Header And Project Title

1 Introduction 3

1.1 Overview (Alan) 3

1.2 Business context (Krystian) 3

1.3 Intended audience and reading suggestions (Krystian) 3

1.4 Glossary (Both) 3

1.5 Reference (Both) 3

2 General Description 3

2.1 Product / System functions (Both) 3

2.2 User Characteristics and Objectives (Krystian) 3

2.3 Operational Scenarios (Alan) 3

2.4 Design and implementation constraints (Krystian) 3

3 System Requirements 3

3.1 Web Application 3

3.2 Web Services 3

3.3 Android Application 3

4 High Level Design 3

5 Schedule 3

# Introduction

## Overview (Alan)

The FleetOnRails application will provide a complete fleet management system for businesses and for personal usage if desired. The application will have the following functionality:

* Manage vehicle maintenance
* Driver management
* Live GPS vehicle tracking
* Vehicle problem alerts through OBD2

We are going to make use of an Android device for the in car data collection. This will interact with the cars OBD2 port and also send GPS data for location calculation and speed calculation.

The data collected from the Android device will be sent to a Ruby on Rails API where it will be processed and stored in a database for persistence. The API will be the backbone of the application and will handle all the data for the Application. The API will be fully tested, documented and versioned so that it should be easy to make a client for any platform and use our API

The Web Application will provide a front end for the Application and will be where the most user interaction will be. We will use Google’s AngularJS framework for the Web Application. This framework will allow us to create a fully tested easily maintainable front end.

## Business context (Krystian)

The aim of this project is to provide a fleet management service for car-based companies such as taxi providers.  In today’s world there is a number of companies, which use cars and vans as the foundation of their business. They either deliver goods to other places like courier's or provide a transportation service for people like taxi companies. In either case the main part of their business is managing a fleet of cars or vans. Businesses, which operate on that basis, must take a great number of things into consideration in order to maximize their profit and minimize any potential repair cost. The task of managing a small fleet of cars is relatively easy for most companies but that task could become more difficult once the size of a fleet increases.

As mentioned above the basic principle of this project is to enable a company to manage their fleet. This product would enable its users to add cars into the system and manage each car in terms of its fuel consumption on other factors, which would be a great help to the company in order to maximize their profit. As well as that the system will enable the user to schedule maintenance repairs or set reminders if a vehicle is due a check. This would result in saving money in terms of book keeping.

Another aspect of the system, which would be beneficial to a company, is a driver profiling system. Each car can be assigned a driver and while driving, the system will gather information about the driver’s behavior on roads. This would result in companies monitoring their drivers and their behavior, which could lead to saving of money. Companies could assess their drivers and take disciplinary actions if the driver’s behavior is dangerous. Dangerous driving could potentially cost the company a loss of money.

## Intended audience and reading suggestions (Krystian)

The intended readers for this document are potential users and project supervisor. The user group may contain school staff as well as any potential company who would intend to use this product. The school staff may be interested in reading how the product has been developed and the company may be interested in the functionality of the product before the development stage. The project supervisor may be interested in reading about the functionality of the product, the planned features to be implemented, interface design, and testing phase.

The rest of this SRS consist of:

* General description of the program (program functions, user characteristics and objectives, operational scenarios and constraints).
* Functional Requirements (user interface, system features)
* System Architecture
* High level Design
* Preliminary Schedule

## Glossary (Both)

* OBD2 – On Board Diagnostics
* Rails – MVC framework for Ruby
* Active Record - Object Relational Mapping used in Rails
* RSpec – Behavior driven development framework for Ruby
* Capistrano – Deployment tool written in Ruby
* Apache – Web Server
* Unicorn – Application Server
* AngularJS – JavaScript Framework by Google
* JSON – JavaScript Object Notation

## Reference (Both)

# General Description

## Product / System functions (Both)

**Live Vehicle Tracking**

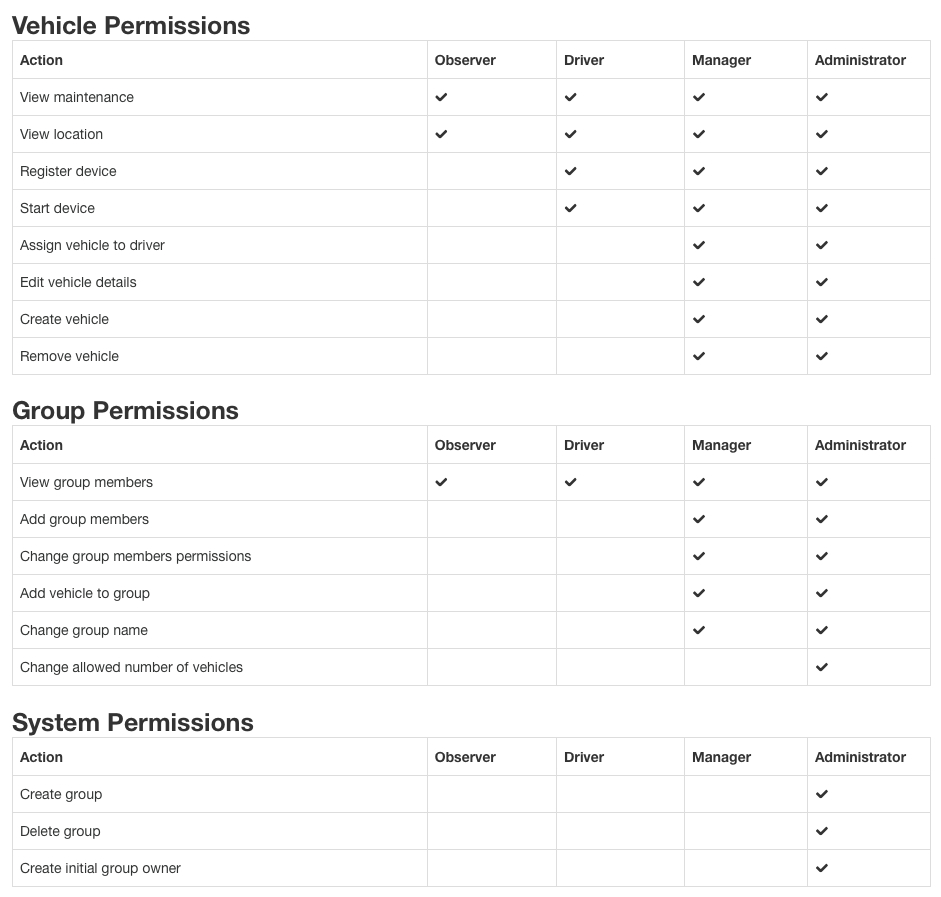
Utilizing an Android Phone we will be able to get highly accurate locations for each car in the fleet that has been assigned an Android Device. The location will be received and then sent to the API where it will be associated with the car that the particular Android device is connected to. We will also be able to get highly accurate speed-readings from the GPS; we can also get the speed from the OBD2 port of the car. However the GPS will be more accurate as to the real speed of the vehicle. Reading just from the GPS is hazardous though as the device might loose signal and then we would have no reading. So it might be beneficial for us to get both GPS speed and OBD2 speed and get the average of the two.

**Fault Code Notification**

Having the functionality to identify fault codes in a vehicle would be beneficial as the company will be able to schedule an immediate vehicle check when the driver gets back to base. We are able to read OBD2 fault codes from the car with the Android device and then send them to the API, We should then be able to send something like a push notification to the Web APP through some kind of socket where the fault code can be identified and suitable action can be scheduled.

**Security**

* Authentication: This is important so that the application is secure. The application would preferably implement OAuth 2.0 for authentication. This will insure that users will be able to authenticate for our Android application and the Web application and any future applications.
* User Permissions: There will be different user permissions. These will define what each user is able to access in the application. This is the proposed permissions table:



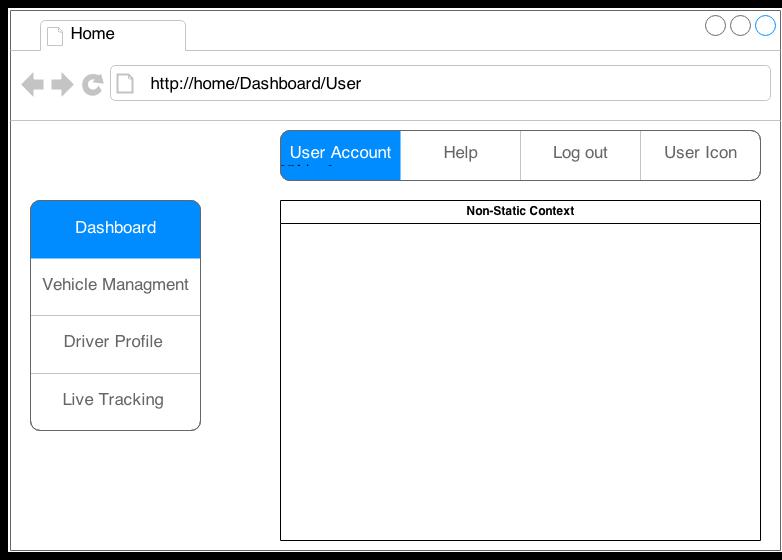
**Data Processing / Handling**

The API will provide all the data handling and processing for the application it will do this through a REST service that will allow any authorized applications to use the basic HTTP methods like [Get, Put, Post, Delete]. By using HTTP we are not restricting the platforms that clients can be made for. The API will accept data from the Android client and allow the Web application to call for data that will be returned in JSON format.

**Dashboard**

All of the functionality of the system will be available to the user in the main Dashboard; the functionality of the dashboard depends on the security group a particular user has been assigned to. In order for the user to use the dashboard a log in is required. After successfully login in, the main dashboard will be displayed to the user.

The dashboard will consist of 3 main containers. The top container will include help button, user settings and log out button. The left container will contain the main functions of the dashboard; these include home page, management of vehicles, management of documentation and services. The main container will display different results based on whatever the user has clicked, this will be the only non-static part of the page.



The home page is composed out of four parts:

* **Dashboard**

After adding at least one vehicle to the system the user will be able to see that vehicle on the main dashboard page. This page will include basic information about the car such as current odometer status, base location, basic car information, statistical information about cost of running the vehicle. The user will be able to change the information about the car, view graph representation of the cost of running the car and add documentation.

* **Vehicle Management**

Under this heading the user will be able to manage each vehicle as well as the entire fleet. The user will be able to add and remove vehicle from a fleet. After selecting a particular vehicle the user will be able to manage service reminders, check date for next maintenance, see the previous history for all performed services and also add parts receipt associated with each receipt. Also the user will be able to search for a vehicle based on filter conditions.

* **Driver Profile**

A significant part of this system is driver profiling. The management of the company will be able to assess each driver based on how are they driving. The system will monitor each driver based on harsh braking, not obeying to the rules of the road (this mainly involves speeding) and possibly checking if goods have been delivered on time. The system will grade each driver in scale one to ten, one being really bad and ten being really good. The system will take into consideration the above conditions and check how many times the driver has had went over the limit.

* **Live Tracking**

Another important part of the system is live tracking of a vehicle or a fleet of vehicles. The user will be able to see the current location of the vehicle on a Google map. The location of the vehicle will pulled from the API which handles the coordinates send from the car. The user would preferably be able to pin point destination of where the vehicle is heading, thus allowing the system to calculate fuel consumption for that route and enabling the user to view this in the main dashboard content.

## User Characteristics and Objectives (Krystian)

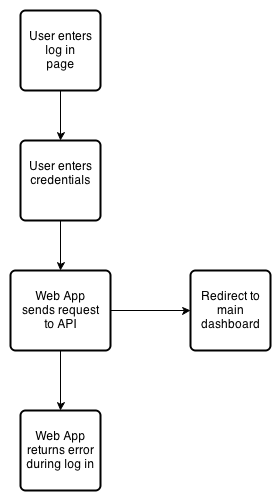
This system is designed for company’s which automotive transport is one of the main part's of their business. Since this user group may not have experience with any similar software or with software in general, the FleetOnRails system will been designed in such a way to ensure that the user will benefit from the product as much as possible. This involved designing a simple, intuitive interface while not lacking in functionality which would maximize the usage of the product. Simplicity was one of the core factors taken into consideration when designing the system. The user objective for this product is to use it in everyday life as a part of a business, thus it is necessary to create a system which would help in the day to day running of a company and also it would be simple to use and provide a company with the necessary functionality to make it worthwhile for them to use it. The current market at present provides similar software but in many cases they lack important features which the business may need. To ensure that this system provides the functionality business look for, a deep research has been done. The result of the research helped with defining the goals of the system, improved functionality and design, cost of the product and new features which were not being exploited before.

Another user group for the FleetOnRails system is car enthusiasts that might like to collaborate all their cars documentation and manage their cars maintenance through a simple easy to use web interface. The FleetOnRails system can be used with or without an in car device so it is not required for everyone to have the live tracking and fault code notification features. This means that the FleetOnRails system does not require the Android device to be present in order to function correctly.

## Operational Scenarios (Alan)

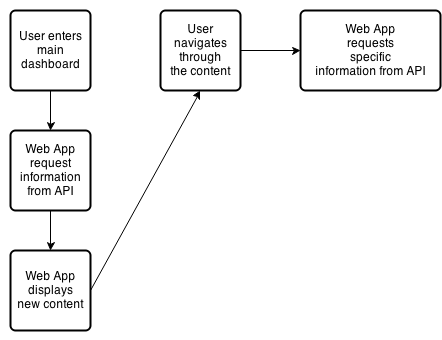
**User logs in to Web Application**

Assuming that the user has already registered in the system, the next step is to log in into the system. The user will be asked to enter credentials which will be send to the API. The API will perform check and is if the user exists in the database and if the credentials are correct. After performing that check the API will send appropriate massage to the Web App where either an error message will be displayed or in the case of successful log in, the user will be re-directed to the main dashboard page.



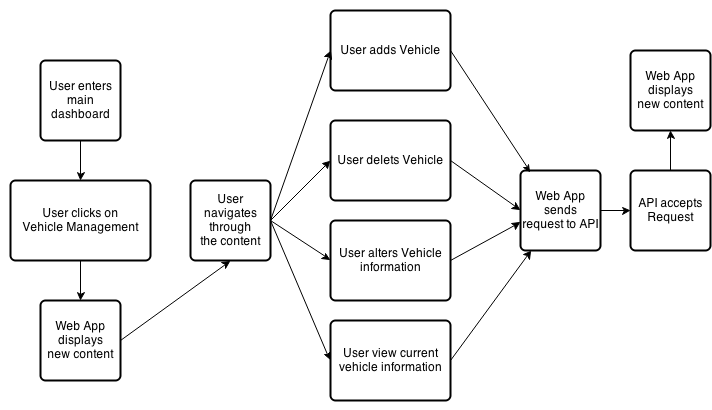
**User uses the Dashboard- Main Dashboard**

After a successful log in, the user will be re-directed to the main dashboard page. The Web App request information about user’s vehicles from the API .The non-static container will display the current information about the vehicle. The user will be able to see and alter basic information about the vehicle. If the user decides to view specific information, a new call will be made to the API. The diagram below shows the flow of actions:

****

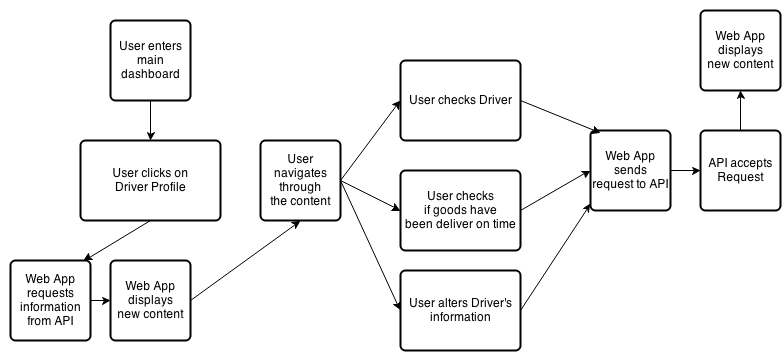
**User uses the Dashboard- Vehicle Management**

After clicking the on Vehicle Management button the Web App will request information from the API about current vehicles based on user id. The Web App will then display the information in the non-static container. The user can alter vehicles information, add vehicle, delete vehicle and view vehicle details (these include service remainders, previous service history). At each of the steps the Web App will either request information from the API or send new information in the case of adding or altering vehicle. After API processes the information, the Web App will display the updated content.



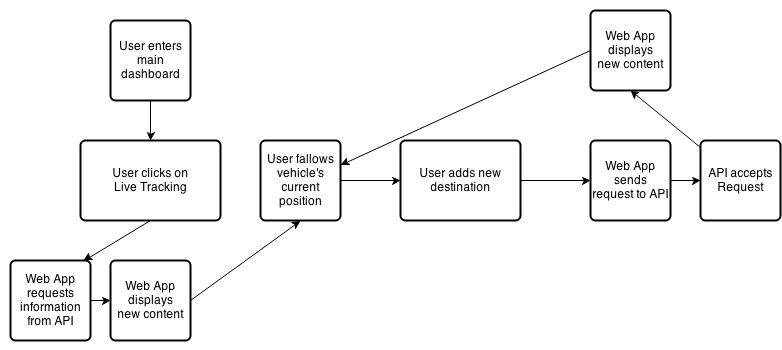
**User uses the Dashboard- Driver Profile**

After clicking on the Driver Profile button the Web App will send request to the API to retrieve current information about all drivers. The user will be able to navigate through the content and check current performance of each driver, update driver information and possibly check if the driver has deliver goods on time. The API will gather all needed information from the car; perform performance check of the driver and send the results to the Web App where the user will be able to see them.



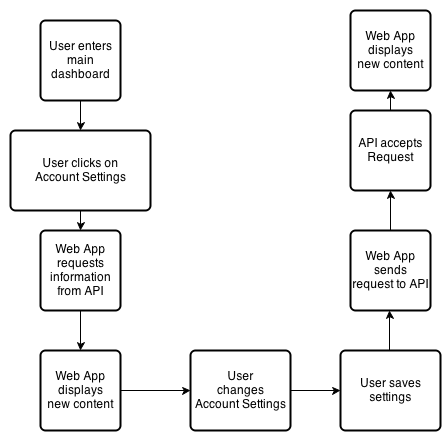
**User uses the Dashboard- Live Tracking**

After clicking on live tracking button, the user will be able to see all vehicles with their current position on the map. The API will receive the co-ordinates from the car and send them to the Web App where they will be placed on a map. The user can check each point on the map and see what vehicle is it. The user would preferably add destination as to where the vehicle is heading. After adding that destination, the Web App will send new information to the API.



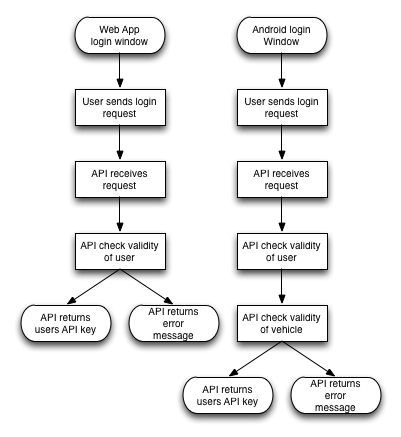
**User uses the Dashboard- Account Settings**

After successful log-in the user can change the account settings by clicking on the Account button. The user will be able to change password, first name, second name. The user can also change the email address and set up profile picture. After changing the settings and clicking on submit button the Web App will send an update request to the API where the information will be processed. The API will then send updated information to the Web App, where they will be displayed.



**API authenticates users**

When a user logs in to either the Web Application or the Android application, the users details such as the username and password will be checked against the database. The username will be stored in plain text in the database however it would be a very high security risk to store the password in plaintext. This is because most users will use passwords common throughout many services such as social media and email, however there is a library in rails to combat this it is called BCrypt it contains functions for hashing passwords securely, it is this has that will then be stored in the database. After a user is authenticated the API will return that user their API key. This will allow them to interact with the API and for all transactions to be recognized as that user. A sample workflow is below

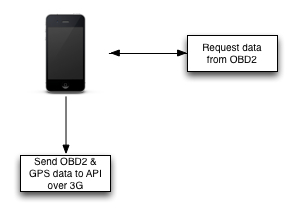


**API handles HTTP requests from Android**

One of the core functionalities of FleetOnRails is the ability to receive information from the cars ECU using the cars OBD2 port. “*All petrol cars sold within Europe since 1 Jan 2001, and diesel cars manufactured from 2003, must have on-board diagnostic systems to monitor engine emissions.”*

The Android application will connect to the OBD2 port through an OBD2 – Bluetooth device this allow us to put the Android device anywhere in the car and it will not be in the way. The Android device will then be able to request data from the adapter through serial commands. OBD2 returns byte arrays that will contain the information requested so that will just have to be parsed and then we have the information. This data will then be sent to the API in JSON format.

The data we are sending will also include GPS data that will be used to get the vehicles current location and speed. Below is an example of the flow.



**API handles HTTP requests Web Application**

The FleetOnRails Web Application will be a purely client side Application, meaning that it will be completely decoupled from the API. By taking this approach with the application we have kept our options open for people to develop client applications for every platform such as iOS and platform native programs. We didn’t want to go down the route of JSP for example where our backend is tightly coupled with our front end.

The way in which our Web Application will function is by sending specific HTTP requests to our API. The API will have many different API endpoints, which will provide any client with access to the data so long as they are authenticated, and have the correct user permissions. An example of a HTTP request to the API in JavaScript would be:

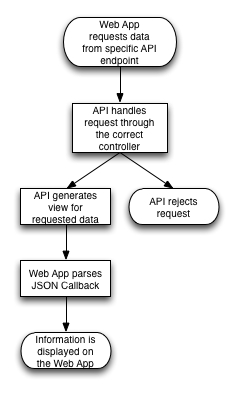
$scope.apiKey = "6f950ea6ed5ae98f5af25c6925488b5c";

$http.get('http://secure.fleetonrails.eu/api/v1/users.json/' + $scope.apiKey + '/?callback=JSON\_CALLBACK').success(**function** (data) {

// Do some stuff with the data

}

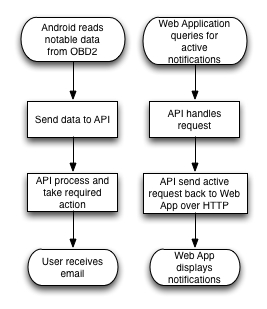
Where scope.apiKey is the logged in users API key. This endpoint in theory will return all the users in the database. However depending on the logged in users permissions they may or may not have access to view all the users in the database. Here is a example of the flow.



**API sends notification emails**

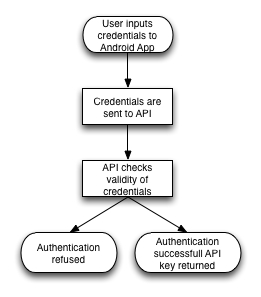
Another function that our API should have is the ability to send notifications. Notifications including email notifications and visual notifications on the dashboard can be sent from the API. This would mean that if a car develops a fault code an email could be sent to users with the correct level of permissions on a group for example the group owner or observer.

This will allow them time to schedule a service or repairs for when the car is back at base. Notifications such as email can be sent through ActionMailer, which is a ruby library for sending email. The dashboard could also request notifications periodically or ideally there can be some kind of socket connection between the Web App and the API. All notification messages will be stored in the database for audit purposes. Here is an example flow of the notifications.



**User logs in to Android application**

When a user logs into the Android Application the user credentials and the cars registration number will be sent to the API the API will either authenticate the user in which case the users API key will be returned or it will return an error the Android device something like “failed to authenticate”. After the Android device has been authenticated it can start to send data to the API such as location data and OBD2 data. Here is the flow diagram for Android device authentication.



## Design and implementation constraints (Krystian)

Below is a list of possible constraints placed upon the design team under which this project is to be developed.

**Time constraints**

The project has a completion date which is the 3/6/2014 for DCU and taking into consideration the workload, time constraints can play a significant part in developing the product as initially designed. As well as that there might be a constraint regarding testing the system in a company and receiving feedback.

**Company constraints**

There might be a constraint regarding receiving a full support from a company where this product may be used, which is a part of user testing.

**User requirements**

Within the time frame given we will be able to fully understand the need for the system and expectations of the end user.

# Functional Requirements

**N.B) Description, Criticality, Technical Issues, Dependencies with other requirements**

## Web Application

**Log in**

*Description*

The system will provide a registration option for first time users. After the registration is complete the Web App will send all needed information to the API where after processing the request the API will send back token with password. The user will be able to log in into the system. At the time of log in the Web App will send the username and credentials to API which will perform the validation process. After validating the user the system will either allow to use the Web App or throw an error.

*Criticality*

The system must ensure that the log in process for the user is intuitive. The Web App must ensure that it sends the username and password correctly to the API, in theory we would like to implement HTTPS and SSL while sending the credentials.

*Technical Issues*

The main issue with this requirement is to securely send the credentials and retrieve the right token from the API. Also it is important the after the log in the user will be re-directed to the right page.

*Dependencies*

This requirement highly depends on the API side and sending and retrieving the right information. Also the log in must ensure that after validating the user and accepting the credentials the user will be directed to the main dashboard page.

**GUI**

*Description*

Since the Web App is a website the GUI plays really important role as a part of the system. The design of it is based on human-computer interaction principle. The system must be design in such a way that the user is being presented with a neat, intuitive and clear interface.

*Criticality*

The interface for the system is very significant. Due to the fact that the main user group is automotive service providers the interface must be design in such way to encourage the use of it. The interface must have the same design throughout the system and give the user a friendly feeling about it.

*Technical Issues*

Issue regarding this requirement is ensuring the features on the system are designed in a friendly manner and that they are intuitive to the user. Any graphical representation of data must be shown in a clear way and give as much information as possible. The static content of the Web App must stay the same at all times and only the non-static container should change. Nevertheless the non-static content should give the same feeling as the static content.

*Dependencies*

The GUI interface will be implemented throughout the system, thus all feature of the system depend on it.

**Account Settings**

*Description*

The system will provide account settings option for the user where the user can change personal details and preferences. After the user updates the information and submits them, the Web App will post the updated information to the API, where it will be processed. The API will send suitable HTTP response code back to the Web App which will display the message back to the user.

*Criticality*

Significant part of this requirement is handling of the HTTP response code and posting the updated information to the API. The Web App must ensure that it handles the response code properly and also that it posts the right information to the API.

*Technical Issues*

Issue with this requirement is handling the HTTP response code.

*Dependencies*

This part of the system depends on the main dashboard since the only way to access account settings is through main dashboard. Another dependency is with the API as makes calls to it.

**Dashboard – Main Dashboard**

*Description*

One of the main functions of the system is the main dashboard. The main dashboard is the first page the user is exposed to after to log in process. From the main dashboard the user can access all the main functions of the system. The Web App request information about a particular user from the API and displays the basic information on the main dashboard.

*Criticality*

This function of the system is one of the most significant parts. It serves as a link between different parts of the system. It is crucial that this part has to be implemented properly as it is the main dashboard.

*Technical Issues*

Issue with this requirement is that it has to send the right get method to the API with the right user token and retrieve basic information about user’s fleet.

*Dependencies*

This requirement serves as a base for the Web App main functionality and link pointed between different functions of the system, thus it is crucial that any dependencies with other requirements are implemented properly.

Also this part depends on the log in of the Web App as the log in re-directs the user to this function.

**Dashboard – Vehicle Management**

*Description*

This function of the system focuses on the vehicle management. The Web App will send user’s id to the API which will retrieve all vehicles based on that id. The API will post the result to the Web App which will then display them to the user. This features also implements filter function for the user which will help to find vehicles in a big fleet.

*Criticality*

This feature of the system is critical for managing single vehicle and a fleet of vehicles. As well as that this feature enables the user to perform any action necessary to add, delete and alter vehicles.

*Technical Issues*

Issue with this feature of the system is to make sure that the right vehicles are retrieved for a particular user. The API has to make sure that it will query the database for the right user.

*Dependencies*

This function highly depends on the main dashboard feature of the system as it will use the same user id for get methods.

**Dashboard – Driver Profile**

*Description*

This part of the system caters for the driver profiling. The system must ensure that it displays all the drivers registered by the user. The system must supply the user with all information needed by the user to check the performance of a particular driver. As well as that the system must ensure that if a user changes details of a driver, the right information is send to the API and the database should be updated accordingly. It would be preferable if the system registered destinations of each delivery and checked if the driver arrived on time, thus making the grade schema more accurate for the management of a company.

*Criticality*

It is crucial that the Web App will retrieve right information about all of the drivers in a user group. It is also significant for the Web App to grade each driver and display the information in a right manner, thus ensuring the user can make a better judgment about each driver.

*Technical Issues*

The main issue with this function is to ensure that the Web App displays the correct list of drivers and that it grade each driver with the right grade.

*Dependencies*

This requirement depends on the main dashboard which uses the user id to retrieve the right list of drivers and also the API which returns the list of drivers and their current grades.

**Dashboard – Live Tracking**

*Description*

This part of the Web App caters for live tracking of vehicles. The system must ensure that the map is updated frequently in order to display the most current position of each of the vehicles. The Web App will request the co-ordinates for each of the vehicles, longitude and latitude, from the API which will get them from the android application. As an add-on to this feature a user would also add destination for each vehicle, thus calculating time it took each vehicle to reach the destination point. This would be then used for the driver profiling and help a company to achieve their goals.

*Criticality*

It is crucial for the Web App to frequently ask the API for the co-ordinates of each vehicle to ensure that the user has the most up to date information. It is also significant for the Web App the calculate distance traveled to a destination point if this feature its added to the system.

*Technical Issues*

Main issue associated with this requirement is retrieving the co-ordinates form the API. The Web App must frequently ask for the co-ordinates for the right vehicle and the right user. The map should only display the right vehicles associated with a particular user.

*Dependencies*

This requirement highly depends on the main dashboard which passes the right user id to it and also depends on the co-ordinates which are being retrieved from the API.

## Rest API

### Database setup

* Description: In order for the FleetOnRails to work it needs a database, the database we recommend using and will be developing for is MySQL. After installing a MySQL database you need to create a database schema for the application. This schema should be called something meaningful such as fleetonrails\_production. After doing this you will need to create a MySQL user for the application something like fleetonrails will do. Don’t be tempted to use the root user for this.
* Criticality: There are many other databases around and allot of people would disagree with MySQL as it does something’s differently than conventional Databases such as Oracle. However since Active Record will be handling all transactions, we believe MySQL is the easiest to setup and administrate.
* Technical Issues: Port 3306 will need to be open on the database server’s firewall this is presuming the default port is not changed. The database user will also need to have access from the server that the API application will be hosted on.
* Dependencies: The system should have at least 2GB of Ram for a small install and preferably about a 2.0GHZ processor. We recommend installing the database on a \*nix based server.

### Production Environment setup

* Description: In order for the FleetOnRails API application to work Ruby has to be setup on the system. Ruby can be installed with most package managers such as apt-get and yum and even rvm (ruby version manager). However we suggest compiling Ruby from source in a production environment. Once Ruby is installed the Bundler gem has to be installed you will also have to ensure that Git is installed on the system.
* Criticality: Installing Ruby from source is time consuming. The previous sentence is true however in a production environment there shouldn’t be multiple versions of ruby installed and it is also easy to debug the ruby installation when installing. Whereas if you download the binary it might just not work sometimes due to a dependency you are missing.
* Technical Issues: Installing Ruby from source isn’t very difficult however there are a few steps you need to follow carefully to do it, you will need to download development tools such as GCC to compile Ruby. This subject will be covered in the User manual in full.
* Dependencies: As with the database server we recommend a \*nix based server for the production environment.

### Installation

* Description: The FleetOnRails API application will be much the same as every other Ruby on Rails application in terms of installation. You will need to clone the repository then checkout the tag that you want to use. After doing so you will need to configure the system this will be described in the next step.
* Criticality: Allowing users to download the full code repository could cause them some confusion. They might for instance checkout the wrong revision or branch leading to a faulty system. However the instructions in the Installation manual will be defined very clearly so this shouldn’t be a problem.
* Technical Issues: Administrator will need some knowledge of Git and \*nix systems.
* Dependencies: Ruby 1.9.3-p448, Git, Development tools such as compilers

### Configuration

* Description: Once the Database and Production environment is setup. The FleetOnRails API application needs to be configured. The database will need to be defined for the application and mail servers for notifications and also there might be some locale settings such as Metric/Imperial units for speed and distance. Here is an example of the database configuration.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | development:  adapter: mysql2  encoding: utf8  database: fleetonrails\_production  username: fleetonrails  password: secure\_password  host: 127.0.0.1  port: 3306 |

* Criticality:
* Technical Issues: The Database might refuse the connection if the username or password is incorrect or if the application is not allowed to connect from its IP address.
* Dependencies: Environment is setup, there is someway to edit the database file on the server.

### Deployment

* Description: The FleetOnRails Application will be deployed on a Unicorn server. Once all the steps above are complete we need to do two things update our gems with bundle. And then migrate the database with rake db:migrate It will then server up to the public through an Apache2 server instance. This will give us the option to use HTTPS, as it will be more secure. We will also create an init.d script so that the FleetOnRails server can be started like any other service such as Apache with a command like service fleetonrails start
* Criticality: Why do I require an apache server in front of the unicorn server? This is to take advantage of the added security benefits when using HTTPS.
* Technical Issues: Setting up Apache is very easy however the full instruction of how to do so will be in the Installation manual.
* Dependencies: Apache, Unicorn

## Android Application

**Log in**

* Description
* Criticality
* Technical Issues
* Dependencies

**Transfer of data between OBD2 port and API**

* Description
* Criticality
* Technical Issues
* Dependencies

**GUI**

* Description
* Criticality
* Technical Issues
* Dependencies

**GPS data transfer to API**

* Description
* Criticality
* Technical Issues
* Dependencies

# System Architecture

## API

* **General Overview**

The backend services will be achieved with a Ruby on Rails application; The Rails application will depend on a MySQL relational database to store the data.

Ruby on Rails is an MVC framework for ruby. Using an MVC framework for this app will mean that our code will be easily maintainable as business logic is split up into controllers and models while the views will define how the data is returned from the app.

For authentication of the API we will be using token-based authentication. This will allow us to identify users on the system in turn producing a secure API. If an incorrect token is produced interactions will be rejected from that user.

* **Authentication Process**

To achieve token-authentication based authentication in our app we will be making use of some Ruby Gems in particular a gem called ‘devise’. This gem will add a few columns to the users table these columns will allow it to control session based authentication tokens. There will be an API endpoint such as

That will take a users name and password. If the users credentials are correct the users authentication token will be returned. The users token will be used to interact with the API for any following requests.

* **HTTP status codes**

HTTP status codes should be returned after all requests. Status codes are necessary in order to let the client of that API that there request was successful or if it had an error processing the request. All status codes will have to be returned in JSON so that the clients can process the return value.

* **Unicorn and Apache**

Unicorn is an HTTP server for Rack applications; Rails applications need an application server in order to deploy to the world we will be using unicorn, which is tried and tested on some well-known applications such as GitHub. However in front of our unicorn server we will have an Apache server instance this will act as a proxy for the moment but if we ever scale up it can do things like load balancing for example which would be a great help. Another advantage of using apache is that we can use HTTPS. Using HTTPS adds huge security advantages because all interactions between any clients are encrypted. This would mean that interactions between the Android Application and Web Application would be encrypted to the API.

* **Bundler**

Every Ruby on Rails application will have some Ruby Gems in it. There is one problem with this and that is when you get conflicting dependencies. However there is a great Ruby Gem called Bundler that can help with this problem.

Bundler is a Gem management system it allows us to do things like install dependencies with one command this will make deployment very easy. You can also execute any ruby command with `bundle exec` prefixed to it, doing this will mean that only Gems from your bundle will be used to execute a command.

## Web Application

## Android

* **General Overview**

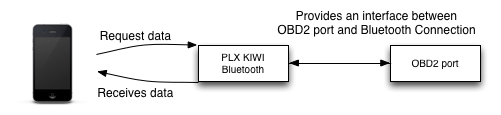
The Android application will make up one of the core services of the FleetOnRails application. The Application will connect into the cars on board diagnostics port, this will allow us to read data from the car such as speed and engine fault codes. We will also be getting data from the phones GPS this data will be sent to the API.

* **Android SDK**

The Android SDK we will be using API level 10 which will work with Android 2.3 Gingerbread and every following version of Android. We are using an older version of Android so our app will work with the vast majority of android devices.

* **OBD2 Bluetooth connection**

In order to connect with the OBD2 port we need to have an OBD2 Bluetooth adapter these are available from various suppliers. The device we have decided to get is the PLX Kiwi Bluetooth; this device comes with a 1-year warranty and is made with high standard materials. The phone will open a Bluetooth connection to the device and then the app will be able to communicate with the OBD2 device over a Bluetooth stream. Here is a diagram of the setup.



* **3G Connection**

The Android device will make use of a 3G data connection in order to interact with the API. When the Android device collects data it will send the data to the API ad JSON. Any network can provide a 3G-sim card. We will be picking the network with the best coverage so that the cars can always interact with the API.

# High Level Design

# Schedule