PLANT VIEW – AN AUGMENTED REALITY ANDROID APPLICATION

Matthew Frost (L1426439) - BSc Computer Science

Tutor: Eudes Diemoz

Abstract

An augmented reality Android application that displays information relevant to the user depending on their location. The application was originally developed for a client that operates on an industrial site, so the data collected would be relevant to their use case, such as the temperature of a pipe. However, the app has been generalised enough that it can work with any numerical data set, for example, the energy usage of buildings. The data is displayed on a graph to show how it changes over time and statistical analysis is applied to highlight any anomalies in the data.

The application uses a range of Android features such as GPS location, camera preview, QR code scanning and the use of the internal compass.

A separate Google Maps web application has been developed to allow the mapping of locations against data in an SQL database. Both the Android app and web app communicate with the data sources using Node JS web services. The web services are used to store and retrieve location points as well as pull the data for each location wherever it is stored.

Acknowledgements

Firstly, I would like to thank Eudes Diemoz for acting as my project supervisor and providing feedback and advice. Next, I would like to thank both Sabisu and Tim Butters for providing the initial project specification as well as guidance on next steps and feedback on work. Finally, a thank to Teesside University for providing the data used during development.

Contents

1. Introduction

1.1 Rationale

The project proposal was initially provided by a local company, Sabisu, who develop reporting tools for customers within the oil and gas industry. They were looking for an augmented reality Android application that would allow a user on an industrial site to walk around with a tablet and view information on what was around them. Sabisu also asked for a web application that would allow points of interest to be plotted on a map, this would be used by admin users.

There were not many technical limitations put in place within the project proposal from Sabisu, all they asked for was an augmented reality Android application, and an admin web application to go along with it. This meant that the project could be approached with any methods or technologies that would best suit the given task.

Furthermore, as this project allows for a range of technologies to be used, this would help with employability as both web applications, and mobile applications are what the current market is moving towards.

1.2 Ethical, Legal and Social Issues

Since this project involves a real client it is important that the British Computer Society code of conduct is adhered to (British Computing Society, 2017). The section that would apply specifically would be "professional competence and integrity" meaning that work should only be undertaken that the developer believes they are competent of and they should be willing to accept criticisms and alternative viewpoints. As the software will be open source, available online via GitHub, and be given to a client it could be useful to a license with the project. The MIT License would be most effective as it states that anyone can use and modify the project as they fit however, the author of the project is not liable for anything that goes wrong (Open Source Initiative, no date).

2. Research & Analysis

2.1 Requirements

The initial proposal provided contained several requirements, some were necessary and some were just "nice to haves". After analysing the requirements provided, it was clear that it would not be possible to complete them all in time and some of them would not be possible to work on outside of the Sabisu offices. Therefore, it was necessary to remove any requirements that would not be feasible as well as anything that would not be implemented in time and would not affect the end product too much. To categorise and prioritise the requirements the MoSCoW(Must have, Should have, Could have, Will not have) system was used (Agile Business Consortium, 2008). This ensured that if all the minimum requirements were met then a shippable product would be produced. Requirements that come under "must have" are requirements that would make the project useless if they were not included. Requirements that come under "should have" are requirements that are important to the project but are not critical to success. Requirements that come under "could have" are “nice to haves”, they may add more polish or functionality to the project but are less important than “should have” requirements. Anything that falls under “will not have” will not be done in this release, they are items that are feasible but will not be able to be included in the given time period.

For example, one of the requirements was to have it connect with the Sabisu platform to integrate with some of their APIs. However, this would require having a VPN for their network during development and it is not a feature that is necessary for the application to work.

Must

- The web application must allow the user to create, edit and delete locations
- The web application must display all created locations on the Google map interface
- The web service must return all created locations that have not been deleted
- The web service must be able to return all locations within one-hundred meters of a given latitude and longitude
- The web service must allow locations to be created, edited and deleted in the database
- The web service must be able to return all data associated with a given location
- The Android application must use the device’s GPS to get the current location
- The Android application must access the device’s camera to create an augmented reality application.
- The Android application must interact with the web service to get the locations near the device
- The Android application must retrieve the data associated with a given location
- The Android application must display the retrieved information on a graph

Should

- The Android application should perform analytics on the data retrieved to identify any anomalies in the data
- The Android application should use the device’s compass to get the direction the device is facing

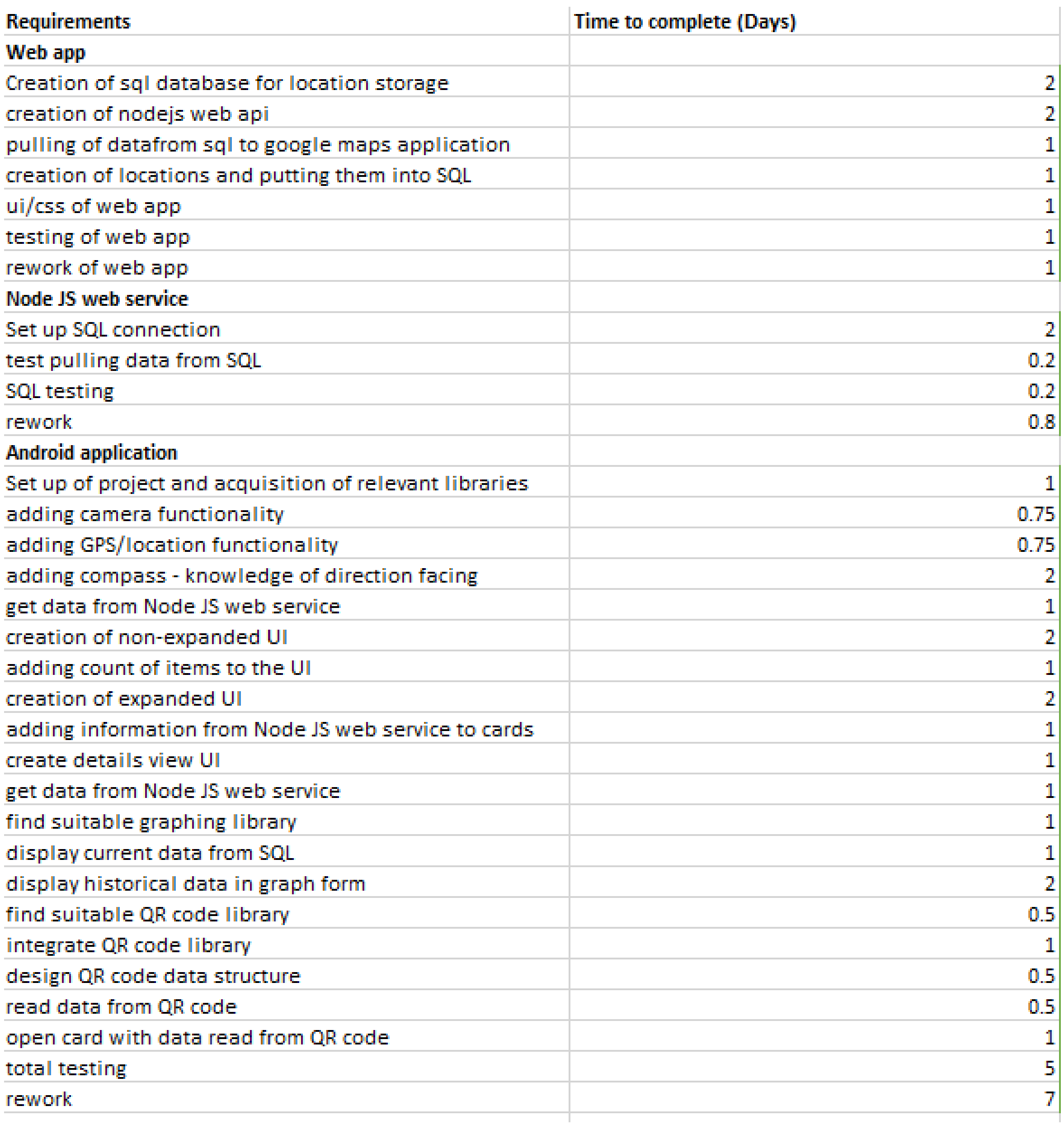
Could

- The Android application could implement QR code scanning to retrieve information about the location associated with that QR code
- The Android application could have offline capabilities to store the data if internet connectivity is lost

Will not have this time

- The Android application will not have connectivity to enable Sabisu log ins
- The Android application will not use the Sabisu API’s
- The web services will not connect to IP21

2.2 Estimates



Requirements	Time to complete (Days)
Web app	
Creation of sql database for location storage	2
creation of nodejs web api	2
pulling of datafrom sql to google maps application	1
creation of locations and putting them into SQL	1
ui/css of web app	1
testing of web app	1
rework of web app	1
Node JS web service	
Set up SQL connection	2
test pulling data from SQL	0.2
SQL testing	0.2
rework	0.8
Android application	
Set up of project and acquisition of relevant libraries	1
adding camera functionality	0.75
adding GPS/location functionality	0.75
adding compass - knowledge of direction facing	2
get data from Node JS web service	1
creation of non-expanded UI	2
adding count of items to the UI	1
creation of expanded UI	2
adding information from Node JS web service to cards	1
create details view UI	1
get data from Node JS web service	1
find suitable graphing library	1
display current data from SQL	1
display historical data in graph form	2
find suitable QR code library	0.5
integrate QR code library	1
design QR code data structure	0.5
read data from QR code	0.5
open card with data read from QR code	1
total testing	5
rework	7

Figure 2.1 – estimates

Before development work began it was important to break down the requirements into smaller tasks, then estimate how long it would take to complete each task. Doing this makes it possible to check if every requirement can be completed during the allotted development time if it is not possible then a re-evaluation of the requirements would be necessary.

Due to the short development time, it was decided that each individual task should only be estimated to a maximum of two days. If a task was estimated to be longer than two days then it should be broken down into separates tasks that would each be less than two days work. A single day was the equivalent of one work day, about 7-8 hours.

2.3 Justification of choices

Android Development Language

For Android development, there are several languages and libraries that can be used. After some research there were a number of languages that seemed to be popular and potentially suitable for the project, these include:

- C/C++ (Android Native Development Kit (NDK))
- Java (Android Software Development Kit (SDK))
- Kotlin (Android SDK)
- C#/Vuforia (Unity)

Since the Android operating system is built in C and C++ (Agarwal, 2011) it is possible to create applications for Android in these languages. Applications built using the NDK are often faster than those written in a Java based language as they do not need to run on the Java Virtual Machine (JVM) (Sims, 2016). However, for this project, the speed limitations of the JVM are not going to be an issue and the lack of support for the NDK compared to the Java SDK mean that C and C++ were ruled out for the choice of Android development language.

During the initial research period, the head of Sabisu suggested that the Unity engine (Unity Technologies, 2017) could provide a solid method for creating augmented reality applications. With some research, it was found that there is a library for Unity called Vuforia (PTC Inc, 2017) which provides an easy way to add augmented reality and image recognition to an application. However, for the image recognition to work, it had to have knowledge of the shape beforehand and shapes had to be complex to enable more accurate recognition. As the application would be looking at 3D shapes such as tanks and pipes it was clear that using Vuforia would not work.

The most popular language for Android development is Java, it is the language used by Google's Android SDK. Therefore, a Java based language was chosen as the language for the creation of the project. Kotlin was chosen as it complies down to Java bytecode so it will run on the JVM and it can also work alongside any existing Java libraries. The advantage of Kotlin for Android development is that it removes some of the potential drawbacks of Java such as, null pointer exceptions and having to reference every Android component by using the findViewById method. Due to the advantages that Kotlin provides, it is the language that has been chosen for the development of the Android application.

Web Service Development Language

The project contains three separate web services, one to store and retrieve the geo-location points for both the web application and the Android application. Another web service retrieves the data about a given location, this data will be stored in an SQL database. The final web service is purely for dealing with anomalies in data.

When developing a web service there is a large range of languages that could have been used. However, the two that are most prevalent were C# .NET and Node.js.

C# .NET was a potential candidate as it has all the features that would be required and being a Microsoft product it had good integration with SQL server using Entity Framework. Furthermore, since C# has been around for seventeen years it is a mature and widely used language with plenty of support.

Node.js is a more modern development language, being initially released in 2009 and only coming into more widespread use within recent years. So, due to the more modern technology being used it could be more appealing to the industry. Furthermore, Node.js is a very light weight solution and as both web services are just reading and writing data to databases it appears something heavier such as .NET would be too much as most its features would not be utilised. This could be especially useful if the web service was to be hosted on a cloud based computing platform such as Microsoft's Azure or Amazon Web Services. On these platforms, memory and processing power come at a premium so being able to run the web services on a low a tier as possible and still be effective would be important for saving costs.

Integrated Development Environments (IDEs)

Android Studio is the official IDE supported by Google and it provides all the tools necessary for creating an Android application. Although most Android development is done in Java, Android Studio provides support for the Kotlin plugin which allows code to be translated from Java to Kotlin, as well as providing code completion for Kotlin.

For the web development aspects of the project, Visual Studio was used. Visual Studio was chosen because it is widely used within in the industry and university so is therefore very familiar. Visual Studio provides support for a lot of different languages including the ones needed for this project which are HTML, CSS, and JavaScript.

Data Storage

Storing the data that the project used came down to the choice between two different paradigms, SQL (Structured Query Language) or No SQL. However, this project would not take need nor take advantage of any of the features of a No SQL data storage mechanism. No SQL databases are generally used for storing unstructured data and data that differs in both type and content (Gentz, Rabella, 2017). For this reason, SQL Server was used for storing any data that the applications will access.

A lot of industrial data that Sabisu accesses is stored in a data historian called InfoPlus21 (IP21), having access to this system would allow the app to return live data as it is updated by the sensors on the industrial site. However, access to this would require a virtual private network (VPN) as well as credentials to log into IP21. Furthermore, after deciding to make the application more generalised it would not make sense to spend the time adding such a niche feature. After speaking to Sabisu they agreed that just connecting to a SQL database would be suitable.

3. Design

3.1 Mock-up designs

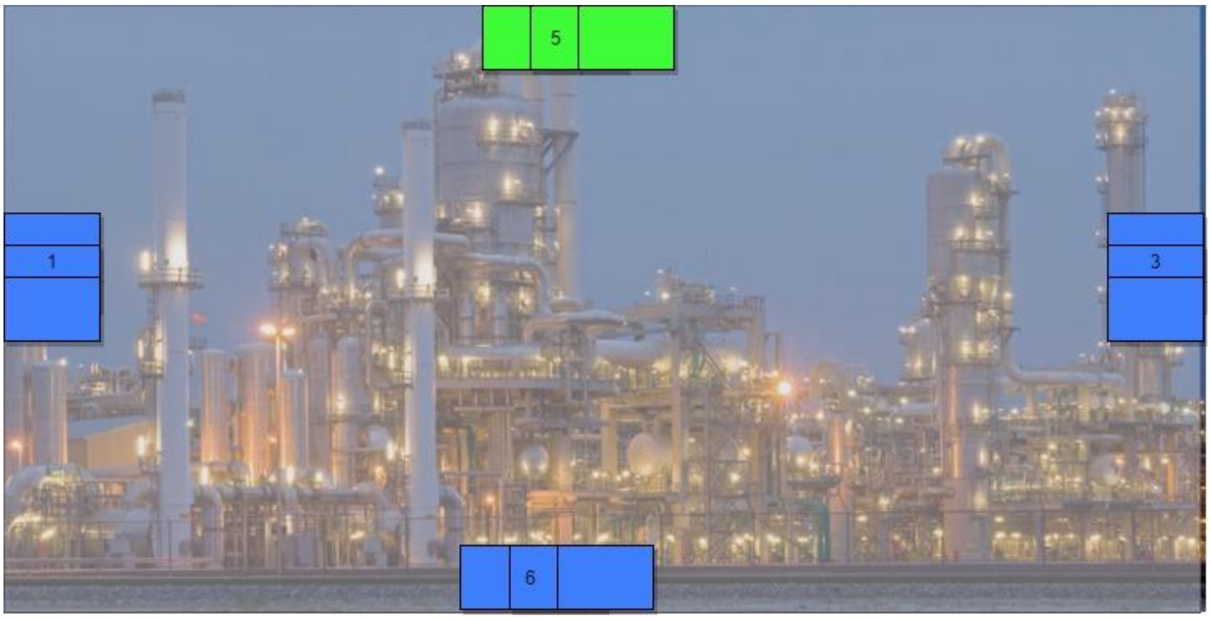


Figure 3.1 - initial mock-up of main application screen

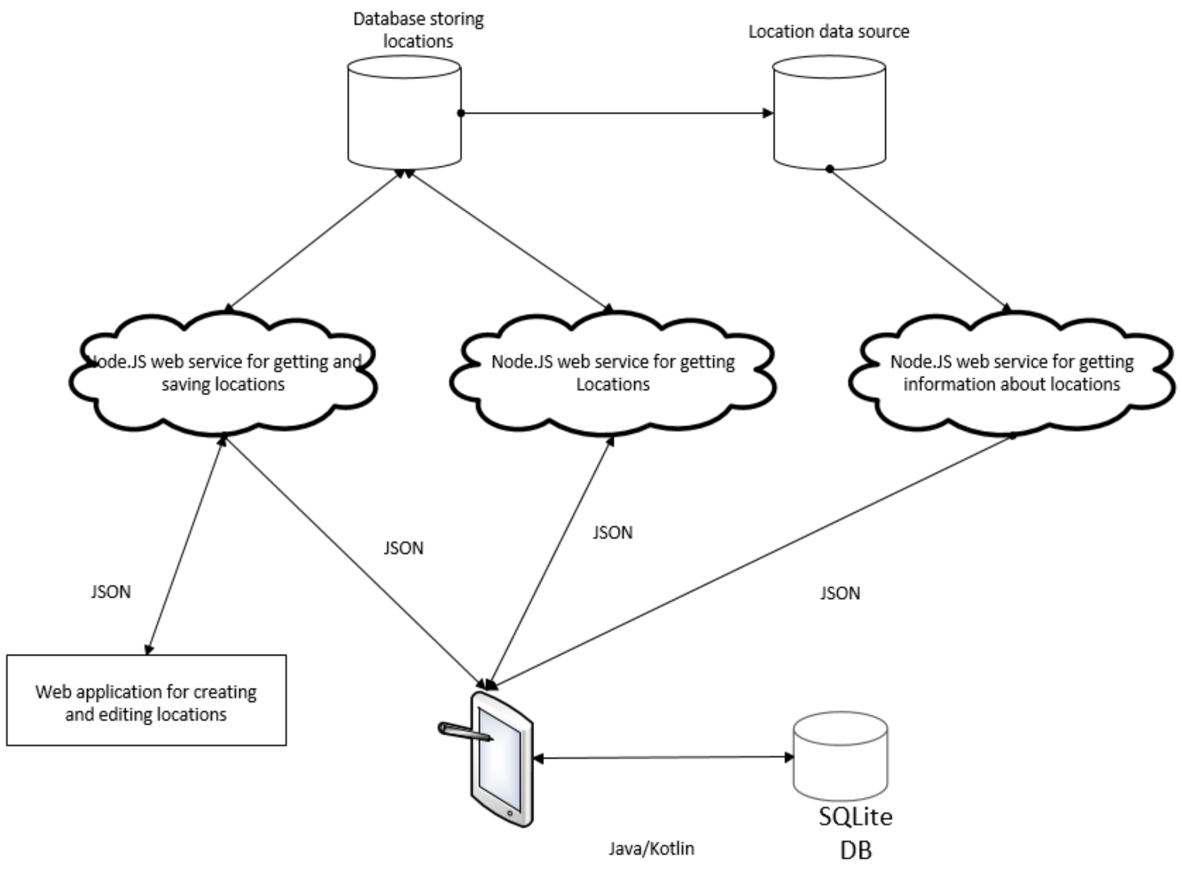
When designing the user interface, it was important to keep in mind the use case of the application. As it was initially intended to be used in an industrial setting where the user would be walking around outside, the user interface needed to be simple to use.

To meet these criteria the entire user interface consists of large buttons that could be accurately pressed even when wearing gloves. Furthermore, since this is an augmented reality application it is important not to clutter up the interface too much as it could obscure the camera view.

As well as being functional, a user interface should also look appealing. To achieve this, Google's material design guide lines were followed (Google, 2017$_a$). Material design is used to help create a flat modern look and use shapes and shadows to help the user understand how the application works. In this application material design is used to help create the impression of a stacked set of cards, this is used to convey the idea that clicking on the stack will allow the user to see each card individually.

Finally, another important item to consider for good user interface design is the choice of colour. As the top button is the only one that can be clicked it has been coloured green. Green typically conveys ideas of safety and correctness (Butters, 2014) so therefore it was chosen for the only clickable button. Red is normally used as the inverse of green however, it did not seem suitable to use red for the other buttons. Therefore, blue was chosen for the other buttons as it comes across as a more neutral colour compared to red and green.

3.2 Architecture



Database storing locations
Location data source
Node.JS web service for getting and saving locations
Node.JS web service for getting Locations
Node.JS web service for getting information about locations
JSON
JSON
JSON
JSON
Web application for creating and editing locations
Java/Kotlin
SQLite DB

Figure 3.2 – The architecture of the full system

Figure three shows the architecture of the system as well as how data flows between each of the separate applications. The main structure of the system is using microservices to enable connectivity and data transfer between the different layers. Martin Fowler describes a microservice architecture as “an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms” (Fowler, 2014). The microservice approach was chosen as it improves both maintainability and scalability. By keeping a piece of functionality inside a single container if anything goes wrong with that feature then it can be immediately isolated to that single service. Having a maintainable architecture is especially important when shipping the application out to a customer or client. Furthermore, once the application starts to be used by multiple users it will be clear which services are put under the most load and a microservice architecture would allow each service to be scaled up or down with demand.

Both the web application and Android application communicate with the web services using Hypertext Transfer Protocol (HTTP) requests, sending and receiving JavaScript Object Notation (JSON) data. Although the applications could communicate with the databases directly, the web services are there to apply any business logic and shape the data in a way that can be directly used to by the applications without further modification.

The database for storing locations is the database specific for the application. It contains information about locations and the data that is mapped against them, as well as a list of locations with anomalous data against them. It also has stored procedures used to retrieve and insert data from and to the database as well as retrieve data from another database. The second database marked 'Location data source' is the database that would be provided by the user, it should store the data related to the locations that the application uses.

The Android application also has a database associated with it, it is used to cache the data retrieved from the web services so that if the Android device loses connectivity to the network it will still be able to show some data.

3.3 Database Diagrams

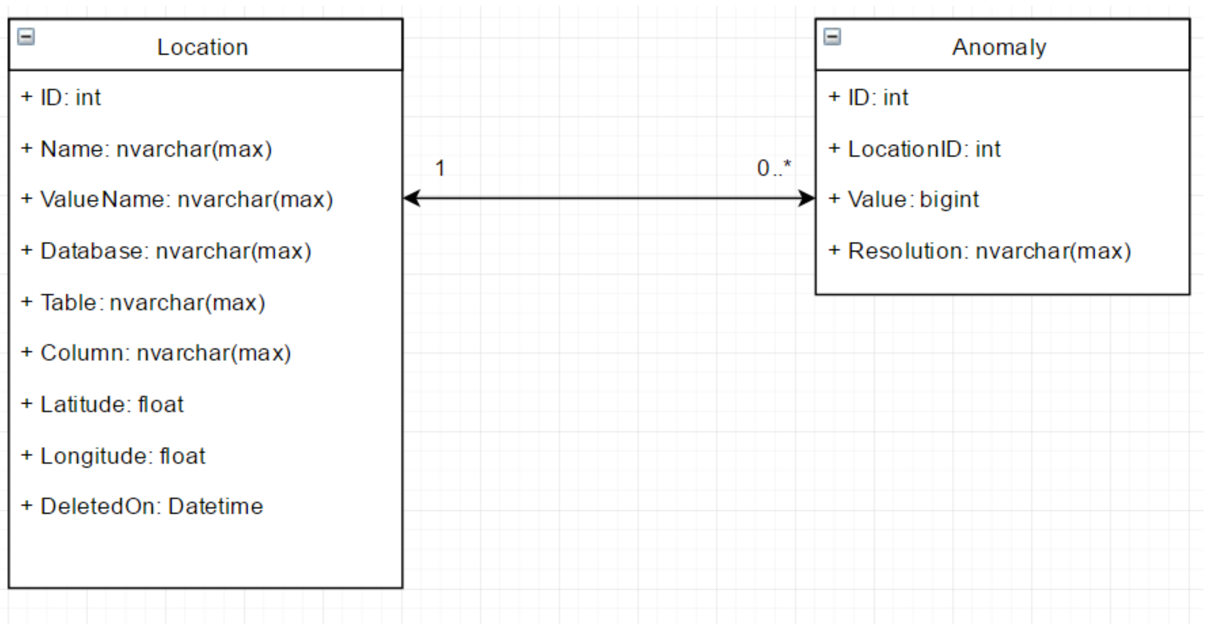


Figure 3.3 – Location database diagram

The database used by the web services only includes two tables. The Location table stores information about locations, this includes the position of the location in the physical world and the location of the data that is mapped against that location in a separate database or table. The location table contains a nullable column called "DeletedOn" this column is used to implement "soft deletion" of records. Doing this means that records marked as deleted can be easily filtered out in queries by including "where DeletedOn is null". However, this clause can be removed by always selecting data from SQL views where deleted data has already been filtered out, this removes repetition of code and stops deleted records from accidently being returned. By having the column as a datetime field it is also possible to see when the data was marked as deleted, this can be useful when going back see what happened and when.

The database also contains a table for the storing of anomalies, the table is related to the Location table using LocationID as a foreign key. Anomalies are not intended to be deleted there for there is no field for marking them as deleted.

3.4 Class Diagrams

3.4.1 Location Web Service

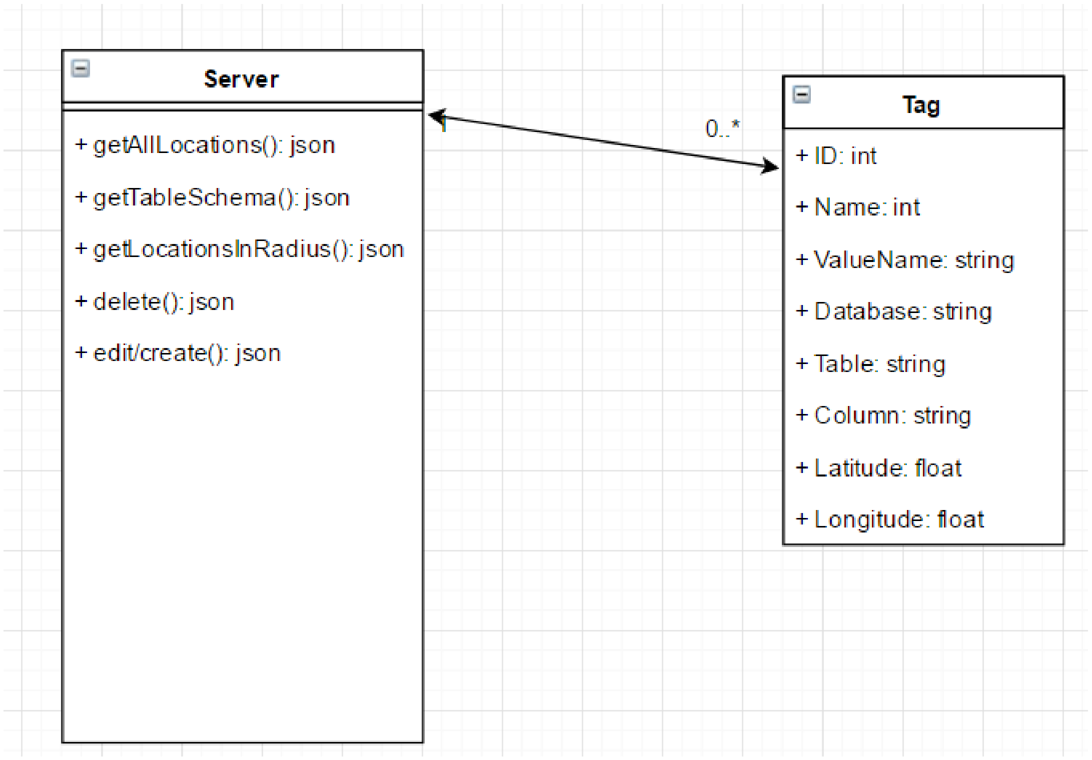


Figure 3.4 – Location web service class diagram

The web service that allows locations to be saved, edited, deleted and retrieved from the database consists of just two classes. The Tag class is used as a data transfer object (DTO), mapping the data that goes into and from the database. Doing this allows the data to be easily manipulated in the form of a JavaScript object.

The Server class is where the logic and functionality is implemented. The getAllLocations method returns all locations from the database that have not been deleted, this method is used for the web based application.

The getTableSchema method takes the name of a database and table, then returns the names of the columns for the given table. This is also used in the web application to help with the creation of locations and mapping data to the selected location.

The getLocationsInRadius method takes a latitude and longitude value for a location and returns all locations from the database that are within one hundred meters of the given location. This functionality is only used by the Android application to show the user all points within their vicinity.

The delete function is used to mark locations in the database as deleted so that they will not be displayed in any of the applications. The function takes the ID of the location to be deleted.

The creation and editing of a location is wrapped up into one function. Traditionally both actions would have their own function, however since the action for editing and creating a location uses the same stored procedure in the database, there was no point splitting them up. The function takes a Tag object when creating and editing a location. The function is only used by the web application to create and edit locations.

3.4.2 Location Data Web Service

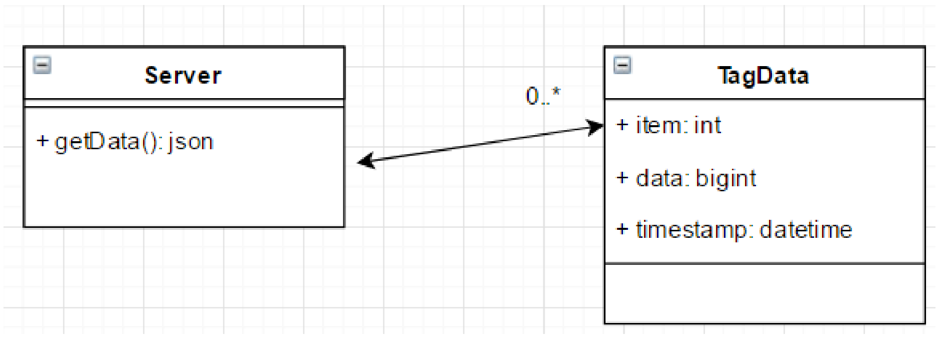


Figure 3.5 – Location Data web service class diagram

The web service for getting data related to a given location consists of two classes, TagData is the DTO class describing what the data should look like. The server is where all the functionality and connectivity to SQL server is handled.

The getData method retrieves all data for a given location, it takes the ID of a location and returns a JSON array of TagData objects.

3.4.3 Android Application Class Diagram

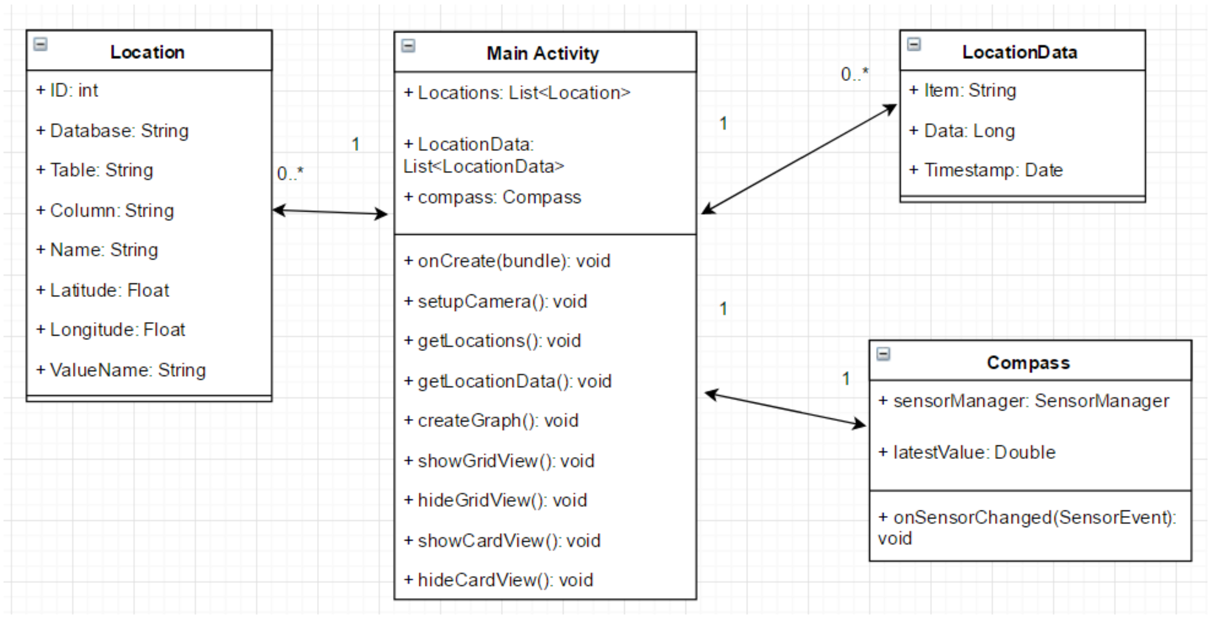


Figure 3.5 – Android application class diagram

Figure seven shows the class diagram for the Android application before any development had taken place. The Android application makes use of the same DTOs as the Node JS web services, this is done to help keep the shape of the data consistent as it moves between applications.

In Android applications, each screen is known as an activity, by default the first activity that is opened in an application is the main activity. The Main Activity class is where most the application logic and functionality is contained. The onCreate method is called by the Android operating system when the activity is loaded up, here all the initialisation of events and listeners is set up. The setupCamera method is called from onCreate and is used to initialise the device's camera and show the camera preview within the application. getLocations calls the location web service to retrieve the locations near the user and stores the locations in the Locations list. getLocationData gets the data related to a given location, it takes the ID of a given location and stores the data in the LocationData list. CreateGraph uses the data in the LocationData list to create a graph from each of the items in the list using the timestamp and data value. Since the application only has one activity the rest of the functions are used to show and hide different aspects of the user interface. By having everything in one activity it removes the need to pass data around different activities. The drawback to having one activity is that a lot of code can end up in the one class.

In order to detect what direction the user is facing the device's compass needed to be utilised. This has been moved into its own class to keeps functionality contained in a single class instead of mixing with other operations, this is known as separation of concerns. To access the device's, compass a sensor manager needs to be used, this can be used to acquire access to the compass values. The onSensorChanged method is fired every time the orientation or angle of the device is changed. This method then gets the latest value for the compass and stores it in latestValue.

3.4.4 Web application

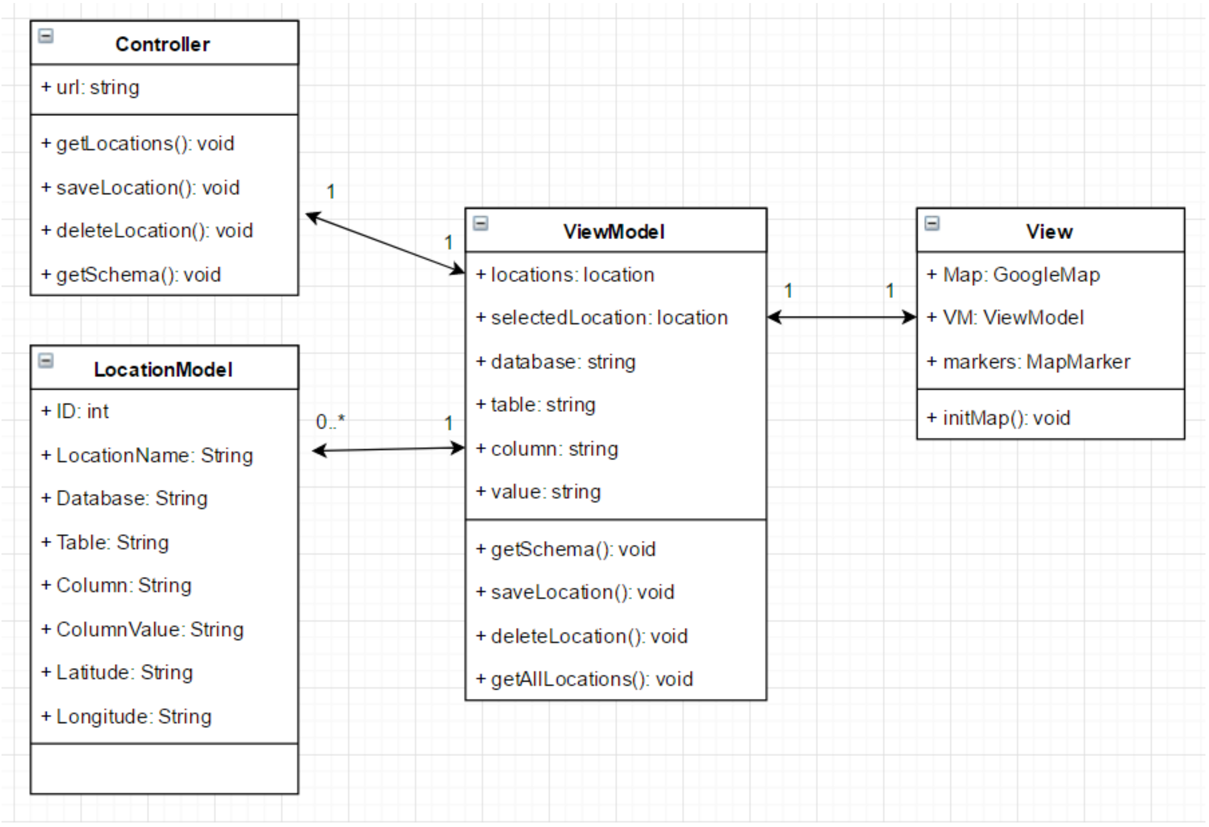


Figure 3.6 – web application class diagram

The web application is used to manage locations, has a model-view-viewmodel (MVVM) architecture. The view contains references to the Map, the markers on the map and the viewmodel. All the view is intended to do is initialise the viewmodel.

The LocationModel acts as the DTO, mapping what is received and transferred to and from the web service.

The ViewModel class takes the LocationModels and applies the logic to display them as points as points on the map. The viewmodel sits in between the controller and the view, taking data from the controller, transforming it into models and displaying it with the view.

3.4.5 AnomalyNotification Service

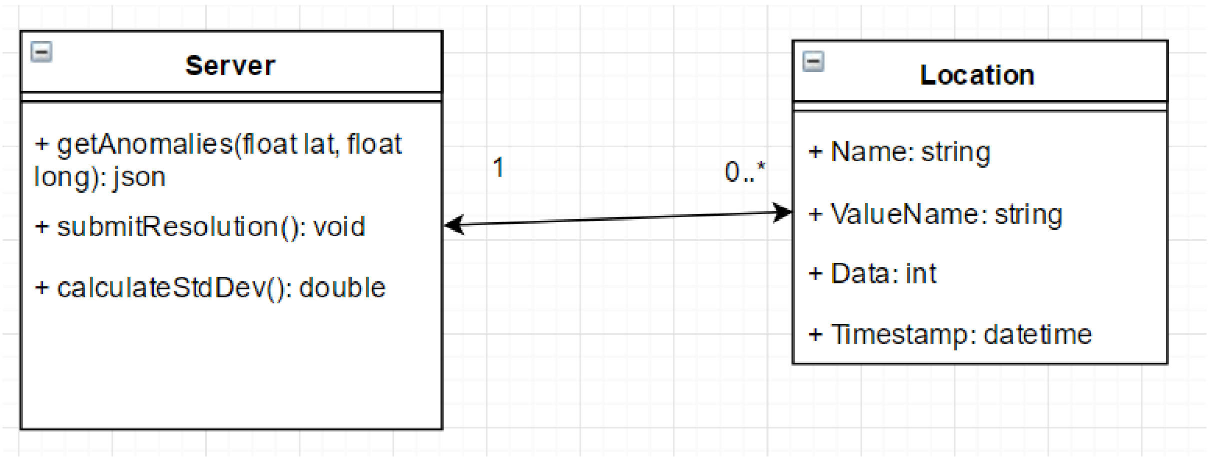

Server
+ getAnomalies(float lat, float long): json
+ submitResolution(): void
+ calculateStdDev(): double
1
0..*
Location
+ Name: string
+ ValueName: string
+ Data: int
+ Timestamp: datetime

Figure 3.7 – AnomalyNotification web service class diagram

The anomaly notification web service is used by the Android application to retrieve any locations that have anomalous data associated with them. The getAnomalies function takes a latitude and longitude and returns any locations within the vicinity of the given position that have anomalous data. The getAnomalies function makes use of the calculateStdDev function to take the data from a location and find out if there are any anomalous data points. The submitResolution function is used to put a resolution or reason next to an anomalous data point in the database.

The location class is used as a DTO for this web service, it mirrors the anomaly table in the database and is used for getting anomalies and submitting anomalies with resolutions.

4. Implementation

4.1 Development methodology

When planning a project there are a range of methods and techniques that can be used to most effectively plan out what needs to be done and when it should be done by. Project management methodologies can be split up into several groups, these include sequential, agile and change management (*Project Methodologies*, no date).

An example of a sequential methodology is waterfall. Waterfall means that each section is fully completed before moving onto the next step and stops backwards steps. This means that ideally, each step will be perfect in order to move onto the next one. However, as a real client is involved this is not always the case, as requirements often change. As the waterfall methodology does not fit around revisiting previous stages it would not be a suitable strategy.

This project was developed using an agile methodology. Each week of development was a sprint, where a single feature would be implemented then it can be fully tested and any rework can be applied in the same sprint. There are many different approaches to agile development such as Scrum, Extreme Programming, and Feature-Driven Development. The flavour of Agile used for this project was Scrum, in Scrum there are three “actors” these are: Product Owner, Scrum Master, and the Development Team. The product owner is the person who has the vision of what the end product should be, in this case the product owner is Sabisu. The Scrum Master acts as an intermediary between the product owner and development team, they help remove any obstacles encountered by the development team and clear up any questions with the product owner. In this case, the Scrum Master was the project manager from Sabisu. The development team in Scrum is responsible for creating the product but they are also entirely self-managed, they are responsible for managing time and resources.

At the beginning of each sprint, there were meetings with a member of Sabisu to discuss the progress of the previous weeks work and what should be worked on in the upcoming week. This methodology was effective since there was a live client. It allowed for changes to be made and features to be added during development so that the client knows what they are getting before development is finished.

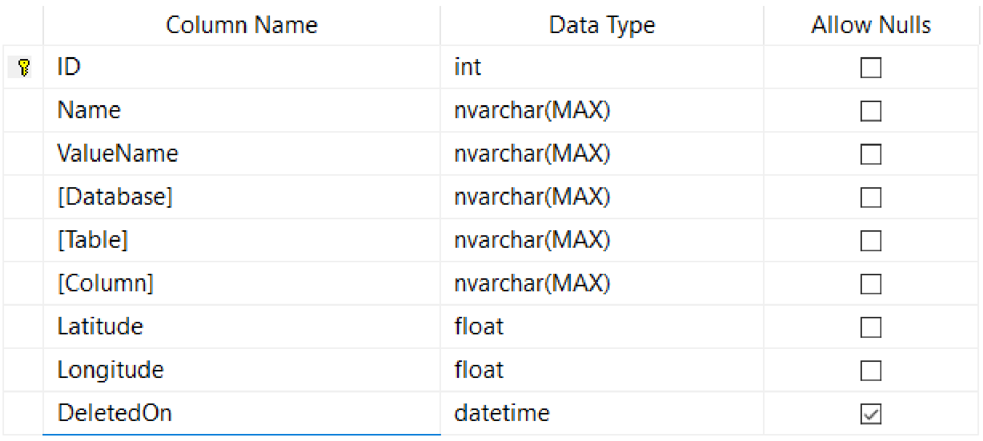
Agile Methodology
Sprint 1
Sprint 2
Sprint 3
Test
Build
Design
Plan
Review
Launch

Figure 4.1 – Agile Methodology (Strivastava, 2017)

4.1.1 Source Control
To make sure that the project was backed up constantly and readily available, Git source control was used. Git was used alongside GitHub to back up the project online, a new commit was made at the end of every day or whenever a feature was implemented. Since there was just one person working on the project there was not any real need to uses separate branches, so a single master branch was used.

4.2 Database
The database behind all the applications created is a SQL Server database. SQL Server Management Studio was used to create the database, it's tables and stored procedures. Using Management Studio meant that tables and views could be easily created with the user interface and stored procedures could be written using SQL.

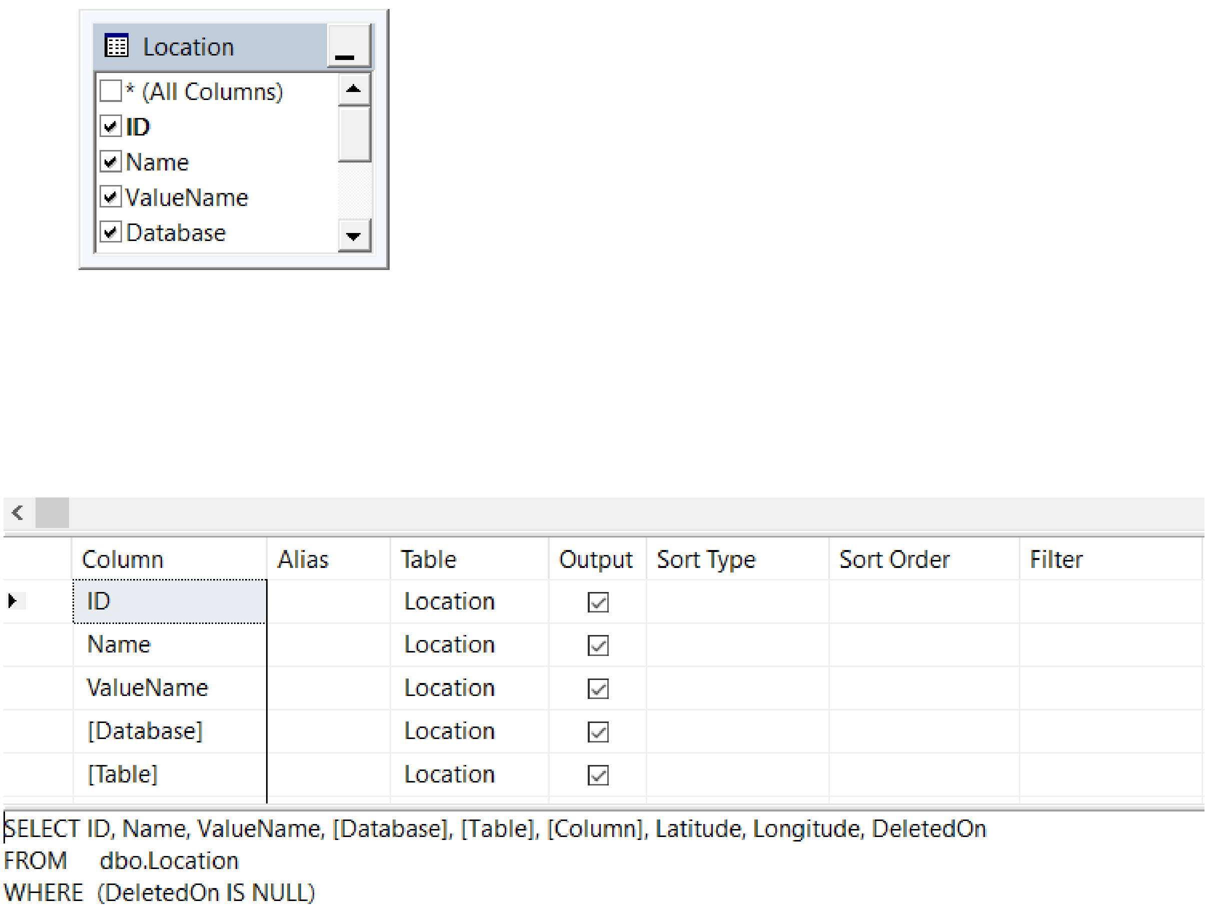
The database was created first as that is what all the applications rely on so there would be no point in developing anything until the database was complete. The first table that needed to be implemented was the one used to store the locations.



	Column Name	Data Type	Allow Nulls
🔑	ID	int	☐
	Name	nvarchar(MAX)	☐
	ValueName	nvarchar(MAX)	☐
	[Database]	nvarchar(MAX)	☐
	[Table]	nvarchar(MAX)	☐
	[Column]	nvarchar(MAX)	☐
	Latitude	float	☐
	Longitude	float	☐
	DeletedOn	datetime	☑

Figure 4.2 – Design of the Location table

The Location table consists of the columns shown in figure 4.2. All the columns apart from DeletedOn are marked as not allowing nulls, this ensures that no data can be omitted when a row is inserted. The DeletedOn column is not null when a record is marked as deleted. Each record has a unique ID which auto-increments every time a record is inserted. The ID also doubles as the primary key for the table, having a primary key is important as it helps with the indexing of rows which makes the data faster to search once the table gets large (Factor, 2013).

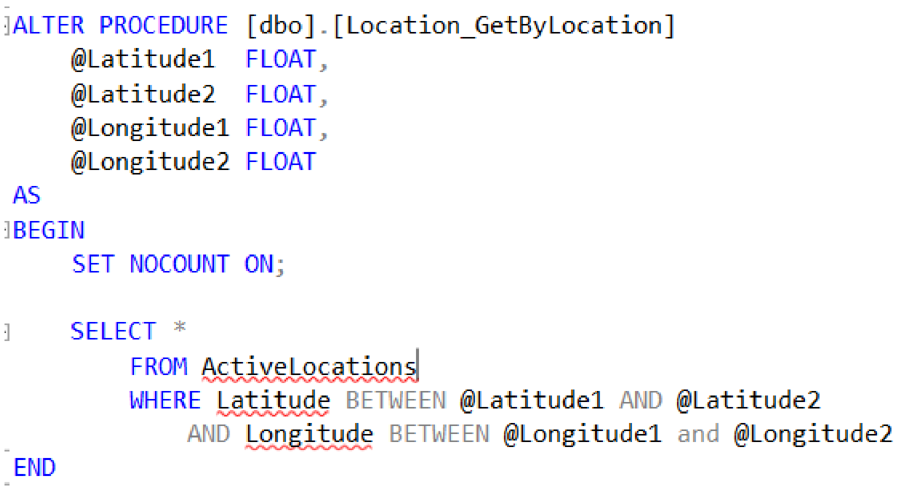


Column	Alias	Table	Output	Sort Type	Sort Order	Filter
ID		Location	☑			
Name		Location	☑			
ValueName		Location	☑			
[Database]		Location	☑			
[Table]		Location	☑			

```
SELECT ID, Name, ValueName, [Database], [Table], [Column], Latitude, Longitude, DeletedOn
FROM    dbo.Location
WHERE  (DeletedOn IS NULL)
```

Figure 4.3 – Active locations view

To make sure that only locations that were not deleted were returned, all stored procedures that read from the Location table go to a view that filters out any deleted records.



```
ALTER PROCEDURE [dbo].[Location_GetByLocation]
    @Latitude1  FLOAT,
    @Latitude2  FLOAT,
    @Longitude1 FLOAT,
    @Longitude2 FLOAT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT *
        FROM ActiveLocations
        WHERE Latitude BETWEEN @Latitude1 AND @Latitude2
            AND Longitude BETWEEN @Longitude1 and @Longitude2
END
```

Figure 4.4 – stored procedure returning certain locations

Figure 4.4 shows the SQL code used to create the stored procedure that returns all the locations within the vicinity of the user. Two sets of latitude and longitude points are passed into the procedure and these are used to return all the points that lay in between. Stored procedures were used as having the logic in the database makes it a lot easier to make changes rather than having it in a server side application. The logic of the stored procedure can be changed without having to make any re-deployments. This method was also chosen as it is similar to the approach that applications at Sabisu use, having a similar architecture will make it easier for the client to pick up and maintain.

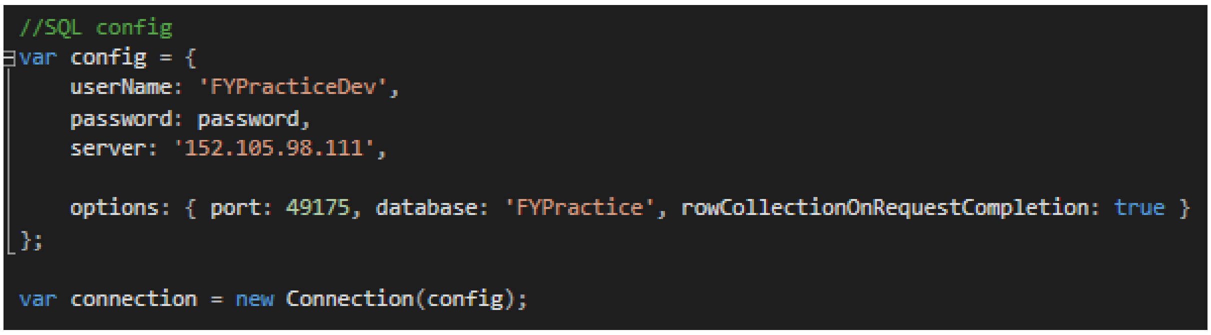
To reduce the number of stored procedures in the database, both the insert and edit functions have been merged into a single merge function. The merge statement works by having a range of variables that can be passed in to create or edit a location, if no ID is passed in then the location is inserted into the Location table. If an ID is passed in and it exists within the Location table, then it updates that record with the new values that have been passed in while keeping the old values for parameters that were not passed in. Doing this also allows the web service to have a single function to either update or create a location.

4.3 Web services

The project contains three web services which are all used to handle communication and data transfer between all the applications and databases. Each web service is written in Node JS which is an event driven JavaScript environment that is used to create server side applications (Node.js Foundation, 2017$_a$).

All the web services have a very similar structure. They all use the Express (Node.js Foundation, 2017$_b$) Node JS framework which is used to create web applications and APIs. Since they all connect to an SQL database they all also use the Tedious Node JS module. Tedious is a Node JS implementation of the TDS Protocol which is used to interact with SQL Server (Pilsbury, 2017).

The web services are initialised from the command line using Node JS, the password for logging into the database is also passed in as a parameter, doing this means that the password is not hardcoded in any file.



```
//SQL config
var config = {
    userName: 'FYPracticeDev',
    password: password,
    server: '152.105.98.111',

    options: { port: 49175, database: 'FYPractice', rowCollectionOnRequestCompletion: true }
};

var connection = new Connection(config);
```

Figure 4.5 – Configuration object for web services

When the web service is initialised a JavaScript object is created which has all of the properties required to initialise a connection to a SQL Server database, this object is then passed to a Tedious Connection object to create the connection.

One of the most difficult features to implement in the web service was finding out the latitude and longitude points either side of the user. This was needed to be able to retrieve the points that are in the immediate vicinity of the user, a coordinate is passed in and all locations within one hundred meters of that coordinate should be returned. As the location data is stored in an SQL database it is not possible to create a query that would return all locations within a given radius, therefore a conceptual bounding box needed to be created. This was an original idea, essentially the query to the SQL database asks for all points that are between two latitudes and two longitudes that are 100m apart from the user's position.

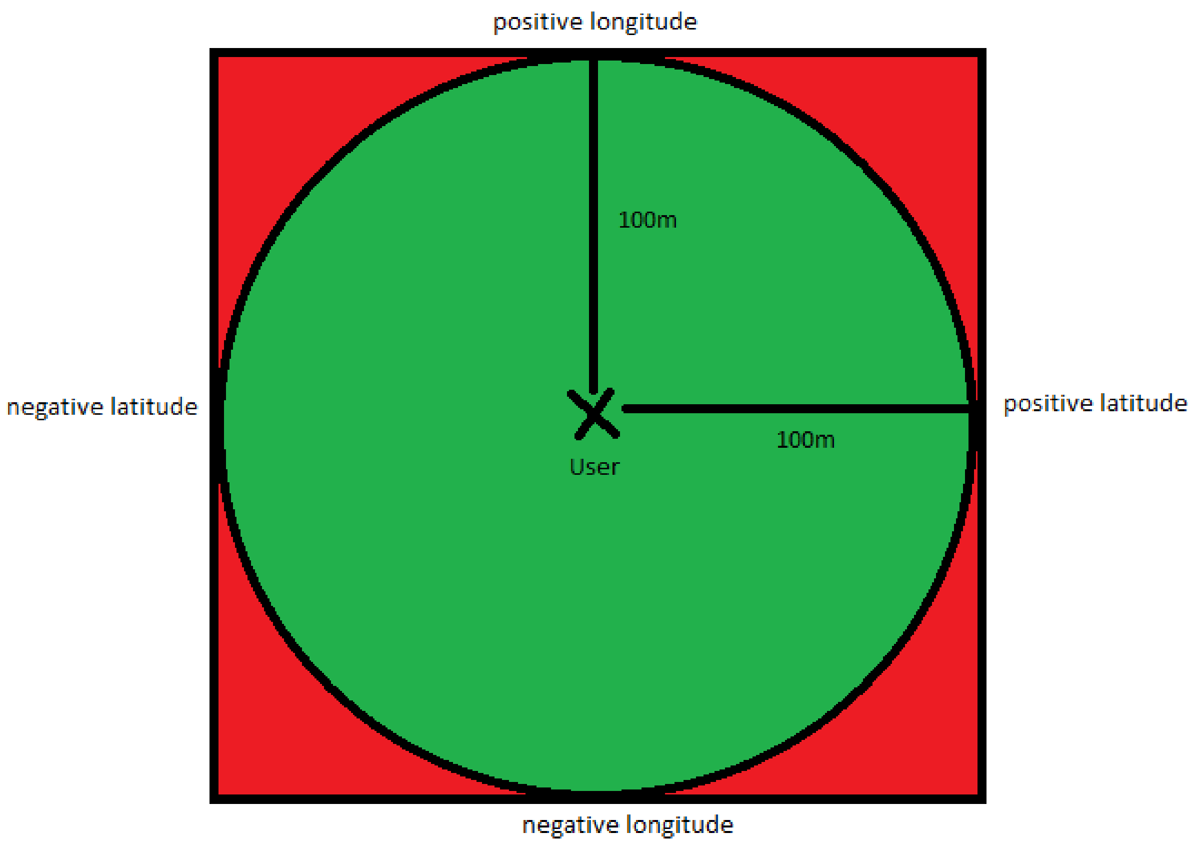


Figure 4.6 – conceptual bounding box

$$newLat = originalLat + \left(\frac{distance}{earthRadius}\right) * (\frac{180}{\pi})$$

Figure 4.7 – Formula to find new latitude

$$newLong = originalLong + \left(\frac{distance}{earthRadius}\right) * (\frac{\frac{180}{\pi}}{\cos\left(originalLat * \frac{\pi}{180}\right)})$$

Figure 4.8 – Formula to find new longitude

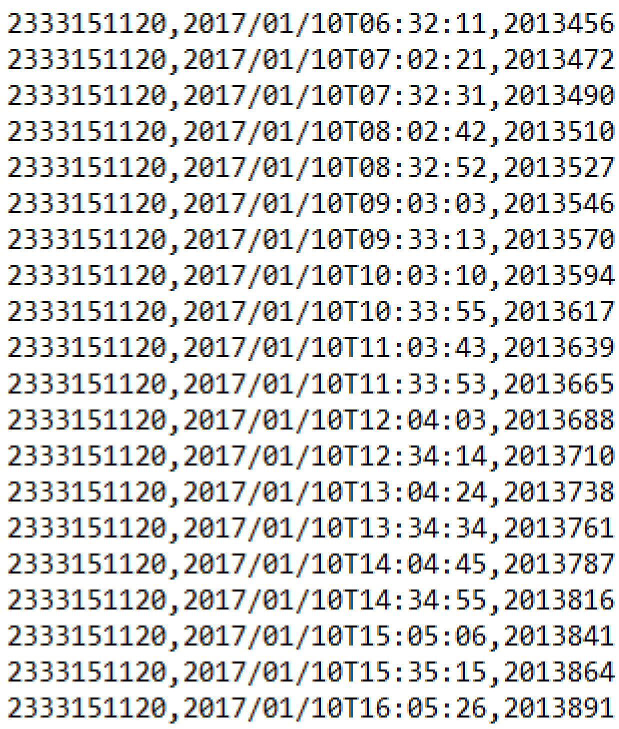
Figure 4.6 shows how the conceptual bounding box works, the query executed by SQL server will return any locations within the red square however the application only wants to display the locations within the green circle. To do this the locations within the red square are returned from the database to the application where they are converted into Android Location objects so any locations that are out of the required radius can be filtered out.

The difficult aspect of this was finding the required values for the negative and positive latitudes and longitudes. To find the new latitude the formula shown in figure 4.7 is used, the latitude in radians is found by dividing the distance by the radius of the earth, it is then converted to degrees by multiplying by 180 divided by pi. The formula to find the new longitude is similar to the latitude formula except that it takes into account the latitude by using the cosine of the original latitude.

4.4 Text File Parser

To get some realistic data, Teesside University was approached to see if there was any data that was numerical, had a timestamp, was historical and was related to a location. Initially, data such as the number of people logged into a lab over a period of time was discussed. However, having this data could allow members of staff to be tracked even if the data was anonymised and this was against the universities policy. A dataset that was possible to obtain was the energy usage of seven university buildings over twenty days, this was ideal as it was measured every ten minutes, fluctuated throughout the day and each set of readings was related to a location.

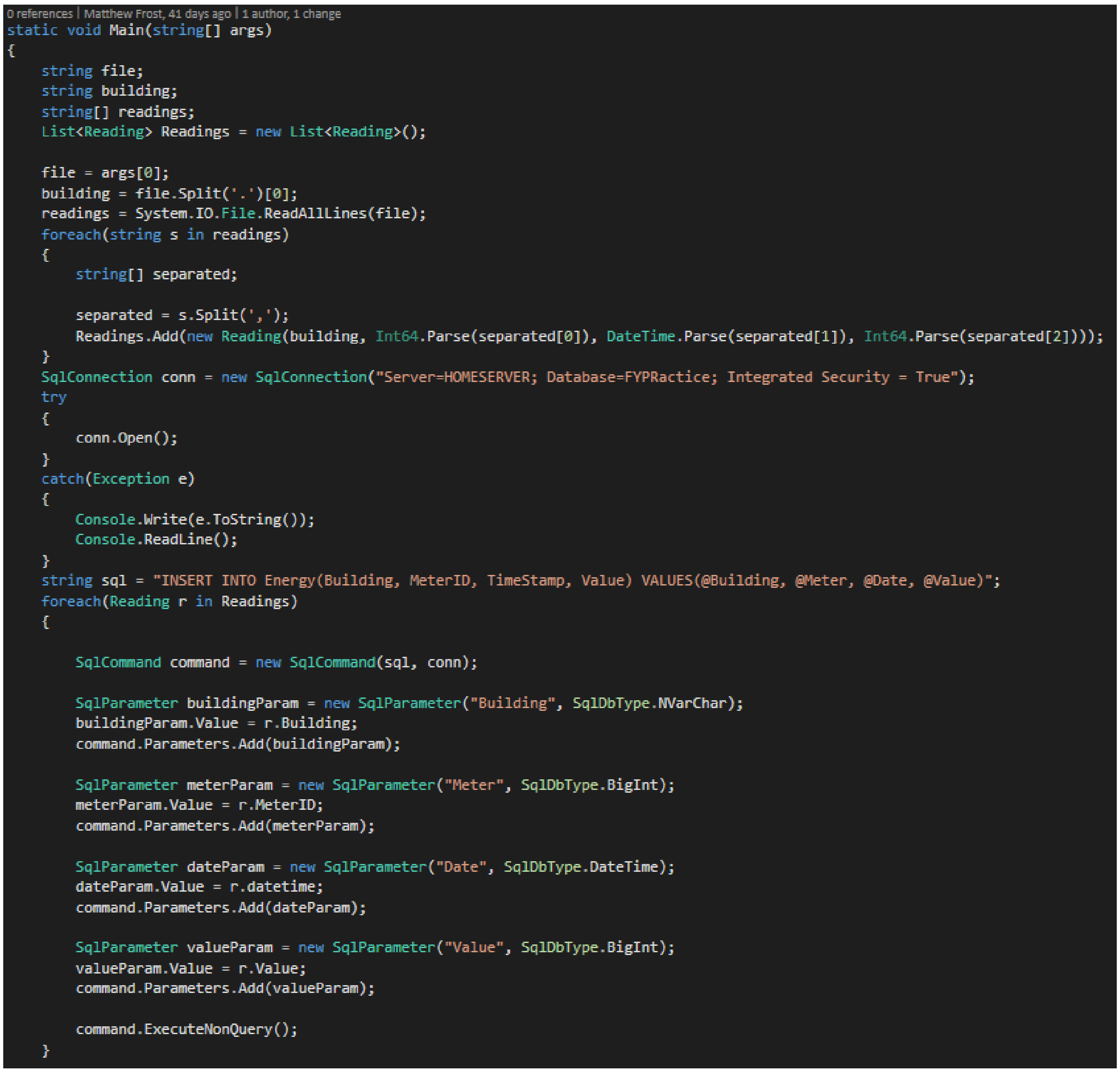
However, when the data was obtained it was given as a series of text files, one file for each building. Since this was not the ideal format for the application the files needed to be parsed and inserted into a database table. The data was in the format: building/meter ID, date-time, reading value, the end of each reading was then terminated by a carriage return character.



```
2333151120,2017/01/10T06:32:11,2013456
2333151120,2017/01/10T07:02:21,2013472
2333151120,2017/01/10T07:32:31,2013490
2333151120,2017/01/10T08:02:42,2013510
2333151120,2017/01/10T08:32:52,2013527
2333151120,2017/01/10T09:03:03,2013546
2333151120,2017/01/10T09:33:13,2013570
2333151120,2017/01/10T10:03:10,2013594
2333151120,2017/01/10T10:33:55,2013617
2333151120,2017/01/10T11:03:43,2013639
2333151120,2017/01/10T11:33:53,2013665
2333151120,2017/01/10T12:04:03,2013688
2333151120,2017/01/10T12:34:14,2013710
2333151120,2017/01/10T13:04:24,2013738
2333151120,2017/01/10T13:34:34,2013761
2333151120,2017/01/10T14:04:45,2013787
2333151120,2017/01/10T14:34:55,2013816
2333151120,2017/01/10T15:05:06,2013841
2333151120,2017/01/10T15:35:15,2013864
2333151120,2017/01/10T16:05:26,2013891
```

Figure 4.9 – Original data from text file

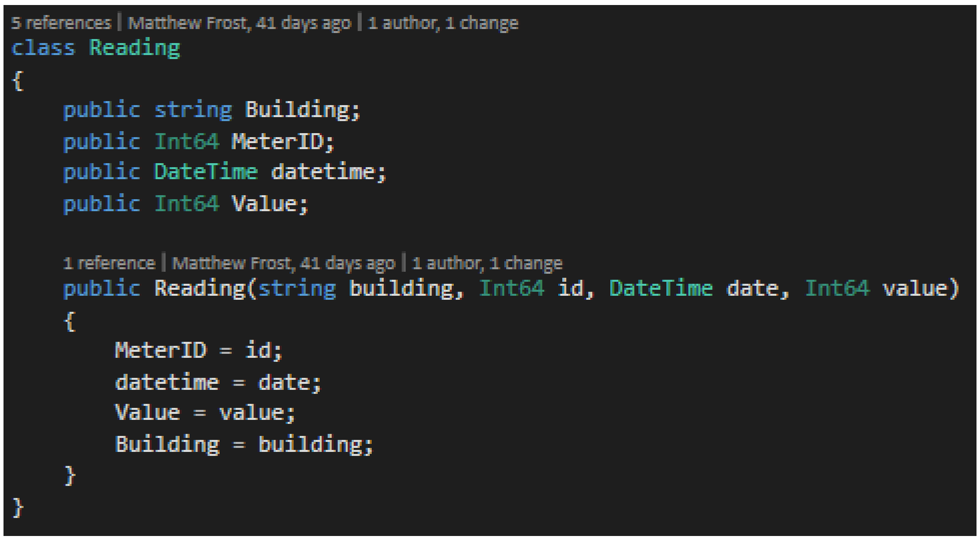
To get the data from the text files and into an SQL database a C# program was created. C# was used as it provided ways to read from files and insert into a SQL database without the need for any external libraries.


0 references | Matthew Frost, 41 days ago | 1 author, 1 change

```
static void Main(string[] args)
{
    string file;
    string building;
    string[] readings;
    List<Reading> Readings = new List<Reading>();

    file = args[0];
    building = file.Split('.')[0];
    readings = System.IO.File.ReadAllLines(file);
    foreach(string s in readings)
    {
        string[] separated;

        separated = s.Split(',');
        Readings.Add(new Reading(building, Int64.Parse(separated[0]), DateTime.Parse(separated[1]), Int64.Parse(separated[2])));
    }
    SqlConnection conn = new SqlConnection("Server=HOMESERVER; Database=FYPRactice; Integrated Security = True");
    try
    {
        conn.Open();
    }
    catch(Exception e)
    {
        Console.Write(e.ToString());
        Console.ReadLine();
    }
    string sql = "INSERT INTO Energy(Building, MeterID, TimeStamp, Value) VALUES(@Building, @Meter, @Date, @Value)";
    foreach(Reading r in Readings)
    {

        SqlCommand command = new SqlCommand(sql, conn);

        SqlParameter buildingParam = new SqlParameter("Building", SqlDbType.NVarChar);
        buildingParam.Value = r.Building;
        command.Parameters.Add(buildingParam);

        SqlParameter meterParam = new SqlParameter("Meter", SqlDbType.BigInt);
        meterParam.Value = r.MeterID;
        command.Parameters.Add(meterParam);

        SqlParameter dateParam = new SqlParameter("Date", SqlDbType.DateTime);
        dateParam.Value = r.datetime;
        command.Parameters.Add(dateParam);

        SqlParameter valueParam = new SqlParameter("Value", SqlDbType.BigInt);
        valueParam.Value = r.Value;
        command.Parameters.Add(valueParam);

        command.ExecuteNonQuery();
    }
```

Figure 4.10 – Text file parser program

Figure 4.10 shows the full source code for the program used to parse the text files. The program is a console application that can be started from the command line with the name of a text file passed in as a parameter. The program gets the name of building from the name of the file and then starts to parse the file line by line. Each line is split into meter ID, Timestamp, and meter reading value. Each line is then used to create a “Reading” object, which is then inserted into a list of “Reading” objects. Once the entire file has been parsed, a connection to the SQL database is made and the list of “Reading” objects is then iterated over and inserted one by one.



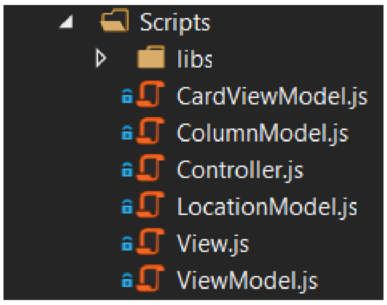
```
5 references | Matthew Frost, 41 days ago | 1 author, 1 change
class Reading
{
    public string Building;
    public Int64 MeterID;
    public DateTime datetime;
    public Int64 Value;

    1 reference | Matthew Frost, 41 days ago | 1 author, 1 change
    public Reading(string building, Int64 id, DateTime date, Int64 value)
    {
        MeterID = id;
        datetime = date;
        Value = value;
        Building = building;
    }
}
```

Figure 4.11 – The C# class for readings

4.5 Web application

The web application was created as a way of being able to manage the locations, to do this the app utilises the Google Maps JavaScript API (Google, 2017$_b$). The application was created using HTML, CSS, and JavaScript, the JavaScript libraries jQuery and Knockout JS were also used. jQuery was chosen as it ensures that the application will work on all browsers, it helps with event handling and AJAX requests which are necessary for interacting with the web services (Waldron, 2010). Knockout JS is a Model-View-ViewModel (MVVM) JavaScript library, this was chosen as it allows for data binding which makes the process of saving and retrieving locations easier.

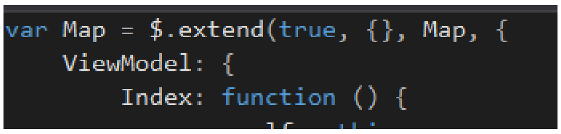
The entire application was structured in an MVVM architecture, doing this helps keep the application organised as it separates each component into its own file.



```
◢ Scripts
    ▷ libs
      CardViewModel.js
      ColumnModel.js
      Controller.js
      LocationModel.js
      View.js
      ViewModel.js
```

Figure 4.12 – structure of the web application

Figure 4.12 shows how the application's JavaScript is split into separate files. The view file is used just for the creation of the ViewModel on page load and the binding of events to buttons. The Model files are used to represent the data that will be sent and received by the application, each file contains a JavaScript object which describes the data that will be sent and retrieved. Each attribute in the object is assigned to a Knockout observable. An observable is a type of object used by Knockout JS to allow data binding, they update the ViewModel with the new value of the property as soon as it is changed without any extra code. The ViewModel file is where all the logic for the events is implemented, it interacts with the Model and Controller files. The ViewModel is used to keep the user interface up to date with the latest changes to the data. The Controller file is used to interact with the Node JS web services, it contains methods for creating/updating locations as well as deleting them.



```
var Map = $.extend(true, {}, Map, {
    ViewModel: {
        Index: function () {
```

Figure 4.13 – jQuery extend function

To be able to access functions and variables across all the files, the jQuery function "extend" was used to create a namespace. The function works by taking an object and either copying it if it already exists, if it does not then it creates a new object. The function takes a Boolean value which states whether the object should be a "deep copy", for example, if the object being copied contains an array then a deep copy will also copy the values in the array. The next parameter is the new object that is being copied into, in this case, a blank object is being created. The next parameter is the object that is being copied, the final parameter is any new properties to be merged into the new object.

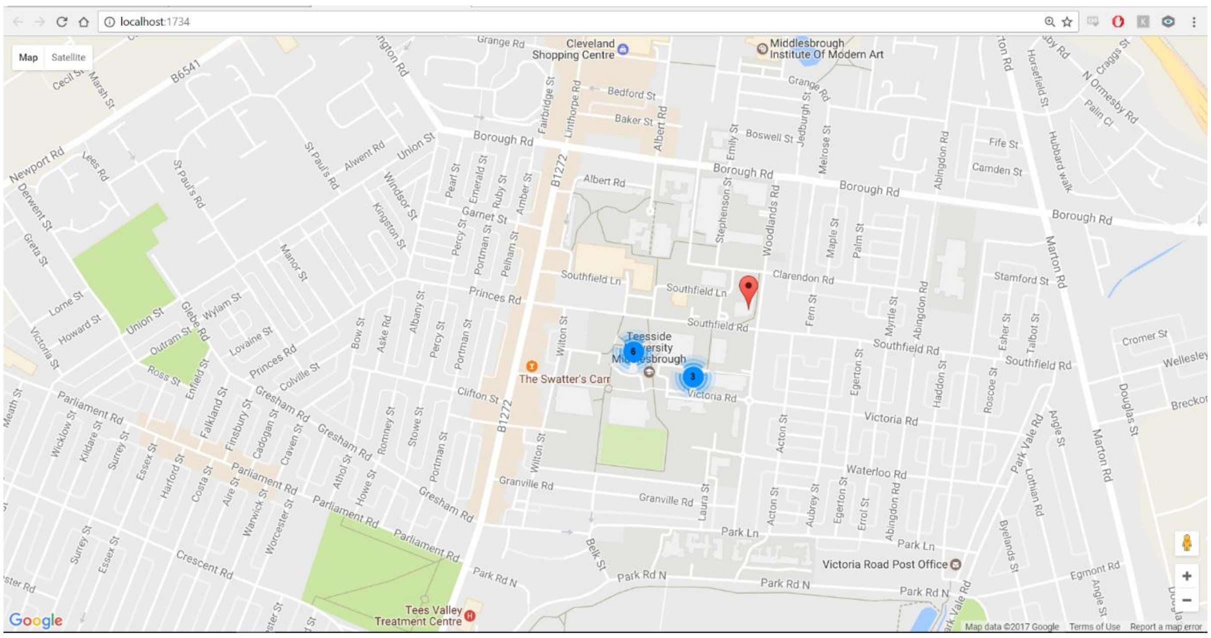


Figure 4.14 – web application overview

When the application is first opened, if there are any locations saved then they are displayed on the map. If there are multiple locations close together they are displayed in a group until the user zooms in closer. This is done using the "marker clustering" feature of the Google Maps API.

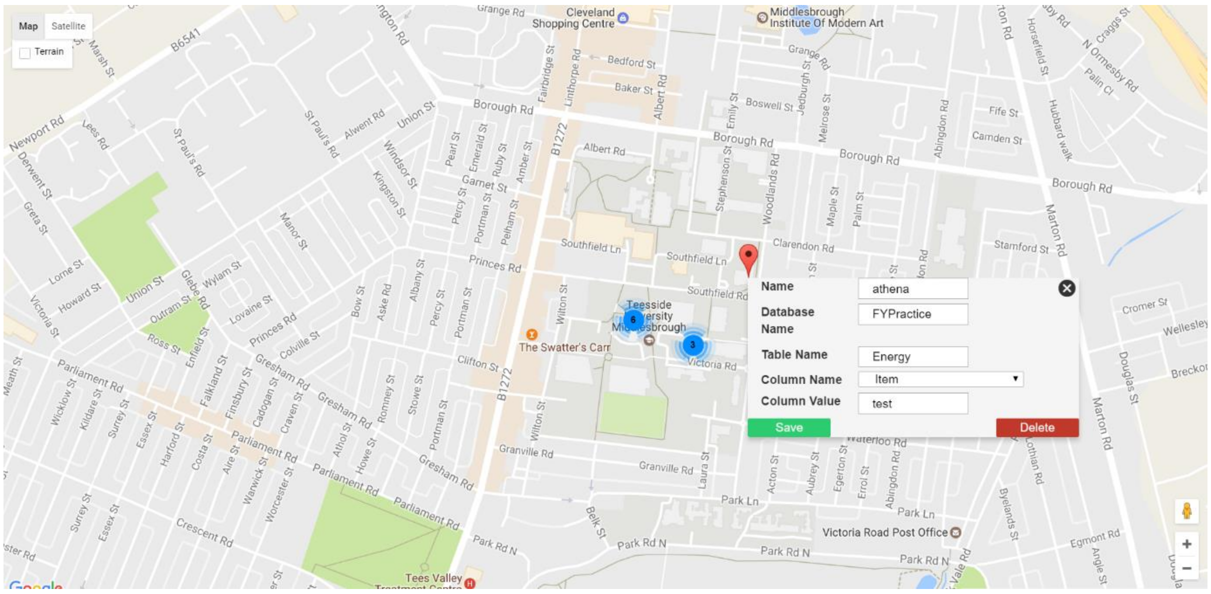
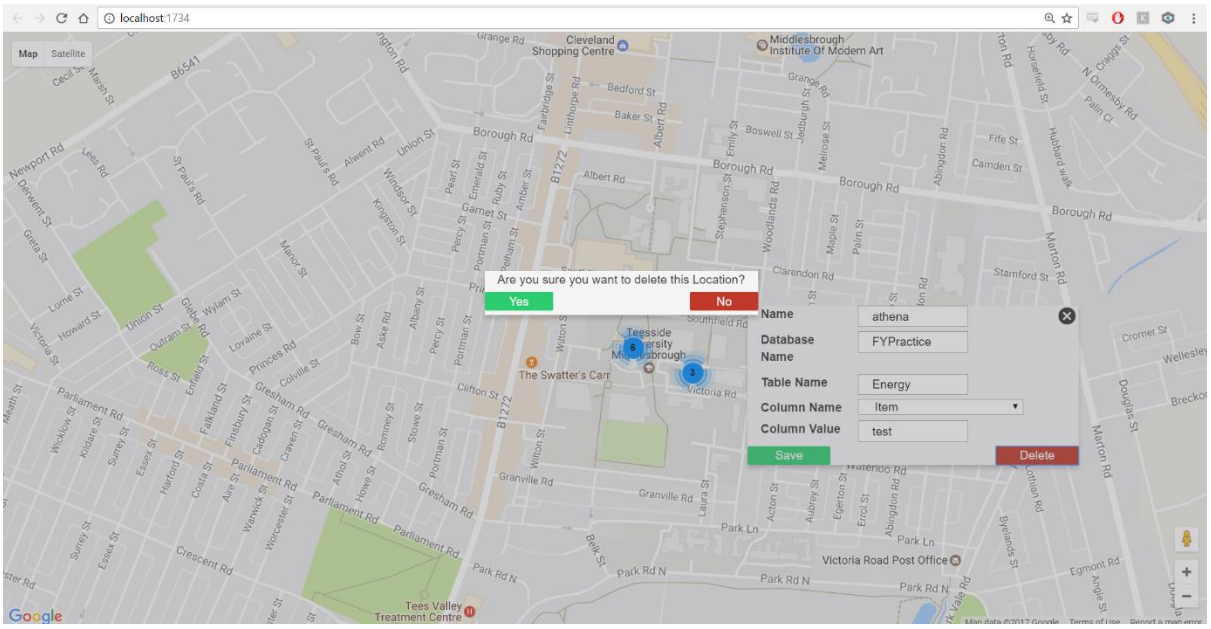

Name: athena
Database Name: FYPractice
Table Name: Energy
Column Name: Item
Column Value: test

Figure 4.15 – dialog when a location is clicked

When an existing location is clicked the dialog in figure 4.15 is displayed at that location. The dialog shows all the information associated with that location when it was created, this dialog can be used to edit and delete a location.


Are you sure you want to delete this Location?

Name: athena
Database Name: FYPractice
Table Name: Energy
Column Name: Item
Column Value: test

Figure 4.16 – confirmation dialog

To make sure that a location cannot be accidently deleted a dialog appears asking the user for confirmation

4.6 Android Application

The Android application is intended to be the main aspect of the project as it is the application that will be interacted with most frequently by the end users. When the application is being used, to the user it seems very simplistic as to what the app can do and how straight forward it is to use, however behind the scenes there is a lot going on with sensors and data management.

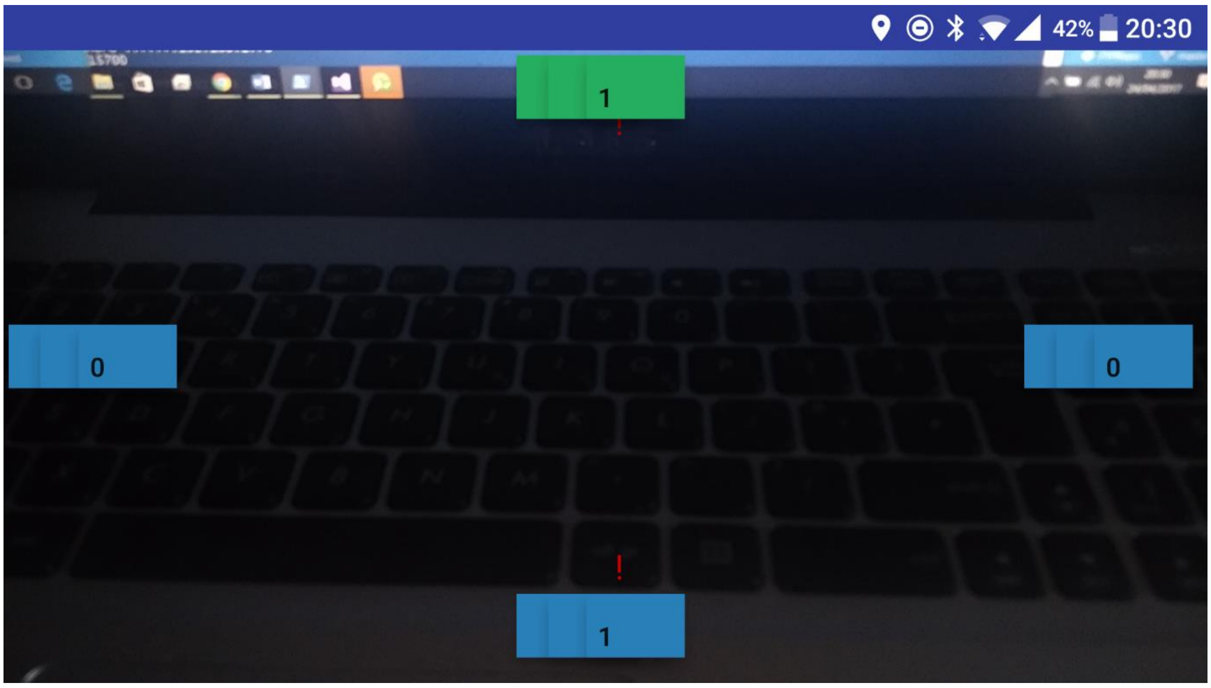
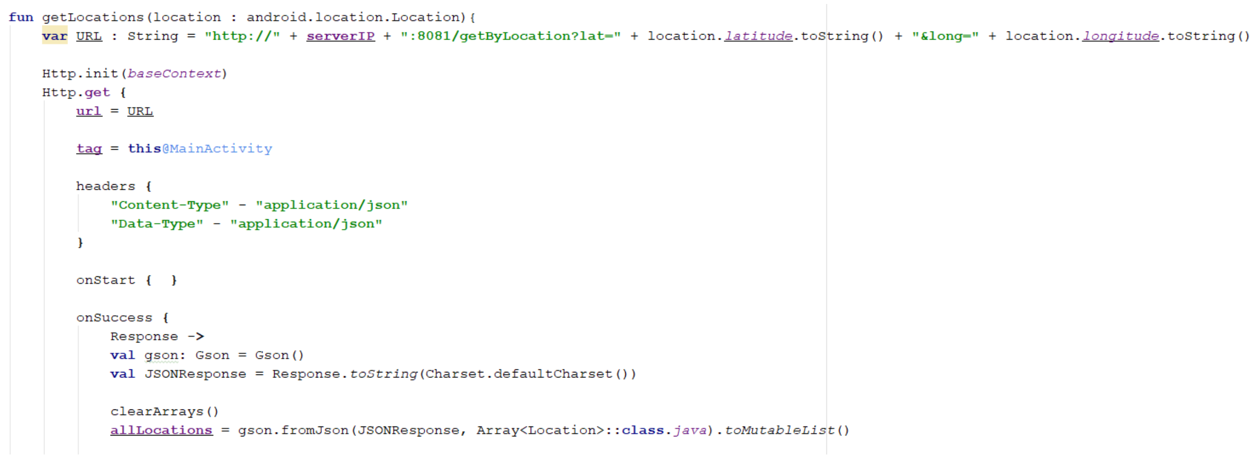


Figure 4.17 – opening screen of the Android application

The opening screen of the Android application has 4 buttons, each button represents a point on the compass (North, South, East and West). The buttons have a number on them which shows the number of locations in that direction. Only the top button can be clicked, this is done so that only the locations within the view of the camera can be viewed which leads to the augmented reality aspect of the application. Sometimes a button will have a red exclamation mark icon next to it, this is displayed when a location in that direction has anomalous data associated with it. Doing this brings the location to the user's attention and prompts them to investigate further.

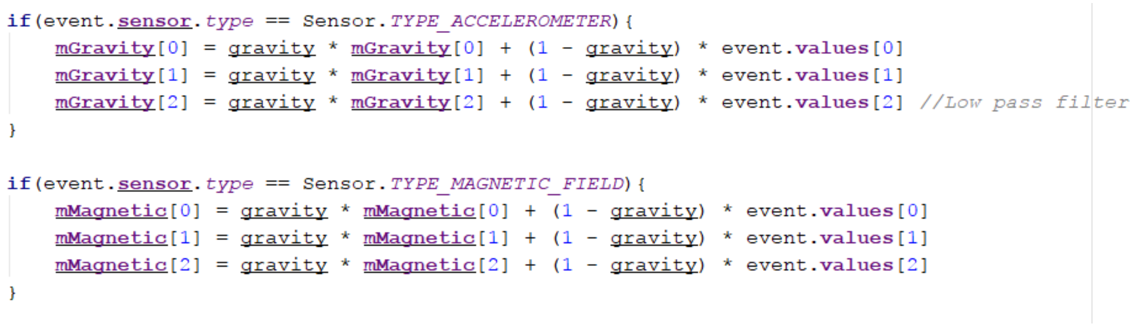
As soon as the application is opened calls to the location web service and anomaly web service are made, these calls are also made again when the location of the device changes to provide the user with the latest locations around them. The location of the device is obtained by using the Android Location API and calls to the web services are executed with a library called Kolley (Omerhe, 2016). Kolley is a Kotlin specific implementation of Android's standard HTTP request implementation, Volley. Kolley was included as there were some issues getting Volley to work in a Kotlin class.



```
fun getLocations(location : android.location.Location){
    var URL : String = "http://" + serverIP + ":8081/getByLocation?lat=" + location.latitude.toString() + "&long=" + location.longitude.toString()

    Http.init(baseContext)
    Http.get {
        url = URL

        tag = this@MainActivity

        headers {
            "Content-Type" - "application/json"
            "Data-Type" - "application/json"
        }

        onStart {  }

        onSuccess {
            Response ->
            val gson: Gson = Gson()
            val JSONResponse = Response.toString(Charset.defaultCharset())

            clearArrays()
            allLocations = gson.fromJson(JSONResponse, Array<Location>::class.java).toMutableList()
```

Figure 4.18 – Code snippet showing a HTTP request with Kolley

As the user turns around the buttons also turn with them so if the user turns ninety degrees clockwise then the values on the buttons will shift one place to the right, this is the mechanism that allows the user to view locations in each direction. This is achieved by using the device's compass, the algorithm for the compass was provided by Lutin (2015) and was originally written in Java but has been translated into Kotlin as part of the project. The compass works by continuously receiving values from the accelerometer and compass, each value is then put through a low pass filter to improve the accuracy. A low pass filter is used when data being received fluctuates a lot, this is known as "noisy" data and a low pass filter helps to "smooth out" that data.



```
if(event.sensor.type == Sensor.TYPE_ACCELEROMETER){
    mGravity[0] = gravity * mGravity[0] + (1 - gravity) * event.values[0]
    mGravity[1] = gravity * mGravity[1] + (1 - gravity) * event.values[1]
    mGravity[2] = gravity * mGravity[2] + (1 - gravity) * event.values[2] //Low pass filter
}

if(event.sensor.type == Sensor.TYPE_MAGNETIC_FIELD){
    mMagnetic[0] = gravity * mMagnetic[0] + (1 - gravity) * event.values[0]
    mMagnetic[1] = gravity * mMagnetic[1] + (1 - gravity) * event.values[1]
    mMagnetic[2] = gravity * mMagnetic[2] + (1 - gravity) * event.values[2]
}
```

Figure 4.19 – Low pass filter implementation

An additional feature that was added to the main screen is the ability to scan a Quick Response (QR) code, this allows the information to be encoded into a QR code so the data can be viewed without having to interact with the application. This could be especially useful if there are a lot of location within a small area or if it is not possible to get a GPS signal in the user's location. The QR code is an image that has been encoded with text by an online generator, the text is in the format of "locationID, locationName". For example, the QR code in figure 4.2 has the text "16, ITBuilding" behind it. As soon as the QR code is scanned and successfully parsed then the card displaying the data appears.

Figure 4.20 – example of QR code used

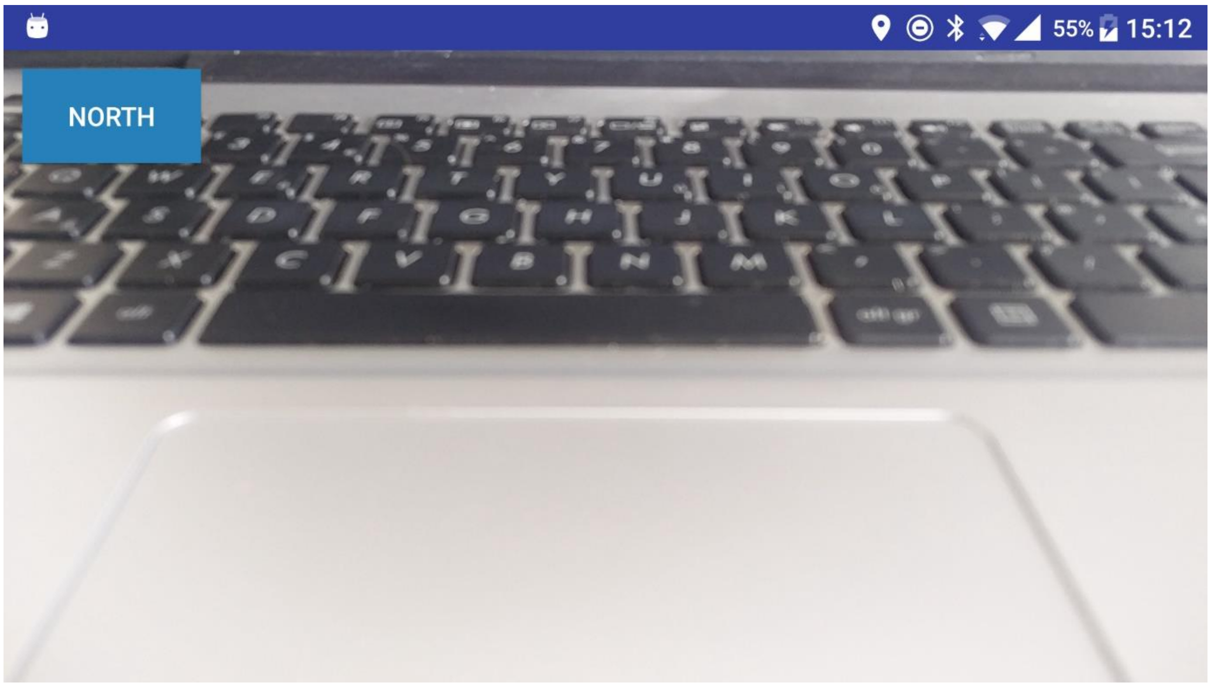


Figure 4.21 – user interface when direction has been selected

When the user selects a direction then the user interface displays each location on its own card which can then be selected to display the information tied to that location.

When a card is selected a call to the location data web service is made, if the data is successfully obtained then the data is cached in the devices SQLite database. Doing this means that if the next time the data for that location is requested and the network connection has been lost then the cached version can be returned instead.

Once the data has been successfully loaded in it is then analysed to determine if there is any anomalous data within the dataset. This is done using standard deviation, which is a mathematical technique used to analyse the variance in data. There are two methods of calculating the standard deviation, there is the “population” method and the “sample” method. The “population” method should be used when analysing a full dataset that is not going to change. The “sample” method should be used when the dataset will change or if the dataset is very large. In this implementation of the application only the “population” method is used as the data being analysed is not going to change. A data point will be marked as anomalous if it lies more than three standard deviations away from the mean. Three standard deviations was chosen because as shown by Whitney (no date), statistically ninety nine percent of a normalised data set will lie within three standard deviations of the mean so anything greater than that could be considered an anomaly.

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Figure 4.22 – Population standard deviation

$$\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i-\bar{x})^2}$$

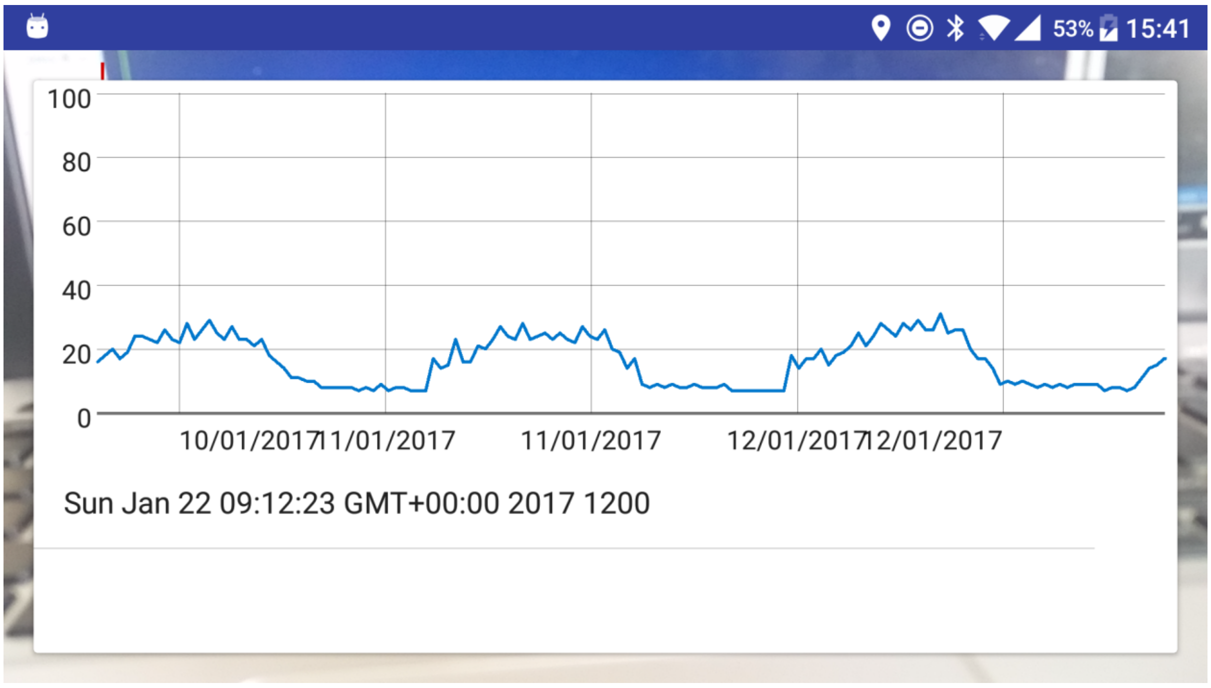
Figure 4.23 – Sample standard deviation

Finally, once the data has been cached and analysed it is displayed on a graph, to create the graph a library by Jonas Gehring was used. The library allowed the creation of a line graph to display the data associated with the selected location. The graph is also scrollable which means that large amounts of data can be displayed without it being cramped on a small screen. There were some issues with getting the graph to scroll properly while have dates on the x-axis as the documentation only showed how to enable scrolling with a numerical x-axis. However, after getting in contact with the Gehring, he explained how to do this. It was done by converting the date to the number of milliseconds since January 1st 1970 and then doing the same but adding 3 days to create the required viewport.

```
graph.getViewport().setMinX(locationData.get(0).Timestamp.getTime().toDouble())
graph.getViewport().setMaxX(locationData.get(0).Timestamp.getTime().toDouble() + (3 * 24 * 60 * 60 * 1000)) //adding 3 days
```

Figure 4.24 – solution to enabling scrolling with dates.

Once the data has been analysed for anomalies it is added to a list that is displayed beneath the graph, each entry in the list shows the date and time of the anomaly as well as the value of the data point. Originally this was all the functionality that was going to be enabled but since there was plenty of time left it was decided to allow users to comment on an anomaly. When the user taps on an entry in the anomaly list, a dialog is displayed which allows them to enter a comment, this comment is then stored in the database. This could be useful for the user as it would allow them to investigate the cause of the anomaly and then leave a comment against it so that anyone else investigating can see that it has been resolved or any other history behind it.



Sun Jan 22 09:12:23 GMT+00:00 2017 1200

Figure 4.25 – graph with location data and anomaly value list

4.7 Deployment

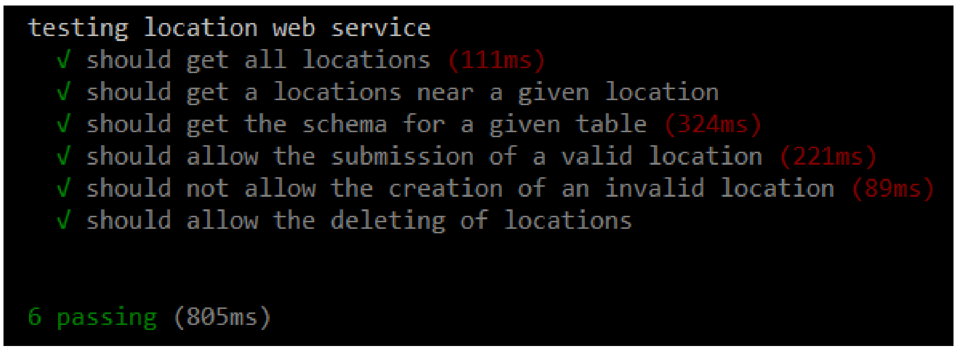
Because the web services are written in Node JS it means that they can be run on any server that can run Node JS. Node JS can be downloaded for Windows, Mac OSX, and Linux so the web services can be deployed to any machine running one of these operating systems. Also as the web services are not very CPU intensive, if there is a low demand on the services it will be able to run on a low powered system. Node JS can handle hundreds of thousands of active connections (Caustik, 2012) at a time which is well above the initial target of the project so it would be easier to scale the hardware before the software would need to be scaled. Scaling a system by upgrading the machine that runs the software is known as scaling vertically, adding more machines to a system is known as scaling horizontally (Beaumont, 2014). In this case, scaling vertically would be easier and more effective as there would be no extra configuration needed. During development, the web services were all run simultaneously on a single machine with an Intel Core 2 Duo,3GB DDR2 RAM and running Windows Server 2012. With this set up there were no issues, the only configuration that was needed was allowing each web service through the windows firewall which allowed other devices to connect to the right port.

Since the 2017 release of SQL Server, it can now be deployed on a Linux machine running either Red Hat or Ubuntu. However, earlier versions require a Windows machine to deploy SQL Server. SQL Server does not need to be on the same machine or even the same network for the web services to connect, so it could be possible to host the SQL database on a platform such as Azure.

The web app does not need any specific hardware or operating system requirements either as it only consists of HTML, JavaScript, and CSS. It could be deployed on a Windows server using Internet Information Services (IIS) or a Linux server with Apache. Alternatively, any other online website host would be able to support the application.

5. Testing & Evaluation

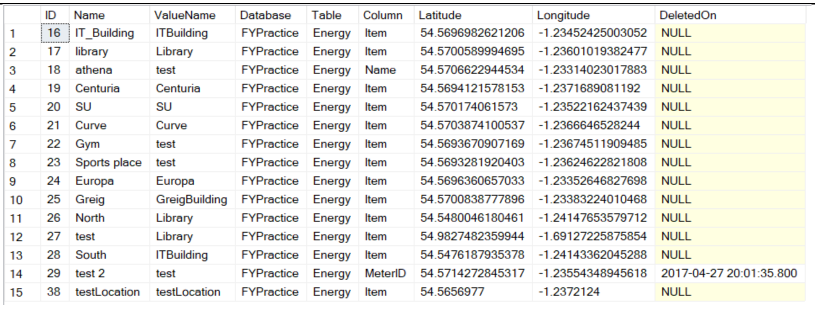
To ensure that a robust and complete product has been delivered it is important that the entire project has been thoroughly tested. To do this all the web services have been unit tested, this allows all the features to be automatically tested. To run the unit tests Mocha, Chai and Supertest are used, these are all JavaScript libraries. Mocha is used to run the unit tests, Supertest is used for the HTTP requests, Chai is used for assertion testing e.g response should have a length. Both the web application and Android application were tested manually as the structure and technologies used did not work well with automated unit tests. Testing was carried out at the end of each sprint to ensure that everything was working as intended before moving on to the next feature.



```
testing location web service
  √ should get all locations (111ms)
  √ should get a locations near a given location
  √ should get the schema for a given table (324ms)
  √ should allow the submission of a valid location (221ms)
  √ should not allow the creation of an invalid location (89ms)
  √ should allow the deleting of locations


6 passing (805ms)
```

Figure 5.1 – unit tests being run

5.1 Location web service testing

Test	Expected Result	Test Data
Get all locations	Should return all locations from the database that have not been deleted. Should get 14 locations	

Get locations near a given location	Should return all locations within 100m of the given location. Should get 9 locations	Latitude = 54.569467
		Longitude = -1.2342727

Get the schema for a given table	Should return the column names for the table provided	Database: FYPractice Table: Location
Should allow the submission of a valid location	Should not receive an error if a valid location is submitted	LocationName: testLocation, ColumnValue: testLocation. Database: FYPractice Table: Energy Column: Item, Latitude: 54.5656977 Longitude: -1.2372124
Should not allow the submission of an invalid location	Should return a HTTP error if an invalid location is submitted	LocationName: testLocation ColumnValue: testLocation Table: Energy Column: Item Latitude: 54.5656977

	ID	Name	ValueName	Database	Table	Column	Latitude	Longitude	DeletedOn
1	16	IT_Building	ITBuilding	FYPractice	Energy	Item	54.5696982621206	-1.23452425003052	NULL
2	17	library	Library	FYPractice	Energy	Item	54.5700589994695	-1.23601019382477	NULL
3	18	athena	test	FYPractice	Energy	Name	54.5706622944534	-1.23314023017883	NULL
4	19	Centuria	Centuria	FYPractice	Energy	Item	54.5694121578153	-1.2371689081192	NULL
5	20	SU	SU	FYPractice	Energy	Item	54.570174061573	-1.23522162437439	NULL
6	21	Curve	Curve	FYPractice	Energy	Item	54.5703874100537	-1.2366646528244	NULL
7	22	Gym	test	FYPractice	Energy	Item	54.5693670907169	-1.23674511909485	NULL
8	23	Sports place	test	FYPractice	Energy	Item	54.5693281920403	-1.23624622821808	NULL
9	24	Europa	Europa	FYPractice	Energy	Item	54.5696360657033	-1.23352646827698	NULL
10	25	Greig	GreigBuilding	FYPractice	Energy	Item	54.5700838777896	-1.23383224010468	NULL
11	26	North	Library	FYPractice	Energy	Item	54.5480046180461	-1.24147653579712	NULL
12	27	test	Library	FYPractice	Energy	Item	54.9827482359944	-1.69127225875854	NULL
13	28	South	ITBuilding	FYPractice	Energy	Item	54.5476187935378	-1.24143362045288	NULL
14	29	test 2	test	FYPractice	Energy	MeterID	54.5714272845317	-1.23554348945618	2017-04-27 20:01:35.800
15	38	testLocation	testLocation	FYPractice	Energy	Item	54.5656977	-1.2372124	NULL

Should allow the deletion of a location	Should return a HTTP success code if the location has been deleted	ID: 29

Table 5.1 – unit tests for the Location web service

5.2 Location Data web service testing

Test	**Expected Result**	**Test Data**
Get all data for a given location	Should return all data for the location provided. Should return 911 rows	All data associated with the ITBuilding location

Table 5.2 – unit test for the Location Data web service

5.3 Anomaly web service testing

Test	**Expected Result**	**Test Data**
Get all anomalies associated with locations in the nearby area	Should return 4 anomalies for the given latitude and longitude. Should return 4 anomalies	Latitude: 53.569467 Longitude: -1.2342727
Return 0 anomalies	Should return 0 anomalies when a location has no anomalies in the given area	Latitude: 54.569467 Longitude: -1.2342727
Submission of valid resolution	Should allow the submission of a valid resolution	LocationID: 16 Resolution: ‘unit test resolution’ Value: 9999
Submission of an invalid resolution	Should not allow the submission of an invalid resolution	Resolution: ‘no LocationID’ Value: 1233

Table 5.3 – unit tests for the Anomaly web service

5.4 Web application testing

Test	**Expected Result**	**Test Data**
Get Locations from database	Should retrieve locations from the location database and display them as points on the map	Locations stored in the location database table
Clicking on a location	Clicking on a location should display a dialog with the locations information correctly filled out	Name: Europa Database Name: FYPractice, Table Name: Energy, Column Name: Item, Column Value: Europa
Double click on map	Double clicking on the map should bring up the dialog to create a new location	-
Should be able to save new location	Correctly filling out the new location dialog should allow a location to be created	Name: locationTest Database: FYPractice, Table Name: Energy,

		Column Name: Item. Column Value: Test
Editing a location	Changing the information on a location dialog and clicking save should commit those changes to the database	Name: locationTest2, Database: FYPractice, Table Name: Energy, Column Name: Item, Column Value: Test
Delete location	Clicking the delete button on the location dialog should display a confirmation dialog. If the deletion is confirmed, then the location should be removed from the map and database.	-

Table 5.4 – tests for the web application

5.5 Android application testing

Test	**Expected Result**	**Test Data**
Camera opens on application start	When the application is first opened, the camera preview is shown	-
Camera displayed when application is resumed	If the app is paused in the background and resumed, then the camera should still be displayed	-
GPS acquired on application start	When the application is first opened, the GPS signal should be acquired. This can be shown by the GPS icon in the status bar	-
GPS should be turned off when the application is closed	When the application is closed then the GPS icon should be removed from the status bar.	-
Application should get a list of all locations near by	When the application is first opened it should execute a HTTP request and retrieve a list of locations within one hundred meters. Should get two locations	Latitude: 54.5428755, Longitude: -1.238414
Compass sensor	As the device is rotated the value reported by the compass sensor should change	-
Correct compass value for North	If the device is facing North then the compass	North: 310 – 49

	reading should have the correct value	
Correct compass value for East	If the device is facing East then the compass reading should have the correct value	East: 50 – 129
Correct compass value for South	If the device is facing South then the compass reading should have the correct value	South: 130 – 229
Correct compass value for West	If the device is facing West then the compass reading should have the correct value	West: 230 – 309
Binding data to each direction	When the locations have been obtained they should be categorised into the correct location. one location should be in the North, one location should be in the South	Latitude: 54.5428755, Longitude: -1.238414 {Name: North, Latitude: 54.5480046180461, Longitude: -1.24147653579712}, {Name: South, Latitude: 54.5476187935378, Longitude: -1.24143362045288}
Mapping data to UI	Once the data has been categorised it should be placed in the correct position on the user interface. If facing North one should be at the top, one should be	One North, one South
UI changes with direction	As the devices turns the UI should change to reflect the direction of each location relative to the user	-
Clicking the top button	Clicking the top button on the user interface should display all of the locations associated with the direction being faced	-
Clicking a location button	Clicking on a button associated with a location should get the data associated with it	If facing North: Latitude: 54.5428755, Longitude: -1.238414 {Name: North, Latitude: 54.5480046180461, Longitude: -1.24147653579712}

Location data graph	Once the data has been obtained it should be displayed on a graph	Data associated with selected location
Scrolling graph	The location data graph should scroll horizontally	-
Caching data	If the device is online then the data retrieved from the web service should be cached in a local SQLite database	Data associated with selected location
Retrieving data from local database	If the device does not have network connectivity then the local database should be checked to see if data for that location exists. If it exists then it should be retrieved and put on the graph	Data associated with selected location
Analysing data	Once the data has been retrieved it should be analysed to see if there are any anomalies	Data associated with selected location
Display anomalies in a list	Once the data has been analysed, if any anomalies have been found they should be displayed in a list below the graph	-
Clicking anomaly in the list	If there are any anomalies in the list, when one is clicked then a dialog should be displayed	Selected anomaly
Dialog cancel button	If the cancel button is clicked then the dialog should close	-
Dialog submit button	If the dialog text field has been filled in and the submit button is clicked then the anomaly resolution should be submitted to the web service	Resolution: “test resolution”, LocationID: 16, Value: 1409
Submit resolution with not text	If the submit button has been clicked but there is no text in the dialog text field then it should not be submitted to the web service	LocationID: 16, Value: 1409

Table 5.5 – Testing for the Android application

6. Conclusion

Looking back at the objectives of the project every one of them has been met and a fully functional product has been created. Extra functionality has also been added in the form of anomaly notifications and reporting. Considering these points, it is reasonable to say that the project has been a success.

Some issues with the project as it stands is that there is currently no data source that is constantly updating, the data being used currently is from January 10th 2017 to January 31st. This means that every time the application is opened it will always return the same results if the user is in the same location. However, if the application was to be used by a real client with their own data source it would not be an issue.

The web and Android applications could also be refactored as to allow automated unit testing to be more easily implementable. Doing this would help improve both the stability and maintainability of the software.

With the project now finished the next steps would be to go back to Sabisu to demonstrate how the project has developed and see if they would want to take it further and test it in the field. The application would need further polish and integration with Sabisu before it could be deployed as a professional application.

6.1 Future improvements

Although all the requirements set out at the start of the project were met there are still a lot of improvements that could be made to the Android application and web application.

The web application could be improved by changing how the dialogs work. At the end of the development period for the entire project, it was discovered that there is a feature in the Google Maps API that allows a dialog to be tied to a location (Google, 2017_c). In the current implementation, the dialogs are tied to a pixel on the map which works as intended if the user does not scroll.

If the application were to continue development in conjunction with Sabisu then it would be good to implement some of the requirements that were ruled out from the initial specification. This would include features such as Sabisu log in and integration with existing Sabisu APIs for better reporting.

In terms of improving on existing features, it would be desirable to change the anomaly notification feature to run as a background service on the Android device. Doing this would allow the user to be automatically alerted as soon as they entered an area that had a location with an anomaly nearby.

Another consideration is the accessibility of the application. If the user is colour blind, then the use of red and green coloured buttons could be a hindrance to them especially since the most common form of colour blindness is red-green colour blindness (Colour Blind Awareness, no date). This is something that was over looked as it was not a requirement in the initial project specification.

An issue that still stands with the application is the time taken when loading the graph to show data. This is caused by having to make a HTTP request to get the data, analysing the retrieved data for anomalies and storing the retrieved data in a SQLite database. This could perhaps be reduced several ways, the anomalies could be calculated on a server and retrieved via a web service, this would reduce the amount of calculations done on the user's device. The caching procedure could also be leveraged further to improve performance, if the data stored in the local database was relatively recent then perhaps that could be used, as reading data from the database on the device is significantly faster than getting it from the web service.

7. Reflection

Overall I believe that the project has been a success, everything was completed on time or before and a fully functional product has been developed while following an Agile methodology. Development of the project went on ahead of time compared to the initial project plan this meant that more time could be spend on certain aspects of the project.

One aspect of the project that did not go well was the design of the class diagrams and other design related documents. They were mainly put to the side and development started without them being fully finished. Once I had the structure of how everything was going to link together and I knew what components everything needed I tended to jump straight into development. I think this was because I felt as though I had a solid idea of what I wanted to include in each application and how it would all work. Furthermore, due to the project being ahead of time and extra features being added, the initial design documents do not contain items that are in the final product.

Developing the project in an agile method helped me understand how software is developed in industry. Working on a feature for a week and then having a review meeting at the start of the next week helped as it allowed me to talk about what had been achieved and any problems that I had encountered. I found that simply talking to someone about an issue or bug I was having would help me come up with a solution as I was describing it. The project started by strictly following the agile methodology but towards the end it started to break down, mainly as the report was beginning to be written. I think that development went on a bit longer than expected since new features were being added so this slightly clashed with the writing of the report.

The project also helped improve my technical skills as a developer. For example, I feel a lot more confident in planning a full system architecture, understanding how everything links together and the protocols behind it all. It also helped improve my Android development skills allowing me to learn techniques that I had not used before such as making HTTP requests. The project also gave me the opportunity to research and implement some technologies that I had heard about but not had chance to consider. For example, I had heard a lot about Mongo DB and No SQL databases but it was not until this project that I had chance to investigate them myself.

In terms of how the project was implemented, there were a few things that would be changed if it was to be attempted again. Firstly, the Android application would be written in Java and not Kotlin, Kotlin was used as it was presented as a better approached to Android development. However, in practice, it felt as though it made some aspects of the development more difficult and that its apparent features became more of a hindrance. For example, Kotlin does not allow variables to be null however there is a way around this, by declaring variables with the 'lateinit' keyword. This feature was used a lot in the application for global variables and it felt as though Kotlin was not being used as it should be. Furthermore, the majority of Android documentation and support is in Java, this meant that either some classes were implemented purely in Java or the Kotlin code was just attempting to mimic something written in Java.

An issue that was encountered during development was that it appeared that the Android application was draining my phones battery even when it had been fully closed. There was a noticeable decrease in battery drain rate when the application had been uninstalled, it was clear that something was not being removed by the operating system that kept it running in the background. After some investigation, it could be seen in the Android debug monitor that the camera was still attempting and failing to send data to the application, this was what was causing the battery drain as constant camera usage is heavy on the battery. This was solved by changing the camera API that was being used, the original method being used had been marked as deprecated by

Google. So, with the new approach more modern and better supported code was being used and the battery drain issue had been removed.

8. Bibliography

Meier (2012) *Professional Android 4 application development*

Deitel, Deitel, and Wald (2015) *Android for programmers: an App-driven approach*

Bott (2005) *Professional Issues in Information Technology*

Shalloway, Trott (2005) *Design Patterns Explained*

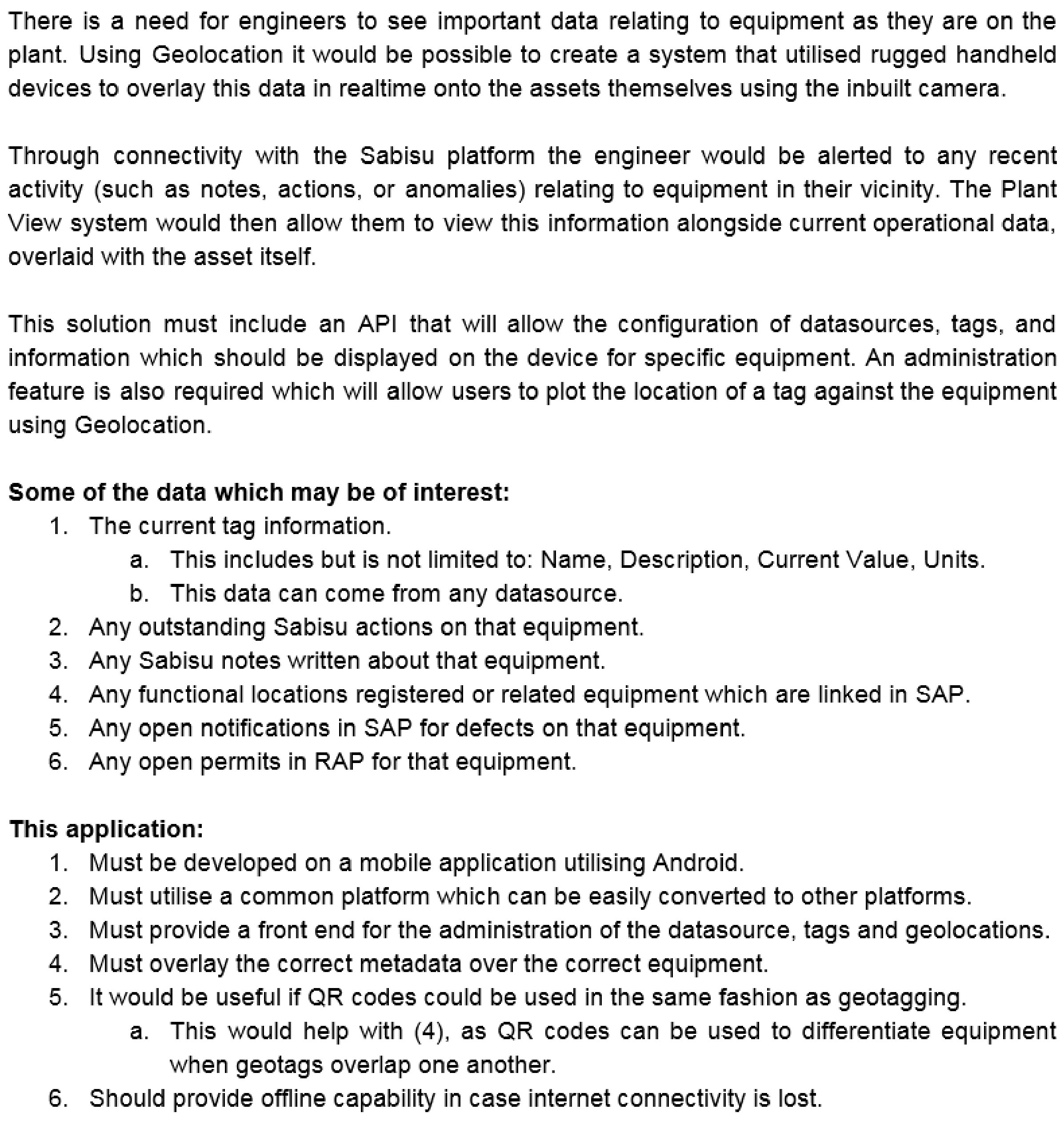
9. References

Agarwal, N. (2011) *What is Android operating system? A beginners read.* Available at: http://www.thewindowsclub.com/what-is-android-operating-system-a-beginners-read
Agile Business Consortium (2008) *MoSCoW Prioritisation.* Available at: https://www.agilebusiness.org/content/moscow-prioritisation-0
Beaumont, D (2014) *How to explain vertical and horizontal scaling in the cloud.* Available at: https://www.ibm.com/blogs/cloud-computing/2014/04/explain-vertical-horizontal-scaling-cloud/
British Computing Society (2017) *BCS Code of Conduct.* Available at: http://www.bcs.org/category/6030
Butters, K. (2014) *UX Dilemma: Red Button vs. Green Button.* Available at: https://www.sitepoint.com/button-ux-red-green/
Caustik (2012) *Scaling Node.js to 100k concurrent connections.* Available at: http://blog.caustik.com/2012/04/08/scaling-node-js-to-100k-concurrent-connections/
Colour Blind Awareness (no date) *Types of Colour Blindness.* Available at: http://www.colourblindawareness.org/colour-blindness/types-of-colour-blindness/
Factor, P. (2013) *Primary Key Primer for SQL Server.* Available at: https://www.simple-talk.com/sql/learn-sql-server/primary-key-primer-for-sql-server/
Fowler, M. (2014) *Microservices.* Available at: https://martinfowler.com/articles/microservices.html
Gentz, M., Rabeler, C. (2017) *NoSQL vs SQL.* Available at: https://docs.microsoft.com/en-us/azure/documentdb/documentdb-nosql-vs-sql
Google (2017$_a$) *Material Design Guidelines.* Available at: https://material.io/guidelines/material-design/introduction.html#introduction-principles
Google (2017$_b$) *Google Maps JavaScript API.* Available at: https://developers.google.com/maps/documentation/javascript/
Google (2017$_c$) *From Info Windows to a Database: Saving User-Added Form Data.* Available at: https://developers.google.com/maps/documentation/javascript/info-windows-to-db
Lutin, V. (2015) *Compass.* Available at: https://github.com/iutinvg/compass
Node.js Foundation (2017$_a$) *Node.js.* Available at: https://nodejs.org/en/
Node.js Foundation (2017$_b$) *Express.* Available at: https://expressjs.com/
Omerhe (2016) *Kolley.* Available at: https://github.com/ohmerhe/Kolley
Open Source Initiative (no date) *The MIT License.* Available at: https://opensource.org/licenses/MIT
Pilsbury, M (2017) *Tedious.* Available at: http://tediousjs.github.io/tedious/

Project Methodologies (no date) Available at: http://1.https://www.wrike.com/project-management-guide/methodologies/
PTC Inc (2017) *Vuforia highlights*. Available at: https://www.vuforia.com/
Sims (2016) *Java vs C app performance.* Available at: http://www.androidauthority.com/java-vs-c-app-performance-689081/
Srivastava, B. (2017) *What is Agile methodology?* Available at: https://www.linkedin.com/pulse/what-agile-methodologydisadvantage-waterfall-model-bikesh-srivastava
Unity Technologies (2017) *Unity - game engine*. Available at: https://unity3d.com/
Waldron, R. (2010) *What cross-browser issues does jQuery solve?* Available at: https://www.quora.com/What-cross-browser-issues-does-jQuery-solve
Whitney, T (no date) *Anomaly Detection.* Available at: http://trevorwhitney.com/data_mining/anomaly_detection

10. Appendices

Project Specification



There is a need for engineers to see important data relating to equipment as they are on the plant. Using Geolocation it would be possible to create a system that utilised rugged handheld devices to overlay this data in realtime onto the assets themselves using the inbuilt camera.

Through connectivity with the Sabisu platform the engineer would be alerted to any recent activity (such as notes, actions, or anomalies) relating to equipment in their vicinity. The Plant View system would then allow them to view this information alongside current operational data, overlaid with the asset itself.

This solution must include an API that will allow the configuration of datasources, tags, and information which should be displayed on the device for specific equipment. An administration feature is also required which will allow users to plot the location of a tag against the equipment using Geolocation.

Some of the data which may be of interest:

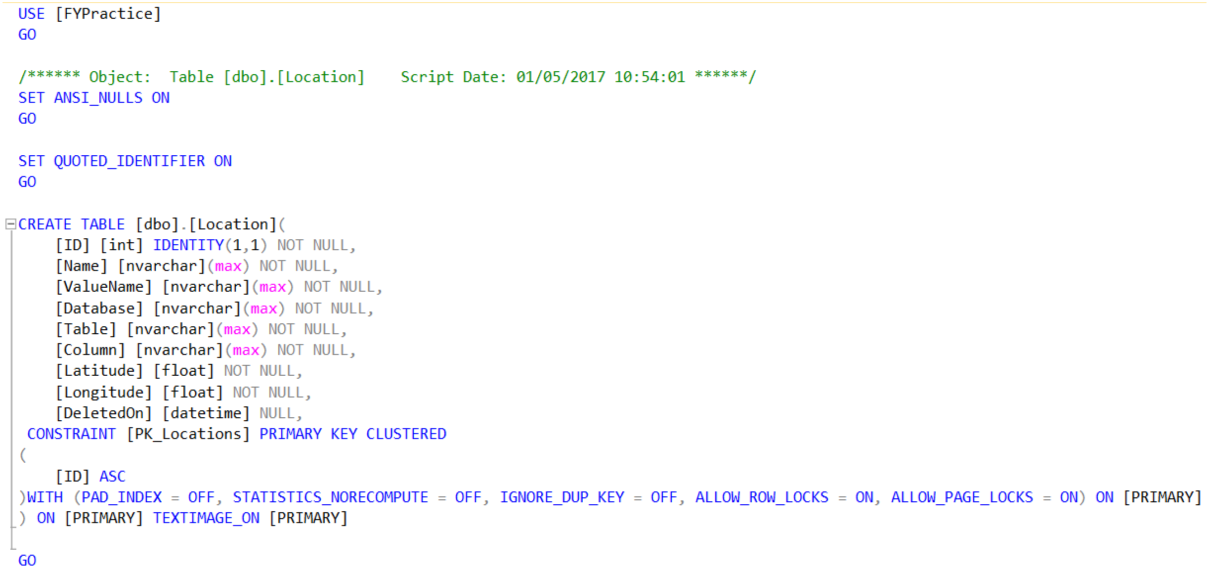
1. The current tag information.
    a. This includes but is not limited to: Name, Description, Current Value, Units.
    b. This data can come from any datasource.
2. Any outstanding Sabisu actions on that equipment.
3. Any Sabisu notes written about that equipment.
4. Any functional locations registered or related equipment which are linked in SAP.
5. Any open notifications in SAP for defects on that equipment.
6. Any open permits in RAP for that equipment.

This application:

1. Must be developed on a mobile application utilising Android.
2. Must utilise a common platform which can be easily converted to other platforms.
3. Must provide a front end for the administration of the datasource, tags and geolocations.
4. Must overlay the correct metadata over the correct equipment.
5. It would be useful if QR codes could be used in the same fashion as geotagging.
    a. This would help with (4), as QR codes can be used to differentiate equipment when geotags overlap one another.
6. Should provide offline capability in case internet connectivity is lost.

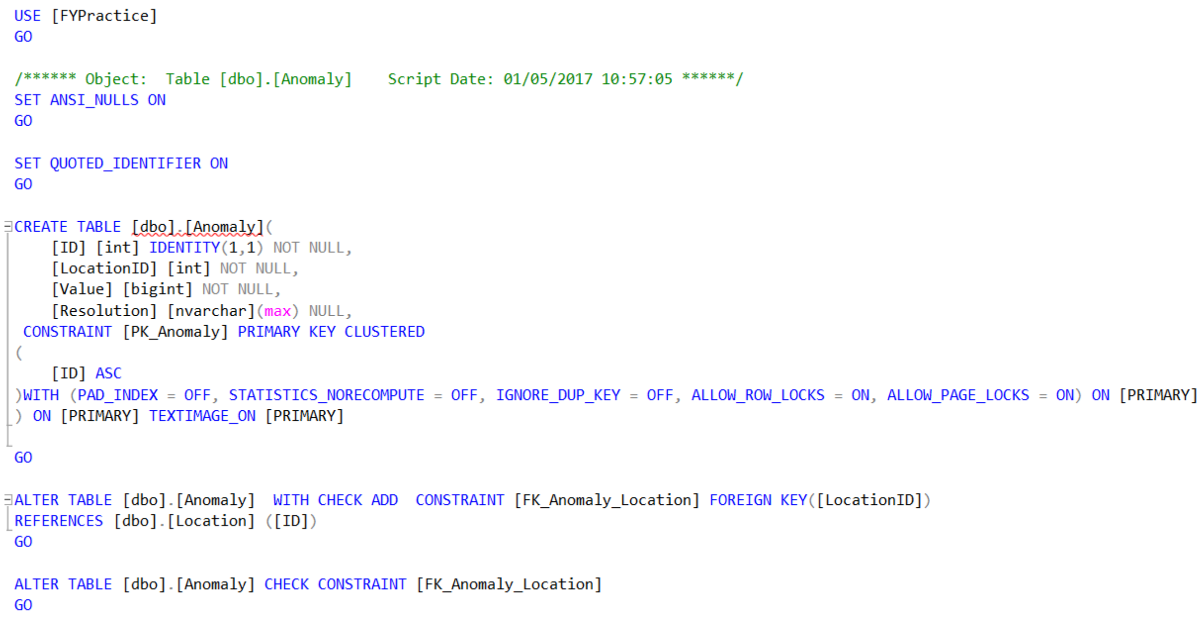
Figure x – the initial project specification provided by Sabisu

SQL Server Code



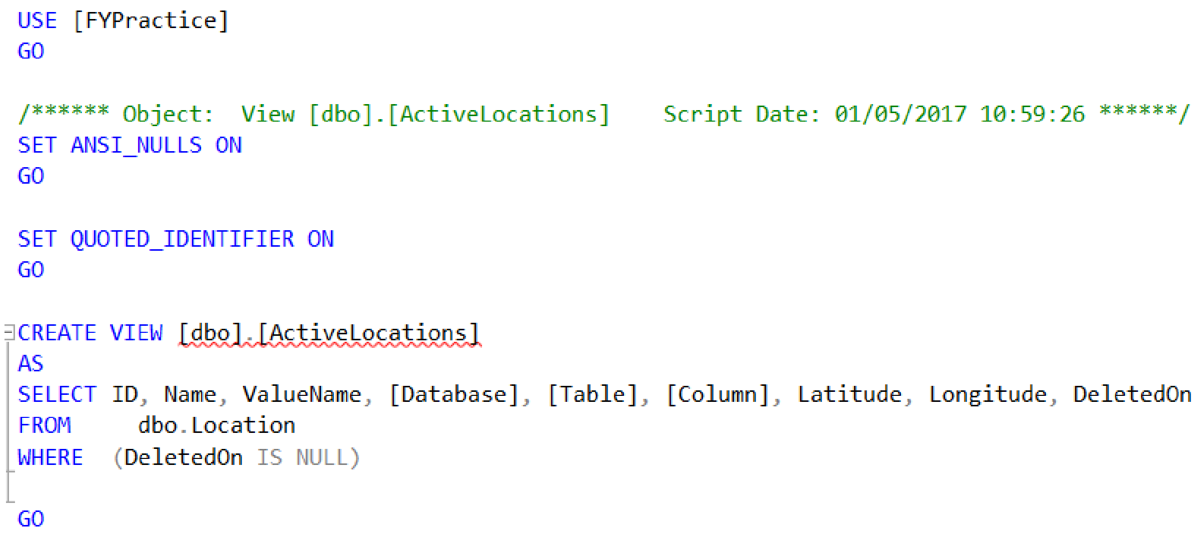
```
USE [FYPractice]
GO

/****** Object:  Table [dbo].[Location]    Script Date: 01/05/2017 10:54:01 ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Location](
    [ID] [int] IDENTITY(1,1) NOT NULL,
    [Name] [nvarchar](max) NOT NULL,
    [ValueName] [nvarchar](max) NOT NULL,
    [Database] [nvarchar](max) NOT NULL,
    [Table] [nvarchar](max) NOT NULL,
    [Column] [nvarchar](max) NOT NULL,
    [Latitude] [float] NOT NULL,
    [Longitude] [float] NOT NULL,
    [DeletedOn] [datetime] NULL,
 CONSTRAINT [PK_Locations] PRIMARY KEY CLUSTERED
(
    [ID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

GO
```

Figure x – SQL code for the Location table



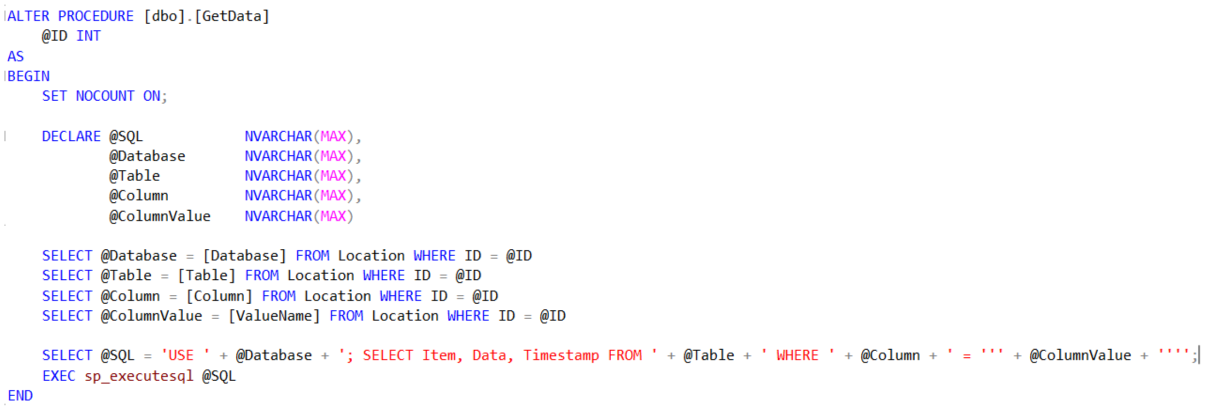
```
USE [FYPractice]
GO

/****** Object:  Table [dbo].[Anomaly]    Script Date: 01/05/2017 10:57:05 ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Anomaly](
    [ID] [int] IDENTITY(1,1) NOT NULL,
    [LocationID] [int] NOT NULL,
    [Value] [bigint] NOT NULL,
    [Resolution] [nvarchar](max) NULL,
 CONSTRAINT [PK_Anomaly] PRIMARY KEY CLUSTERED
(
    [ID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

GO

ALTER TABLE [dbo].[Anomaly]  WITH CHECK ADD  CONSTRAINT [FK_Anomaly_Location] FOREIGN KEY([LocationID])
REFERENCES [dbo].[Location] ([ID])
GO

ALTER TABLE [dbo].[Anomaly] CHECK CONSTRAINT [FK_Anomaly_Location]
GO
```

Figure x – SQL code for the Anomaly table



```sql
USE [FYPractice]
GO

/****** Object:  View [dbo].[ActiveLocations]    Script Date: 01/05/2017 10:59:26 ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE VIEW [dbo].[ActiveLocations]
AS
SELECT ID, Name, ValueName, [Database], [Table], [Column], Latitude, Longitude, DeletedOn
FROM      dbo.Location
WHERE   (DeletedOn IS NULL)

GO
```

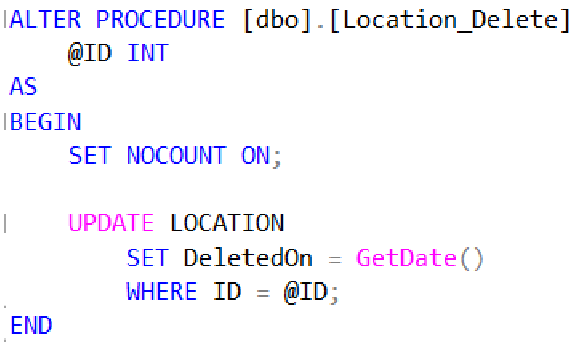
Figure x – SQL code for the ActiveLocations view



```sql
ALTER PROCEDURE [dbo].[GetData]
    @ID INT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @SQL            NVARCHAR(MAX),
            @Database       NVARCHAR(MAX),
            @Table          NVARCHAR(MAX),
            @Column         NVARCHAR(MAX),
            @ColumnValue    NVARCHAR(MAX)

    SELECT @Database = [Database] FROM Location WHERE ID = @ID
    SELECT @Table = [Table] FROM Location WHERE ID = @ID
    SELECT @Column = [Column] FROM Location WHERE ID = @ID
    SELECT @ColumnValue = [ValueName] FROM Location WHERE ID = @ID

    SELECT @SQL = 'USE ' + @Database + '; SELECT Item, Data, Timestamp FROM ' + @Table + ' WHERE ' + @Column + ' = ''' + @ColumnValue + '''';
    EXEC sp_executesql @SQL
END
```

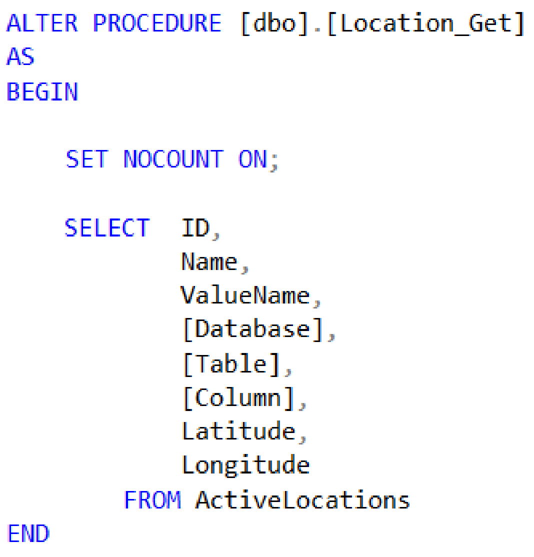
Figure x – SQL code for the GetData stored procedure

In Figure x, dynamic SQL is used to build up a query, this is done to allow another database and table to be queried from this stored procedure.



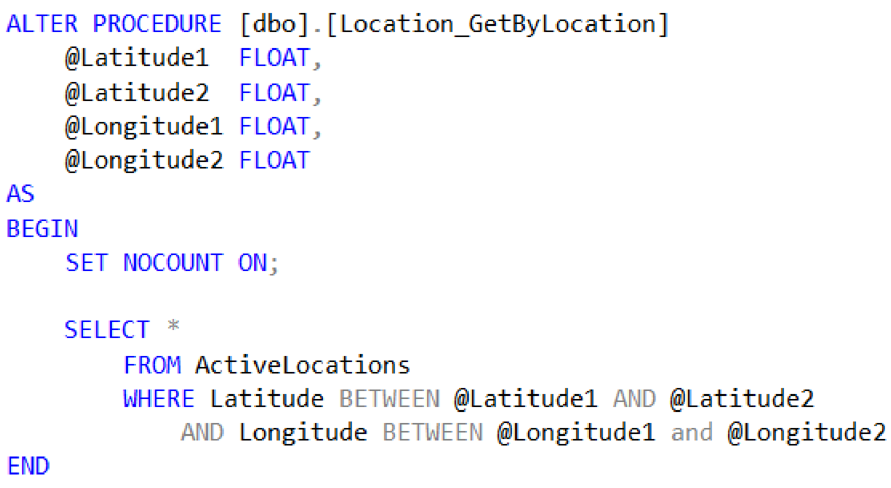
```sql
ALTER PROCEDURE [dbo].[Location_Delete]
    @ID INT
AS
BEGIN
    SET NOCOUNT ON;

    UPDATE LOCATION
        SET DeletedOn = GetDate()
        WHERE ID = @ID;
END
```

Figure x – SQL code for the stored procedure to delete Locations



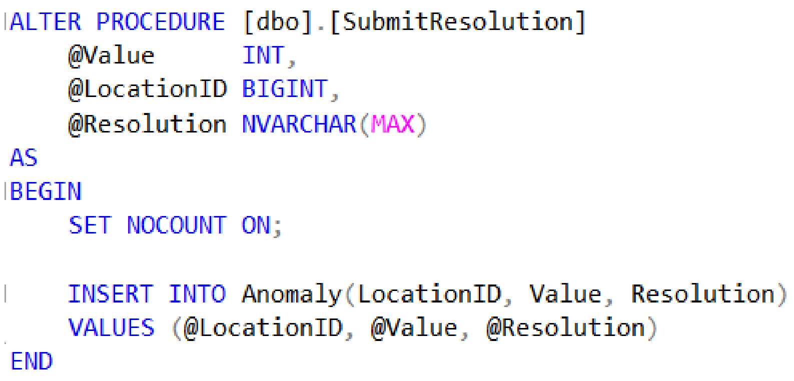
```sql
ALTER PROCEDURE [dbo].[Location_Get]
AS
BEGIN

    SET NOCOUNT ON;

    SELECT  ID,
            Name,
            ValueName,
            [Database],
            [Table],
            [Column],
            Latitude,
            Longitude
        FROM ActiveLocations
END
```

Figure x – SQL code for the stored procedure to get all locations that have not been deleted



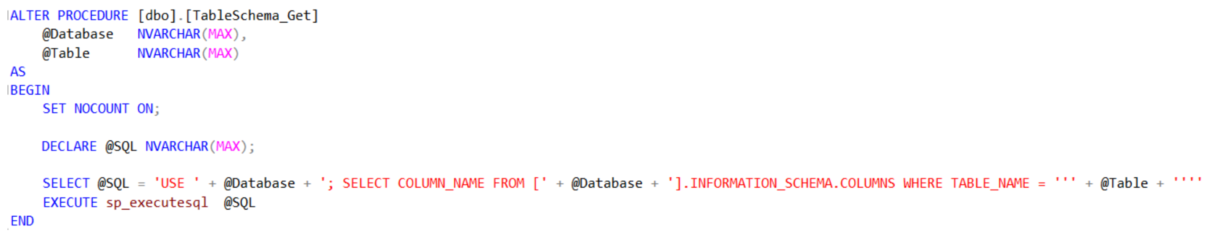
```sql
ALTER PROCEDURE [dbo].[Location_GetByLocation]
    @Latitude1  FLOAT,
    @Latitude2  FLOAT,
    @Longitude1 FLOAT,
    @Longitude2 FLOAT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT *
        FROM ActiveLocations
        WHERE Latitude BETWEEN @Latitude1 AND @Latitude2
            AND Longitude BETWEEN @Longitude1 and @Longitude2
END
```

Figure x – SQL code for the stored procedure to get all locations within a given area



```sql
ALTER PROCEDURE [dbo].[SubmitResolution]
    @Value      INT,
    @LocationID BIGINT,
    @Resolution NVARCHAR(MAX)
AS
BEGIN
    SET NOCOUNT ON;

    INSERT INTO Anomaly(LocationID, Value, Resolution)
    VALUES (@LocationID, @Value, @Resolution)
END
```

Figure x – SQL code for the stored procedure to submit a resolution



```
ALTER PROCEDURE [dbo].[TableSchema_Get]
    @Database    NVARCHAR(MAX),
    @Table       NVARCHAR(MAX)
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @SQL NVARCHAR(MAX);

    SELECT @SQL = 'USE ' + @Database + '; SELECT COLUMN_NAME FROM [' + @Database + '].INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = ''' + @Table + ''''
    EXECUTE sp_executesql  @SQL
END
```

Figure x – SQL code for getting table schema information

The stored procedure in figure x is used to get the names of the columns in a given table in a given database. This also uses dynamic SQL to allow any database and table to be passed in.

```
ALTER PROCEDURE [dbo].[Location_Merge]
       @ID                    INT = NULL,
       @Name           NVARCHAR(MAX) = NULL,
       @ValueName      NVARCHAR(MAX) = NULL,
       @Database       NVARCHAR(MAX) = NULL,
       @Table          NVARCHAR(MAX) = NULL,
       @Column                NVARCHAR(MAX) = NULL,
       @Latitude       FLOAT = NULL,
       @Longitude      FLOAT = NULL
AS
BEGIN
       SET NOCOUNT ON;

       DECLARE @NewID TABLE (
              ID INT
       );

       MERGE [dbo].[Location] AS Target
       USING (
              SELECT @ID,
                            @Name,
                            @ValueName,
                            @Database,
                            @Table,
                            @Column,
                            @Latitude,
                            @Longitude
       ) AS Source (ID, Name, ValueName, [Database], [Table], [Column], Latitude,
Longitude)
       ON (Source.ID = Target.ID)
       WHEN MATCHED THEN UPDATE
              SET Target.Name =     CASE
                                                        WHEN Source.Name IS NULL OR
Source.Name = N''
                                                        THEN Target.Name
                                                        ELSE Source.Name
                                                 END,
              Target.ValueName =    CASE
                                                        WHEN Source.ValueName IS NULL
OR Source.ValueName = N''
                                                        THEN Target.ValueName
                                                        ELSE Source.ValueName
                                                 END,
              Target.[Database] =   CASE
                                                        WHEN Source.[Database] IS NULL
OR Source.[Database] = N''
                                                        THEN Target.[Database]
                                                        ELSE Source.[Database]
```

```
                                                                    END,
              Target.[Table] =     CASE
                                                                            WHEN Source.[Table] IS NULL OR
Source.[Table] = N''
                                                                            THEN Target.[Table]
                                                                            ELSE Source.[Table]
                                                                    END,
              Target.[Column] =    CASE
                                                                            WHEN Source.[Column] IS NULL OR
Source.[Column] = N''
                                                                            THEN Target.[Column]
                                                                            ELSE Source.[Column]
                                                                    END,
              Target.Longitude =   CASE
                                                                            WHEN Source.Longitude IS NULL
                                                                            THEN Target.Longitude
                                                                            ELSE Source.Longitude
                                                                    END
       WHEN NOT MATCHED BY TARGET THEN INSERT
       (
              Name,
              ValueName,
              [Database],
              [Table],
              [Column],
              Latitude,
              Longitude
       )
       VALUES (
              Source.Name,
              Source.ValueName,
              Source.[Database],
              Source.[Table],
              Source.[Column],
              Source.Latitude,
              Source.Longitude
       )
       OUTPUT inserted.ID;
END
```

Figure x – SQL code for location merge statement

```
ALTER PROCEDURE [dbo].[GetAnomalyLocations]
       @PosLat              FLOAT,
       @NegLat              FLOAT,
       @PosLong      FLOAT,
       @NegLong      FLOAT
AS
BEGIN
       SET NOCOUNT ON;
       SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

       DECLARE @TempName           NVARCHAR(MAX);
       DECLARE @TempValueName      NVARCHAR(MAX);

    DECLARE @Locations TABLE (
              Name          NVARCHAR(MAX),
              ValueName     NVARCHAR(MAX)
       )

       DECLARE @LocationData TABLE (
              Name          NVARCHAR(MAX),
              ValueName     NVARCHAR(MAX),
              Data          BIGINT,
              [Timestamp]   DATETIME,
              Latitude      FLOAT,
              Longitude     FLOAT
       )

       INSERT INTO @Locations
       SELECT L.Name,
                     L.ValueName
              FROM Location L
              WHERE L.Latitude BETWEEN @NegLat AND @PosLat
              AND L.Longitude BETWEEN @NegLong and @PosLong;

       DECLARE location_cursor CURSOR FOR
       SELECT * FROM @Locations

       OPEN location_cursor

       FETCH NEXT FROM location_cursor
       INTO @TempName, @TempValueName

       WHILE @@FETCH_STATUS = 0
       BEGIN
              PRINT @TempName;
              INSERT INTO @LocationData
              SELECT L.Name,
                            L.ValueName,
                            E.Data,
                            E.[Timestamp],
                            L.Latitude,
                            L.Longitude
                     FROM Location L
                     INNER JOIN Energy E
                     ON E.Item = L.ValueName
                     WHERE L.Name = @TempName;

              FETCH NEXT FROM location_cursor INTO @TempName, @TempValueName
       END

       CLOSE location_cursor
       DEALLOCATE location_cursor
       SELECT * FROM @LocationData
END
```

Figure x – SQL code for stored procedure to get locations with the data associated with it

Web Service Code

```
var Connection = require('tedious').Connection;
var Request = require('tedious').Request;
var Tag = require("./tag.js");
var express = require('express');
var app = express();
var parser = require('body-parser');
var types = require('tedious').TYPES;
var password = process.argv[2];

app.all('*', function (req, res, next) {
    res.header('Access-Control-Allow-Origin', '*');
    res.header('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS');
    res.header('Access-Control-Allow-Headers', 'Content-Type');
    next();
});

app.use(parser.urlencoded({
    extended: true
}));

app.use(parser.json());
var json = [];

//SQL config
var config = {
    userName: 'FYPracticeDev',
    password: password,
    server: '192.168.1.73',

    options: { port: 49175, database: 'FYPractice', rowCollectionOnRequestCompletion:
true }
};

var connection = new Connection(config);

connection.on('connect', function (err) {
    console.log(password);
    console.log("connected");
});

app.get('/getSchema', function (req, res) {
    var db, table;
    var data = req.query;

    db = data.database;
    table = data.table;
    var sql = 'dbo.TableSchema_Get'

    request = new Request(sql, function (err, rowCount, rows) {
        if (err) {
            console.log(err);
        } else {
            var rowArray = [];
            rows.forEach(function (columns) {
                json.push({
                    columnName: columns[0].value
                });
            });
            var ip = req.headers['x-forwarded-for'] || req.connection.remoteAddress
||req.socket.remoteAddress || req.connection.socket.remoteAddress;
            console.log("here " + ip);
            res.json(json);
            json = [];
        }

    });
    request.addParameter('Database', types.VarChar, db);
    request.addParameter('Table', types.VarChar, table);
    connection.callProcedure(request);

});


app.get('/getAll', function (req, res) {
    request = new Request("select * from Location where DeletedOn is null", function
(err, rowCount, rows) {
        if (err) {
            console.log(err);
        } else {
            var rowArray = [];
            rows.forEach(function (columns) {
                var tag = new Tag(columns[0].value, columns[1].value,
columns[2].value, columns[3].value, columns[4].value, columns[5].value,
columns[6].value, columns[7].value);
                json.push(tag);
            });
            console.log("here")
            res.status(200).send(json);
            json = [];
        }

    });
    var ip = req.headers['x-forwarded-for'] || req.connection.remoteAddress ||
req.socket.remoteAddress || req.connection.socket.remoteAddress;
    console.log("here " + ip);
    connection.execSql(request);
});

var deleteTag = function (id, res) {
    var sql = 'dbo.Location_Delete';
    var request = new Request(sql, function (err) {
        if (err) {
            res.sendStatus(500);
        }
        else {
            res.sendStatus(200);
        }
    });

    request.addParameter('ID', types.Int, id);
    connection.callProcedure(request);
}

app.get('/getByLocation', function (req, res) {
    var data = req.query;
    var latitude, longitude, posLat, posLong, negLat, negLong, earthRadius, sql;
    var LatDiff, LongDiff

    latitude = parseFloat(data.lat);
    longitude = parseFloat(data.long);
    earthRadius = 6378;

    posLat = latitude + (0.1 / earthRadius) * (180 / Math.PI);
    posLong = longitude + (0.1 / earthRadius) * (180 / Math.PI) / Math.cos(latitude *
Math.PI / 180);
    negLat = latitude + (-0.100 / earthRadius) * (180 / Math.PI);
    negLong = longitude + (-0.100 / earthRadius) * (180 / Math.PI) / Math.cos(latitude
* Math.PI / 180);

    sql = 'dbo.Location_GetByLocation';

    request = new Request(sql, function (err, rowCount, rows) {
        if (err) {
            console.log(err);
        } else {
            var rowArray = [];
            rows.forEach(function (columns) {
                var tag = new Tag(columns[0].value, columns[1].value,
columns[2].value, columns[3].value, columns[4].value, columns[5].value,
columns[6].value, columns[7].value);
                json.push(tag);
            });
            var ip = req.headers['x-forwarded-for'] || req.connection.remoteAddress ||
req.socket.remoteAddress || req.connection.socket.remoteAddress;
            console.log("here " + ip);
            res.json(json);
            json = [];
        }

    });

    request.addParameter('Latitude1', types.Float, negLat);
    request.addParameter('Latitude2', types.Float, posLat);
    request.addParameter('Longitude1', types.Float, negLong);
    request.addParameter('Longitude2', types.Float, posLong);

    connection.callProcedure(request);
});

app.put('/delete', function (req, res) {
    var id;
    debugger;
    id = req.body;

    deleteTag(id.data, res);
});

app.post('/submit', function (req, res) {
    var item;
    item = req.body;

    var createTag = function (item) {
        var sql = 'dbo.Location_Merge';
        var request = new Request(sql, function (err, rowCount, rows) {
            var item = rows[0];
            if (err) {
                res.json(500);
            }
            else {
                res.json(item[0].value);
            }
        });

        request.addParameter('ID', types.Int, item.ID);
        request.addParameter('Name', types.VarChar, item.LocationName);
        request.addParameter('ValueName', types.VarChar, item.ColumnValue);
        request.addParameter('Database', types.VarChar, item.Database);
        request.addParameter('Table', types.VarChar, item.Table);
        request.addParameter('Column', types.VarChar, item.Column);
        request.addParameter('Latitude', types.Float, item.Latitude);
        request.addParameter('Longitude', types.Float, item.Longitude);

        request.on('returnValue', function (parameterName, value, metadata) {
            res.json["{'ID': " + value + "}"];
        });

        connection.callProcedure(request);
    }
    createTag(item);
});


app.listen(process.env.PORT || '8081');

exports = module.exports = app;
```

Figure x – Node JS code for the Location web service

```
var method = Tag.prototype;

function Tag(ID, Name, Value, Database, Table, Column, Lat, Long){
    this.ID = ID;
    this.Name = Name;
    this.ValueName = Value;
    this.Database = Database;
    this.Table = Table;
    this.Column = Column;
    this.Latitude = Lat;
    this.Longitude = Long;
}

method.getName = function(){
    return this.Name;
}

module.exports = Tag;
```

Figure x – JavaScript code for the Tag class used by the Location web service

```
var express = require('express');
var Connection = require('tedious').Connection;
var Request = require('tedious').Request;
var parser = require('body-parser');
var app = express();
var TagData = require('./TagData.js');
var types = require('tedious').TYPES;
var password = process.argv[2];

//SQL config
var config = {
    userName: 'FYPracticeDev',
    password: password,
    server: '152.105.199.178',

    options: { port: 49175, database: 'FYPractice', rowCollectionOnRequestCompletion:
true }
};

var connection = new Connection(config);

connection.on('connect', function (err) {
    console.log("connected");
});

app.all('*', function (req, res, next) {
    res.header('Access-Control-Allow-Origin', '*');
    res.header('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS');
    res.header('Access-Control-Allow-Headers', 'Content-Type');
    next();
});

app.use(parser.urlencoded({
    extended: true
}));
app.use(parser.json());
var json = [];

app.get('/getData', function(req,res){
    var db, table, column, columnValue;

    var data = req.query;
    ID = data.id

    var query = "dbo.GetData"

    request = new Request(query, function (err, rowCount, rows) {
        if (err) {
            console.log(err);
        } else {
            var rowArray = [];
            rows.forEach(function (columns) {
                var tagData = new TagData(columns[0].value, columns[1].value,
columns[2].value);
                json.push(tagData);
            });
            console.log("here")
            res.json(json);
            json = [];
        }

    });
    request.addParameter('ID', types.Int, ID);

    connection.callProcedure(request);
});

app.listen(process.env.PORT || '8082');

exports = module.exports = app;
```

Figure x – Node JS code for the Location Data web service

```
var method = TagData.prototype;

function TagData(Item, Data,Timestamp){
    this.Item = Item;
    this.Data = Data;
    this.Timestamp = Timestamp;
}

module.exports = TagData;
```

Figure x – JavaScript code for the TagData class used by the LocationData web service

```
var express = require('express');
var tedious = require('tedious');
var Connection = require('tedious').Connection;
var Request = require('tedious').Request;
var parser = require('body-parser');
var app = express();
var password = process.argv[2];
var types = require('tedious').TYPES;
var Location = require("./Location.js");

app.all('*', function (req, res, next) {
    res.header('Access-Control-Allow-Origin', '*');
    res.header('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS');
    res.header('Access-Control-Allow-Headers', 'Content-Type');
    next();
});

var json = [];
app.use(parser.urlencoded({
    extended: true
}));
app.use(parser.json());

var config = {
    userName: 'FYPracticeDev',
    password: password,
    server: '192.168.1.73',
    options: {
        port: 49175,
        database: 'FYPractice',
        rowCollectionOnRequestCompletion: true
    }
}

var connection = new Connection(config);

connection.on('connect', function (err) {
    console.log("connected");
});
```

```
app.post('/submitResolution', function (req, res) {
    var data, sql, request;
    console.log("submit");
    data = req.body;
    console.log(data);
    console.log(req);
    sql = 'dbo.submitResolution';

    request = new Request(sql, function (err, rowCount, rows) {
        if (err) {
            res.sendStatus(500);
        }
        else {
            res.sendStatus(200);
        }
    });

    request.addParameter('LocationID', types.Int, data.LocationID);
    request.addParameter('Value', types.BigInt, data.Value);
    request.addParameter('Resolution', types.VarChar, data.Resolution);

    connection.callProcedure(request);
});

app.get('/getAnomalies', function (req, res) {
    var lat, long, posLat, posLong, negLat, negLong, sql, request, data,
currentLocation, currentIndex;

    var url = req.query;

    lat = parseFloat(url.lat);
    long = parseFloat(url.long);

    earthRadius = 6378;

    posLat = lat + (0.1 / earthRadius) * (180 / Math.PI);
    posLong = long + (0.1 / earthRadius) * (180 / Math.PI) / Math.cos(lat * Math.PI /
180);
    negLat = lat + (-0.100 / earthRadius) * (180 / Math.PI);
    negLong = long + (-0.100 / earthRadius) * (180 / Math.PI) / Math.cos(lat * Math.PI
/ 180);

    sql = 'dbo.getAnomalyLocations'
    data = [];
    currentLocation = '';
    currentIndex = -1;
    request = new Request(sql, function (err, rowCount, rows) {
        if (err) {
            console.log(err);
        } else {
            var rowArray = [];
            rows.forEach(function (columns) {
                var location = new Location(columns[0].value, columns[1].value,
columns[2].value, columns[3].value, columns[4].value, columns[5].value);
                if(location.Name != currentLocation){
                    data.push([]);
                    currentLocation = location.Name;
                    currentIndex++;
                }
                data[currentIndex].push(location);
            });
            for(var i = 0; i < data.length; i++){
                calculateStDev(data[i]);
            }

            var ip = req.headers['x-forwarded-for'] || req.connection.remoteAddress ||
req.socket.remoteAddress || req.connection.socket.remoteAddress;
            console.log("here " + ip);
            res.json(json);
            json = [];
        }

    });

    request.addParameter('negLat', types.Float, negLat);
    request.addParameter('posLat', types.Float, posLat);
    request.addParameter('negLong', types.Float, negLong);
    request.addParameter('posLong', types.Float, posLong);

    connection.callProcedure(request);
});

function calculateStDev(data) {
    var total, mean, diffData, diff, sum;

    total = 0;
    diffData = [];
    for (var i = 1; i < data.length; i++) {
        diff = 0;
        diff = data[i].Data - data[i - 1].Data;
        diffData.push({ Location: data[i].Name, Value: diff, Timestamp:
data[i].Timestamp, Latitude: data[i].Latitude, Longitude: data[i].Longitude });
        total += diff;

    }
    console.log(total);

    mean = total / data.length;
    var x, temp;
    x = 0;
    temp = 0;
    for (var i = 0; i < diffData.length; i++) {
        x = diffData[i].Value - mean;
        temp += Math.pow(x, 2.0);
    }

    var y = temp / diffData.length;
    var stdDev = Math.pow(y, 0.5);
    for (var i = 0; i < diffData.length; i++) {
        var current = diffData[i];
        if (current.Value > (mean + (4 * stdDev)) || current.Value < (mean - (4 *
stdDev))) {
            json.push(current);
        }
    }
}

app.listen(process.env.PORT || '8083');

exports = module.exports = app;
```

Figure x – Node JS code for the Anomaly Notification web service

```
var method = Location.prototype;

function Location(Name, ValueName, Data, Timestamp, lat, long) {
    this.Name = Name;
    this.ValueName = ValueName;
    this.Data = Data;
    this.Timestamp = Timestamp;
    this.Latitude = lat;
    this.Longitude = long;
}

module.exports = Location;
```

Figure x – JavaScript code for the Location class used by the Anomaly Notification web service

```
var request = require('supertest');
var chai = require('chai'), expect = chai.expect, should = chai.should();
describe('testing anomaly notifier', function () {
    var url;
    url = 'http://localhost:8083/'
    beforeEach(function (done) {
        done();
    });
    afterEach(function () {
    });
    it('should return 4 anomalies', function testAnomalyService(done) {
        request(url)
            .get('getAnomalies?lat=54.569467&long=-1.2342727')
            .end(function (err, res) {
                if (err) {
                    throw err;
                }
                res.body.should.have.length(4);
                done();
            });

    });
    it('should return 0 anomalies', function testNoAnomalies(done) {
        request(url)
            .get('getAnomalies?lat=53.569467&long=-1.2342727')
            .end(function (err, res) {
                if (err) {
                    throw err;
                }
                res.body.should.have.length(0);
                done();
            });
    });
    it('should allow submission of a valid resolution', function
testResolutionSubmission(done) {
        var resolution = {
            LocationID: 16,
            Resolution: 'unit test resolution',
            Value: 9999
        }
        request(url)
            .post('submitResolution')
            .send(resolution)
            .end(function (err, res) {
                res.statusCode.should.equal(200);
            });
        done();
    });
    it('should not allow the submission of an invalid resolution', function
invalidResolution(done) {
        var resolution = {
            Resolution: 'no LocationID',
            Value: 1233
        }
        request(url)
            .post('submitResolution')
            .send(resolution)
            .end(function (err, res) {
                res.statusCode.should.equal(500);
            });
        done();
    });
});
```

Figure x – Node JS code for Anomaly Notification web service unit tests

```
var request = require('supertest');
var chai = require('chai'), expect = chai.expect, should = chai.should();
describe('testing location data web service', function () {
    var url;
    url = 'http://localhost:8082/'
    it('should get data for a given location', function (done) {
        request(url)
            .get('getData?id=16')
            .end(function (err, res) {
                if (err) {
                    throw err;
                }
                res.body.should.have.length(911);
                done();
            });
    });
});
```

Figure x – Node JS code for Location Data web service unit test

```
var request = require('supertest');
var chai = require('chai'), expect = chai.expect, should = chai.should();
describe('testing location web service', function () {
    var url;
    url = 'http://localhost:8081/'
    beforeEach(function (done) {
        done();
    });
    it('should get all locations', function getAll(done) {
        request(url)
            .get('getAll')
            .end(function (err, res) {
                if (err) {
                    throw err;
                }
                res.body.should.have.length(14);
                done();
            });
    });
    it('should get all locations near a given location', function byLocation(done) {
        request(url)
            .get('getByLocation?lat=54.569467&long=-1.2342727')
            .end(function (err, res) {
                if (err) {
                    throw err;
                }
                res.body.should.have.length(4);
                done();
            });
    });
    it('should get the schema for a given table', function getSchema(done) {
        request(url)
            .get('getSchema?database=FYPractice&table=Location')
            .end(function (err, res) {
                if (err) {
                    throw err;
                }
                res.body.should.have.length(9);
                done();
            });
    });
    it('should allow the submission of a valid location', function
createLocation(done) {
        var location = {
            LocationName: 'testLocation',
            ColumnValue: 'testLocation',
            Database: 'FYPractice',
            Table: 'Energy',
            Column: 'Item',
            Latitude: '54.5656977',
            Longitude: '-1.2372124'
        }
        request(url)
            .post('submit')
            .send(location)
            .end(function (err, res) {
                res.body.should.not.equal(500);
                done();
            });
    });
    it('should not allow the creation of an invalid location', function
invalidLocation(done) {
        var location = {
            LocationName: 'testLocation',
            ColumnValue: 'testLocation',
            Table: 'Energy',
            Column: 'Item',
            Latitude: '54.5656977'
        }
        request(url)
            .post('submit')
            .send(location)
            .end(function (err, res) {
                res.body.should.equal(500);
                done();
            });
    })
    it('should allow the deleting of locations', function deleteLocation(done) {
        var id = {
            data: 29
        }
        request(url)
            .put('delete')
            .send(id)
            .end(function (err, res) {
                res.statusCode.should.equal(200);
            });
        done();
    });
});
```

Figure x – Node JS code for the Location web service unit tests

Compass code

```
public class
Compass implements
SensorEventListener
{
        private static final String TAG = "Compass";


        private SensorManager sensorManager;
        private Sensor gsensor;
        private Sensor msensor;
        private float[] mGravity = new float[3];
        private float[] mGeomagnetic = new float[3];
        private float azimuth = 0f;
        private float currectAzimuth = 0;


        // compass arrow to rotate
        public ImageView arrowView = null;


        public Compass(Context context) {
                sensorManager = (SensorManager) context

        .getSystemService(Context.SENSOR_SERVICE);
                gsensor =
sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
                msensor =
sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
        }


        public void start() {
                sensorManager.registerListener(this, gsensor,
                                SensorManager.SENSOR_DELAY_GAME);
                sensorManager.registerListener(this, msensor,
                                SensorManager.SENSOR_DELAY_GAME);
        }
```

```
		public void stop() {
			sensorManager.unregisterListener(this);
		}


		private void adjustArrow() {
			if (arrowView == null) {
				Log.i(TAG, "arrow view is not set");
				return;
			}


			Log.i(TAG, "will set rotation from " +
currectAzimuth + " to "
					+ azimuth);


			Animation an = new RotateAnimation(-currectAzimuth,
-azimuth,
					Animation.RELATIVE_TO_SELF, 0.5f,
Animation.RELATIVE_TO_SELF,
					0.5f);
			currectAzimuth = azimuth;


			an.setDuration(500);
			an.setRepeatCount(0);
			an.setFillAfter(true);


			arrowView.startAnimation(an);
		}


		@Override
		public void onSensorChanged(SensorEvent event) {
			final float alpha = 0.97f;


			synchronized (this) {
				if (event.sensor.getType() ==
Sensor.TYPE_ACCELEROMETER) {


					mGravity[0] = alpha * mGravity[0] +
(1 - alpha)
```

```
                                                            * event.values[0];
                                        mGravity[1] = alpha * mGravity[1] +
(1 - alpha)
                                                            * event.values[1];
                                        mGravity[2] = alpha * mGravity[2] +
(1 - alpha)
                                                            * event.values[2];


                                        // mGravity = event.values;


                                        // Log.e(TAG,
Float.toString(mGravity[0]));
                                 }


                                 if (event.sensor.getType() ==
Sensor.TYPE_MAGNETIC_FIELD) {
                                        // mGeomagnetic = event.values;


                                        mGeomagnetic[0] = alpha *
mGeomagnetic[0] + (1 - alpha)
                                                            * event.values[0];
                                        mGeomagnetic[1] = alpha *
mGeomagnetic[1] + (1 - alpha)
                                                            * event.values[1];
                                        mGeomagnetic[2] = alpha *
mGeomagnetic[2] + (1 - alpha)
                                                            * event.values[2];
                                        // Log.e(TAG,
Float.toString(event.values[0]));


                                 }


                                 float R[] = new float[9];
                                 float I[] = new float[9];
                                 boolean success =
SensorManager.getRotationMatrix(R, I, mGravity,
                                              mGeomagnetic);
                                 if (success) {
                                        float orientation[] = new float[3];
```

```
                                                SensorManager.getOrientation(R,
                         orientation);
                                                // Log.d(TAG, "azimuth (rad): " +
                         azimuth);
                                                azimuth = (float)
                         Math.toDegrees(orientation[0]); // orientation
                                                azimuth = (azimuth + 360) % 360;
                                                // Log.d(TAG, "azimuth (deg): " +
                         azimuth);
                                                adjustArrow();
                                        }
                                }
                        }


                        @Override
                        public void onAccuracyChanged(Sensor sensor, int accuracy)
                {
                        }
                }
```

Figure x – original code provided by Lutin in Java

```
class Compass : SensorEventListener
{
    var sensorManager : SensorManager
    var gsensor : Sensor
    var msensor : Sensor
    var mGravity : FloatArray = FloatArray(3)
    var mMagnetic : FloatArray = FloatArray(3)
    var azimuth : Double = 0.0


    constructor(c : Context){
        sensorManager = c.getSystemService(Context.SENSOR_SERVICE) as
SensorManager
        gsensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
        msensor = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD)
    }

    fun start(){
        sensorManager.registerListener(this, gsensor,
SensorManager.SENSOR_DELAY_GAME)
        sensorManager.registerListener(this, msensor,
SensorManager.SENSOR_DELAY_GAME)
    }

    fun stop(){
        sensorManager.unregisterListener(this)
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {
        //throw UnsupportedOperationException("not implemented") //To change
body of created functions use File | Settings | File Templates.
    }

    override fun onSensorChanged(event: SensorEvent) {
        var gravity : Float = 0.98f

        synchronized(this){
            if(event.sensor.type == Sensor.TYPE_ACCELEROMETER){
                mGravity[0] = gravity * mGravity[0] + (1 - gravity) *
event.values[0]
                mGravity[1] = gravity * mGravity[1] + (1 - gravity) *
event.values[1]
                mGravity[2] = gravity * mGravity[2] + (1 - gravity) *
event.values[2] //Low pass filter
            }

            if(event.sensor.type == Sensor.TYPE_MAGNETIC_FIELD){
                mMagnetic[0] = gravity * mMagnetic[0] + (1 - gravity) *
event.values[0]
                mMagnetic[1] = gravity * mMagnetic[1] + (1 - gravity) *
event.values[1]
                mMagnetic[2] = gravity * mMagnetic[2] + (1 - gravity) *
event.values[2]
            }

            var R : FloatArray = FloatArray(9)
            var I : FloatArray = FloatArray(9)

            var success = SensorManager.getRotationMatrix(R, I, mGravity,
mMagnetic)

            if(success){
                var orientation : FloatArray = FloatArray(3)
                SensorManager.getOrientation(R, orientation)
                azimuth = Math.toDegrees(orientation[0].toDouble() + 45)
                azimuth = ((azimuth + 360) % 360)  //to account for device
rotation

            }
        }
    }
}
```

Figure x – original compass code translated into Kotlin by myself