to be 75%.

eCPM varies a lot by country (Adtapsy.com, 2017). By the time writing this report, last month's (3/2017) eCPM in Finland was 1.74$, UK 7.01$, Germany 3.94$ and France 4.31$. With these four countries, we get average eCPM of 4.25$.

The equation for revenue calculation is multiplying the amount of impressions divided by thousand times eCPM value. Amount of impressions is calculated from amount of users' times minutes spend in application times impressions per minute. Amount of

impressions is divided by 1000 because eCPM is revenue per thousand ad impressions. This leads to following equation.

$$\frac{users * minutes\ used * imperssions\ per\ minute}{1000} * eCpm\$ = revenue\ \$$$

For the users who try and leave our application during the first day, we predict that they will stay 5 minutes in our application before leaving. Above mentioned equation gives us

$$\frac{9000 * 5 * 2}{1000} * 4.25\$ = 3825\$$$

For the 10000 users who stay with our service we need to calculate another variable – CLV or customer lifetime value. This metric represents the amount of money single user spends on premium services, in-app purchases or advertisement. Since we are only looking for advertisement revenue, we ignore the first two cases.

For the calculation, let's assume that user stays with our service for 6 months before he/she loses the interest. During the 6-month period, user needs our application twice per month and spends 6 minutes per usage inside the application. Please note that these values are estimates based on writers' personal experience.

Based on the values mentioned above, we get 144 impressions per user for the whole time they spend with our application.

$$6\ months * (2 * 6\ \text{mins}) * 2\ ipm = 144\ impressions\ per\ user$$

We can add those 10 impressions they will get from the first use, the same number we get for those who do not stay with our application. This leads to total 154 impression per user for the application lifetime.

$$144\ impressions\ over\ 6\ months + 10\ impression\ for\ the\ first\ use = 154\ total$$

Total revenue for these users is calculated with following equation.

$$\frac{(total\ impressions\ per\ user * amount\ of\ users)}{1000} * eCPM\ \$ = revenue\ \$$$

With the actual values, we get the result

$$\frac{154 * 10000}{1000} * 4.25\$ = \ 6545\$$$

When we add the revenue from one time users and returning users together we get

$$3825\$ + 6545\$ = 10\ 370\$$$

This means that for 100 000 downloads, we would generate a bit over 10 000$ revenue. But this is not the whole truth because it assumes 100% fill rate, which is impossible to achieve. With the 75% estimation mentioned earlier we get 7777.5$ revenue per 100 000 downloads.

$$10370\ \$ * 0.75 = 7777.5\ \$$$

It's also worth to note that this calculation uses average value for eCPM which varies a lot based on the country. If the application would only be available in Finland, these values would be very different. Obviously other variables are also subject to change based on application but this gives and rough idea how much revenue could be generated with advertising.

5 PROOF OF CONCEPT ANALYSIS

The goal for MyCar.com is to be a SaaS solution with web and native mobile application user interfaces. It aims to serve two distinct user groups, consumers and business users with specialized features and dedicated interfaces for each group. Ultimately, consumers will share the benefits created by business users.

That being said, the correlation between number of business users and consumer benefits is very clear. In practice, it comes down to the fact that more business' there are adding the information about their offerings to the system, more relevant this service is for the consumers. Unfortunately, it's also vice versa. More consumers there are using the application, more appealing the service is for the business users.

Like any other business, reason for MyCar.com existence is to become financially profitable. To achieve profitability the revenue model must be carefully selected that suits the service and business environment. There are multiple models to choose from, each one with pros and cons. While there is no silver bullet to this matter, some choices suit the MyCar.com better than others.

5.1 Architecture

MyCar.com architecture is made of three layers (figure 17). Basis for the whole service is data warehouse or database. Data which is stored to the system is both structured and relational. As these are characteristics best served by relational SQL database, it is an obvious choice (Gent & Rabeler, 2017).

In the middle is server, which communicates with the database and serves the data to web and mobile clients. Server must keep response times to a reasonable minimum at all times to keep the users happy making it imperative to implement scaling solutions.

On the top layer, there are two web and two mobile clients. Business users should have own client or at least own user interface separated from consumer interface. Obviously, the business users interface could also include consumer interface, but not the other way around.

Mobile clients would be developed for major platforms. By the time of this project, there are two such platforms in the market, iOS made by Apple and Android made by Google. While there are others too, their market share is negligible causing any development for these platforms to be wasted effort (Statista.com, 2017).

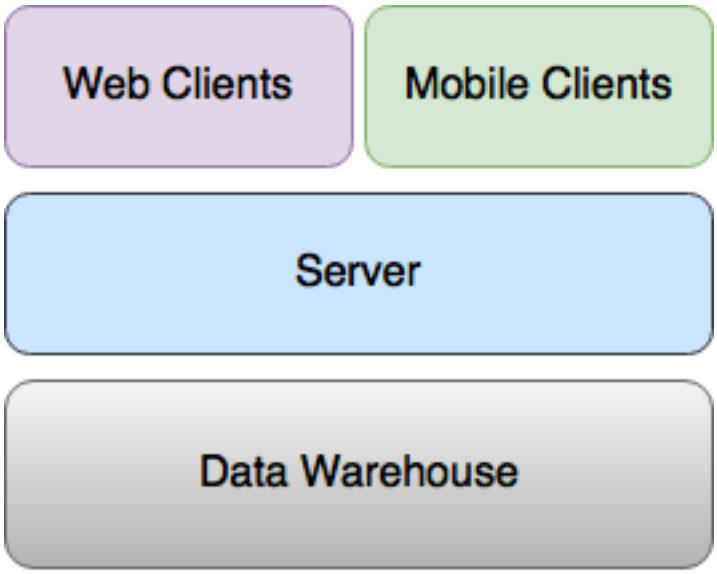

FIGURE 17. MyCar.com architecture

Chosen architecture suits the MyCar.com very well. Each part of the service is well isolated to independent entities which makes the development and maintenance of the service much easier. Biggest architectural decision is whether to go with native applications or implement one web application that has mobile view. While both are viable options, there are characteristics that lean towards native applications (Summerfield). It's also notable that 90% of the time people spend with mobile devices is spent on applications (Chaffey, 2017).

Native applications will allow the use of all the features including navigation. Should the service be developed further to cover for example car repairs, a calendar integration for repair time reservations would also be very nice.

Division of web clients to two distinct applications, one for business users and one for consumers enables separated development efforts and once again improves the maintainability of the whole code base. While it's not mandatory to separate those two, personally I'd recommend it for the before mentioned reasons.

5.2 Technological Selections

Each layer in application has its own requirements. Database needs to match the needs of inserted data and while server has implementation, maintenance and performance requirements. Client applications, web and native ones, have needs for features, user experience and performance. All of the above also require workforce should the service be developed into full blown SaaS solution, which also affects the technological selections for the product.

5.2.1 Database

The stored data is mostly made of numerical values and dates. Almost all of the stored data is relational and linked to specific user, either business or consumer. These two use cases are what the relational databases were designed for, making the SQL database an obvious choice (Gent & Rabeler 2017).

When it comes to the choosing the SQL provider, first choice is between open source and commercial solution. Oracle is leading SQL provider in commercial versions and MySQL is Oracles counterpart in open source world (Solid IT, 2017). There are no clear benefits for commercial SQL provider with MyCar.com, so the selection will lean towards open source alternatives.

Like said, MySQL is probably the most used open source SQL engine making it a safe bet. The newcomer in the market, MariaDB 10 is also an option. MariaDB is plug-and-play replacement for MySQL making migration easier. MariaDB comes with security features that are not available in free version of MySQL such as data-on-rest encryption and has improved performance in highly multithreaded systems over MySQL (Fournier, 2015). For the production deployment, I'd recommend using the MariaDB over MySQL, but it's a matter of taste. Both of these engines suit the needs very well.

There is also PostgreSQL as one open source option. Quite widely used, comes with lots of features but lacks in development tools. As the database needs to be actively developed the lack of good tools can slow down the development process.

5.2.2 Server

First version of the server was made with php. While it's suitable and fairly straightforward to implement these kinds of APIs in php, it comes with a cost specially to those who are not familiar with the peculiar syntax of that language.

The implementation is based on open source library called Slim (Appendix 6). It's actively developed at least for the time being, but since it's the product of few interested individuals the support and documentation can be found lacking at times. During the initial development, there wasn't any features that could not be implemented in php, but at the same time there were other languages that some of the features could be implemented in easier way.

For the commercial version of MyCar.com server, I'd personally choose other language, namely Java. It's strongly typed, objects oriented language that suits better to implementations with complex object and data models (Lowmiller, 2013).

At the time of writing Java comes with very well documented and supported REST API framework JAX-RS and has ORM libraries like Hibernate that offer wide range of features (Appendix 6). With the full potential of object oriented programming making the REST API with Java is an easy operation.

Obviously, there are also other possibilities, for example JavaScript. Node.js is implementation that runs server side JavaScript and there are multitude of readymade libraries that can be used to make such server. While JavaScript does not offer all the features of Java, it's still viable language for server side implementation. However, JavaScript and Node.js are better suited to implementations that work with JSON and therefore NoSQL databases (Wayner, 2017). This is not the case with MyCar.com, thus selection between Java and PHP makes more sense.

It's also worth to note that making MyCar.com commercial product, it requires hired workforce and it's currently easier to find JavaScript developers than Java developers in the market. On the side note, I feel this is more of imago issue of Java than anything else.

Like with databases there are other options too. For example, Python comes with the framework called Django. However, it's designed and therefore better suited for server side rendered web pages than REST APIs.

5.2.3 Web Clients

Web client was developed with Angular 2. It's fairly new framework, first production release was published late 2016, only couple of weeks before the implementation started. As with all new frameworks, the support was hard to find and documentation was often outdated describing some beta versions that had been changed many times before production release. Situation has improved since and is likely to improve still.

Angular 2 applications are written in Typescript. A new language emerged from the ECMAScript definition behind JavaScript. Typescript tries to solve some of the lacking's of JavaScript but as it's essentially compiled to JavaScript it's more of syntactic sugar than different language. Typescript comes with nice features like classes and strong typing, but there is definitely a learning curve when adopting it.

Angular 2 itself is not a next version of Angular 1 but total rewrite of the framework. Like language used for development, it tries to solve problems of its predecessor. It's said to be more lightweight and faster when processing the UI. Angular 2 comes with modular and componentized architecture that promotes writing modular code. Modularity can be achieved in any language and is matter of choice so selecting Angular 2 for task just for that reason is wrong.

After the initial learning curve, Angular 2 application was easy to develop. Componentized structure makes code isolation easy which in turn makes the whole development easier. Compared to other frameworks, for example React.js, Angular 2 has clear advantage as it includes all the necessary components and are maintained as one framework. Developer will find the need for external dependencies scarce.

Angular 2 also has nice command line toolset that makes starting the new project faster. CLI also automates new file generation and other recurring tasks easing the development effort. For commercial implementation, I'd choose Angular 2 taken that skilled workforce could be found. If not, then the latest version of Angular 1 could also be used.

5.2.4 Mobile Clients

MyCar.com POC introduced only Android application. There are not that many technical selections that can be made, namely the UI native vs. HTML and toolset native vs. cross-platform.

Concerning the mobile application UI, going with native is a no-brainer when trying to optimize user experience. While HTML5 provides decent options, it's very easy to create an application that looks and smells like web application, which will lead to unified user interface and experience across platforms.

Ignoring platform conventions is a major factor why applications fail to attract users (Kamal, 2015). For MyCar.com to succeed a high conversion rate, which means users will stay with the application, is a must. As unified HTML5 interface is risk to conversion rate, it's a risk not worth taking.

On the matter of native vs. cross-platform, I'd once again go for native. It's not uncommon that hybrid apps suffer from performance issues, especially with animated user interfaces (Kuz, 2016). Poor performance is major contributor to poor user experience, which is also a big contributor to failed apps section (Whittle, 2014).

In my personal experience, cross-platform solutions will need branching the software anyways, so after the initial development impact there is no real reason to go with cross-platform solution.

For commercial version, an iOS application is also required. Development effort is going to be considerably smaller than for Android due to the feedback that can be collected from Android users.

5.3 Development Process

Development process is divided to distinct phases (figure 18). This report covers the first phase, Proof of Concept. Findings made in this report and those made during the development of POC are used as feedback when designing the MVP.

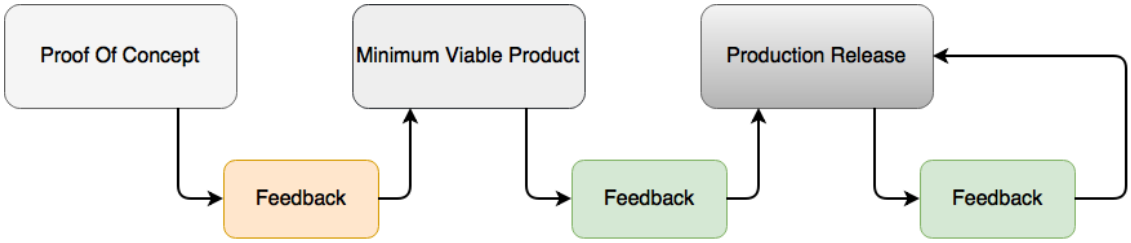

FIGURE 18. MyCar.com development process

This model works very well with experimenting new application ideas. Making the very small and very fast proof of concept lowers the risk of failure to a point that not trying out new ideas is a bigger risk than making them (Dogra, 2013).

As POC is developed, the understanding of the end result comes clearer and clearer. It also gives a great opportunity to evaluate new techniques and languages and their suitability for the task. It's possible that whatever is made during the POC phase, it's not reused as such in MVP.

Next step for MyCar.com would be making the minimum viable product to validate business viability for MyCar.com (Singla, 2016). MVP implementation includes also some technical selections that will or at least should be carried to production releases also, therefore decisions for MVP should be made with production mind-set.

MyCar.com has four components; database, server, web clients and mobile clients and while it would not be the end of the world if one of those components had to be rewrite after MVP, rewriting the all would not be an option.

Minimum viable product would most likely include real-time fuel pricing info and account management with web client. UI should look clean and finalized even though it is subject to change when going for production. This is because people tend to give feedback on the matters they see first and it does not provide much information if majority of the feedback is similar to "UI is ugly". Especially if that is known beforehand.

For the deployment of MVP, it would be beneficial to move the solution to public cloud. There are multiple providers, some better than the others. The requirements for the MVP deployment are so low that any public cloud provider can meet them, making this a matter

of personal preference. I'd go with the AWS, as it has easy to use web interface and powerful CLI tools available and I'm already familiar with it. Cloud provider should be evaluated during and after MVP and corrective actions should be taken if for some reason the original selection is found to be lacking.

5.4 Monetizing

Monetizing provides unique problems, that rarely have one correct answer. There is no silver bullet that would suit every situation and therefore it takes considerable effort to find out the correct approach.

As with all B2C SaaS products, the key metrics are user acquisition and conversion rate. No matter what the monetizing solution for MyCar.com is, this precondition remains true and should be taken into account when choosing the business model.

Paid applications model does not seem to suit very well. It makes user acquisition harder and requires relatively big investment to up-front development and marketing. Any revenue generated with this model is one-time income only, which makes it difficult to improve the solution over time. Also, it's quite difficult to determine who would be the target of paid applications and what would be the price of the application. Service should provide wide range of features to be appealing as paid application and it affects both business users and consumers alike.

The cost of the application is also relevant question, especially for the consumers. Spending 10€ for application that can save you 1€-2€ twice a month might not be seen as very attractive offer.

Free trial would help on the user acquisition, but does not solve the fundamental issues with the paid applications. After the trial period conversion rate will be low if the feature offering is not wide enough and it's very unlikely that users who have once decided against a service would come back to it later on. That being said, it's also worth to note that Apple App Store does not support free trials for paid applications, making the implementation of it very difficult across platforms.

Best option for user acquisition point of view would be Free Forever business model but it makes the profitability difficult (Bishop, 2014). With Free Forever model service must include other options for generating revenue, such as premium features, other applications and advertisement.

As discussed in chapter 4.4, advertisement requires very large user base in order to make enough revenue. The example calculation shows that every 100 000 new users will generate revenue around 7000$ to 10000$. If the profitability target is 500 000$ yearly revenue it takes 6.4 million new users each year to achieve this. It'll take quite a long time before service is attracting that many new users each year.

Advertisements also have negative impact on user experience. Ads are often seen too intrusive and distracting which in turn can lower the user conversion rate. In order to attract and retain as many consumer users as possible the user experience must be very good. Anything affecting negatively to user experience should be removed from the application.

Based on the discussion above, MyCar.com would be best served with Freemium revenue model. For businesses, there is monthly or yearly fee and for consumers' service is free, including applications. This should optimize the new user acquisition of consumers making the service more attractive to businesses. While advertising seems to be very popular choice for generating revenues, subscriptions seem to be more profitable (Munir, 2014).

To be successful this model still requires heavy marketing efforts, but those can be targeted to businesses making those marketing investments more beneficial.

MyCar.com could also benefit from other Freemium strategies. Feature gating of business user features could very well be possible. With the basic fee, they would get basic features, but for example detailed data of consumers and other businesses would require additional payments.

Same approach could easily be applied to advertisements. The applications could be implemented to show ads created by business users and ability to post them to consumers could be behind additional payments. And each shown ad comes with the price.

The real challenge is this; what features should the service offer for the business users that the service is still attractive to them even though there is a monthly fee?

Currently in Finland there is hard competition in fuel prices and it's known that business owners tend to drive across the city few times a day, just to see competitors current pricing. With MyCar.com, you could see that info online, taken that your competitor is part of the system. Are the reduced expenses enough? If the service implements a way for the businesses to advertise their products, it might make the it more appealing.

These are the questions that remain unanswered at this time. Should the MyCar.com move to a next phase, interviews and questionnaires focusing on these open questions should be conducted for possible business users. Feedback from these interviews should then be used to decide what features MVP should have.

6 DISCUSSION

The proof of concept clearly shows that the original problem can be solved. Real time information of fuel prices, other fuel station services and offerings can be made available for most of the smart phone users in 2017. SaaS has been proven to be efficient way to implement this kind of service and hard competition in public cloud is likely reducing costs in coming years.

The development process starting from proof of concept and leading to incremental production releases will lower the risk and mitigate the problems caused by locking to specific feature set or technology too soon in the process.

But SaaS does not come without challenges. Monetizing the service is difficult task, as users don't tend to stick with applications very long even when they can benefit from it. Constant development of new features and active marketing are must haves when going to production phase.

Despite the fact that technical feasibility and solid ground for monetizing options has been established, there are still unknowns that need to be solved. How much are the station owners willing to pay for the service? What about those who do not compete with fuel price, why would they join this kind of service? What is correct amount of advertisement to maximise the effectiveness without compromising user experience?

In overall I think this project did accomplish what it was supposed to. Architectural and technical feasibility has been proven and the project is ready to move on. Next steps are market research, finding the pilot customer, defining and implementing the MVP features.

REFERENCES

Singleton, Derek, 28.07.2011. What is SaaS? 10 Frequently Asked Questions about Software as a Service. Read 22.1.2017
http://blog.softwareadvice.com/articles/enterprise/saas-faqs-1072811/

Gartner, 25.01.2016. Gartner Says Worldwide Public Cloud Services Market Is Forecast to Reach $204 Billion in 2016. Read 22.1.2017
http://www.gartner.com/newsroom/id/3188817

Manish, Mundra, 12.5.2016. Multi-Tenant Vs. Single-Tenant Architecture (SaaS). Read 22.01.2017.
https://blogs.sap.com/2015/07/12/multi-tenant-vs-single-tenant-architecture-saas

Eisenhauer, Tim, 2015. Single Tenant vs Multi Tenant Business Software. Read 22.01.2017.
https://axerosolutions.com/blogs/timeisenhauer/pulse/146/single-tenant-vs-multi-tenant-business-software

Polous, Nic, 08.05.2015. Vertical SaaS: What Is It & How Is It Different. Read 22.01.2017.
http://www.bowerycap.com/blog/themes/vertical-saas-what-is-it-how-is-it-different/

English Oxford Dictionary, Proof of Concept. Read 22.01.2017.
https://en.oxforddictionaries.com/definition/proof_of_concept

Swallow, Claire, 05.10.2016. Proof of Concept V Minimal Viable Product. Read 24.01.2017
http://www.cucumber.co.nz/blog/2016/october/05/proof-of-concept-v-minimal-viable-product/

Chang, Elijah, 9.6.2016. 4 steps to defining a minimum viable product. Read 09.04.2017
https://www.devbridge.com/articles/4-steps-to-defining-a-minimum-viable-product/

Brainhub, 17.11.2016. The Ultimate Guide to Minimum Viable Product. Read 24.01.2017.
https://hackernoon.com/the-ultimate-guide-to-minimum-viable-product-59218ce738f8#.cu0zj7ii2

1redDrop, 16.01.2017. Why Infrastructure-as-a-Service (IaaS) Growth is Critical to Cloud Computing. Read 25.01.2017.
http://1reddrop.com/2017/01/16/infrastructure-service-iaas-growth-critical-cloud-computing/

Savvas, Anthony, 07.05.2014. The benefits of public cloud computing. Read 25.01.2017.
http://www.itproportal.com/2014/05/07/benefits-public-cloud-computing/

Ramel, David, 02.08.2016. Microsoft No. 2 Behind Amazon in Cloud Market Share. Read 25.01.2017.
https://rcpmag.com/articles/2016/08/02/microsoft-behind-aws-in-cloud.aspx

Solid IT. February 2017. DB-Engines Ranking. Read 12.02.2017.
http://db-engines.com/en/ranking

BuiltWith.com. Framework Usage Statistics. Read 19.02.2017.
https://trends.builtwith.com/framework

Statista.com. Global mobile OS market share in sales. Read 19.02.2017.
https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

Internet Engineering Task Force (IETF), 05.2015. JSON Web Token RFC7519. Read 19.02.2017.
https://tools.ietf.org/html/rfc7519

Bishop, Stacey, 23.01.2014. What's your SaaS revenue Model? Read 19.02.2017.
https://www.scalevp.com/blog/whats-your-saas-revenue-model

Vogel, Jeffrey, 15.07.2015. 5 SaaS-ready revenue models you should know about. Read 19.02.2017.
https://venturefizz.com/blog/5-saas-ready-revenue-models-you-should-know-about

Dilger, Daniel Eran, 19.7.2016. Apple's iOS App Store now generating 4x revenues per app vs Android Google Play. Read 19.03.2017.
http://appleinsider.com/articles/16/07/19/apples-ios-app-store-now-generating-4x-revenues-per-app-vs-android-google-play

Munir, Annum, 10.09.2014. App Monetization: 6 Bankable Business Models That Help Mobile Apps Make Money. Read 19.03.2017.
http://info.localytics.com/blog/app-monetization-6-bankable-business-models-that-help-mobile-apps-make-money

Vineet, Kumar, May 2014. Making "Freemium" Work. Read 19.03.2017.
https://hbr.org/2014/05/making-freemium-work

Chaffey, Dave, 01.03.2017. Mobile Marketing Statistics compilation. Read 19.03.2017.
http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/

Adtapsy.com, Mobile Ad Networks Interstitial eCPM Report. Read 01.04.2017.
http://ecpm.adtapsy.com

Fournier, Jocelyn, 22.11.2015. MariaDB 10.1 vs MySQL 5.7: Real world performances. Read 01.04.2017.
https://www.softizy.com/blog/mariadb-10-1-mysql-5-7-performances-ibm-power-8/

Gent, Mimi & Rabeler, Carl, 14.3.2017. NoSQL vs SQL. Read 09.04.2017
https://docs.microsoft.com/en-us/azure/documentdb/documentdb-nosql-vs-sql

Summerfield, Jason. Mobile Website vs. Mobile App. Which is Best for Your Organization? Read 09.04.2017.
https://www.hswsolutions.com/services/mobile-web-development/mobile-website-vs-apps/

Lowmiller, Joe, 06.09.2013. Which is the Code for Developing a Bright Future. Read 09.04.2017
https://blog.udemy.com/php-vs-java/

Wayner, Peter, 09.02.2017. PHP vs. Node.js: An epic battle for developer mindshare. Read 09.04.2017.
http://www.infoworld.com/article/3166109/application-development/php-vs-nodejs-an-epic-battle-for-developer-mind-share.html

Kuz, Maria, 12.07.2016. Native vs. Cross-Platform App Development. Read 09.04.2017.
https://mlsdev.com/blog/74-native-vs-cross-platform-app-development

Kamal, Shiv, 07.2017. The Top 4 Reasons Why App Fail. Read 09.04.2017.
https://clearbridgemobile.com/youre-doing-it-wrong-the-top-4-reasons-why-apps-fail/

Whittle, Dustin, 16.7.2014. Nearly 90 Percent Surveyed Stop Using App Due to Poor Performance. Read 09.04.2017.
http://www.apmdigest.com/nearly-90-percent-surveyed-stop-using-apps-due-to-poor-performance

Dogra, Rohit, 17.05.2013. Why developing a POC or an MVP before developing the final, complete product makes sense. Read 09.04.2017.
https://www.netsolutionsindia.com/blog/benefits-poc-mvp/

Singla, Lalit, 08.07.2016. POC vs MVP And Why Confusing Them Could Be a Mistake. Read 09.04.2017.
https://www.netsolutionsindia.com/blog/how-proof-of-concept-poc-differs-from-minimum-viable-product-mvp-and-confusing-them-could-be-disastrous/

APPENDICES

Appendix 1. Database Initialization Script

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,
ALLOW_INVALID_DATES';

-- -----------------------------------------------------
-- Schema mycar
-- -----------------------------------------------------
CREATE SCHEMA IF NOT EXISTS `mycar` DEFAULT CHARACTER SET utf8 ;
USE `mycar` ;

-- -----------------------------------------------------
-- Table `mycar`.`user`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`user` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `firstname` VARCHAR(45) NOT NULL,
  `lastname` VARCHAR(45) NULL,
  `email` VARCHAR(120) NOT NULL,
  `salt` VARCHAR(21) NOT NULL,
  `password` CHAR(90) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE INDEX `email_UNIQUE` (`email` ASC))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mycar`.`company`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`company` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mycar`.`brand`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`brand` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `company_id` INT NOT NULL,
  `name` VARCHAR(55) NULL,
  `description` VARCHAR(255) NULL,
  PRIMARY KEY (`id`, `company_id`),
  UNIQUE INDEX `name_UNIQUE` (`name` ASC),
  INDEX `fk_brand_company_idx` (`company_id` ASC),
  CONSTRAINT `fk_brand_company`
    FOREIGN KEY (`company_id`)
    REFERENCES `mycar`.`company` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

```
-- -----------------------------------------------------
-- Table `mycar`.`stationtype`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`stationtype` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `type` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE INDEX `type_UNIQUE` (`type` ASC))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mycar`.`station`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`station` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  `brand_id` INT UNSIGNED NOT NULL,
  `brand_company_id` INT NOT NULL,
  `stationtype_id` INT UNSIGNED NOT NULL,
  `streetaddress` VARCHAR(125) NULL,
  `zipcode` CHAR(8) NULL,
  `city` VARCHAR(45) NULL,
  `state` VARCHAR(10) NULL,
  `country` VARCHAR(65) NULL,
  `longitude` DOUBLE(9,6) NULL,
  `latitude` DOUBLE(9,6) NULL,
  `manager_user_id` INT NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_station_brand1_idx` (`brand_id` ASC, `brand_company_id` ASC),
  INDEX `fk_station_stationtype1_idx` (`stationtype_id` ASC),
  INDEX `fk_station_user1_idx` (`manager_user_id` ASC),
  CONSTRAINT `fk_station_brand1`
    FOREIGN KEY (`brand_id` , `brand_company_id`)
    REFERENCES `mycar`.`brand` (`id` , `company_id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_station_stationtype1`
    FOREIGN KEY (`stationtype_id`)
    REFERENCES `mycar`.`stationtype` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_station_user1`
    FOREIGN KEY (`manager_user_id`)
    REFERENCES `mycar`.`user` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mycar`.`fueltype`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`fueltype` (
  `id` SMALLINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,
  `type` VARCHAR(25) NOT NULL,
  `description` VARCHAR(255) NULL,
  `unit` ENUM('L', 'USG') NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE INDEX `type_UNIQUE` (`type` ASC))
ENGINE = InnoDB;
```

```
-- -----------------------------------------------------
-- Table `mycar`.`fuelprice`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`fuelprice` (
  `id` BIGINT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `station_id` INT NOT NULL,
  `fueltype_id` SMALLINT(3) UNSIGNED NOT NULL,
  `description` VARCHAR(45) NULL,
  `pricePerUnit` DOUBLE NOT NULL,
  `validFrom` TIMESTAMP NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_fuelprice_station1_idx` (`station_id` ASC),
  INDEX `fk_fuelprice_fueltype1_idx` (`fueltype_id` ASC),
  CONSTRAINT `fk_fuelprice_station1`
    FOREIGN KEY (`station_id`)
    REFERENCES `mycar`.`station` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_fuelprice_fueltype1`
    FOREIGN KEY (`fueltype_id`)
    REFERENCES `mycar`.`fueltype` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mycar`.`token`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`token` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `token` VARCHAR(255) NOT NULL,
  `user_id` INT NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_token_user1_idx` (`user_id` ASC),
  CONSTRAINT `fk_token_user1`
    FOREIGN KEY (`user_id`)
    REFERENCES `mycar`.`user` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mycar`.`advertisement`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`advertisement` (
  `id` BIGINT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `title` VARCHAR(50) NOT NULL,
  `validFrom` TIMESTAMP NULL,
  `validUntil` TIMESTAMP NULL,
  `image_name` VARCHAR(30) NULL,
  `station_id` INT NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_advertisement_station1_idx` (`station_id` ASC),
  CONSTRAINT `fk_advertisement_station1`
    FOREIGN KEY (`station_id`)
    REFERENCES `mycar`.`station` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

```
-- -----------------------------------------------------
-- Table `mycar`.`vehicletype`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`vehicletype` (
  `id` TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,
  `type` CHAR(20) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE INDEX `type_UNIQUE` (`type` ASC))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mycar`.`vehicle`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`vehicle` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `user_id` INT NOT NULL,
  `vehicletype_id` TINYINT(3) UNSIGNED NOT NULL,
  `license` CHAR(10) NOT NULL,
  `make` VARCHAR(20) NULL,
  `model` VARCHAR(60) NULL,
  `engine` VARCHAR(20) NULL,
  `year` SMALLINT(4) NULL,
  `fueltype_id` SMALLINT(3) UNSIGNED NOT NULL,
  PRIMARY KEY (`id`, `user_id`),
  INDEX `fk_vehicle_vehicletype1_idx` (`vehicletype_id` ASC),
  INDEX `fk_vehicle_user1_idx` (`user_id` ASC),
  INDEX `fk_vehicle_fueltype1_idx` (`fueltype_id` ASC),
  CONSTRAINT `fk_vehicle_vehicletype1`
    FOREIGN KEY (`vehicletype_id`)
    REFERENCES `mycar`.`vehicletype` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_vehicle_user1`
    FOREIGN KEY (`user_id`)
    REFERENCES `mycar`.`user` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_vehicle_fueltype1`
    FOREIGN KEY (`fueltype_id`)
    REFERENCES `mycar`.`fueltype` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mycar`.`fuelling`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mycar`.`fuelling` (
  `id` BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
  `vehicle_id` INT UNSIGNED NOT NULL,
  `vehicle_user_id` INT NOT NULL,
  `fuelprice_id` BIGINT(10) UNSIGNED NOT NULL,
  `units` FLOAT UNSIGNED NOT NULL,
  `timestamp` TIMESTAMP NOT NULL DEFAULT now(),
  `fulltank` TINYINT(1) NULL DEFAULT 1,
  `odometer` INT UNSIGNED NULL,
  INDEX `fk_fuelling_vehicle1_idx` (`vehicle_id` ASC, `vehicle_user_id` ASC),
  INDEX `fk_fuelling_fuelprice1_idx` (`fuelprice_id` ASC),
  PRIMARY KEY (`id`),
  CONSTRAINT `fk_fuelling_vehicle1`
    FOREIGN KEY (`vehicle_id` , `vehicle_user_id`)
```

```
    REFERENCES `mycar`.`vehicle` (`id` , `user_id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_fuelling_fuelprice1`
    FOREIGN KEY (`fuelprice_id`)
    REFERENCES `mycar`.`fuelprice` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

Appendix 2. Server Routing

```
<?php
header("Access-Control-Allow-Origin: *");
header("Access-Control-Allow-Methods: GET,PUT,POST,DELETE");
header("Access-Control-Allow-Headers: Content-Type");

use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

require 'vendor/autoload.php';
require 'common/db.php';
require 'common/baseController.php';
require 'common/embedController.php';

/*
   Modules
*/
require 'controllers/stations.php';
require 'controllers/users.php';
require 'controllers/attributes.php';
require 'controllers/companies.php';
require 'controllers/vehicles.php';
require 'controllers/fuellings.php';
require 'controllers/prices.php';
require 'controllers/authentication.php';

$config = [
   'settings' => [
      'displayErrorDetails' => true,
   ],
];
$app = new \Slim\App($config);

/* Stations */
$app->post('/stations', '\StationController:create');
$app->get('/stations', '\StationController:getAll');
$app->get('/stations/{id}', '\StationController:getOne');
$app->put('/stations/{id}', '\StationController:update');
$app->post('/stations/{id}/prices', '\StationController:addPriceForStation');
$app->get('/stations/{id}/prices', '\StationController:getPricesForStation');

/* Prices (utility) */
$app->get('/prices', '\PriceController:getAll');

/* Users */
$app->post('/users', '\UserController:create');
$app->get('/users', '\UserController:getAll');
$app->get('/users/{id}', '\UserController:getOne');
$app->put('/users/{id}', '\UserController:update');

/* Vehicles */
$app->get('/users/{uid}/vehicles', '\VehicleController:getAll');
$app->get('/users/{uid}/vehicles/{vid}', '\VehicleController:getOne');
$app->post('/users/{uid}/vehicles', '\VehicleController:create');
$app->put('/users/{uid}/vehicles/{vid}', '\VehicleController:update');
$app->delete('/users/{uid}/vehicles/{vid}', '\VehicleController:delete');

/* Fuellings */
$app->get('/users/{uid}/fuellings', '\FuellingController:getAll');
$app->post('/users/{uid}/vehicles/{vid}/fuellings', '\FuellingController:create');
$app->put('/users/{uid}/vehicles/{vid}/fuellings/{fid}', '\FuellingController:update');
```

```
$app->delete('/users/{uid}/vehicles/{vid}/fuellings/{fid}', '\FuellingController:delete');

/* Attributes */
$app->get('/attributes', '\AttributeController:selectAttributes');
$app->get('/stationtypes', '\AttributeController:selectStationTypes');
$app->get('/fueltypes', '\AttributeController:selectFuelTypes');
$app->get('/vehicletypes', '\AttributeController:selectVehicleTypes');

/* Companies */
$app->post('/companies', '\CompanyController:create');
$app->get('/companies', '\CompanyController:getAll');
$app->get('/companies/{id}', '\CompanyController:getOne');
$app->put('/companies/{id}', '\CompanyController:update');

/* Brands */
$app->get('/companies/{id}/brands', '\CompanyController:getBrandsForCompany');
$app->post('/companies/{id}/brands', '\CompanyController:addBrandForCompany');
$app->put('/companies/{cid}/brands/{bid}', '\CompanyController:updateBrandForCompany');

/* Authentication */
$app->post('/authentications', '\AuthenticationController:doLogin');


$app->run();
```

Appendix 3. Web Client Application Module

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { UIRouterModule} from 'ui-router-ng2';

import { AppComponent } from './app.component';
import { NavbarComponent } from './navbar/navbar.component';
import {APP_STATES} from "./routes/app.states";
import {AppRouterConfig} from "./routes/router.config";
import { WelcomeComponent } from './welcome/welcome.component';
import {StationManagerModule} from "./station-manager/station-
manager.module";
import {AttributeService} from "./attributes/attribute.service";
import {AgmCoreModule} from "angular2-google-maps/core";
import {WelcomeModule} from "./welcome/welcome.module";
import { LoginComponent } from './login/login.component';
import {UserService} from "./shared/user.service";

@NgModule({
    declarations: [
        AppComponent,
        NavbarComponent,
        LoginComponent
    ],
    imports: [
        BrowserModule,
        FormsModule,
        HttpModule,
        UIRouterModule.forRoot({
            states: APP_STATES,
            useHash: true,
            configClass: AppRouterConfig,
            otherwise: { state: 'home', params:{} }
        }),
        AgmCoreModule.forRoot({
            apiKey: *****
        }),
        StationManagerModule,
        WelcomeModule
    ],
    providers: [AttributeService, UserService],
    bootstrap: [AppComponent]
    })
export class AppModule { }
```

Appendix 4. Implementation of Menu Bar base class

```
package fi.tamk.eng.mika.menovesi.base;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;

import fi.tamk.eng.mika.menovesi.R;
import fi.tamk.eng.mika.menovesi.maps.MapsActivity;
import fi.tamk.eng.mika.menovesi.settings.SettingsActivity;
import fi.tamk.eng.mika.menovesi.stationlist.StationListActivity;

/**
 * Created by mika on 01/11/2016.
 */

public abstract class MenuBarActivity extends AppCompatActivity {

    private static final String TAG = MenuBarActivity.class.getName();

    public MenuBarActivity() {}

    protected abstract int getMenuId();

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        Intent intent;
        switch (item.getItemId()) {
            case R.id.listViewMenuItem:
                intent = new Intent(getBaseContext(),
StationListActivity.class);

intent.setFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);


                startActivity(intent);
                return true;
            case R.id.mapViewMenuItem:
                Log.i(TAG, "Map view clicked");
                intent = new Intent(getBaseContext(),
MapsActivity.class);

intent.setFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);

                startActivity(intent);
                return true;
            case R.id.settingsViewMenuItem:
                Log.i(TAG, "Settings clicked");
                intent = new Intent(getBaseContext(),
SettingsActivity.class);

intent.setFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);

                startActivity(intent);
                return true;
            default:
                Log.e(TAG, "I should not be here");
        }
        return super.onOptionsItemSelected(item);
```

```
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        Log.i(TAG, "onCreateOptionsMenu");

        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main, menu);

        MenuItem item = menu.findItem(this.getMenuId());
        if(item != null)
            item.setVisible(false);


        return true;
    }
}
```

Appendix 5. Implementation of data manager singleton

```
package fi.tamk.eng.mika.menovesi.station;

import android.content.Context;
import android.content.Intent;
import android.location.Location;
import android.util.Log;
import android.widget.Toast;

import java.util.ArrayList;
import java.util.Collections;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import fi.tamk.eng.mika.menovesi.R;
import fi.tamk.eng.mika.menovesi.base.AppConstants;
import fi.tamk.eng.mika.menovesi.networking.DownloadStationInfoTask;

/**
 * Created by mika on 05/12/2016.
 *
 * Singleton
 * Handles all Station List related actions
 */
public class StationManager implements
DownloadStationInfoTask.StationInfoResponseDelegate {

    // Debugging
    private static final String TAG = StationManager.class.getName();

    // Singleton
    private static StationManager ourInstance = new StationManager();
    public static StationManager getInstance() {
        return ourInstance;
    }

    // Variables
    private ArrayList<StationPrice> stationList;
    private Location currentLocation = null;

    private Context context;
    private ScheduledExecutorService scheduler = null;


    private StationManager() {
    }

    public void setStationList(ArrayList<StationPrice> stationList) {
        this.stationList = stationList;
    }

    public ArrayList<StationPrice> getStationList() {
        if(this.stationList == null)
            this.stationList = new ArrayList<>();

        return this.stationList;
    }

    /**
     * Get station list ordered by price per unit
     * if list is null, creates new list
```

```
     *
     * @return ordered list of stations
     */
    public ArrayList<StationPrice> getPriceList() {
        if(this.stationList == null)
            this.stationList = new ArrayList<>();

        if(!this.stationList.isEmpty())
            Collections.sort(stationList, new
StationPriceComparator());
        return stationList;
    }

    public void setCurrentLocation(Location currentLocation) {
        this.currentLocation = currentLocation;

        for(StationPrice sp : getStationList()) {
            sp.setDistance(currentLocation);
        }
    }

    /**
     * Get list of stations ordered by distance
     * If list is null, creates new empty list
     *
     * @return ordered list of stations
     */
    public ArrayList<StationPrice> getDistanceList() {
        if(this.stationList == null)
            this.stationList = new ArrayList<>();

        if(!this.stationList.isEmpty())
            Collections.sort(stationList, new
StationPriceComparator());

        if(currentLocation == null)
            return this.stationList;

        for(StationPrice sp : this.stationList) {
            sp.setDistance(currentLocation);
        }

        Collections.sort(stationList, new
StationDistanceComparator());
        return this.stationList;
    }

    /**
     * Uses background task to reload station data
     *
     * @param context Context for message broadcasts
     * @param refreshTimer Timer how often data should be refreshed in
minutes, 0 to disable
     */
    public void reloadData(final Context context, int refreshTimer,
boolean force) {

        if(scheduler != null && !scheduler.isShutdown() && !force)
            return;

        if(scheduler != null) {
            Log.i(TAG, "Force Refresh");

            scheduler.shutdownNow();
```

```
            scheduler = null;
        }

        this.context = context;

        if(refreshTimer > 0) {
            scheduler = Executors.newSingleThreadScheduledExecutor();
            scheduler.scheduleAtFixedRate(new Runnable() {
                @Override
                public void run() {
                    executeDataReload();
                }
            }, 0, refreshTimer, TimeUnit.MINUTES);
        } else {
            executeDataReload();
        }
    }

    public void cancelDataReload() {
        scheduler.shutdownNow();
        scheduler = null;
    }

    private void executeDataReload() {
        new DownloadStationInfoTask(this, this.context).execute();
        Toast.makeText(this.context,
R.string.downloading_notification, Toast.LENGTH_SHORT).show();
    }

    @Override
    public void processFinish(ArrayList<StationPrice> prices) {
        Log.i(TAG, "Data download is finished, time to notify UI");
        this.stationList = prices;

        Intent intent = new Intent(AppConstants.INTENT_DATA_READY);
        this.context.sendBroadcast(intent);
    }
}
```

Appendix 6. Frameworks

Android Developer Console https://developer.android.com/about/dashboards/index.html

Angular 2 https://angular.io

Angular 2 Architecture https://angular.io/docs/ts/latest/guide/architecture.html

Angular 2 Command Line Interface. https://cli.angular.io

Angular 2 Google Maps https://angular-maps.com

Angular 2 UI Rounter https://github.com/ui-router/quickstart-ng2

Atlassian Bitbucket https://bitbucket.org/product

Bootstrap 4 https://v4-alpha.getbootstrap.com

Java Hibernate ORM http://hibernate.org/orm/

Java JAX-RS https://jax-rs-spec.java.net/nonav/2.0/apidocs/

MySQL https://www.mysql.com

MySQL Workbench http://mysqlworkbench.org

Slim Framework https://www.slimframework.com