Declaration Cover Sheet for Project Submission

SECTION 1 *Student to complete*

Name: Dan Downey
Student ID: X14728599
Supervisor: Dr. Dominic Carr

SECTION 2 Confirmation of Authorship

The acceptance of your work is subject to your signature on the following declaration:

I confirm that I have read the College statement on plagiarism (summarised overleaf and printed in full in the Student Handbook) and that the work I have submitted for assessment is entirely my own work.

Signature: Dan Downey **Date:** May 13th, 2018

Vehilytics

Technical Report

Dan Downey - x14728599
IoT specialization
x14728599@student.ncirl.ie

Supervisor: Dr. Dominic Carr

Contents

i. Definitions, Acronyms, and Abbreviations

RPi	Raspberry Pi
OBD	On-Board Diagnostics
API	Application Programming Interface
GUI	Graphical User Interface
ECU	Engine Control Unit
Diagnostics reader	Raspberry Pi installed into the vehicle that polls the OBD port for vehicle diagnostics, and sends these diagnostics to the web service
Diagnostic report / vehicle diagnostics	The raw data for a given report sent by the diagnostics reader
Vehicle analytics	An easy to read analysis of the historic diagnostic reports. Used to abstract away the technical terms and numbers into friendly useful groupings of information to help inform decision making
Android APK	Android Application Package Kit - Android application installer file extension

ii. Executive Summary

This report discusses my attempts to design and build a system that allows vehicle owners to more easily understand the information gathered and reported by their vehicle, and to manage routine maintenance tasks all from an Android application. Currently there are similar systems that exist for expensive vehicles that require the use of manufacturer specific hardware and have no simple equivalent for cheap or mid-range cars.

The motivation for building this system arose from seeing vehicles break down from issues that would have been easily avoidable had the gathered data been exposed to the owner of the vehicle such as, the battery voltage gradually becoming too low to be able to start the car. This battery voltage can be graphed to show a fall in voltage over days or weeks leading up to the breakdown. In more expensive cars, this information is easily accessed by the owner through a dashboard display or a proprietary mobile application. My aim is to provide this functionality to any vehicle regardless of the manufacturer or value of the vehicle.

The project consists of 3 subsystems: a diagnostics reader, an Android application, and a Ruby on Rails API. The diagnostic reader provides an interface with the vehicle to gather and report various sensor readings. The Android application allows the user to view the sensor readings in a readable format and set preferences for which sensors should be reported. The Rails API acts as the bridge between the Android application and the diagnostic reader by persisting the data and exposing it to both. The API is designed in such a way that the reports can come from any device authenticated to use the endpoints and the consuming client could be any mobile application or web page that provides a login option for the user to allow them to authenticate with the web service.

1 Introduction

Vehilytics is a system that provides the means for a user to easily access and understand the data provided by their vehicle so they may attempt to prevent breakdowns by performing their own preventative maintenance or having the vehicle repaired by a mechanic while the damage is minimal before a more serious failure occurs.

The system is intended to allow owners of cheaper vehicles to access the services normally found in expensive vehicles as these users are far more likely to experience part failure and less likely to be able to afford the repairs necessary. The hope is that it will also allow people who are not technically savvy to gain an understanding of the maintenance needs of their vehicle.

1.1 Background

My idea began as a sensor array that could be placed within a car to detect when a baby was left unattended and notify parents, guardians, and finally emergency services as necessary to ensure the survival of the child. The inspiration for this idea came from reading an article which described the surprisingly common phenomenon of parents simply forgetting they had the child with them in the car on their way to/from work and discovering hours later that the child had perished in the heat or cold.

Developing a system to detect the presence of a baby that was left alone would end up being an unwieldy amount of sensor configuration and processing for a singular purpose to be served. However, a car already has a host of sensors installed, reading, and reporting at most times to monitor things like RPM, or external temperature, or anti-theft devices. So, the idea came about to publish these diagnostics in some form of readable report for the vehicle owner to be viewed remotely and allow the system to be generic and extensible so additional modules can be added at a later stage, including the baby monitoring device.

The beauty of this general, extensible platform is that the limit of what can be reported and viewed is based on what the user and any third-party manufacturers can design, build and install following what I can determine to be allowable report formats. These reports could elicit different responses based on the nature of the data. For example, a sudden increase in engine RPM while the user is not near the vehicle could indicate the car has been stolen.

The reports from the On-Board Diagnostics itself could save huge amounts of time and money by allowing the user to identify faults that they can fix themselves and provide their mechanic with the report so they don't need to run, and can't bill for, their own diagnostics services.

Additional modules could also leverage the sensor readings to manipulate things like air conditioning so that the vehicle is not too hot when the user enters the vehicle, or control an array of suitable systems for use cases such as, theft deterrence, calculating insurance costs, or settling traffic disputes.

There is a similar system called “Carista” but it differs from Vehilytics in a few ways. Carista uses a proprietary OBD socket which is not suitable for some cheaper / older vehicles, while Vehilytics is aimed at using any available OBD reader meaning it can connect to any vehicle with an OBD port.

1.2 Technologies

The system has a wide variety of tools used throughout the technology stack so it was tricky to decide on the best tools for each sub system to allow efficient operation and effective communication between each of its subsystems.

1.2.1 Development

The Android application was developed using Android Studio and is written entirely in Kotlin. The choice to use Kotlin was made because it is a newly supported language, it is quite similar to Java, it has some very nice syntax, and it forces good practices due to immutable and non-nullable types.

Ruby on Rails was chosen for the web service due to its expressive syntax, third party library support, and how quickly it can be used to build a working system. The library support allowed an effective authentication system to be built quickly without the need to invest a large amount of time encrypting data and providing mechanisms to restrict access to endpoints. The web service was developed in a Cloud 9 instance to reduce the chances of errors due to manual configuration in a local installation of Rails.

The diagnostic reader was built using a Raspberry Pi since it does not require much power to operate, and is small enough to be hidden from sight once installed inside a vehicle. The code on the Pi was written in Python due to its simplicity and library support, allowing me to quickly build a fully working system and have it communicating with both the vehicle and web service. All the Python code was written using PyCharm on a separate desktop PC and transferred using Git to allow development to progress quickly.

1.2.2 Testing

JUnit and Mockito were the primary testing frameworks used for unit testing the Android application. JUnit allowed me to express assertions about how my code should be behaving, while Mockito allowed me to mock the behaviour of complex classes that I shouldn’t need to test such as, Context or SharedPreferences, and to verify that operations were happening in a specific order.

A full suite of integration tests was built for the web service to ensure consistency with each of the endpoints throughout development. I used gems like RSpec, Factory Bot Rails, and Faker for building powerful expressive tests. RSpec is a testing framework for Rails that allowed me to build request specifications to make actual requests to the endpoints and test the behaviour. Factory Bot Rails allowed me to generate models to populate the test database as required and Faker provided the actual data for the models.

1.2.3 Version Control

Each of the systems was tracked using Git to give me the ability to quickly move between development environments and revert code when there were issues. The diagnostic reader code was pushed to Github because it began as a fork of another library, while the Android application and web service were pushed to Bitbucket to make use of the free private repositories.

All the codebases were moved to Bitbucket and made public after completion of the project for ease of access in the event of any issues with the upload of this report. I have provided links to the final commit before submission of each of the repositories.

Python code available at:
https://bitbucket.org/danthedrummer/vehilytics_reporter/src/cabb15d572c0eb1f33b834ba0a1e561f61de83d0/

Web service code available at:
https://bitbucket.org/danthedrummer/vehilytics_api/src/61451c0ae20e2a2380e936d5273a119d7c1e6717/

Android application code available at:
https://bitbucket.org/danthedrummer/vehilytics_android/src/f14469eddb2a809331921f03fe860e604493cd01/

1.2.4 Deployment

The web service was deployed to Heroku, which is a cloud based hosting platform. The useful thing about Heroku is that it allows you to quickly deploy using a git repository, which I already had in place developing the entirety of the system.

1.2.5 OBD Readers

OBD readers

OBD readers are devices that plug into the OBD port in the vehicle and allow communication with the vehicle ECU. The image to the left shows the 3 readers I purchased. From top to bottom they use Bluetooth, Wi-fi, and USB technologies to enable a connection.

2 User Requirements Definition

To help elicit some user requirements I ran a brainstorming session, and circulated a survey. Between both elicitation techniques I was able to glean some valuable information about the end users, their reaction to the product, and how they would like to use the product.

2.1 Brainstorming

The brainstorming session prompted some interesting discussion on how the application could present information to users based on their technical skill and understanding of their vehicle, and features that could be included in the mobile application to promote awareness of the routine of maintaining the vehicle.

For a user with little knowledge on the internals of the vehicle, the raw diagnostic data is meaningless. So, for these users, the data should be analysed and broken down into useful readouts that give general, high-level information about the health of the vehicle.

For more technically able users, the ability to opt in to viewing the raw data could be important in properly diagnosing a problem that could be fixed by a hobbyist user without needing to involve a mechanic.

An interesting feature that was requested is a mechanism to identify warning symbols displayed by the car either through the diagnostic reports or allowing the user to take a picture of the symbol and maintaining a glossary of these meanings.

2.2 Survey

The survey was useful for determining the types of vehicles the responders are using, how often they use them, how familiar they are with vehicle maintenance, and whether they would like to receive warnings about potential maintenance issues. 70.5% of responders claim to own a vehicle so there was a small subset of responses with no information for familiarity with vehicle maintenance and overall vehicle usage.

Responses from vehicle owners

93.5% of the vehicle owners who responded drive either a passenger (car, people carrier) or goods (truck, van) vehicle. This is good to see as the means of gathering the diagnostics data is through a port available in these vehicles. For the remaining 6.5% who drive motorcycles the product will not work as the port is not available.

71% of the vehicle owners who responded say that they have some basic knowledge of vehicle maintenance, 22.6% say they are fully capable of performing maintenance on their own vehicle, while 6.5% say they don’t know the first thing about it.

Not surprisingly, 64.5% of the vehicle owners who responded use their vehicles daily, with the rest split between weekly and monthly use. This is a good number to see as with regular use, the need for routine maintenance becomes more important and the possibility of failure increases.

Product reaction from all respondents

All respondents expressed an interest in being able to receive warnings about potential maintenance needs of their vehicle, while only a small minority of people have an issue with the idea of installing the reader in their vehicle dependent on how secure the device would be.

To gather some information about the usefulness of the potential feature for determining the meaning of symbols displayed in the vehicle, the respondents were asked how many of the symbols displayed do they understand. Only a small number (13.6%) recognise all of them, 36.4% recognise most of the symbols, while 47.7% are only able to recognise a few of the symbols, and 2.3% of responders don’t understand any of the symbols.

2.3 Results

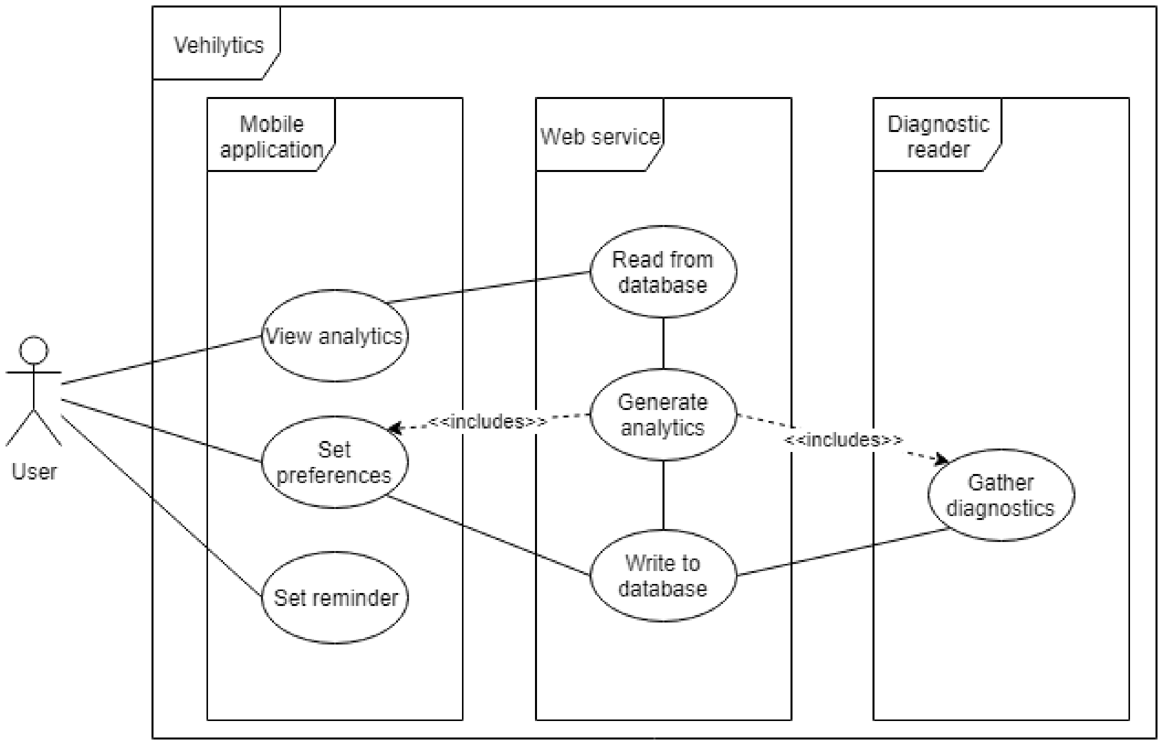
From my requirements gathering, I can see an interest in using this product as long as it meets a few requirements:

- The analytics are easy to read and informative
- The reader is easy to install and secure
- The user can be reminded of routine maintenance tasks
- The user is alerted to potential maintenance issues
- The mobile application provides a means to help understanding the vehicle

3 Requirements Specification

3.1 Functional Requirements

3.1.1 Use Case Diagram



3.1.2 Requirement 1: View analytics

Description and Priority

The analytics system will allow a user to view their vehicle diagnostics data in a readable format or access the raw diagnostic data if necessary. It has high priority as it is the primary function of the system.

Use Case

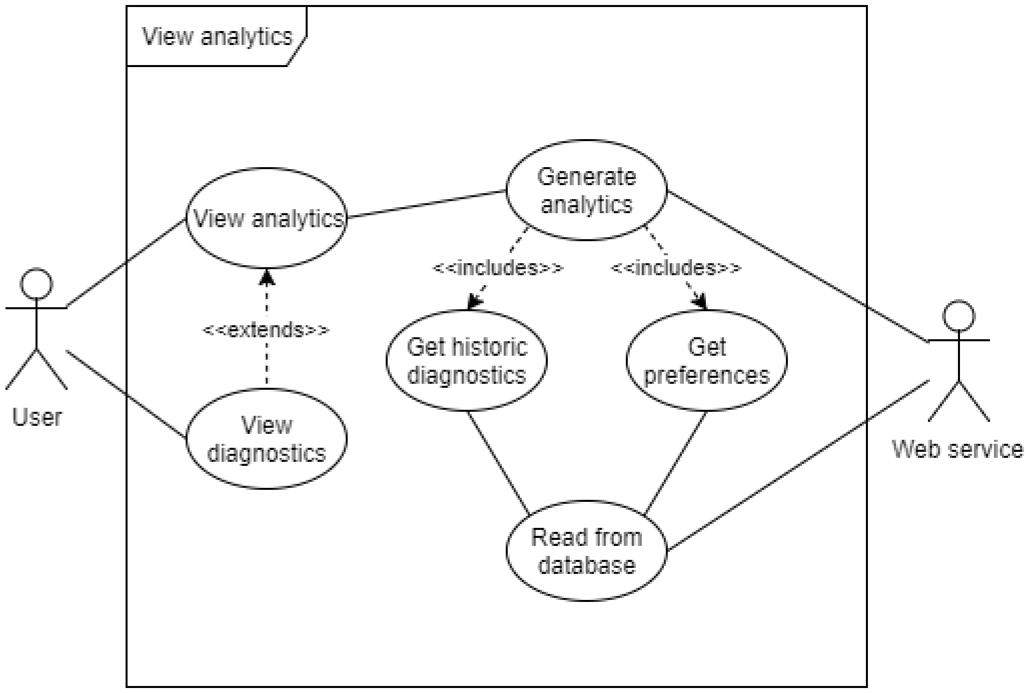
Scope

The scope of this use case is to allow a user to view their vehicle analytics.

Description

This use case describes the steps involved in requesting and viewing vehicle analytics.

Use Case Diagram



Precondition

- The user is logged in to the application
- The diagnostic reader is installed in the user's vehicle

Activation

The use case starts when the user has logged into the application.

Main Flow

1. The system displays the main screen
2. The user taps “View Readings” button
3. The system gathers the latest diagnostic reports [see E1]
4. The system displays the reports to the user
5. Main flow ends

Exceptional Flow

E1: The network is unavailable

1. The system displays an error message to the user
2. Use case ends

Termination

The user leaves the “View Readings” screen.

Post condition

The user is presented with the vehicle analytics that they can examine.

3.1.3 Requirement 2: User preferences

Description and Priority

The preferences system will allow a user to specify which analytics information they want to see or hide. Some data gathered might not be relevant to a user so they might want to ignore it entirely in the application. This system has a low priority as it doesn't affect the main function of the application.

Use Case

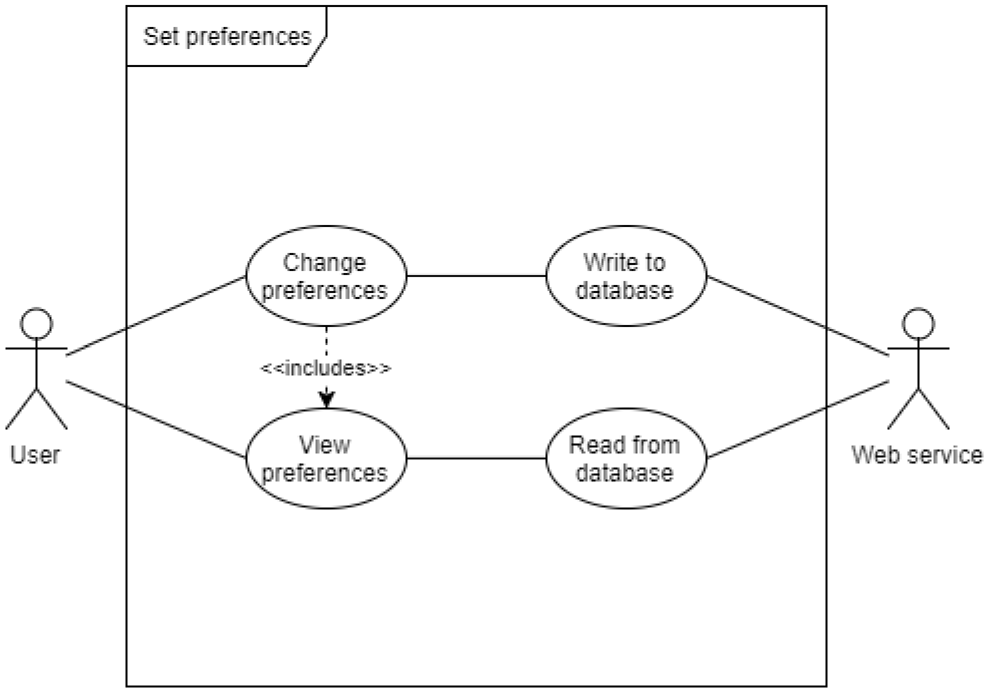
Scope

The scope of this use case is to allow a user to define their preferences.

Description

This use case describes the steps involved in updating user preferences.

Use Case Diagram



Precondition

- The user is logged in to the application

Activation

The use case starts when the user has logged into the application.

Main Flow

1. The system displays the main screen
2. The user taps the “User Preferences” button
3. The system displays the current preferences
4. The user edits their preferences
5. The system updates the stored preferences [see A1]
6. Main flow ends

Alternate Flow

A1: The network is unavailable

1. The system stores the changed preferences locally
2. The system updates the persistent storage when connection is re-established
3. Use case ends

Termination

The user discards any changes to their preferences.

Post condition

The user preferences are updated.

3.1.4 Requirement 3: Reminders

Description and Priority

The reminders system will allow a user to set reminders for them to perform routine maintenance checks, e.g. tyre pressure, changing oil. The user might want to adjust frequency of updates if their vehicle is older or if a fault has occurred before because of a particular type of maintenance being neglected. The priority for this feature is low as it doesn't affect the main function of the system.

Use Case

Scope

The scope of this use case is to allow a user to set reminders for performing routine maintenance checks.

Description

This use case describes the steps involved in creating a reminder.

Precondition

- The user is logged in to the application

Activation

The use case starts when the user has logged into the application.

Main Flow

1. The system displays the main screen
2. The user taps the "Task Reminders" button
3. The system displays the reminders screen
4. The user taps the add icon
5. The system displays the create reminder screen
6. The user sets the reminder details
7. The system creates the reminder [see A1]
8. Main flow ends

Alternate Flow

A1: The task already has a reminder

1. The system asks the user if they want to replace the current reminder
2. The user replaces the current reminder [see E1]
3. Return to main flow position 7

Exceptional Flow

E1: The reminder is discarded

1. The system displays the main screen
2. Use case ends

Termination

The user discards the reminder.

Post condition

The device will now remind the user about the task described.

3.2 Non-Functional Requirements

3.2.1 Performance/Response time

The web service should be able to quickly respond to user requests for authentication and displaying diagnostics information. The web resource will be responsible for most of the heavy computation, such as generating useful metrics for decision making based on historical diagnostics so it will be important that these can be served to the user in a small amount of time.

3.2.2 Security

The web service will be collecting and storing sensitive user information such as, email and password. This information must be handled correctly such that malicious users cannot gain access to these credentials. All user credentials and requests should be suitably encrypted.

The diagnostics reader will be communicating with the sensor kits via Bluetooth so it will be necessary for it to be able to distinguish between Bluetooth transmitters that are verified and not verified so as not to gather erroneous data.

A mobile client should only be able to connect to a device that the user possesses. A user should not be able to get diagnostics reports from a device they don't own by guessing some identifying information.

3.2.3 Maintainability

An important feature of this project is for users to be able to integrate additional sensor kits into the diagnostics reader, have the data report correctly up to the web service and for that information to be readable by the mobile client. It should be as easy as possible to integrate or remove these sensor kits, or replace them if they are faulty.

3.2.4 Extensibility

The mobile application could evolve in the future to include features separate from the diagnostics reader but still related to maintaining vehicle health. This could be simple reminders to check tire pressure, wiper fluid, and oil, or more advanced features like allowing the user to take a picture of a symbol displayed in the car and display information about that symbol. Building the application in such a way that these features can be implemented without breaking the remainder of the system should be a primary concern.

4 Interface requirements

4.1 Mobile application GUI

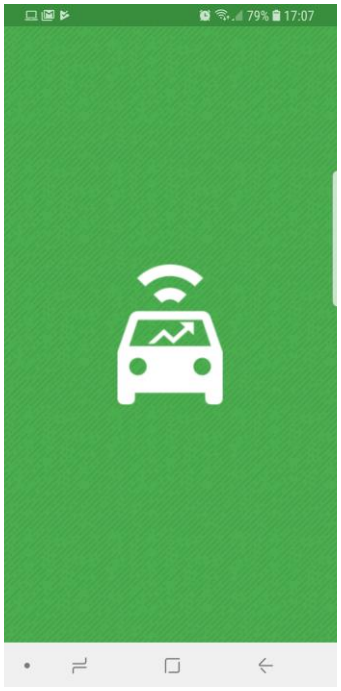


Fig 1. Splash screen

The application opens to a simple splash screen (Fig 1) which displays the application logo. The splash screen acts as a nice introduction to the colour scheme of the application and in the background, it checks for any existing credentials stored on the device. If any credentials have been stored it will validate them with the server and either pass the user along to the login screen if they are invalid or straight to the home screen if they are still valid.

The login screen (Fig 2) displays the application logo and name, and prompts the user to enter their email and password if they are registered. If any fields are invalid, an error message will display. If they are not registered, they can tap the "Sign up now" label to be taken to the registrations page when they can create a new account. After a successful login, the user will be taken to the Home screen.

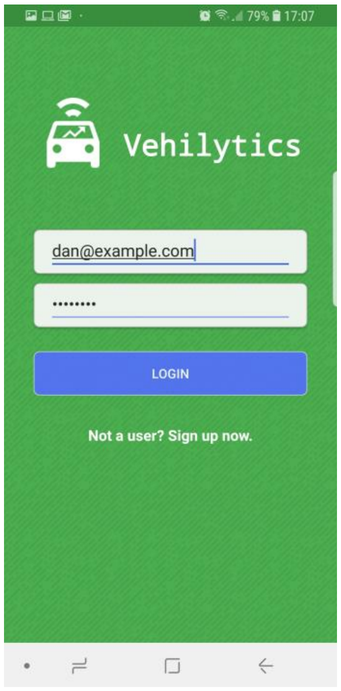


Fig 2. Login screen

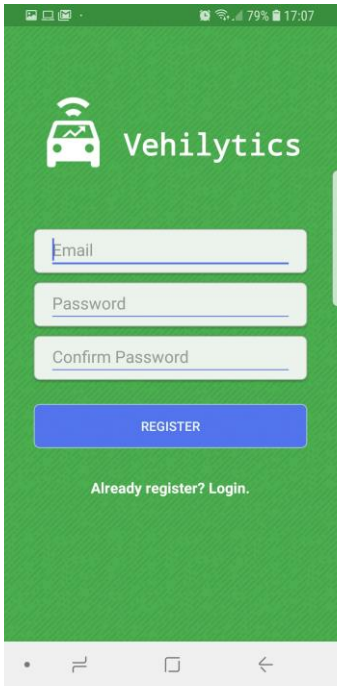


Fig 3. Register

The registration screen (Fig 3) displays the app logo and name, and allows the user to create a new account by entering a valid email, a password and confirming that password. If any fields are invalid, then an error message will display. If the user already has an account, they can tap the “Login” label to be taken to the Login screen. After a successful registration, the user will be taken to the Home screen.

After logging in successfully, the user is presented with the home screen (Fig 4) from which they can navigate to any other part of the application. The home screen consists of a number of views that display information about which activity they lead to and act as buttons to allow the application to transition to that activity.

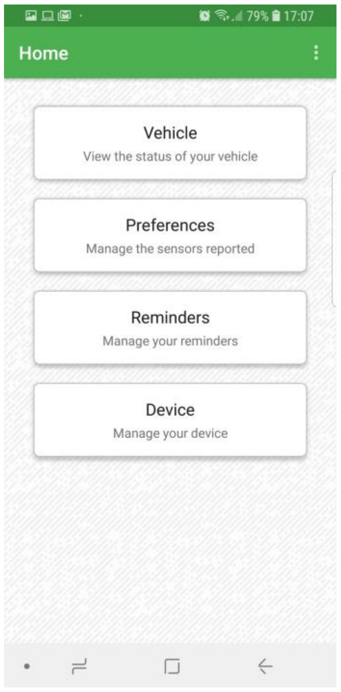


Fig 4. Home screen

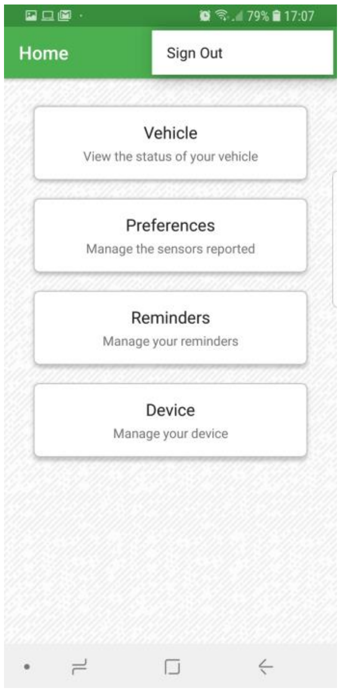


Fig 5. Menu Options

A menu (Fig 5) can be accessed from any screen with an application toolbar by tapping the menu button in the top right. This gives access to the "Sign Out" function which will signal to the web service to reset any authentication tokens and take the user back to the Login screen. Once a sign out request has been made, and the tokens have been reset, any users attempting to use the old token will be forcibly logged out of the application.

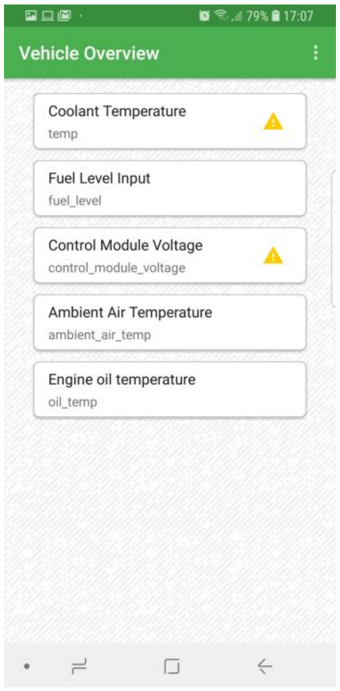
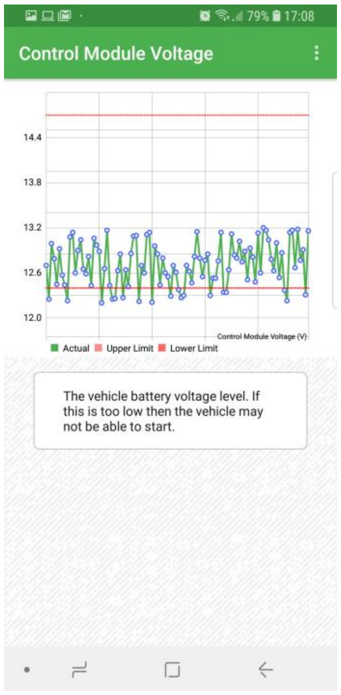


Fig 6. Vehicle screen

Tapping the "Vehicle" button will bring the user to the Vehicle Overview screen (Fig 6) where they can see which sensors have been reported by the system. If there are a few readings for that sensor that fall outside of safe ranges then a warning will be issued and a yellow indicator will display. If there are a lot of readings for that sensor fall outside safe ranges then a red error indicator will display. Tapping on a sensor will graph the readings to provide a visual representation of that sensor.

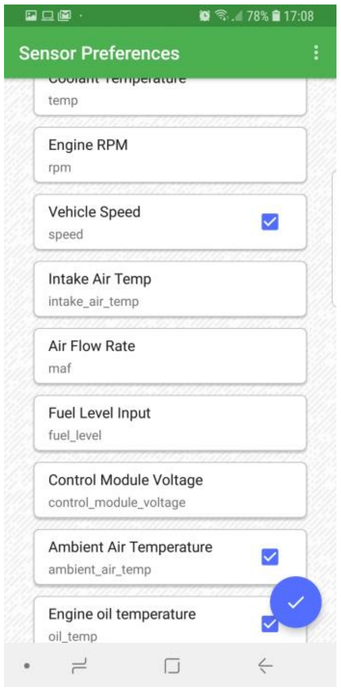

79% 17:08
Control Module Voltage
14.4
13.8
13.2
12.6
12.0
Control Module Voltage (V)
🟩 Actual 🟥 Upper Limit 🟥 Lower Limit

The vehicle battery voltage level. If this is too low then the vehicle may not be able to start.

Fig 7. Graph screen

The Graph screen (Fig 7) displays the readings data in a way that effectively communicates the state of that sensor over the previous 100 readings. If the sensor has upper or lower safe ranges then they will be displayed as red lines to allow the user to understand how the component is behaving at a glance. Information about the selected sensor is displayed at the bottom of the screen and is updated based on what information is available on the server.

The Sensor Preferences screen (Fig 8) allow the user to update which sensors they want the diagnostic reader to observe and report to the web service. The user can tap each of the sensors to add it to the list of preferences locally and then submit it to the web service by tapping the floating action button in the bottom right. Any preferences saved to the web service will be received by the diagnostic reader and will populate the preferences screen the next time it is opened.



Coolant Temperature
temp

Engine RPM
rpm

Vehicle Speed ✓
speed

Intake Air Temp
intake_air_temp

Air Flow Rate
maf

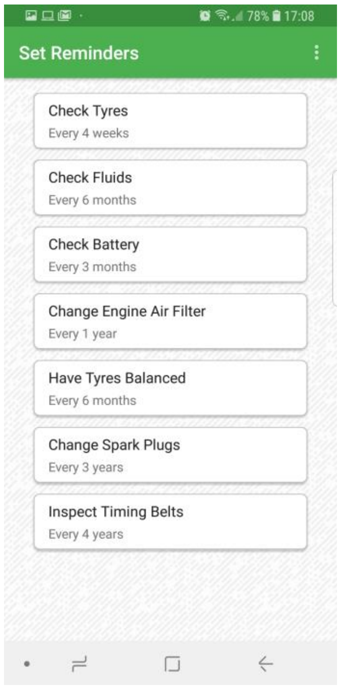
Fuel Level Input
fuel_level

Control Module Voltage
control_module_voltage

Ambient Air Temperature ✓
ambient_air_temp

Engine oil temperature ✓
oil_temp

Fig 8. Preferences



Check Tyres
Every 4 weeks

Check Fluids
Every 6 months

Check Battery
Every 3 months

Change Engine Air Filter
Every 1 year

Have Tyres Balanced
Every 6 months

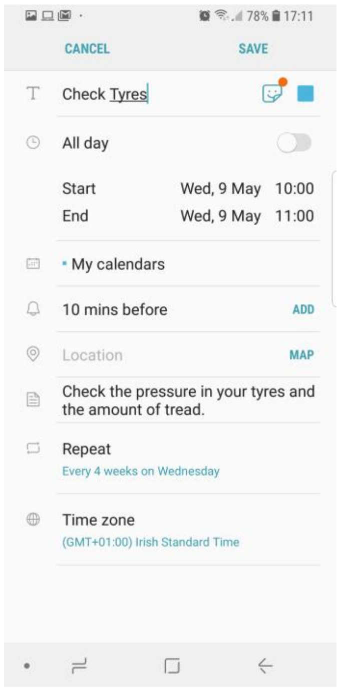
Change Spark Plugs
Every 3 years

Inspect Timing Belts
Every 4 years

Fig 9. Reminders

The Reminders screen (Fig 9) displays a list of reminders recommended by the web service. The list displays a title for the reminder and how often the task should be performed. Tapping on a reminder recommendation will open the users calendar and allow them to create the reminder.

When a reminder recommendation is selected a new calendar entry will be created and populated with a title, time, frequency, and description (Fig 10). The user can then fine tune the reminder to their own preference or save the default.



Check Tyres

All day

Start	Wed, 9 May	10:00
End	Wed, 9 May	11:00

My calendars

10 mins before

Location

Check the pressure in your tyres and the amount of tread.

Repeat
Every 4 weeks on Wednesday

Time zone
(GMT+01:00) Irish Standard Time

Fig 10. Calendar

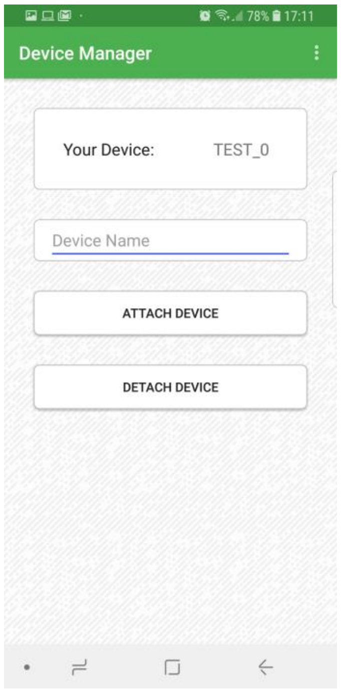
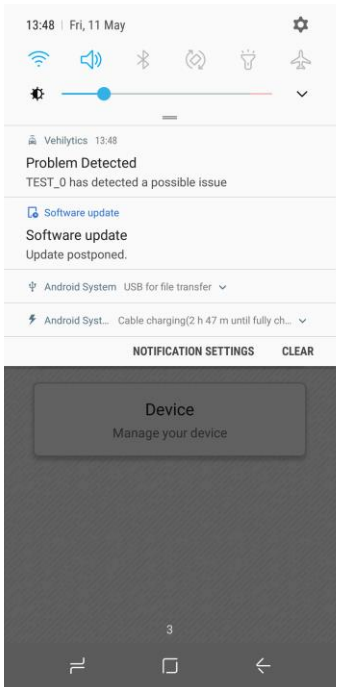


Fig 11. Device

The device manager screen (Fig 11) allows the user to attach or detach a diagnostic reader from their account. A device can only be attached to a single user to prevent multiple users from receiving information which doesn't belong to them. It displays the currently attached device at the top. The device name is required to perform any attaching or detaching.

The user will be alerted via push notifications (Fig 12) is any problems have been detected by the web service. Readings are examined when they are published from the diagnostic reader and notifications pushed out immediately once an issue is detected. Tapping the notification will bring the user to the Vehicle Overview screen.


13:48 | Fri, 11 May

Vehilytics 13:48
Problem Detected
TEST_0 has detected a possible issue

Software update
Software update
Update postponed.

Android System USB for file transfer

Android Syst... Cable charging(2 h 47 m until fully ch...

Fig 12. Push Notifications

4.2 Web API

The web service is responsible for handling user/device authentication, gathering diagnostic reports from diagnostic readers, and making these reports available to users. The mobile application and diagnostic readers cannot communicate directly with each other for security reasons and simplicity in restricting access to certain resources.

There are several authenticated and unauthenticated endpoints. The unauthenticated endpoints allow clients to sign in, register new accounts, or to access unrestricted resources such as, the sensors supported by the system. The authenticated endpoints provide means for a client to access restricted resources such as, user preferences, viewing vehicle diagnostic, or publishing diagnostic readings.

These endpoints will be restricted and only accept requests made by the appropriate device. This means that only a diagnostic reader will be able to submit reports to that specific endpoint. This is to prevent malicious users who know the resource location from submitting erroneous data through their own client.

4.3 OBD

The OBD port in a vehicle is a standard that was introduced to allow consistent communication with the vehicle’s numerous ECUs. The diagnostic reader will interface with this port using a Bluetooth adapter which will allow the device to sit in the vehicle glovebox after installation without leaving cables visible.

With the diagnostic reader acting as the master in this Bluetooth network, it may be possible to add in additional Bluetooth slave devices that will also generate usable data. This could mean that the add-on modules would form a wireless sensor network around the car to serve multiple purposes with needing cables.

5 System Architecture

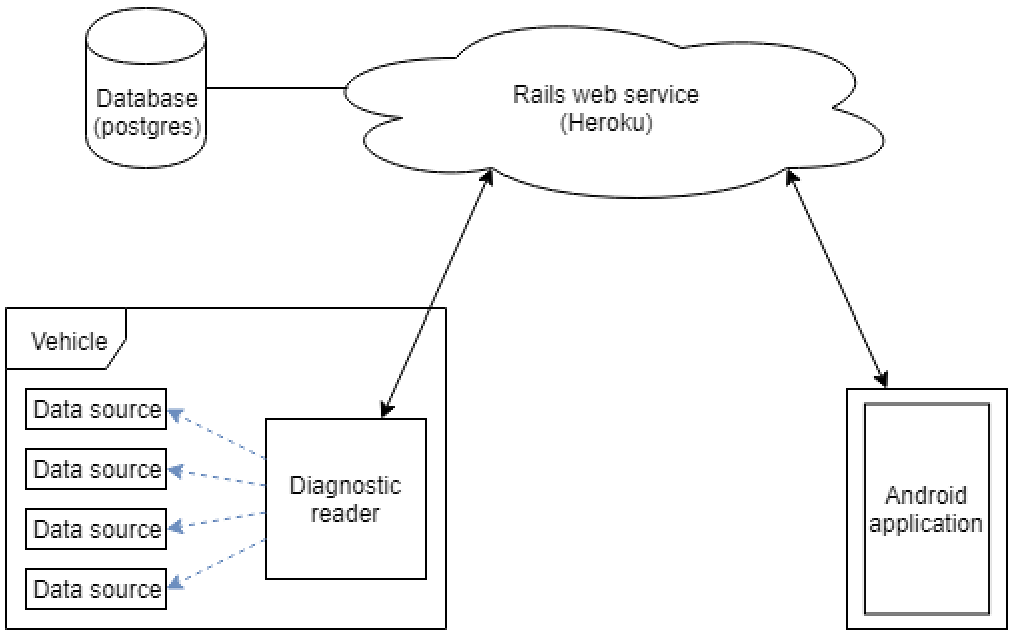


Fig 1. Architecture overview

There are 5 main components to the Vehilytics platform architecture:

- Diagnostic reader
- Web service
- Mobile application
- Persistent storage
- Data sources

5.1 Diagnostic Reader Overview

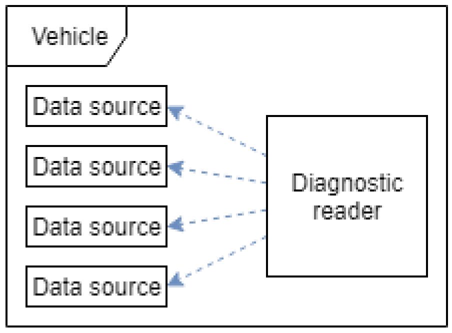


Fig 2. Diagnostic reader / Data sources

The diagnostic reader is responsible for gathering data from the vehicle in the form of diagnostics or other useful sensor readings. It receives this data from various data sources around the vehicle. These data sources include the vehicle OBD port, and any additional modules which might be installed into the vehicle. The diagnostic reader can use Bluetooth or USB to communicate with the various data sources so there is no need to install numerous cables around the vehicle.

The code for the diagnostic reader was built on a fork from a library called PyOBD which handles the low-level communications through the serial port. The existing library provided a graphical user interface for the user to interact with but for my project I need it to run headless (without a user interface). I extended the project to include a headless reporter which runs automatically when the Pi turns on, connecting to the API and gathering the required data.

I hit a major obstacle here in that I purchased a number of OBD reader units, a device that plugs into the actual OBD port and provides the communication point to the vehicle ECU, but none of them actually worked. I was unable to retrieve any data from the vehicle so I was forced to create a dummy interface for the reporter that generated fake data instead of pulling real data from the sensors. It’s unfortunate that this didn’t work as intended but by pushing up any data I could see the rest of the system work perfectly since the web service doesn’t care how the data is created.

5.2 Android Application Overview

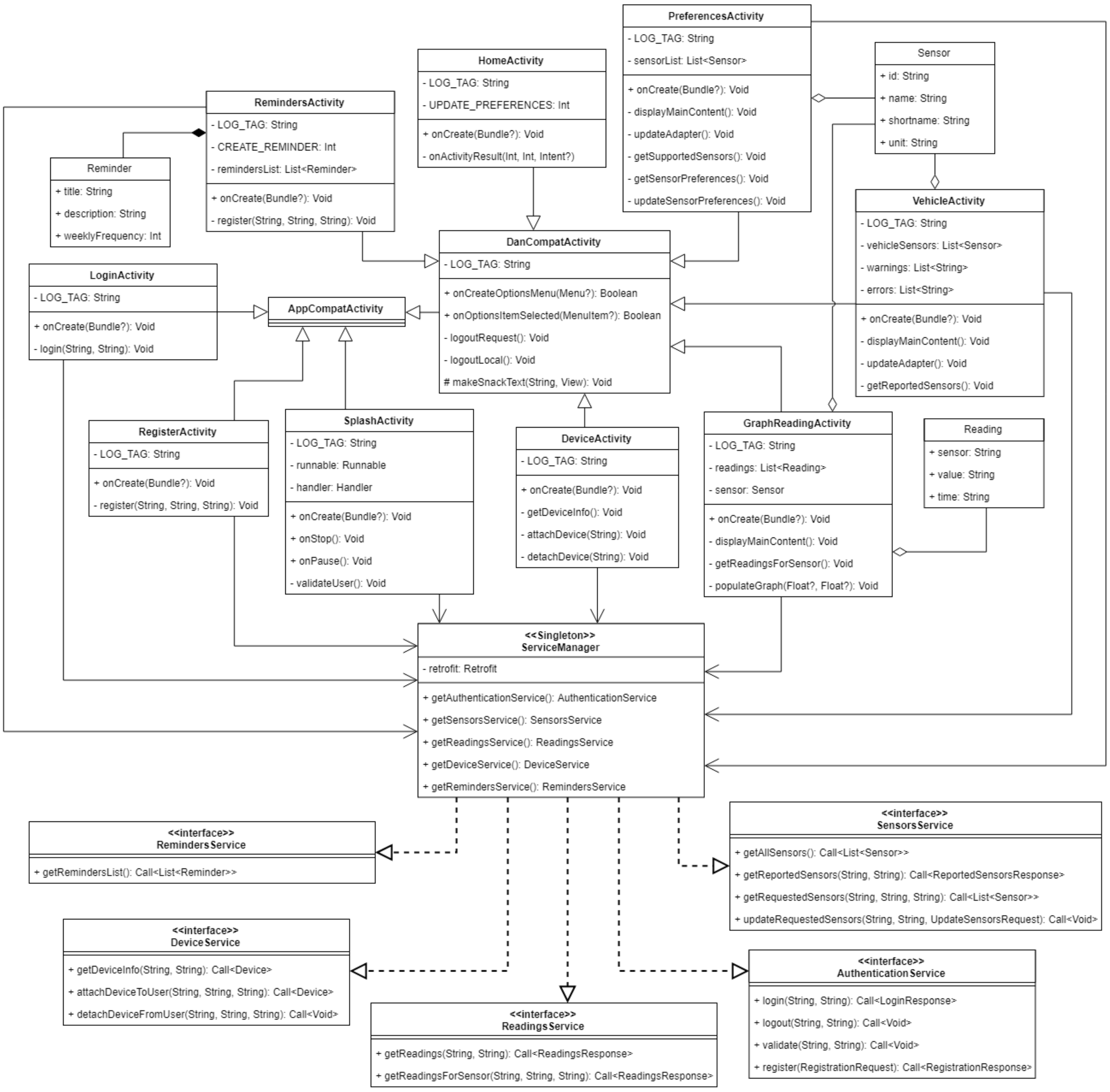

PreferencesActivity
- LOG_TAG: String
- sensorList: List<Sensor>
+ onCreate(Bundle?): Void
- displayMainContent(): Void
- updateAdapter(): Void
- getSupportedSensors(): Void
- getSensorPreferences(): Void
- updateSensorPreferences(): Void
HomeActivity
- LOG_TAG: String
- UPDATE_PREFERENCES: Int
+ onCreate(Bundle?): Void
- onActivityResult(Int, Int, Intent?)
Sensor
+ id: String
+ name: String
+ shortname: String
+ unit: String
RemindersActivity
- LOG_TAG: String
- CREATE_REMINDER: Int
- remindersList: List<Reminder>
+ onCreate(Bundle?): Void
- register(String, String, String): Void
Reminder
+ title: String
+ description: String
+ weeklyFrequency: Int
DanCompatActivity
- LOG_TAG: String
+ onCreateOptionsMenu(Menu?): Boolean
+ onOptionsItemSelected(MenuItem?): Boolean
- logoutRequest(): Void
- logoutLocal(): Void
# makeSnackText(String, View): Void
VehicleActivity
- LOG_TAG: String
- vehicleSensors: List<Sensor>
- warnings: List<String>
- errors: List<String>
+ onCreate(Bundle?): Void
- displayMainContent(): Void
- updateAdapter(): Void
- getReportedSensors(): Void
LoginActivity
- LOG_TAG: String
+ onCreate(Bundle?): Void
- login(String, String): Void
AppCompatActivity
RegisterActivity
- LOG_TAG: String
+ onCreate(Bundle?): Void
- register(String, String, String): Void
SplashActivity
- LOG_TAG: String
- runnable: Runnable
- handler: Handler
+ onCreate(Bundle?): Void
+ onStop(): Void
+ onPause(): Void
- validateUser(): Void
DeviceActivity
- LOG_TAG: String
+ onCreate(Bundle?): Void
- getDeviceInfo(): Void
- attachDevice(String): Void
- detachDevice(String): Void
GraphReadingActivity
- LOG_TAG: String
- readings: List<Reading>
- sensor: Sensor
+ onCreate(Bundle?): Void
- displayMainContent(): Void
- getReadingsForSensor(): Void
- populateGraph(Float?, Float?): Void
Reading
+ sensor: String
+ value: String
+ time: String
<<Singleton>>
ServiceManager
- retrofit: Retrofit
+ getAuthenticationService(): AuthenticationService
+ getSensorsService(): SensorsService
+ getReadingsService(): ReadingsService
+ getDeviceService(): DeviceService
+ getRemindersService(): RemindersService
<<interface>>
RemindersService
+ getRemindersList(): Call<List<Reminder>>
<<interface>>
SensorsService
+ getAllSensors(): Call<List<Sensor>>
+ getReportedSensors(String, String): Call<ReportedSensorsResponse>
+ getRequestedSensors(String, String, String): Call<List<Sensor>>
+ updateRequestedSensors(String, String, UpdateSensorsRequest): Call<Void>
<<interface>>
DeviceService
+ getDeviceInfo(String, String): Call<Device>
+ attachDeviceToUser(String, String, String): Call<Device>
+ detachDeviceFromUser(String, String, String): Call<Void>
<<interface>>
AuthenticationService
+ login(String, String): Call<LoginResponse>
+ logout(String, String): Call<Void>
+ validate(String, String): Call<Void>
+ register(RegistrationRequest): Call<RegistrationResponse>
<<interface>>
ReadingsService
+ getReadings(String, String): Call<ReadingsResponse>
+ getReadingsForSensor(String, String, String): Call<ReadingsResponse>

Fig 3. Android Class Diagram

The above diagram shows the structure of the Android application architecture while omitting a number of data classes, OnClickListener interfaces, RecyclerView adapters, and custom Callback objects.

I wrote a class to extend AppCompatActivity so that I could implement shared functionality across a number of different activities in the form of the sign out menu option. The DanCompatActivity class handles creating the options menu and performing actions when an option is selected. It also has a protected method to allow its child activities to display Snackbar messages when an event occurs.

I used Retrofit, a library that uses Kotlin interfaces to expose API endpoints, to handle all the networking in the application and built the ServiceManager singleton to handle the retrofit instance and expose the interfaces. The advantage of this design is that if I add some headers to the retrofit instance, then every request will have the new headers without the need to update every class making requests. It also means that when I have a new endpoint I simply have to add it to the interface/manager and it will be available throughout the application.

Retrofit provides a Callback interface with 2 methods, onError and onResponse, that gets used when making requests. I created an abstract class that implements this Callback and provides an onResponse method that will force the application to sign out the user if any requests ever return a status code 401 (unauthorised). If the server ever returns this status code, it means that the user authentication token was invalid so the token was either forged or the user signed out on another device.

There are a lot of small data classes in the application for holding information about entities specific to my system such as, a sensor, a reminder, and a sensor reading. These models allow me to juggle the information between classes, pass the information into requests, and parse JSON responses into meaningful objects.

There are a number of lists throughout the application and they are all built as RecyclerViews with their own custom adapters to maintain efficiency as the lists grow longer and longer. Each of these lists contain items with nested views (multiple TextViews, checkboxes, ImageViews), and having custom adapters allows me to fine tune how they all behave and look.

5.3 Web Service Overview

For security and performance reasons, the diagnostic reader and mobile application should never communicate directly. The web service acts as the communication point for every device in the system. This ensures that diagnostic reports and preferences are stored for later use by the diagnostic reader or the mobile application.

The web service has 2 types of clients interacting with it, Users and Devices. Users are people using the Android application and they have access to historic diagnostic data, can update preferred sensors to be reported. Devices are diagnostic readers and they can view the preferred sensors of their associated user and publish sensor readings in the form of reports consisting of a timestamp and collection of sensor values. Both types of client have their own authentication endpoints that give them authentication tokens to allow them to make use of the endpoints.

The data is persisted in a Postgresql database that contains encrypted user credentials, sensor information, historic diagnostic data, and recommendations for maintenance task reminders. The web service contains many endpoints, most of which are authenticated, that are described by the API documentation in the appendices.

5.3.1 Web Service Endpoints

5.3.1.1 Sessions Controller

Validate user credentials

URL	/v1/sessions
Method	GET
URL Params	None
Data Params	None

Header Params	X-User-Email=<user_email>
	X-User-Token=<user_auth_token>

Success Response	Code: 200 OK
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X GET -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" "https://vehilytics-api.herokuapp.com/v1/sessions"

User sign in

URL	/v1/sessions
Method	POST
URL Params	None

Data Params	{ “email”: <user_email>, “password”: <user_password> }
	OR to enable push notifications on Android
	{ “email”: <user_email>, “password”: <user_password>, “firebase_token”: <firebase_instance_id_token> }

Header Params	Content-Type=application/json
Success Response	Code: 201 CREATED
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X POST -H "Content-Type: application/json" "https://vehilytics-api.herokuapp.com/v1/sessions" -d ‘{ “email”: <user_email>, “password”: <user_password> }’

User sign out

URL	/v1/sessions
Method	DELETE
URL Params	None
Data Params	None

Header Params	X-User-Email=<user_email>
	X-User-Token=<user_auth_token>

Success Response	Code: 200 OK
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X DELETE -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" "https://vehilytics-api.herokuapp.com/v1/sessions"

5.3.1.2 Device Sessions Controller

Validate device credentials

URL	/v1/device_sessions

Method	GET
URL Params	None
Data Params	None

Header Params	X-Device-Email=<device_email>
	X-Device-Token=<device_auth_token>

Success Response	Code: 200 OK
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X GET -H "X-Device-Token: <device_auth_token>" -H "X-Device-Email: <device_email>" "https://vehilytics-api.herokuapp.com/v1/device_sessions"

Device sign in

URL	/v1/device_sessions
Method	POST
URL Params	None
Data Params	{ “email”: <device_email>, “password”: <device_password> }
Header Params	Content-Type=application/json
Success Response	Code: 201 CREATED
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X POST -H "Content-Type: application/json" "https://vehilytics-api.herokuapp.com/v1/sessions" -d ‘{ “email”: <device_email>, “password”: <device_password> }’

Device sign out

URL	/v1/device_sessions
Method	DELETE
URL Params	None
Data Params	None

Header Params	X-Device-Email=<device_email>
	X-Device-Token=<device_auth_token>

Success Response	Code: 200 OK
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X DELETE -H "X-Device-Token: <device_auth_token>" -H "X-Device-Email: <device_email>" "https://vehilytics-api.herokuapp.com/v1/device_sessions"

5.3.1.3 Devices Controller

Get user's device info

URL	/v1/devices
Method	GET
URL Params	None
Data Params	None

Header Params	X-User-Email=<user_email>
	X-User-Token=<user_auth_token>

Success Response	Code: 200 OK Body: { “email”: <device_email>, “device_name”: <device_name> }
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X GET -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" "https://vehilytics-api.herokuapp.com/v1/devices"

Attach device to user

URL	/v1/devices
Method	POST
URL Params	function=attach
Data Params	{ “device_name”: <device_name> }

Header Params	X-User-Email=<user_email>
	X-User-Token=<user_auth_token>
	Content-Type=application/json

Success Response	Code: 201 CREATED Body: { “email”: <device_email>, “device_name”: <device_name> }
Error Response	Code: 400 BAD REQUEST Code: 401 UNAUTHORISED

Curl	curl -X POST -H “Content-Type: application/json” -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" -d { “device_name”: <device_name> } "https://vehilytics-api.herokuapp.com/v1/devices?function=attach"

Detach device from user

URL	/v1/devices
Method	POST
URL Params	function=detach
Data Params	{ “device_name”: <device_name> }

Header Params	X-User-Email=<user_email>
	X-User-Token=<user_auth_token>
	Content-Type=application/json

Success Response	Code: 200 OK
Error Response	Code: 400 BAD REQUEST Code: 401 UNAUTHORISED
Curl	curl -X POST -H “Content-Type: application/json” -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" -d { “device_name”: <device_name> } "https://vehilytics-api.herokuapp.com/v1/devices?function=detach"

5.3.1.4 Readings Controller

Get all readings

URL	/v1/readings
Method	GET
URL Params	None
Data Params	None

Header Params	X-User-Email=<user_email>
	X-User-Token=<user_auth_token>

Success Response	Code: 200 OK Data: { “readings”: [{ “sensor”: <sensor_shortname>, “value”: <value>, “time_reported”: <timestamp> }, …] }
Error Response	Code: 401 UNAUTHORISED

Curl	curl -X GET -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" "https://vehilytics-api.herokuapp.com/v1/readings"

Get all readings for sensor

URL	/v1/readings
Method	GET
URL Params	sensor=<sensor_shortname>
Data Params	None

Header Params	X-User-Email=<user_email>
	X-User-Token=<user_auth_token>

Success Response	Code: 200 OK Data: { “readings”: [{ “sensor”: <sensor_shortname>, “value”: <value>, “time_reported”: <timestamp> }, …] }
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X GET -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" "https://vehilytics-api.herokuapp.com/v1/readings?sensor=<sensor_shortname>"

5.3.1.5 Registrations Controller

Register new user

URL	/users
Method	POST
URL Params	None
Data Params	{ “email”: <user_email>, “password”: <password>, “password_confirmation”: <password> }
Header Params	Content-Type=application/json
Success Response	Code: 201 OK Body: { “token”: <user_auth_token> }
Error Response	Code: 401 UNAUTHORISED Body: { “message”: “Email address already registered / New user not saved” }

Curl	curl -X POST -H “Content-Type: application/json” -d ‘{ “email”: <user_email>, “password”: <password>, “password_confirmation”: <password> }’ "https://vehilytics-api.herokuapp.com/users"

Register new device

URL	/devices
Method	POST
URL Params	None
Data Params	{ “email”: <device_email>, “device_name”: <device_name>, “password”: <password>, “password_confirmation”: <password> }
Header Params	Content-Type=application/json
Success Response	Code: 201 OK Body: { “token”: <device_auth_token> }
Error Response	Code: 401 UNAUTHORISED Body: { “message”: “Email address or device name already registered / New device not saved” }
Curl	curl -X POST -H “Content-Type: application/json” -d ‘{ “email”: <device_email>, “device_name”: <device_name>, “password”: <password>, “password_confirmation”: <password> }’ "https://vehilytics-api.herokuapp.com/devices"

5.3.1.6 Reminders Controller

Get all recommended reminders

URL	/v1/reminders
Method	GET
URL Params	None
Data Params	None
Header Params	None
Success Response	Code: 200 OK Data: [{ “id”: <id>, “title”: <reminder_title>, “weekly_frequency”: <frequency>, “description”: <description> }]
Error Response	Code: 401 UNAUTHORISED

Curl	curl -X GET "https://vehilytics-api.herokuapp.com/v1/reminders"

5.3.1.7 Reports Controller

Create a report with readings

URL	/v1/readings
Method	POST
URL Params	None
Data Params	{ “time_reported”: <timestamp>, “device_name”: <device_name>, “readings”: [{ “shortname”: <sensor_shortname>, “value”: <value> }] }
Header Params	X-Device-Email=<device_email> X-Device-Token=<device_auth_token> Content-Type=application/json
Success Response	Code: 201 CREATED Data: { “time_reported”: <timestamp>, “device_name”: <device_name>, “readings”: [{ “shortname”: <sensor_shortname>, “value”: <value> }] }
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X POST -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" -H “Content-Type: application/json” -d ‘{ “time_reported”: <timestamp>, “device_name”: <device_name>, “readings”: [{ “shortname”: <sensor_shortname>, “value”: <value> }] }’ "https://vehilytics-api.herokuapp.com/v1/readings"

5.3.1.8 Sensors Controller

Get all supported sensors

URL	/v1/sensors
Method	GET
URL Params	None
Data Params	None
Header Params	None
Success Response	Code: 200 OK Data: [{ “id”: <id>, “name”: <sensor_name>, “shortname”: <sensor_shortname>, “unit”: <measurement_unit> }, ...]
Error Response	-

Curl	curl -X GET "https://vehilytics-api.herokuapp.com/v1/sensors"

Get all sensors with available readings

URL	/v1/sensors
Method	GET
URL Params	None
Data Params	None

Header Params	X-User-Email=<user_email>
	X-User-Token=<user_auth_token>

Success Response	Code: 200 OK Data: [{ “id”: <id>, “name”: <sensor_name>, “shortname”: <sensor_shortname>, “unit”: <measurement_unit> }, ...]
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X GET -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" "https://vehilytics-api.herokuapp.com/v1/sensors"

Get all user requested sensors

URL	/v1/sensors
Method	GET
URL Params	None
Data Params	None

Header Params	X-Device-Email=<device_email>
	X-Device-Token=<device_auth_token>

Success Response	Code: 200 OK Data: [{ “id”: <id>, “name”: <sensor_name>, “shortname”: <sensor_shortname>, “unit”: <measurement_unit> }, ...]
Error Response	Code: 401 UNAUTHORISED
Curl	curl -X GET -H "X-Device-Token: <device_auth_token>" -H "X-Device-Email: <device_email>" "https://vehilytics-api.herokuapp.com/v1/sensors"

Update requested sensors

URL	/v1/sensors
Method	POST
URL Params	None
Data Params	{ “sensors”: [<sensor1_shortname>, <sensor2_shortname>, ...] }

Header Params	X-User-Email=<user_email>
	X-User-Token=<user_auth_token>

Success Response	Code: 201 CREATED
Error Response	Code: 401 UNAUTHORISED Code: 400 BAD REQUEST
Curl	curl -X GET -H "X-User-Token: <user_auth_token>" -H "X-User-Email: <user_email>" "https://vehilytics-api.herokuapp.com/v1/sensors"

6 Testing

Several tests were implemented throughout the building of this project to ensure the integrity of features that were built early on while more features were added and code was refactored. The 2 main types of testing performed were unit tests and integration tests. Unit testing was performed on the Android application while Integration testing was performed on the web service controllers. The diagnostic reader was built upon a forked project that wasn't testable but unit tests were added for the portions I had written.

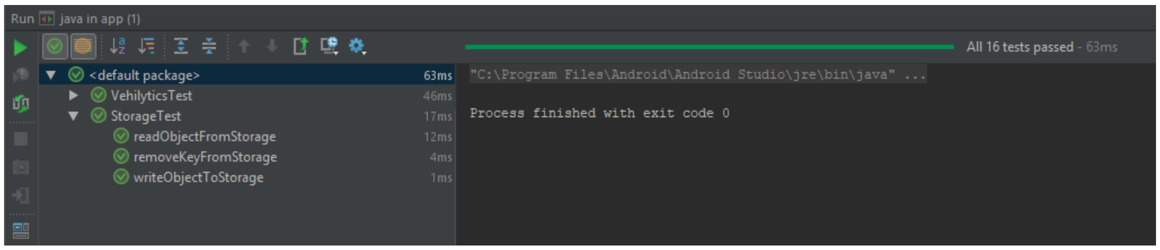
6.1 Unit Testing

Unit testing was performed for some of the Kotlin classes in the Android application using JUnit and Mockito, and some portions in the Python code for the diagnostic reader using PyUnit.

6.1.1 Android Application

Unit testing on Android can be tricky because of the dependencies on framework classes like Context or SharedPreferences. I opted to avoid testing my activities because they have a lot of framework dependencies and don't actually have a lot of logic contained within them. I instead focused on testing classes that offered reusable functionality with minimal framework dependencies. The 2 main classes I tested were "Vehilytics.kt", a singleton object that allowed access to user details throughout the application, and "Storage.kt", a class that handled reading from and writing to SharedPreferences.

The Vehilytics class was easy enough to test as it's only dependency is on the Storage class which I could easily mock using Mockito. Mocking the class allows me to control what happens when methods from that class are called by returning values I specify so I can focus on verifying the order of operations for a given method. The Storage class was a little trickier to test as it has a dependency on Context and SharedPreferences. Using Mockito allowed me to mock both classes once again and avoid any difficulties they presented.



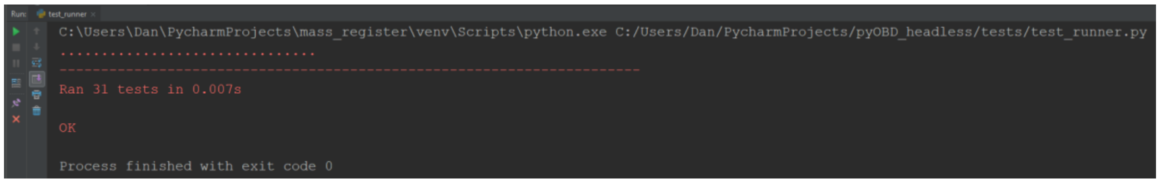
```
<default package>              63ms
  VehilyticsTest               46ms
  StorageTest                  17ms
    readObjectFromStorage      12ms
    removeKeyFromStorage        4ms
    writeObjectToStorage        1ms
```

```
"C:\Program Files\Android\Android Studio\jre\bin\java" ...

Process finished with exit code 0
```

6.1.2 Diagnostic Reader

The diagnostic reader was not easy to test since the majority of the code was not written in a testable way. I've tried to keep the code I wrote on top testable and have written a series of unit tests for the classes and methods that warranted testing. Testing for this system was written using the unittest module (also known as PyUnit).



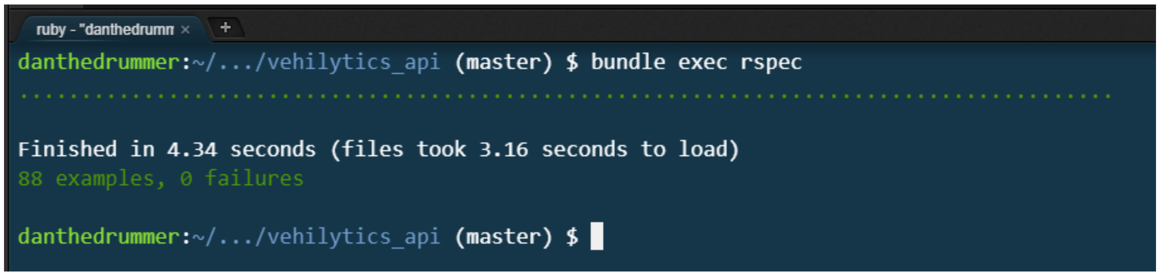
```
C:\Users\Dan\PycharmProjects\mass_register\venv\Scripts\python.exe C:/Users/Dan/PycharmProjects/pyOBD_headless/tests/test_runner.py
...............................
----------------------------------------------------------------------
Ran 31 tests in 0.007s

OK

Process finished with exit code 0
```

6.2 Integration Testing

Most of the testing for the web service was done as request specifications using the RSpec library and allowed me to test the behaviour of the endpoints under various conditions. I could easily test the status code of a requests when credentials were valid or invalid, or the return body for requests asking for collections of data. It also enabled me to test all of my models in terms of their relationships and validators to ensure that all of the entries in the database were acting correctly when created through requests to the controllers.



```
danthedrummer:~/.../vehilytics_api (master) $ bundle exec rspec
........................................................................................

Finished in 4.34 seconds (files took 3.16 seconds to load)
88 examples, 0 failures

danthedrummer:~/.../vehilytics_api (master) $
```

6.3 System Usability Scale

The system usability scale is a quick way to determine the effectiveness of a system that has proven itself to be dependable but does not provide a 100% accurate representation of the actual practicality of the system. The scale is a questionnaire with 10 questions to be answered by users of the system to help determine the effectiveness of the design and implementation.

Here is a list of the 10 questions I used, adapted from some template questions (Thomas, 2018):

1. I would regularly use this system.
2. I found it difficult to understand the information presented.
3. I thought setting up the system was easy.
4. I might need the help of someone technical to setup the system.
5. I liked all of the features offered.
6. I felt that the application struggled to express its message.
7. I feel like the barrier of entry for this application is low.
8. I found it tiring to use this system.
9. I knew what I was doing while using this application.
10. I'll need to learn a lot before I can effectively use this system.

The scale gives a score from 0-100 to help understand the effectiveness of the system. Anything above 68 is above average and means I did an ok job building the system, while getting above 80 means the system is practical and usable by clients. I had some friends and family fill out the system usability scale after showing them my project.

The system averaged just over 70 which I'm delighted about, although the score might be a little biased due to the audience that answered the questions. Even though the results may be a little biased this scale is only supposed to give a rough idea of the practicality of the system, and is not a means of determining exactly how effective the design may be.

7 Conclusion

After having built the entire system for Vehilytics, I have a much greater appreciation of the interactions between frontend and backend systems. It's all well and good building and entire API but it isn't until you start consuming it with some frontend client that you realise that endpoint A isn't quite what you need, or it would be nice to have endpoint B, or a user should be able to access this resource as well as a device. It was nice to be able to jump into the web service to extend endpoints as I needed but I can see that being a huge problem for larger systems.

One design aspect that I'm very pleased with is the fact that none of the data is coded into the Android application so if there is ever a need to update any information about sensor descriptions, or reminder recommendations, I just need to update the server database and the user will receive the updated information the next time they make a request to the service.

Implementing a suite of integration tests turned out to be a massive time saver while building the API as it alerted me to old working code beginning to break based on new features I was implementing. I could have easily implemented 2-3 new features while break 5-6 and not noticing for a few days/weeks, leading to hours of backtracking to discover the commit where the issue first arose.

7.1 Future Extensions

The Android application subsystem gathers all information to be displayed to the user from the web service, meaning it is easy to roll out updated information when available without needing to update the Android APK. Currently the only way to update this information is directly updating the database via a rails console application in the web service. A nice feature to implement would be an Android application that allows an administrator account to update information remotely. This admin application could also handle registering new devices with the system as they are constructed before sale.

7.2 Difficulties

One major roadblock that I ran into was how unreliable the third party OBD readers can be after having purchased and tested 3 devices with no success. If I was to begin this project again I would attempt to create my own OBD reader using a PIC microprocessor, the ELM327 protocol and C language so that I could directly control the interface between the vehicle and the Raspberry Pi.

Another problematic trait of the OBD readers is that they will constantly drain the battery of the vehicle so long as it is plugged in, meaning that the system must be plugged in during use and removed at all other times to prevent the battery from going flat. This problem could be solved by constructing a reader that only access the OBD port when required and would be accomplished as mentioned above using a PIC microprocessor.

8 Bibliography

Obdii.com (2017), *OBD-II - On-Board Diagnostic System.* [online]
Available at: http://www.obdii.com/ [Accessed 2017].

Carista. (2018). *Carista OBD2 App* **|** *Diagnose, customize and service your car.* [online]
Available at: https://caristaapp.com/ [Accessed 2017].

Thomas, N. (2018). *How To Use The System Usability Scale (SUS) To Evaluate The Usability Of Your Website - Usability Geek.* [online] Usability Geek. Available at:
https://usabilitygeek.com/how-to-use-the-system-usability-scale-sus-to-evaluate-the-usability-of-your-website/ [Accessed 2018].

Kabiru, A. (2017). *Build a RESTful JSON API With Rails 5 - Part One.* [online] Scotch.
Available at: https://scotch.io/tutorials/build-a-restful-json-api-with-rails-5-part-one [Accessed 2018].

Codemy.net. (2017). *Rails API: Generating an API Only Rails App - [001].* [online]
Available at: https://www.codemy.net/posts/rails-api-generating-an-api-only-rails-app-001 [Accessed 2018].

Devise. (2018). plataformatec/devise. [online]
Available at: https://github.com/plataformatec/devise [Accessed 2018].

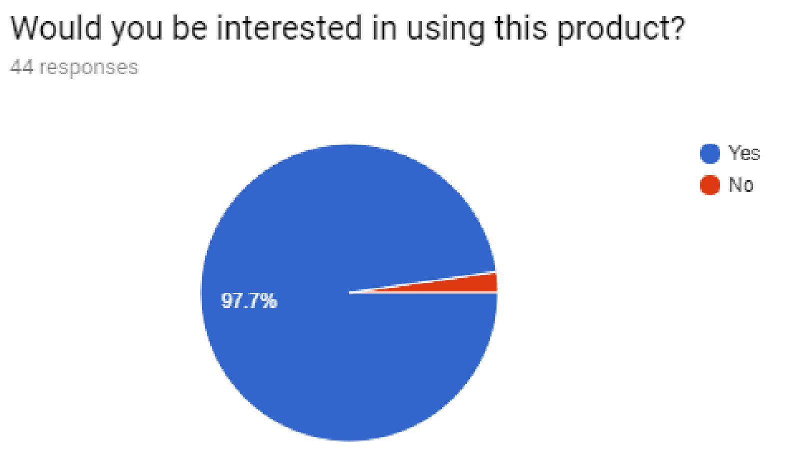
Square.github.io. (2018). *Retrofit*. [online]
Available at: http://square.github.io/retrofit/ [Accessed 2018].

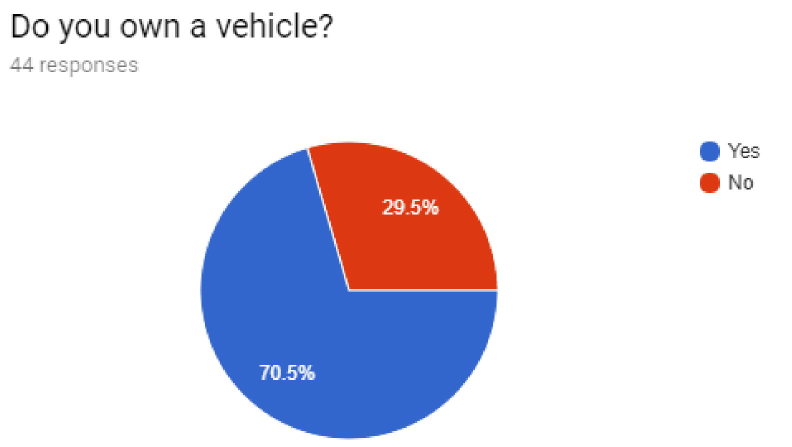
Site.mockito.org. (2018). *Mockito framework site*. [online]
Available at: http://site.mockito.org/ [Accessed 12 May 2018].

Rspec.info. (2018). *RSpec: Behaviour Driven Development for Ruby*. [online]
Available at: http://rspec.info/ [Accessed 12 May 2018].

9 Appendix

9.1 Survey results

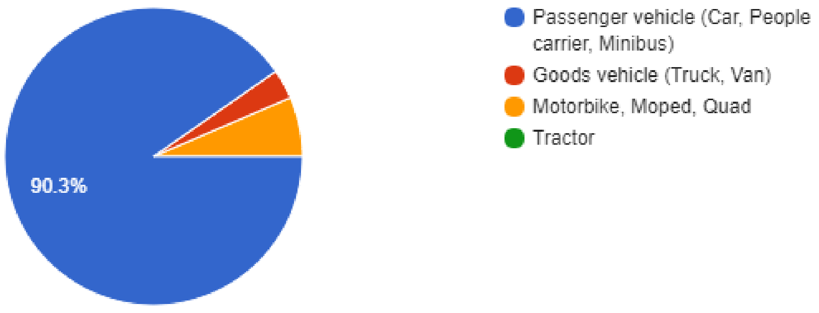




Vehicle Owners

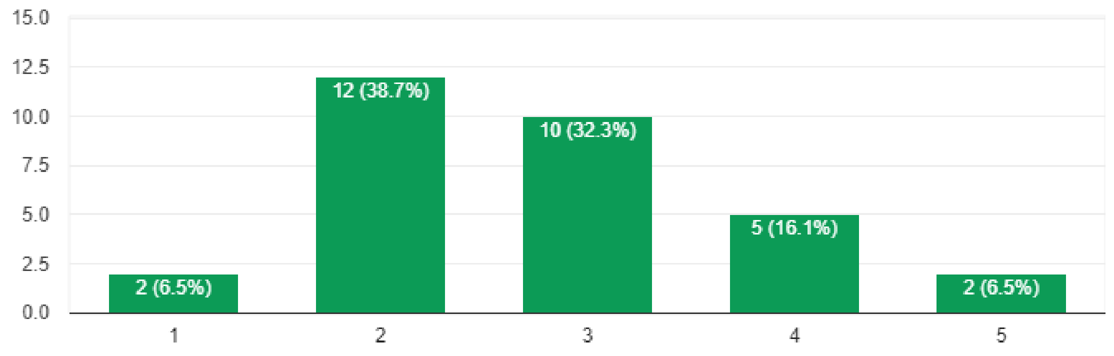
What type of vehicle do you own?

31 responses



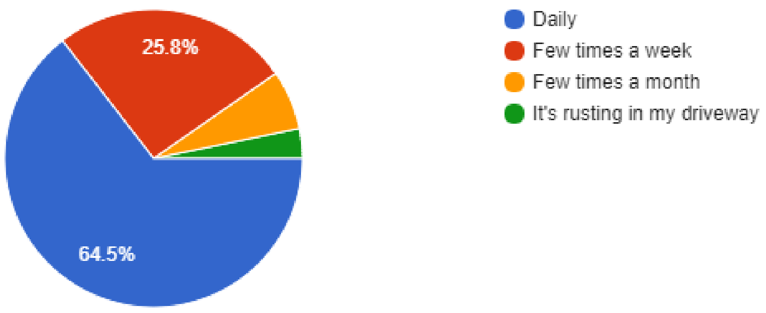
How familiar are you with vehicle maintenance?

31 responses



How often do you use your vehicle?

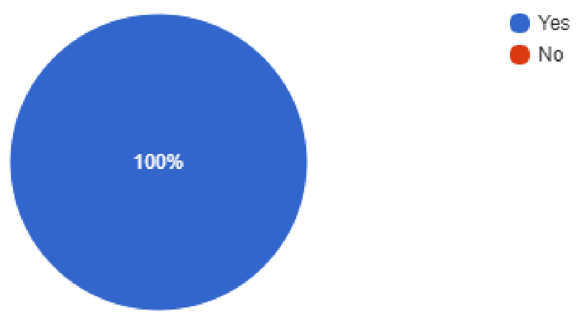
31 responses



Product questions

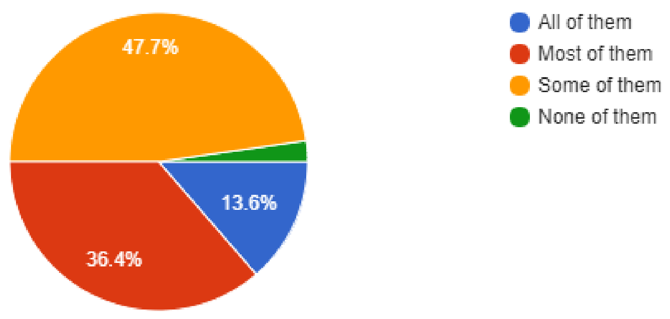
Would you like to receive warnings about potential maintenance needs?

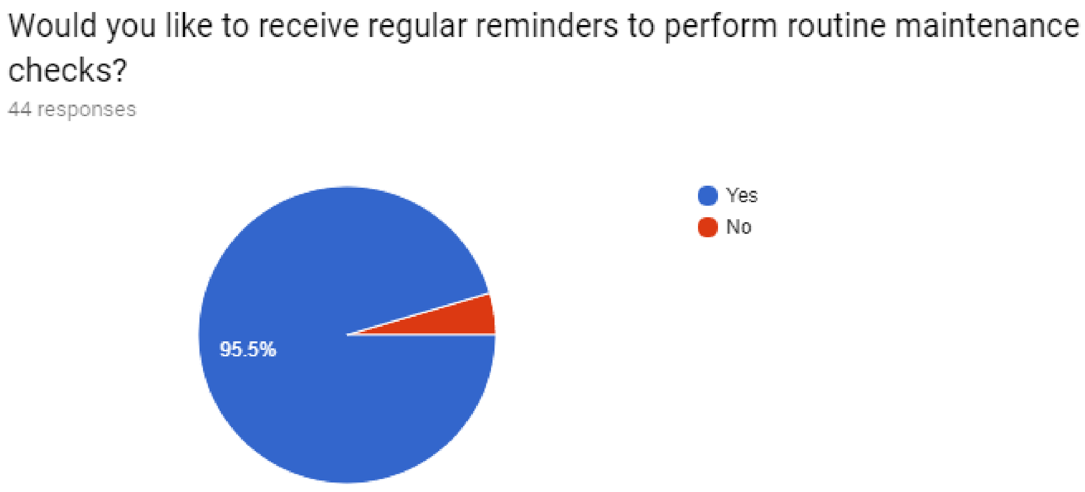
44 responses



How many of the symbols displayed inside a vehicle do you understand?

44 responses





Would you like to receive regular reminders to perform routine maintenance checks?

44 responses

The final question asked the responders would they be comfortable installing the diagnostic reader in their vehicles. Most respondents said they would have issue with it but I've included some notable answers below.

- Depends on how secure it is
- It would depend how secure the device would be once installed, where it would be installed and how complex installation is and whether I could even do it myself. But if it were not too difficult or likely to come loose then definitely.
- Depends how secure it is
- Depends on how complicated the device is
- No but I'd want to know who would have access to the data the device gathers and how far back it stores historic data. Also, depends on how difficult the device is to install and maintain.
- Only concern would be. Who would have access to it?

9.2 Project Proposal

Car Diagnostics Platform

Project Proposal - Draft

Dan Downey - x14728599
x14728599@student.ncirl.ie
BSc. in Computing
Internet of Things (IoT)

Objectives

My primary objective for this project is to build a car diagnostics platform that will allow an end user to view their vehicles diagnostic reports remotely. These reports should be easily readable by the end user, while being informative enough for a mechanic to be able to repair the vehicle in the event of a failure. The system should be extensible and allow additional modules to add information to the diagnostic reports. As a broad overview I can break down the project into 4 sections/milestones.

Data Sources:
This will primarily be the On-board Diagnostics (OBD) port on a car as it gives access to a lot of pre-installed sensors and data that is used by mechanics in diagnosing issues. It may also include additional sensors installed by the user. These data sources will be accessible via Bluetooth and will have no function other than to provide data.

Diagnostics Reader:
This will be a Raspberry Pi that will interface with the various data sources, generate a full diagnostics report, and will log these reports until they are requested. It will use Bluetooth to continuously poll the data sources in a round-robin fashion to keep the logs up-to-date. This device will sanitize the data as appropriate to maintain proper form and order in the reports. If the reader detects an emergency it should be able to send an alert to the web service.

RESTful Web Service:
This will be an API that can be accessed by the end user application and the Raspberry Pi. It will act mainly as a communication point between the user and the vehicle diagnostics reader. It will also host credentials linking an end user to a diagnostics reader, and user settings that will allow the user to omit certain data from the reports and add additional modules to the reports.

End User Application:
This will be an Android application that will present an end user with a diagnostics report, allow them to forward this report to additional contacts, allow them to change settings to their preference, and alert them if there is an emergency.

Each of these systems should be completely unaware of how the other sections operate so that each part can be properly tested, replaced, or mocked without compromising the system as a whole. Making the diagnostics reader vehicle agnostic would be a major goal. The OBD-II standard was introduced to simplify interfacing with vehicle diagnostics, but depending on the manufacturer, it implements one of 5 different signal protocols which will need to be considered when creating the diagnostics reader.

Background

My idea began as a sensor array that could be placed within a car to detect when a baby was left unattended and notify parents, guardians, and finally emergency services as necessary to ensure the survival of the child. The inspiration for this idea came from reading an article which described the surprisingly common phenomenon of parents simply forgetting they had the child with them in the car on their way to/from work and discovering hours later that the child had perished in the heat or cold.

Developing a system to detect the presence of a baby that was left alone would end up being an unwieldy amount of sensor configuration and processing for a singular purpose to be served. However, a car already has a host of sensors installed, reading, and reporting at most times to monitor things like RPM, or external temperature, or anti-theft devices. So, the idea came about to publish these diagnostics in some form of readable report for the vehicle owner to be viewed remotely and allow the system to be generic and extensible so additional modules can be added at a later stage, including the baby monitoring device.

The beauty of this general, extensible platform is that the limit of what can be reported and viewed is based on what the user and any third-party manufacturers can design, build and install following what I can determine to be allowable report formats.

The type of data reported by these systems may also elicit different responses and alerts to be issued. For example, if the user is away from their vehicle and suddenly the RPM shoots up, we can be relatively certain that the car has been stolen and can quickly alert the authorities. Building on that, we could install a GPS module so we can provide coordinates of the car to help in recovering the vehicle in the event of theft.

The reports from the On-Board Diagnostics itself could save huge amounts of time and money by allowing the user to identify faults that they can fix themselves and provide their mechanic with the report so they don’t need to run and can’t bill for their own diagnostics services.

On higher end cars, where the user can access features like air conditioning remotely, this means that an internally installed temperature sensor can be used to view the environment in the car remotely and enable the air-con before reaching the vehicle so that it is comfortable for the driver to hop straight in and drive away.

Currently these kinds of diagnostics systems are only available on high end cars, and for specific manufacturers. Building this system to work independently of manufacturer and OBD-II protocol would be a huge step toward automating vehicle maintenance, vehicle safety, anti-theft, calculating insurance costs, and settling traffic accident disputes.

Widening the scope of analysis for the project idea, we can see major use for this technology platform in data mining. If we can continuously log vast amounts of data about how a car is being used, then we can examine the skill of the driver, the efficiency of the car, traffic patterns and vehicle crime.

This data could all feed into adjusting insurance, determining which roads need to be examined and possibly reworked, and even suggesting new driving routes to the user. Through the mobile application we could begin to target advertisements to the user based on maintenance needs of the car, or which stores they frequent based on GPS.

Technical Approach

A key component to this project is the actual diagnostics data. This means that acquiring an OBD-II reader and analysing the data I can retrieve in terms of both its structure and usefulness should be my first milestone. This will allow me to determine a suitable model for passing reports from the diagnostics reader, up to the web service, and finally on to the end user. My hope is that it will be simple enough that I can just represent the report in JSON format.

Once I have a good model to represent my diagnostic reports, then I can properly design my API contracts using API Blueprint. This will allow me to configure the public endpoints at a high level without regard for how the system internals will handle manipulation of the data being passed out or taken in through these endpoints. This will allow the different sections of the system to operate regardless of who is interacting with it, which will make the entire system easily testable and extensible.

The Diagnostics reader will be built using a Raspberry Pi coded with Python. Python is a nice lightweight language that should place no unnecessary strain on the device and gives access to libraries, such as requests for making network requests, and json for parsing to and from json files. It will use Bluetooth to communicate with the data sources and act as the master device in the pairing. This will also allow me to poll a number of Bluetooth slave devices for data, including the OBD-II port or any other modules I can create which also use Bluetooth.

The Web service will be a simple RESTful API that handles storage and management for diagnostic reports, user credentials, and user preferences. The Web service itself will be created with Ruby on Rails, a SQL database for storage, and it will all be hosted on Heroku.

The end user mobile application will be an Android application written using Kotlin. It will allow the user to register their vehicle to a reader, view their vehicle diagnostics remotely, change their preferences to change the format of the reports, save reports to their device or Google Drive, and forward these reports to their contacts through email or SMS. The application will use libraries, such as Retrofit, Gson, and RxJava to handle network requests, json parsing, and updating UI reactively.

Appropriate testing frameworks will be used to ensure each subsystem continues to function correctly throughout the build process. This means JUnit and Mockito for the Android application, the built in Ruby testing framework or Cucumber for the web service, and PyUnit on the Diagnostics Reader.

I am also toying with the idea of building the entire project in Kotlin using Android Things on the Raspberry Pi, Kara Web Framework for the Web Service and JetBrains Exposed for the SQL storage. This may not be feasible as the Exposed library is only a lightweight prototype and I'm unsure if Android Things will allow me to access the Bluetooth protocol directly.

Special Resources Required

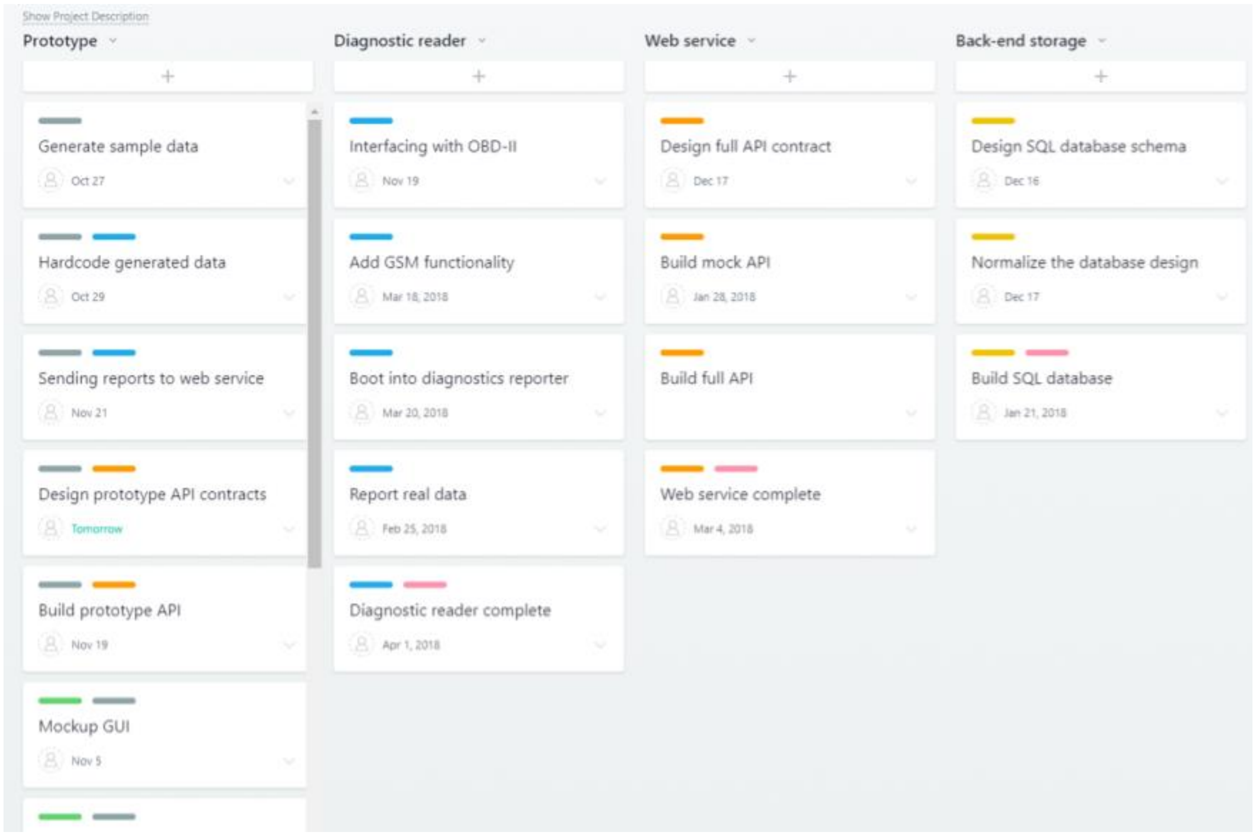
OBD-II Bluetooth adapter
This adapter will allow me to access the OBD-II port wirelessly via Bluetooth. Hopefully it's possible to access this adapter directly from the Bluetooth on a Raspberry Pi, although reading in the data through USB should be fine.

Device with an OBD-II port
I'll need a source of data and this will likely come from an OBD-II port on a car that I can access. Some alternatives to this could be a vehicle ECU pulled from a car or a secondary Raspberry Pi that just emulates and emits data via Bluetooth.

Raspberry Pi Mobile Data HAT
The Raspberry Pi will need some way to communicate with the Web Server and most likely it will not have access to Wi-Fi. A mobile data HAT (Hardware Attached on Top) like PiAnywhere GSM will allow the Pi to access GSM networks.

Project Plan

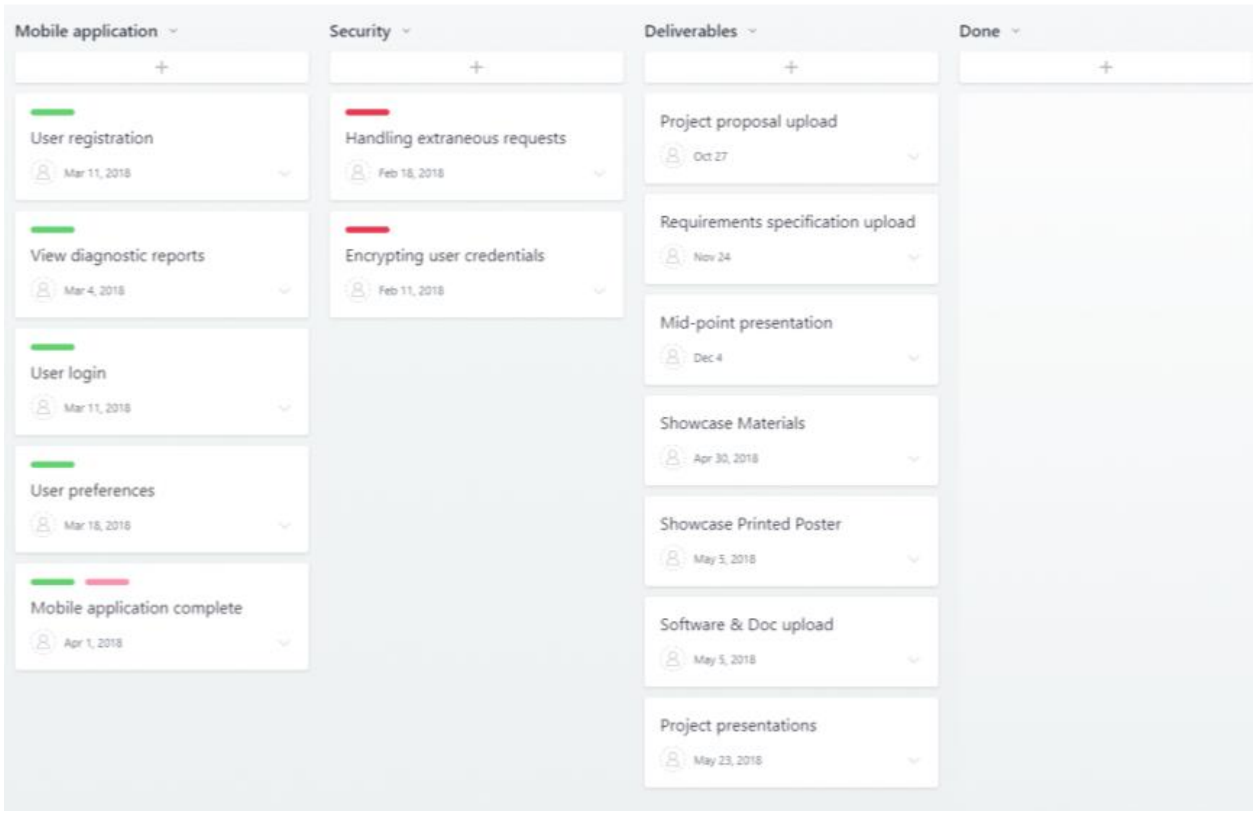
I am using Asana to organise my project tasks and deadlines. Below I have added pictures of my project board, organized into columns for the major milestones. These milestones include the prototype, the diagnostic reader, the web service, the back-end storage, the mobile application, and the finished product. I have also included a Gantt chart for my projected timelines. This will change as the project progresses as I suspect some tasks may take longer than



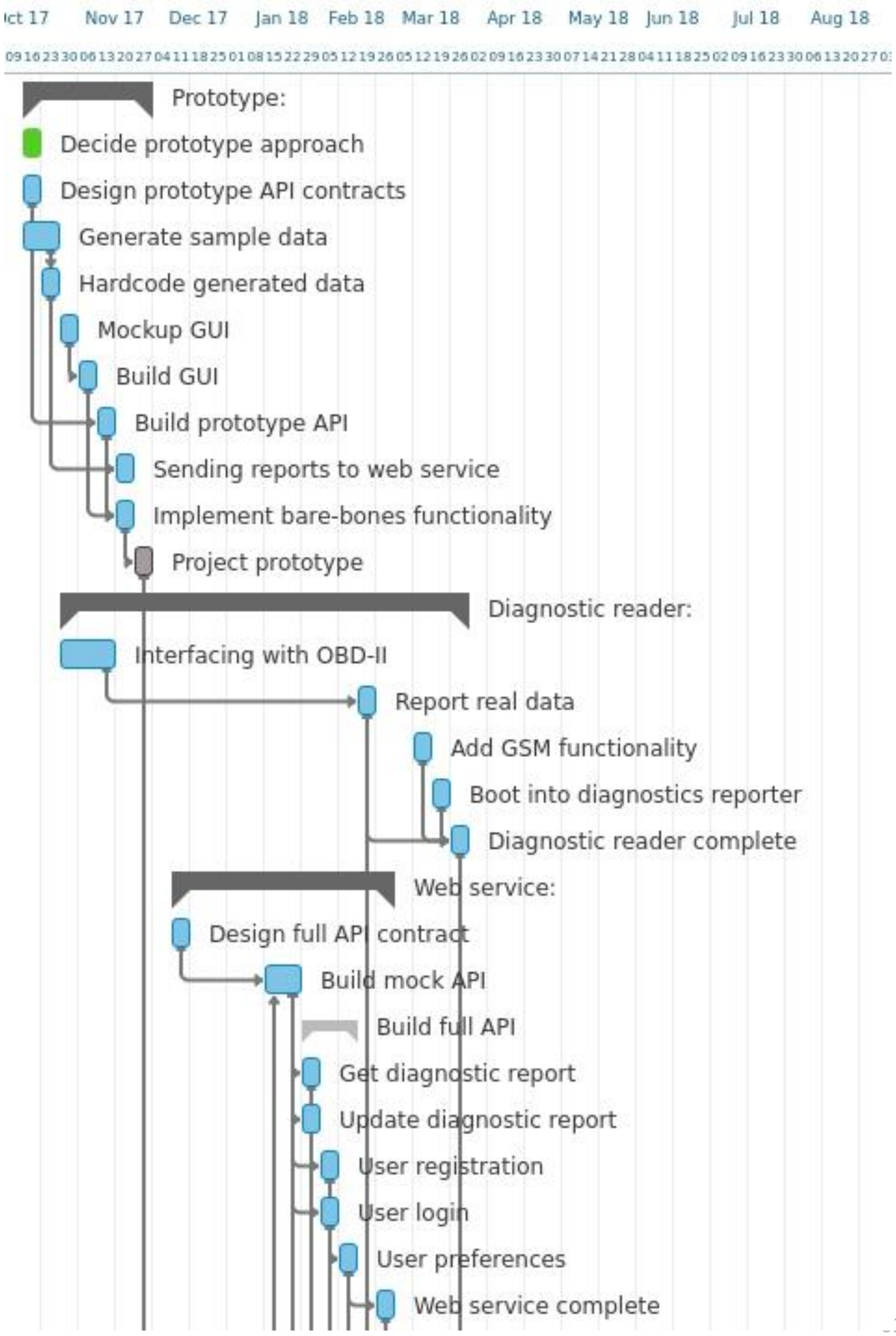
Show Project Description

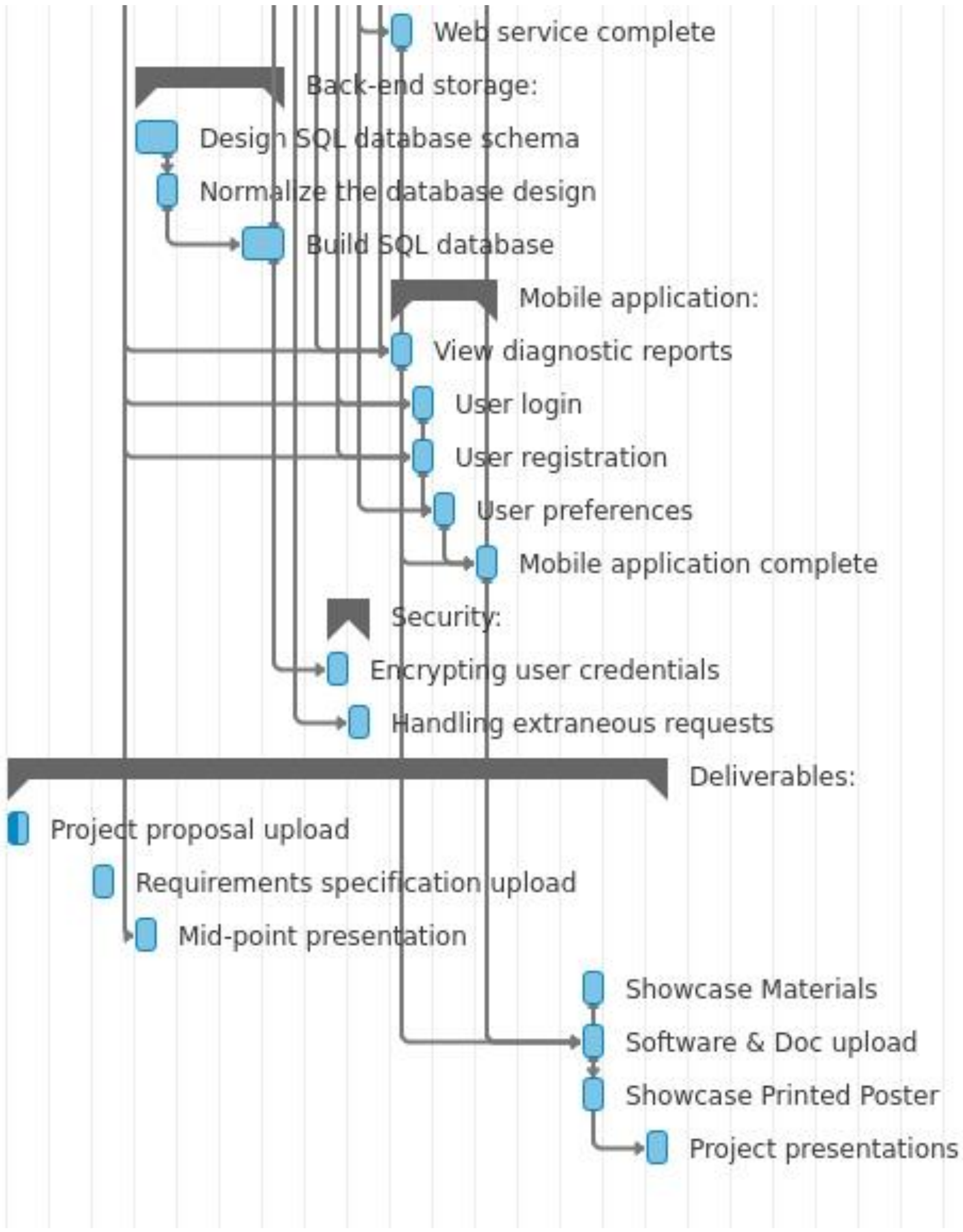
Prototype	Diagnostic reader	Web service	Back-end storage
Generate sample data Oct 27	Interfacing with OBD-II Nov 19	Design full API contract Dec 17	Design SQL database schema Dec 16
Hardcode generated data Oct 29	Add GSM functionality Mar 18, 2018	Build mock API Jan 28, 2018	Normalize the database design Dec 17
Sending reports to web service Nov 21	Boot into diagnostics reporter Mar 20, 2018	Build full API	Build SQL database Jan 21, 2018
Design prototype API contracts Tomorrow	Report real data Feb 25, 2018	Web service complete Mar 4, 2018	
Build prototype API Nov 19	Diagnostic reader complete Apr 1, 2018		
Mockup GUI Nov 5			

expected, while others may prove to be less complex.

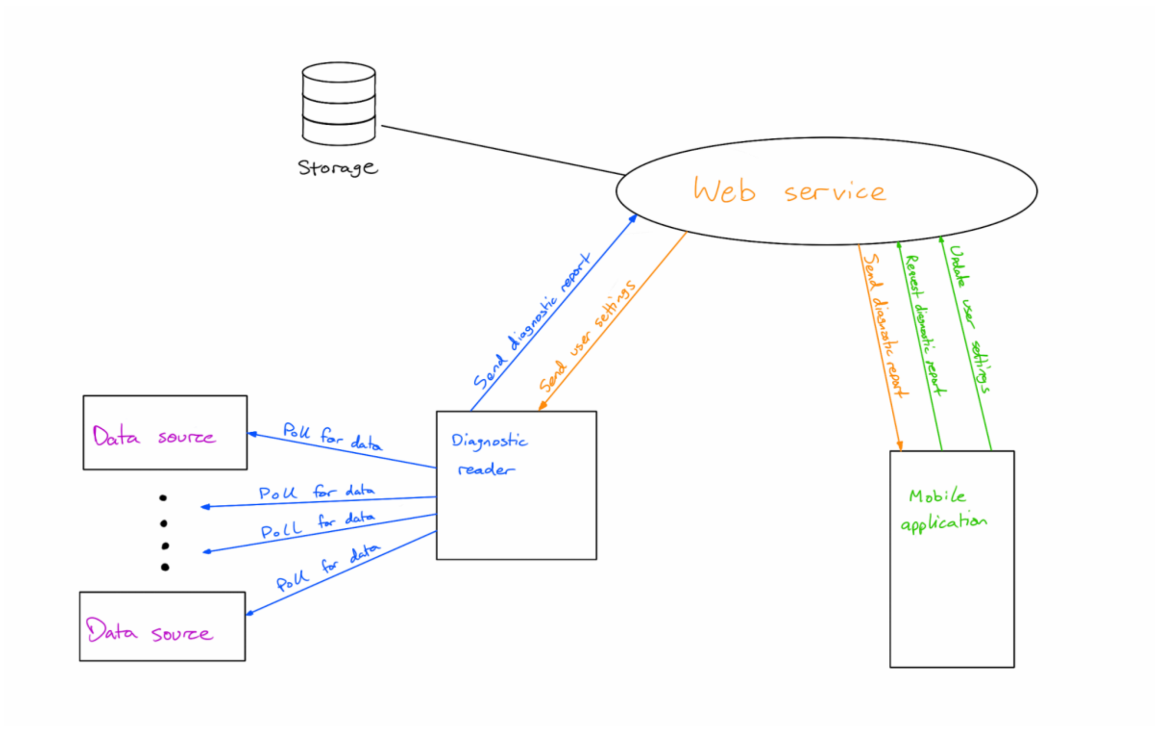


Mobile application	Security	Deliverables	Done
User registration Mar 11, 2018	Handling extraneous requests Feb 18, 2018	Project proposal upload Oct 27	
View diagnostic reports Mar 4, 2018	Encrypting user credentials Feb 11, 2018	Requirements specification upload Nov 24	
User login Mar 11, 2018		Mid-point presentation Dec 4	
User preferences Mar 18, 2018		Showcase Materials Apr 30, 2018	
Mobile application complete Apr 1, 2018		Showcase Printed Poster May 5, 2018	
		Software & Doc upload May 5, 2018	
		Project presentations May 23, 2018	


Oct 17
Nov 17
Dec 17
Jan 18
Feb 18
Mar 18
Apr 18
May 18
Jun 18
Jul 18
Aug 18
Prototype:
Decide prototype approach
Design prototype API contracts
Generate sample data
Hardcode generated data
Mockup GUI
Build GUI
Build prototype API
Sending reports to web service
Implement bare-bones functionality
Project prototype
Diagnostic reader:
Interfacing with OBD-II
Report real data
Add GSM functionality
Boot into diagnostics reporter
Diagnostic reader complete
Web service:
Design full API contract
Build mock API
Build full API
Get diagnostic report
Update diagnostic report
User registration
User login
User preferences
Web service complete


Web service complete
Back-end storage:
Design SQL database schema
Normalize the database design
Build SQL database
Mobile application:
View diagnostic reports
User login
User registration
User preferences
Mobile application complete
Security:
Encrypting user credentials
Handling extraneous requests
Deliverables:
Project proposal upload
Requirements specification upload
Mid-point presentation
Showcase Materials
Software & Doc upload
Showcase Printed Poster
Project presentations

Technical Details



Diagnostic reader

I'll be using *BlueZ* for accessing the Bluetooth protocol stack so I can poll Bluetooth devices for data. Libraries like *request* and *json* will allow me to easily make network requests and parse json. After some research into other people attempting to access their own OBD-II ports with Raspberry Pis I found *pyOBD*, a python library for interfacing with the OBD-II port in a car. Using this, I can place less attention on attempting to interface with the hardware, and focus more strongly on utilising the data.

Web service

I don't currently have any knowledge of Ruby or Rails, but I would like to use these technologies for the Web Service layer of this project as it is the popular web framework in industry at the moment. For storage I will be using a MySQL database so I can design a formal schema for the report data and user credentials.

Mobile application

Networking in the Android application will be performed using Retrofit and OkHttp. Retrofit is an HTTP client that allows me to map my API within the application using interfaces, and OkHttp handles making HTTP requests. I will be using GSON to parse JSON objects and map them onto Kotlin data classes. I would like to use RxJava for reacting to events like server responses but it may be unnecessary.

Version control, quality assurance & testing

I will be using Git and GitHub for saving my project so that I can easily work across multiple machines, and handle code reversion if necessary. This will help for discovering the cause of breaking bugs that may arise during development. I will be using the various testing frameworks available for each implementation language to continuously write appropriate tests. This will mean unit tests for individual blocks of logic, integration tests for hardware/software integration points, and usability testing for the end-user application, to name a few.

Evaluation

During development of my project I will be continuously writing unit tests to be certain that my individual blocks of logic remain unbroken as I progress. This will allow me to discover and fix bugs as they arise instead of surfacing many modifications later and causing me to spend time debugging.

Since my project involves a hardware component and multiple software components, it will be necessary for me to write integration tests to ensure that these components all fit together and operate end to end as per my requirements.

Once the GUI for my mobile application is created I will begin usability testing by asking people to test the application and see where the navigation may be confusing or difficult. Beginning this testing before adding the core functionality will be crucial as it will allow me to tailor the design to better suit the needs of the user and remove anything which is seen as clutter.

Performance testing will be important as the diagnostic reader will be required to report consistently over long periods of time and the user will want to see updated reports as soon as possible. Leaving the diagnostic reader to monitor and report data over a few days and graphing the data should allow me to see any inconsistencies. For testing the response time of updated reports, I can push up modified, and easily identifiable data to see how long of a delay there will be before the report can be requested by the user.

9.3 Monthly Journals

9.3.1 September

After 7 months of working in Zendesk, I was sad to be leaving. I enjoyed the structure and the challenging work. I also enjoyed not having business modules to worry about. That being said I was excited to get back into the college and dive into some new projects with all of the experience and knowledge I gained throughout the internship.

This year I'm working as a Lab Assistant in the college and it's really been a great reminder of how far along my ability has come. Being able to quickly review code written by students and give them guidance or even just to answer unusual questions on specifics of whatever language they are learning, like the choice tag in XSD.

The best sign for me so far that this year is going to be interesting is that we were immediately handed a practical CA, coding a chess engine. I grow bored having nothing but reports and research papers to write, but actually being challenged with some code from the get-go was incredible for grabbing my attention and motivating me to get the work done early.

At the suggestion of one of my coworkers in Zendesk I began reading more tech blogs which led me to start collecting articles and libraries that I felt might be useful in deciding upon a project idea. I was delighted that I decided to do this as it opened up so many brainstorming routes, some of which were fruitful while others fizzled out. So luckily, I was walking into this year with my idea somewhat worked out in my head. I didn't spend too much time locking down details or fully committing to the idea in case I needed to rework it or the idea was rejected outright. This turned out to be a good call.

I was due to make my project pitch at 11am but I headed in for around about 9am so I could get some work done in the free room. As I reached the top of the stairs I was greeted by Eamon who asked if I was interested in making my pitch ahead of schedule. Not one to shy away from a presentation I jumped at the opportunity.

I strolled into the room with my coffee in one hand, earphones in the other, hat falling out of my pocket and my extremely heavy laptop bag weighing me down. It was a bit of a struggle setting it all down on the table but I think I stuck the landing. I pitched my idea for a sensor array that could be fitted into a car for detecting whether a baby had been left unattended and was delighted to find my idea being challenged.

At the suggestion of the lecturers, the project shifted more towards being a car diagnostics platform using a Raspberry Pi to interface with the car OBD-2 port. The Pi would report the data to a web service and the web service would push this data down to a client mobile application, allowing a user to remotely monitor their car’s diagnostics. My view of this project is that the Pi should report the car’s own diagnostics along with whatever data it gathers itself from add-on “modules”. These “modules” will be sensors that provide some valuable data to the user or platform itself (e.g. GPS for location discovery, accelerometer for crash detection).

9.3.2 October

I dove straight into brainstorming for my project at the beginning of October. The OBD technology is new to me so I began researching as soon as I could. I gained some decent insight into the technology at a high level and an idea of how I will make use of it.

I used this information to begin sketching out high level architecture diagrams for the entire system to help visualise the flow of the system and allow me to better choose the appropriate technologies for building the other sections. All of this work went a long way toward writing my project proposal, allowing me to be better able to defend my potential design choices.

An interesting design choice that came about was to build the entire project end to end in Kotlin. From researching various frameworks for the subsystems in the project, I noticed people had created Kotlin frameworks that fit everything I needed. There's arguments to be made for and against this route.

If I use different languages and technologies throughout then I can demonstrate flexibility and ability to pick up these technologies. But if I use only Kotlin, then I can show extreme competence in this new, and quite popular language. The main issue I see with this approach is difficulty in accessing the Bluetooth protocol stack.

At the beginning of October, I was offered a role in computing support tutoring people struggling with various subjects. I jumped at the opportunity because it forces me to reevaluate my own aptitude with these subjects and learn them to the degree that I can teach it to someone else.

This has given me a great help in thinking about the design of my own projects and in developing simpler, less gimmicky/hacky solutions to problems I have been working on. A great example of this was relearning database design and that informing my decisions in how to store and retrieve data for my final project.

I will admit however that I haven't done enough groundwork for the requirement specification at this point and I will have to make up for that in the coming weeks. Hopefully I have been passively picking up on enough information over the last while through tutoring and reading articles that I can quickly and effectively do my requirements gathering and use case modelling.

I have been doing some thinking about how I will showcase my project in May and what kind of form my prototype will take in December. For the prototype my plan is to just pass dummy data from a raspberry pi to some kind of web service and read that on a simple mobile app. The web service might be something as basic as dweet.io so I don't need to build my own endpoints.

The showcase however will be a different game entirely as my project relies on using the OBD port on a car. It would be awfully strange if I was to bring a car up into the Kelly Theatre on the day so instead I have acquired an ECU stripped out of a car and my plan is to power it up and wire it to an OBD port so I can interface using a Bluetooth adapter.

9.3.3 November

Well these last 3 months definitely went by like a flash. I can’t believe it’s been a month since my last journal. I’ve been extremely busy between project work, assignments for other modules, lab assisting, and acting as a support tutor. It’s tiring but I’m enjoying it. Most of my project work this month has been toward the requirements specification and technical report. I performed some requirements elicitation techniques to gather user requirements for the system. The 2 techniques I employed were a survey and a brainstorming session.

The brainstorming session provided me with some potential feature requests for the mobile application such as, user views that can alter the nature of the data being displayed based on the technical competence of the user, and a mechanism for learning the meaning of symbols displayed around the vehicle. The survey highlighted some valuable information about the potential users of the system including, how often they drive, how familiar they are with vehicle maintenance, and if they have an issue with installing the diagnostic reader in their car.

The user requirements I gathered helped me shape my initial prototype mockups for the android application. Rather than designing the web service around the diagnostics being gathered, I have decided to build it around what information the user would like to see.

I started building my prototype quite late considering the scope of my project. I had designed mockups for the mobile application GUI as part of the requirements specification so I set aside a Saturday to build the entire application with no function and it turned out quite well. I was delighted to be able to get graphs working so I can graph out diagnostic trends and highlight potential concerns.

To complement the mobile application, I also spent a Sunday building a Ruby on Rails API that simply accepts diagnostic reports with dummy data that I can then retrieve from the mobile application. This was a tough one because I have never written Ruby before but I'm happy that I got it working in the end.

As of writing this journal I'm just putting the finishing touches on the technical report and have yet to pull the API data into the mobile application but I have a few days before they are due so I might set aside some time to add it to my prototype demonstration.

9.3.4 December

There was a healthy amount of project work to be done throughout December between the Intro AI chess assignment, the IoT research paper and Mobile Application Development project. The chess assignment was tricky but manageable once the codebase was cleaned up. I enjoyed writing the research paper on ubiquitous computing and internet of things because I got to have a massive rant and Snap Inc. and Spectacles and how IoT is basically just a massive gimmick at the moment.

I was delighted with the outcome of the Mobile Application Development project as it led to me releasing my first Android app to the Play Store. I managed to pick up a few Android tricks while building it so hopefully they'll come in handy when I'm building the final Android app for the software project. I'll be trying to squeeze Android in wherever I can to cook up some new tricks.

I don't see myself getting a whole lot of software project done this side of Christmas after lab assisting, tutoring, and the hail of assignments. I'll be focusing on the exams so that I'm heading into semester 2 with a decent foot forward not having to worry about results.

9.3.5 January

January was spent preparing for my final exams and getting setup for semester 2. As a result, there was not a lot of work done toward the final project. I'm happy enough to have left it aside for the time being however as it meant that the exams were less stressful and it was easier to get down to working through class assignments with delay.

Aside from Strategic Management, I'm happy enough with how the exams went. The business modules just do not click with me so it's a chore to sit down and learn the notes, and even then, the material just doesn't stick. I was a little worried about Intro to

AI but the notes were helpful and the material was interesting to learn. In the end I'm happy with how I answered the questions. The IoT exam was enjoyable, as exams go. I found the course content fascinating and easy to read about. The exam itself was interesting as it was mostly just brainstorming IoT systems and I was happy with how I answered the paper.

Overall, I was delighted at my results and they put me in a great mood heading into semester 2, especially since the IoT stream doesn't have any formal exams. Hopefully we can get the briefs for all our assignments early in the semester so there's not a rush in the last few weeks to get 3 projects done alongside the final project.

9.3.6 February

February was an interesting month. I had some interviews for a really good company that was spread across 4 rounds, one of which was 9 hours in length. The stress of going through this process was probably not worth it as it made it very difficult to focus on assignments and project work. Even though I didn't get the job, the relief when the process was finished was unbelievable and led on to me getting a lot of assignments done.

As a result, there was no direct progress made on my final project, but I was forced to quickly learn Rails for the last interview which I need to learn for both a class project and my final project. So even though direct progress wasn't made, I have landed in a position where I am more capable of completing those 2 systems in a tighter time frame.

Throughout all of this though, I have been constantly asking for information regarding module projects and very little was shared with us. My fear of ending up with 4 big projects all at once is looking very likely at this point. I'm not so much worried about the Cloud Application Development or IoT Application Development projects as I am about the Data Mining project. The Data Mining project involves a hefty report and feels like we are missing prerequisite knowledge for the class so that project will be a struggle.

I have been doing some investigating into the OBD portion of my project and have discovered some existing tools that provide some ability to parse the raw data from the vehicle that will be of great use to me. The problem is that these tools all have a GUI, and don't allow the data to be sent anywhere. Luckily these tools are open source so I will be forking those projects and creating a headless version that will allow the data to be pushed up to my API. I might make a pull request on the existing tools but they haven't seen any updates in years so the headless version will most likely just live on my own GitHub.

9.3.7 March

My work on the headless fork of the existing pyOBD library has been going well. I have it *attempting* to connect to the OBD port and make requests. There is an issue with it in that it will write the request out to the serial port, wait, and then consume the response but the response is just the initial request again because the OBD is never actually seeing the request. It took some weird maths to discover this issue because I was just receiving strange values back instead of errors.

For example, I would get back -29 degrees for the ambient air temperature. The formula for getting the air temperature from the response code is given by:

$$\begin{aligned} \text{temp} &= \text{response} - 40 \\ -29 &= \text{response} - 40 \\ \text{response} &= 40 - 29 \\ \text{response} &= 11_{10} \\ \text{response} &= B_{16} \end{aligned}$$

At this point you might be wondering why I labelled the last lines with their bases. I went diving into the existing codebase and discovered that the code was raising the string “Bogus Code” as an exception which has a weird side effect since the string class doesn’t extend BaseException. It turns out that the code was trying to parse the string as hex, which evaluates the first character as 11 and gives -29 for ambient air temperature.

A worrying detail I discovered about the OBD readers is that they drain the car battery as long as they are plugged in, even if the car is off. I realised this when my Dad’s car battery died while we were testing the system. This really hurts my idea of this device being left plugged in constantly and only being in use while the car is on. If I was building the microcontroller to interface with the OBD port myself I would attempt to address this problem but I’ll just have to say that it must be plugged in before you start driving and removed when the car is off.

I was hoping that I would get a lot of work done during reading week but it ended up being a miserable time as I lost a close friend very suddenly. I wasn’t able to focus and get some deep work done for the 2 weeks so I have a lot of work to catch up on coming into April.

We have a Data Mining project due 2 weeks into April and I’m honestly terrified by it. The concepts are tough to grasp and I can’t immediately see the practical use of the algorithms. After speaking to someone in the data analytics stream, he was surprised that we have a project of this scope to do solo while they did a similar project in groups of 4. I’m amazed that we’ve gotten a more difficult data analytics project than the people studying data analytics.

9.3.8 April

April was possibly the most stressful month since I started college. 3 sizable projects all due a week apart from each other. The tight deadlines were only made worse by the fact that 2 of the projects were not enjoyable at all. Data Mining was an overly difficult module with a poorly executed project. The Cloud Application Dev project should have been to build a Rails API with a consuming client of our choice instead of a full Rails application.

The IoT Application Development project was a lot of fun though. We had a lot of freedom to build any system we wanted and I ended up having a shopping spree on Adafruit buying parts that I didn't even use in the end, but I have them now for future projects!

There's not a whole lot to write about for this month as it was mostly busy work just getting the projects done with no time to tinker with my final project. I did manage to figure out a few cool Android tricks that I plan on using in the final project. I figured out how to control buttons nested inside of a RecyclerView, how to trick a relative layout into behaving like a button, and how to fully customise button animations. I plan on pulling out all the stops for the Android portion of my final project.

The Cloud Application Dev project did give me a solid footing in understanding the Rails framework ahead of building the API for my final project, so I'm sure I'll be thankful for having to build it in a weeks' time.

One area I'm a little concerned about moving forward is testing. I haven't had to do any of it outside of my internship where there was a whole collection of tests already built to use as a reference. This time it's all from scratch, for multiple systems, all in different languages, and performing different types of tests.

I'm happy enough that I'm in a position now that I have a decent enough level of knowledge across a number of domains to make a decent dent in the project without much issue. Let's just hope that I don't run out of steam too quickly and end up stuck on one section for more than a few days.

9.4 Code Guide

Vehilytics Code Guide
Dan Downey - x14728599
May 10th, 2018

Contents

Introduction

This document will go through how to run/use each of the subsystems that makes Vehilytics. Before we look into each of the systems, I'll provide some credentials to dummy accounts that exist in the system for examination. There are 2 types of clients that can login to the system:

- User
- Device

A user is a client consuming resources via the Android application, while a device is a diagnostic reader that publishes data to the web service.

User Credentials

	email	password	device	description
0	dan@example.com	password	TEST_0	Has 2 warnings
1	paul@example.com	password	TEST_1	Has a warning and an error
2	glenn@example.com	password	TEST_2	Has a warning
3	aaron@example.com	password	TEST_3	New account, no problems

Device Credentials

	email	device_name	password	In use
0	test_device_0@vehilytics.com	TEST_0	device_pass	Yes
1	test_device_1@vehilytics.com	TEST_1	device_pass	Yes
2	test_device_2@vehilytics.com	TEST_2	device_pass	Yes
3	test_device_3@vehilytics.com	TEST_3	device_pass	Yes
4	test_device_4@vehilytics.com	TEST_4	device_pass	No
5	test_device_5@vehilytics.com	TEST_5	device_pass	No

Subsystems

The 3 subsystems that make up Vehilytics are:

- The diagnostic reader
- The web service
- The Android application

Diagnostic Reader

The diagnostic reader is written purely in Python and has a few different entry points due to it being a forked library. The library was originally written with an extensive GUI but we are interested in the obd_headless.py entry point. I have modified the system to allow it to run without a UI so that it can be placed inside the users' vehicle and allowed to operate without interaction. The code should be placed on a Raspberry Pi with an active internet connection. The system currently has 2 modes:

- Headless Reporter
- Dummy Reporter

Headless Reporter mode means that the system will attempt to connect to an OBD reader, gather data from the vehicle ECU, and interact with the web service. For this mode to function correctly, you must connect an OBD reader to the OBD port in a vehicle and turn on the diagnostic reader. The device will login to the web service, request user preferences and begin pulling data from the vehicle ECU before reporting it to the web service.

The optional Dummy Reporter mode is a way of running the system independently of a vehicle in the event that there is no vehicle close by or there are no working OBD readers. This mode was built out of necessity as none of the 3 OBD readers I purchased would actually work. The Dummy Reporter functions similarly to the Headless Reporter but differs in that it generates fake data to publish to the web service. This mode does not need to be run on a Raspberry Pi and can run on any machine with a Python interpreter.

For either mode to function you must populate the file headless_device_config with some configuration data. The first line is the API root URL, the second is the device email, and the third is the device password. You can optionally place the word dummy on the fourth line to enable Dummy Headless mode. Below is a sample of a configuration setup for using device TEST_0 in dummy mode.

```
https://vehilytics-api.herokuapp.com/
device_test_0@vehilytics.com
device_pass
dummy
```

Once the configuration has been setup, the system can be run using the command `python obd_headless.py` with an optional `--debug` to enable logging. For demonstration purposes the system will publish every 10 seconds, but in a real system this timing would be 5/10/15 minutes to reduce the computation and communication done by the low power device.

Since this system is based on a forked code base I should clarify that the files written by me in this system include:

- dummy_headless.py
- headless_device_config
- headless_reporter.py
- obd_data_generator.py
- obd_headless.py
- obd_utils.py
- Everything in /tests
- Minor modifications in other files such as obd_io.py to remove hard dependencies on UI libraries

Testing

There is a set of unit tests written for my portion of the code in this system located in the tests directory. These tests can be run using the command `python -m unittest tests/test_runner.py`.

Web Service

The web service is a Ruby on Rails API meaning that it has no views, it only accepts and produces JSON. It handles authentication for 2 client types as mentioned at the beginning of the document, which restricts access to certain resources as appropriate.

There are a number of controllers for handling each of the resources and they can be found in `app/controllers/v1`.

Device Registrations Controller

This controller is overriding the Devise gem registration controller so that I can fine tune the creation of new devices for the system.

Device Sessions Controller

Handles authenticating a device login by providing a unique authentication token when a login request is valid, a means of resetting the token when logging out, and a method for validating a token.

Devices Controller

The devices controller allows a user to get information about their device, to attach a new device to their account, or to detach their current device from their account.

Readings Controller

The readings controller allows a user to view diagnostic readings published by their device and can be filtered by sensor.

Reports Controller

The reports controller allows a device to publish diagnostic readings to the web service in the form of a report which consists of a time stamp, the device name, and a list of values for each sensor requested by the user.

Sensors Controller

The sensors controller allows access to a list of sensors and the view varies based on the client attempting access. A device will see the list of sensors requested by its user, a user will see the list of sensors that have been reported by their device, and unauthenticated clients will see a list of sensors supported by the system.

User Registrations Controller

This controller is overriding the Devise gem registration controller so that I can fine tune the creation of new users for the system.

User Sessions Controller

Handles authenticating a user login by providing a unique authentication token when a login request is valid, a means of resetting the token when logging out, and a method for validating a token.

Testing

All of the tests for the web service can be located in ./spec. There are 2 types of tests and a number of helpers within this directory. The folder factories contains factory methods for creating models to populate the test database as the tests run. The models directory contains tests for each of my ActiveRecord models, and tests them on their relationships, and validators. The requests directory contains the bulk of the tests in the form of request specifications which test the functionality of the endpoints.

All of the tests can be run using the command bundle exec rspec.

Android Application

The Android application is the entry point for users to manage their device, preferences, reminders, and view vehicle diagnostics. All the information displayed in the application is pulled directly from the web service so there is no need to hard code anything and new data can be pushed to the device by updating the web service. There are a lot of classes in this project so I will breakdown some of the more interesting ones.

`Vehilytics.kt`

This is a singleton object that holds onto the user credentials and preferences while the application is open, and is responsible for adding or removing them from storage.

`/storage/Storage.kt`

This class handles reading/writing to and from `SharedPreferences` so I can persist some small amounts of data on the device. Currently the only data that gets stored is user authentication token so that the login screen can be skipped.

`/activities/SplashActivity.kt`

Splash screens are usually just a way for a developer to waste a few seconds of time for no good reason or to get extra marks, but I used it instead momentarily distract the user while I check local storage and validate any stored tokens to possibly skip the login screen.

/adapters

This directory contains a number of custom adapters for RecylcerViews that allow me to customise the contents of each list item to my liking so that I can nest TextViews, a Checkbox, or some ImageViews.

/helpers/callbacks/VehilyticsCallback.kt

This is a special callback object for handling responses for any web requests I make. It implements a nice feature where if any of the web requests return 401 to signify that the user was unauthorised, it wipes the local storage and forces the user to login again.

/firebase

My web service handles authentication and data persistence so I didn't have much use for Firebase but it did allow me to implement push notifications so that when a potential problem is detected with a sensor I can notify the user immediately.

/services/ServiceManager.kt

I used a library called Retrofit to expose the API endpoints as Kotlin interfaces. I created the ServiceManager singleton object to manage the retrofit instance and act as the single point of entry to all the available services.

Testing

The tests for this project can be run by right clicking on /app/src/test/java in the project pane in Android Studio and selecting Run 'tests in 'java''.

Version Control

I have made each of the systems available publicly via Bitbucket in case there is any issue with the uploaded code.

I will be continuing to work on the code leading up to the showcase so I will provide a link to the last commit before this upload

Python code available at:
https://bitbucket.org/danthedrummer/vehilytics_reporter/src/cabb15d572c0eb1f33b834ba0a1e561f61de83d0/

Web service code available at:
https://bitbucket.org/danthedrummer/vehilytics_api/src/61451c0ae20e2a2380e936d5273a119d7c1e6717/

Android application code available at:
https://bitbucket.org/danthedrummer/vehilytics_android/src/f14469eddb2a809331921f03fe860e604493cd01/