University of the West of England
BRISTOL

Copyright Notice

Library Digitisation Service

Rail Performance Data Foundation

Matthew Herring

15003409

Acknowledgements

I would like to acknowledge and thank the following people who have supported me, not only throughout this project, but through my University career as well.

Firstly, I would like to thank my supervisor, Mehmet Aydin, for his continued support and guidance throughout the course of this project and academic year.

I would also like to thank Mark Boston and the IS Team at Leidos Europe, without whom this project would not have come to fruition.

I would also like to thank Frank Collins, Nigel Smedley and the rest of the RPS for approaching me with the initial idea for the RPDF.

And finally, I would like to thank all my close family and friends for continually encouraging, supporting and believing in me throughout my academic career.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Data analysis is a procedure that all businesses and charities should make integral to their day to day operations, no matter how large or small they are. 'Data analysis allows businesses to compile and analyse data in order to present findings and trends to management to help inform business decision making' *(Wulff, 2018)*. It is the process of examining and arranging raw data in such a way that allows us to generate useful information and draw conclusions from the data being analysed. As businesses grow and become more technically capable, so does the amount of data that they create, store and maintain. As a result of this, data analysis and analytical capabilities are becoming more of a focus for businesses as the process of data analysis allows them to find trends in markets and exploit those trends in order to reduce costs and increase profits. However, data analytics tools for modern businesses of a large scale come at a large financial cost.

Data analytics tools that are currently available are fantastic, albeit costly, at providing companies with insights into the way their market is heading and allowing businesses to spot trends to inform their future decisions. The advantages of current analytical tools include but are not limited to allowing businesses to learn about the overall performance of a product or service, allowing businesses to assess the reaction to a specific product by its target audience or demographic. Analytical tools allow businesses to perform analysis on trends such as gender, age and location to help inform their decisions about where they should invest their money and resources to increase their profit margins for a specific product or set of products.

The disadvantages of data analytics tools currently available are free tools that are currently available do not give a detailed enough view on the information available hence larger companies will pay a large premium for tools that provide more in-depth analysis. Analytical tools need to be constantly maintained by experts in order to reduce the security risk posed by potential hackers. It is difficult to remove unwanted noise from the data you are analysing, especially with the free tools

currently available on the market. Companies need to employ specialists in order to manage and present data, especially if it is in a large volume. If data analytics tools and practices are not implemented properly, it could cause many different problems for the business. 'If a business is not used to handling data at a fast pace, it could lead to incorrect interpretation and analysis of data' *(Pal, 2018)*, which could cause problems for the business further down the line.

This large cost of data analytics tools makes it increasingly difficult for small businesses and charities who may not have a very high profit margin, if any at all, to make any impact in their respective markets. The cost and complexity of data analytics tools currently on the market makes it very difficult for small businesses to perform any kind of data analysis, outside of an Excel sheet with some graphs and pivot tables. It is not only the cost of the software itself that poses a major problem, but also the financial impact of purchasing user licenses for the software. Due to the complexity of most modern-day data analytics tools, if a small business or charity could afford the software and licenses, they would also have to employ someone who is proficient with that software. Skilled data analysts are highly sought after by many large corporations and in-house knowledge of software such as IBM's '*Cognos Business Intelligence Analytics*' is rare, especially in smaller businesses and charitable organisations.

One such organisation who is affected by the problem of data analytics is the Rail Performance Society (**RPS**), a charitable organisation founded thirty-five years ago that is dedicated to the recording and analysis of the performance of diesel, steam and electric trains and traction engines across the world *(Collins, 2008)*. As a small charitable organisation, the **RPS** regularly encounters problems with compiling and analysing the large amount of data that is collected by its members. As the data the members of the **RPS** collect is very complex and their current database structure has a lot of interlinked tables, all with varying degrees of complexity and size, it makes it very difficult for them to perform any kind of analysis on the data as they do not have the budget or technical capability to do so.

At present the **RPS**' current solution for data analytics is a table that is displayed on their website after a user performs a search on a particular table in the database. One such example of a search that is carried out is a search by 'Recorder'; a recorder is a member of the **RPS** who collects (or records) data on rail performance. The issue with searching by Recorder is that it pulls data from as far back as the **RPS** has existed and displays the results in one big list, so for some Recorders this could be up to thirty years. Therefore, the amount of analysis that can be performed on this data set is severely limited by the underlying factor that there is no functionality in place to reduce the number of results you get when searching the database by adding filters to the search parameters.

The proposed solution to this problem is to create a web-based application called the Rail Performance Data Foundation, or **RPDF** for short. The **RPDF** will be totally separated from the **RPS** website which is where the current data analysis capabilities are held. This will allow the **RPDF** to be completely tailored to the needs of the member of the **RPS**. The overall aim of the **RPDF** is to allow the member of the **RPS** to collate the data they produce on rail performance and to view and analyse this data so that they can produce reports and articles based on their findings. The application will allow the users to produce reports on various metrics of their choosing and each query will be totally customisable to allow the users full flexibility when performing data analysis.

2 Research

2.1 Data Analytics Tools

Data analytics tools that are currently available come with many different features and levels of service. These vary from complex dashboards to allow businesses to see live representations of their data in real time, to report building functionality that produce Microsoft Excel spreadsheets containing raw data pulled directly from a database.

2.1.1 Dundas BI

One such example is Dundas BI, a Business Intelligence tool created by Dundas Data Visualization. 'Dundas BI is a browser-based platform aimed at businesses of all sizes' *(Marchand, 2019)*; Dundas BI provides the ability for varying degrees of customisation when it comes to building bespoke reports and performing analysis on any number of metrics the user may wish to. Compared to other data analytics tools, Dundas BI is fairly cheap, they offer an introductory free trial to allow businesses to decide whether or not they wish to use the service. If a business decides they wish to continue using the service, the price of the service is then tailored to the business depending on how many employees they have *(Marchand, 2019)*; for example, for a small business with 50 employees the price would be significantly lower than the price for a business with over 1000 employees. This ultimately makes Dundas BI a very versatile tool for businesses of all sizes and budgets, allowing users to create a totally bespoke product whilst on a budget.

2.1.2 Cognos Analytics

On the other end of the cost spectrum is Cognos Analytics, by IBM. Cognos Analytics is the recently upgraded version of IBM's Cognos Business Intelligence, or Cognos BI for short. 'Cognos Analytics provides medium to large sized businesses with cognitive guidance, a web-based platform, data visualisation, data governance and security functionalities for their data' *(Picciano, 2019)*. Compared with Dundas BI, IBM Cognos Analytics is very expensive for a very similar service; similarly, to Dundas BI, with Cognos users can query multiple data sources to produce visualisations and reports of that data. When it comes to pricing, Cognos is similar to Dundas in the respect that it offers a free trial period to allow businesses time to assess whether or not they want to continue with the service. However, unlike Dundas, once the trial period is over Cognos charges a flat monthly rate for each authorised user *(Picciano, 2019)*. While in a larger business this may be an option as the budget may allow for this kind of expense; in a smaller business this would not be a viable option because the total cost of using Cognos will grow exponentially as more and more employees require access to create dashboards and visuals for presentations and reports for meetings and regular reviews on different metrics.

2.2 Programming Language

Due to the size and complexity of the project, extensive research needed to be carried out into how this problem was to be tackled. As this project is almost entirely reliant on the backend architecture and database access to be successful, the choice of technologies to accomplish this was of vital importance.

The first topic of research was the programming language that the application was going to be written in. As this is a web-based application designed to be cross-platform and used on multiple devices, Java Enterprise Edition (EE) was the best fit for this scenario.

'**Java EE** is a programming language that consists of a set of over 28 specifications and a runtime environment, it is designed to take full advantage of all **Java SE API**s as it is a superset of the **Java SE** platform' *(Theedom, 2017)*. This means that it is perfect for designing and building multi-level, component-reliant applications, in the context on this project the database is the main component that the application is reliant on. **Java EE** allows developers to provide greater levels of interaction between users and their web applications. Due to its cross-platform nature; **Java EE** prevents developers from encountering problems with the platform the application is running on, it does not need to be configured on a per **OS** or per platform basis which allows for full "plug and play" usage *(Tuan, 2019)*. **Figure 21** *(Theedom, 2017)* shows that, since its conception in 1998, **Java EE** has evolved to the point where it can now provide all of the functionality needed for this project, including integration with multiple different types of Web Services such as **JAX-RS** and **SOAP**, it also includes Context and Dependency Injection and JavaServer Faces (**JSF**) support which will allow for a more custom **UI** to be developed in accordance with the requirements for this project.

2.3 Frameworks

After the decision to use **Java EE** was finalised, research needed to be carried out into whether or not any frameworks should be used for the application and if so, which ones would be used. A Framework, in the context of Java applications is a body of resources, usually code packaged up as Java classes, who's primary purpose it is to aid and ease the application development process. The sole purpose of a framework is to provide a developer with the tools to produce a superior finished application compared to the product they could produce if they did not use frameworks. 'Frameworks provide design patterns and structure to the application, as well as the backbone and container for the components the developer creates for their application to operate within' *(Nash, 2002)*.

JavaServer Faces (**JSF**) is just one example of a Java framework; **JSF** is a component based **MVC**, (Model, View, Controller), framework that is used by developers to build event-oriented web interfaces. '**JSF** allows access to server-side data and logic through the use of **XML** documents that represent components in a logical tree' *(Tyson, 2018)*. In comparison with JavaServer Pages (**JSP**), **JSF** components are backed by Java objects and methods are injected into the view pages when they are needed. This keeps the main Java code and **HTML** view pages independent of one another and allows for full use of Java abilities, ultimately removing the need to put Java code in the view pages, a practice which is commonplace amongst **JSP** view pages. **JSF** also makes the development process much easier for developers as it minimalizes the interaction between the developer and client-side technologies such as **HTML**, **CSS** and **JS**.

The fundamental idea behind the use of **JSF** is to create components that can be reused across a web application. The idea is similar to that of the reusable tags in **JSP**, however **JSF** has a more formal approach when it comes to creating these reusable components. **JSF** pages can be used inside of **JSP** pages, however, it is common practice to create standalone **JSF** pages using Facelets. Facelets are **XHTML** pages that are designed to define **JSF** interfaces. 'To create a Facelet, the developer

uses **XML** tags to create a component tree that becomes the backbone for the **JSF** user interface' *(Tyson, 2018)*.

As mentioned previously, **JSF** is an **MVC** framework, it implements the model, view, controller pattern; the idea behind which is to separate the three layers of a user interface into distinct parts to make them easier to manage. In the **MVC** pattern, the view is responsible for displaying data from the model, the controller is responsible for setting up the model and directing the user to the correct view and the model is a normal Java object or plain old Java object, or **POJO** for short.

BootsFaces is a commonly used **JSF** framework which adds Bootstrap, a popular **JSP** design tool, to **JSF** pages. BootsFaces is well known for having built-in responsive design which is a very attractive trait when it comes to developing web-based applications *(Rauh, 2015)*. Responsive design enables applications to be used on a multitude of devices, ranging from smartphones and tablets, to laptops and desktop computers without having to worry about configuration issues. Responsive design means that screen elements are designed to automatically resize to fit on smaller screens. If resizing the elements does not work and they are still too big for the screen, then BootsFaces will stack elements over each other if they cannot be displayed side by side. Even though this will not look as good as the original design built for large screens, it makes the application more accessible as users who may not have access to a laptop or desktop computer can still use the application on a smartphone or tablet.

After conducting this research, it was concluded that the best course of action was to use **JSF** and base the **UI** on BootsFaces. This is due to the level of customizability that is available from BootsFaces as it will allow for a totally unique **UI** to be built in accordance with the requirements.

2.4 Web Application Servers

In order to develop a web-based application, a decision must be made into what kind of web application server the application will run on. There are many different types of web application servers to choose from depending on the needs and scope of the project at hand. There are an outstanding range of application servers to choose from, however as this project does not have any funding, only open source application servers will be taken into account. 'The purpose of a web application server is to support the construction of dynamic pages and implementation of services such as clustering, fail-over and load balancing so that developers do not have to worry about configuration and can solely focus on the implementation of business logic' *(Marshall, 2015)*.

2.4.1 GlassFish

GlassFish is a **Java EE** application server developed by Oracle and is commonly regarded as the 'reference implementation of the **Java EE** standard' *(Zukanov, 2018)*. This means that GlassFish supports **EJB**, **JPA**, **JSF**, **JMS**, **RMI** and **JSP**, just to name a few. 'GlassFish allows developers to create applications that are portable, scalable and integrate with legacy technologies and supports the installation of optional components to provide additional services' *(Marshall, 2015)*.

Unfortunately, GlassFish does lack commercial support, which would be a concern that a developer would need to take into account if their project was predicted to grow and become financially successful. The long-term security and customer service that commercial support provides is vital when it comes to choosing an application server because if it is needed in the future, the application will need to be migrated to a suitable application server. This is a process that could potentially cause a lot of undue stress and complications for the developer.

2.4.2 Wildfly

Wildfly is an open source application server built by JBoss and is now being developed by Red Hat, it is written in Java which means that it can run on multiple platforms. 'Wildfly includes features such as clustering, Deployment **API**, **EJB** version 3, Failover, persistence programming, **JAAS**, **JCA** integration, **JMS** integration, **JNDI**, **JTA API**, **JSF**, **JSP** and Java Servlet 2.5' *(Marshall, 2015)*. Wildfly also supports web services such as **JAX-WS** and **SOAP**. Wildfly also includes a web-based management console for ease of use when it comes to deployments and server configuration such as setting up database connections.

Unlike GlassFish, Wildfly supports commercial development as applications developed on it can be seamlessly migrated to the commercially supported application server, JBoss Enterprise Application Platform. For developers this means that they can develop an application on Wildfly quickly and easily and migrate the application over to **JBoss EAP** later down the line should their circumstances require them to do so *(Zukanov, 2018)*.

2.4.3 Apache Tomcat

'Apache Tomcat is widely regarded as the most popular application server on the market to date with over 60% of the market share of all Java application server deployments' *(Zukanov, 2018)*.

Tomcat is an open source implementation of Java Servlet and JavaServer Pages technologies developed by the Apache Software Foundation. Tomcat implements multiple **Java EE** specifications such as Java Servlet, **JSP**, **Java EL** and WebSocket. It also provides a "pure Java" **HTTP** web server environment to allow Java code to run in *(Marshall, 2015)*.

However, Tomcat does not actually provide all of the features required of a **Java EE** application server, therefore, Tomcat can only be classified as either a "web server" or a "servlet container". Despite this fact, most of the features of a **Java EE** application server are available to the developer. To include these features would require some additional configuration from the developer in the initial setup of the application, as the features would have to be added as third-party dependencies in the server configuration files. This can cause a lot of difficulties for the developer as the extra configuration could be time consuming; however, it would provide the developer the option to tailor their application to their exact needs for their project.

2.4.4 Jetty

Jetty is an open source application server developed by the Eclipse Foundation. Similarly to Apache Tomcat, Jetty does not provide all of the features of a **Java EE** application server, thus it can only be classified as a "web server" or a "servlet container". However, should a developer want to, the features can be added to the server as third-party dependencies. This allows for greater flexibility as developers can tailor the server to their exact requirements. 'Due to its compact size and small footprint, Jetty is perfect for use in constrained environments and for embedding into other products' *(Zukanov, 2018)*.

Jetty is popular among large companies such as Google and Twitter, it has been used in produces such as Apache ActiveMQ, Alfresco, Apache Geronimo, Apache Maven, Apache Spark, Google App Engine, Eclipse and Twitter's Streaming **API**. 'Jetty also supports Java Servlet **API**, **AJP**, **JASPI**, **JMX**, **JNDI**, **OSGi** and WebSocket' *(Marshall, 2015)*.

In conclusion, the best application server for this project is Wildfly. This is due to the fact that it includes all of the features you would expect a fully functional **Java EE** application server to have. It also has a management console **UI** that is very easy to navigate, which makes performing deployments and server-side configuration very simple. Wildfly also supports commercial scaling, so if the application would require commercial support it could be seamlessly migrated to a commercially sized application server.

2.5 Database Providers

As this project is heavily reliant on the integrity of its backend database architecture, it was vital that the correct database provider was chosen for this project.

The **RPS**' data is currently held on a MySQL server, however, alternatives needed to be explored to establish whether or not they would be more suitable for this project.

Java EE is compatible with most database providers, so from that perspective there were no constraints as to which database provider was chosen. However, as the data the **RPS** collects is stored over multiple tables, all linked together via a multitude of primary and foreign keys, the one condition for selecting a database provider was that the database provider provided a relational database.

With that in mind, and the fact that the database provider needed to be open source, the following were chosen as the top three relational database providers.

2.5.1 MySQL

As stated previously, MySQL is currently what the **RPS** uses to house and manipulate their data. This is with good reason, as MySQL is widely regarded as the market leader when it comes to relational databases.

MySQL has been around for over 30 years and can be used through a variety of different **GUI** tools and also via a command line interface.

The open source community edition, however, includes features such as a pluggable storage engine, MySQL replication, partitioning and connectors. 'MySQL can run on most major **OS**', including Linux, OS X and Windows and supports most widely used programming language including, Java, C++ and PHP' *(Walker, 2017)*.

MySQL is also known to power apps such as Uber, WordPress and Facebook and also offers an enterprise variation, which is more suited to much larger scale, high-volume sites such as these.

2.5.2 Microsoft SQL

Microsoft **SQL** is another well-known relational database provider that has been around since the 1980s, powering massive companies such as NASDAQ, Yahoo and Dell.

Microsoft **SQL** offers an open source variation which is ideal for smaller applications with only 10GB of data. The open source edition comes with a variety of development tools, as well as cloud backup and restore provided by the Microsoft Azure service.

However, due to the relatively small amount of storage offered by the open source version, Microsoft **SQL** is really only targeted at mid to large scale companies. Of the companies that use Microsoft **SQL**, the majority of these are IT and **SaaS** companies.

'Microsoft **SQL** is also only available on Linux and Windows, however it is compatible with the majority of the commonly used programming languages, such as C++, Java and PHP' *(Walker, 2017)*.

2.5.3 PostgreSQL

In comparison with MySQL and Microsoft **SQL**, PostgreSQL is not as old or as widely used. However, 'PostgreSQL has been around for over 20 years and is known to power huge businesses such as ADP, Fujitsu and Cisco and as of 2012 over 30% of tech companies use PostgreSQL as the core of their applications' *(Walker, 2017)*.

PostgreSQL has a generous open source license, which allows users to modify code at will for their company's needs. Some of the main features of PostgreSQL include table inheritance, nested transactions and asynchronous replication.

'PostgreSQL boasts the greatest range of supported **OS**', it supports **OS**' such as Windows, OS X, Linux, Unix and Solaris' *(Walker, 2017)*. However, in comparison with MySQL and Microsoft **SQL**, it does not support as many programming languages but does support the most commonly used programming languages such as C++, Java and PHP.

As the **RPS** currently holds their data on a MySQL database, the logical option is to use a MySQL database. This would prevent compatibility issues from arising when the application is deployed and connected to the currently live database. As MySQL is also regarded as the market leader for relational databases, it is one of the most widely supported databases, so if problems do arise support will be readily available online. MySQL has multiple different **GUI**s available to it which make database administration very easy, it will also allow queries to be written and tested with ease.

3 Requirements

As there are a lot of requirements of varying degrees of effort and difficulty in this project, they needed to be prioritised in order to produce a viable product within the timeframe given. The decision was made to prioritise the requirements using the MoSCoW method, details of which are given below. **Figure 22** *(Walker, 2014)* shows how requirements are typically prioritised using the MoSCoW method.

The reasoning behind using the MoSCoW method is because it overcomes the problems associated with much simpler prioritisation techniques. For example, the use of a high, medium or low prioritisation technique is much weaker than the MoSCoW method because the priorities do not have a clear definition. Therefore, expectations cannot be managed as the customer cannot be provided with a precise idea of what can and cannot be done within the timeframe given.

Also, another example of an even weaker prioritisation technique is a basic sequential numbering technique. This is even weaker that using high, medium and low prioritisation because there will always be debate as to whether or not a requirement should be placed higher or lower on the list.

3.1 MoSCoW Method

'The MoSCoW method is a prioritisation technique used to help understand and manage priorities within the Agile software development process' *(Walker, 2014)*.

The MoSCoW method is split up into four different categories, they are:

- Must Do
- Should Do
- Could Do
- Won't Do

3.1.1 Must Do

Requirements marked as 'Must Do' are those that are vital to the project being delivered. Requirements in this category can be defined by using the following reasoning; 'the product will not be deliverable by the target date without it, the product is not legal without it, the product is unsafe without it or a practical solution cannot be delivered without it' *(Walker, 2014)*.

If the requirement is not met and the result of not meeting the requirement is that the whole project should be cancelled, then the requirement can be classed as 'Must Do'. If, however, there is a workaround, even if it is a very painful and manual process then it can be marked as either 'Should Do' or 'Could Do'. If a requirement is marked as 'Should Do' or 'Could Do' it means that delivery will not be guaranteed within the specified timeframe.

3.1.2 Should Do

Requirements marked as ‘Should Do’ are the second highest priority in the MoSCoW method. These requirements can be defined using the following reasoning; ‘the requirement is important to the project but not vital to it being delivered, leaving it out may be painful but the overall solution is still viable and without this requirement being delivered a workaround, albeit temporary, may be needed, for example, some kind of paperwork or manual process’ *(Walker, 2014)*.

The difference between ‘Should Do’ and ‘Could Do’ is the amount of pain caused to the customer by the requirement not being met. This is usually measured in terms of business value or number of people affected by the requirement not being met.

3.1.3 Could Do

If a requirement is marked as ‘Could Do’ it is the third highest priority dictated by the MoSCoW method. These requirements can be defined using the following reasoning; ‘they are wanted by the customer but are less desirable than those marked as ‘Must Do’ or ‘Should Do’ and if they are left out they will have less of an impact in comparison with requirements marked as ‘Should Do’’ *(Walker, 2014)*.

These requirements are the ones that would only be delivered in their entirety in a best-case scenario. When problems and setbacks occur during the project timeline that put the deadline at risk, one or more of the ‘Could Do’ requirements are the first to be dropped from the project timeline.

3.1.4 Won't Do

Requirements marked as 'Won't Do' are ones that will not be carried out during this timeframe. These requirements are still listed on the requirements list so that the full scope of the project can be gauged; this is to avoid having to formally reintroduce them to the project team later down the line. Requirements marked as 'Won't Do' allow the development team to manage expectations with the customer that some of the requirements will not be in the final deployed solution at this time. The use of 'Won't Do' requirements allows the development team to maintain focus on the more important requirements such as 'Must Do', 'Should Do' and 'Could Do'.

3.2 Functional Requirements

By definition, a functional requirement is 'something that the system should do and more often than not these requirements will specify a behaviour or function of the system' *(Eriksson, 2012)*. The functional requirements for this project are defined below in **Table 1**.

Identifier	Requirement	Rationale	Priority (MoSCoW Method)	Success Criteria
FR-1	RPS website should be accessible via a link on the home page.	To allow seamless transition between RPDF and RPS.	Must Do	On clicking the link, the RPS site should open in a new browser tab.
FR-2	Members should be able to log on using unique credentials.	To control who has access to the application.	Must Do	Upon entering their credentials, users will be redirected to the welcome page.
FR-3	Users should only have access to areas defined by their security level.	To prevent unauthorized users from having access to sensitive information (user account details etc.)	Must Do	Upon logging in, each user type will have a slightly different UI, i.e. different menu icons will be available for different user types.
FR-4	Search facility only accessible	To prevent unauthorized users from	Must Do	When trying to navigate to the search page, users

	to signed up users.	having access to the database.		will only be able to access it if they are logged in. If they are not logged in, they will be redirected to the login page.
FR-5	Searches will be able to be refined by up to 7 secondary filters.	To allow users to search the database to the level of granularity they require.	Must Do	After performing an initial search. A user may enter any number of secondary filters up to 7, the results list should become smaller as more filters are added.
FR-6	The results of each search should be displayed on the screen in a table.	To allow users to easily see the raw data they have extracted.	Must Do	Once a search is carried out, a table is produced containing the expected amount of results.
FR-7	The results table should be able to be printed.	To allow users to have a hard copy of their results.	Should Do	After a results list has been displayed and the print button pressed, the print manager should automatically pop

				up an provide the option to print.
FR-8	The results table should be downloadable in the following formats: CSV, Excel and PDF	To allow users to save copies of their results to their computers.	Should Do	After a results list has been displayed and the appropriate download button pressed, the results list should automatically be downloaded in the selected format.
FR-9	Results page will display page x of y and number of total records	This will allow users to see the complete length of the report produced	Should Do	On the results page a pagination system should be displayed and the total number of records for that result set should also be displayed and should be accurate.
FR-10	A processing bar should be displayed when a search is being carried out.	To give users a visual representation of how long the query has left to run.	Could Do	Once the search button is pressed a pop up should display a processing bar that runs for as long as the query is taking.

FR-11	A term of use and bulletin board should be displayed on login.	To show the terms of use of the application to all registered users.	Could Do	Upon login a text box containing the terms of use appears and is visible to all users.
FR-12	A ‘statistics’ page should be created containing graphs and tables with details of the information the RPDF has on hand.	To show prospective users the kind of information the RPDF stores.	Could Do	When clicking on the ‘statistics’ page, accurate graphs and tables will be displayed detailing the contents of the database in a visual format.
FR-13	A live arrivals and departures board should be available to all registered users.	To allow users to check arrivals and departures for stations they may be visiting on their journeys and to check for delays on their current service.	Could Do	Upon entering a valid station code, a table containing the arrivals or departures for the selected station will appear. The table will contain the service, platform, due time and any reason for delay.
FR-14	The application should be available on	To allow users to use the application if	Could Do	The application will provide the same functionality

	mobile devices such as smartphones and tablets.	they do not have access to a laptop or desktop computer.		and mostly the same display on mobile devices as it does on desktop computers and laptops.
FR-15	Any errors displayed should be in meaningful text and should be summarised to the administrator in a daily log.	To allow errors to be captured and corrective action to be taken against them.	Won't Do	Upon receipt of a generic error an error page is shown, and the in-depth detail is written to the application server logs.
FR-16	Login details for members should be synchronised between the RPS website and the RPDF.	To prevent users from needing multiple login credentials.	Won't Do	Logins to the RPDF and RPS website will be successful when the user is using the same credentials for both systems.
FR-17	All records will be linked to their corresponding photos stored on the RPS' FTP site.	To allow users to view the original logs for the record they are viewing.	Won't Do	After searching, a link will be displayed next to each record which will open a new tab with the correct image for that log.

***Table 1**: Functional Requirements for RPDF*

FR-1 to FR-6 are the 'Must Do' requirements for this project. FR-1 was chosen as a 'Must Do' because it is essential that both the **RPS** website and **RPDF** application are linked to one another, so if users wish to go from one to the other, they have the ability to do so. Although the **RPDF** is being separated from the **RPS** website they are not becoming totally separated so this is why FR-1 is a 'Must Do' requirement.

FR-2 is a 'Must Do' because it is essential that each user has their own unique set of login credentials and that they are able to log in to the system individually. This is a very basic requirement but is fundamental to the application being viable.

FR-3 is heavily linked to FR-2 as it is vitally important that users are only given access to the areas of the application that they actually need to use. For example, you would not want a regular user having access the user management area of the application because with that level of access they could easily delete every user on the system. FR-3 could possibly be considered the most important requirement in this list.

FR-4 is similar to FR-3 in that no outside users should be able to directly access any of the webpages that would require a user to be logged in. This is to prevent unauthorised access to the application from anyone with malicious intent.

FR-5 is listed to provide the very basic search functionality, without this the application would just be a collection of web pages, with no functionality whatsoever. FR-5 is essential to the performance and basic implementation of the solution.

FR-6 is again linked to FR-5. It is a 'Must Do' because it is an essential requirement to provide basic functionality for the application. Without it there would be no display for the results of a user's search query, thus rendering it useless.

Requirements FR-7 to FR-9 are the 'Should Do' requirements. These are the requirements that would mean that the application could ideally do with them, but if

they were not included a work around could be put in place to deal with their exclusion.

FR-7 and FR-8 are both linked to the results table and what happens to it after a search is carried out. FR-7 would be good to have included as it would mean that users could print out a hard copy of their results to keep for their own records. FR-8 is very similar in that it would be good to include the functionality to save their results onto their computer, but it would not be essential to the overall performance of the application.

FR-9 is again linked to the results table, however it is not essential to have the total number of retrieved records displayed on the bottom of the table. While it is a nice addition, not having it does not take anything away from the basic functionality of the application.

FR-10 to FR-14 are the requirements marked as ‘Could Do’, these are the requirements that will most likely be missed off of the final implementation should problems with the higher priority requirements arise.

FR-10 is marked as ‘Could Do’ as it is purely cosmetic and does not really add anything to the application, most browsers will provide some kind of loading bar to show how long the web page has left to load.

FR-11 is again, a cosmetic feature of the system that will not impact overall performance, thus it is a ‘Could Do’ requirement that can easily be implemented at a future date.

FR-12 is a very database intensive requirement that can be created but will take a lot of work to implement and configure, therefore it may be a struggle to implement within this timeframe. However, not including it will not reduce the overall effectiveness of the application as it is purely to offer prospective users an insight into the contents of the database.

FR-13 is reliant of the successful implementation of the National Rail's webservices in order to function. However, it is just an addition to the main functionality of the application that will allow the **RPDF** to become a more rounded product, rather than simply a search engine tool. Not including it will not take anything away from the main application and it could be implemented at another date so therefore it is being classified as 'Could Do'.

FR-14 is being classified as 'Could Do' because, while indeed very useful, it will not really take away from the overall functionality of the system. Although it would reduce its overall usefulness if it is not available on mobile devices it could be implemented at a later date without affecting the initial deployment of the **RPDF**. The use of **JSF** should prove to overcome this problem as responsiveness is part of the selling point of **JSF** and BootsFaces so that very little configuration is needed in order to get it working correctly on mobile devices.

The final three requirements, FR-15 to FR-17, are all marked as 'Won't Do'. This is because in the timeframe given, it would not be possible to implement them as they would cause too many problems and would rely too heavily on external systems in order to function correctly.

FR-15 has been marked as 'Won't Do' because all errors of any significance are captured by the application server and written to the server logs which can be viewed on the administration console by the system admin. A generic view of all errors is given when they are received, for example a 404-error page will appear if a user enters the incorrect URL.

FR-16 is a 'Won't Do' simply because it is not known how user credentials are managed at the moment, so trying to integrate the two would be physically impossible for this timeframe. Most browsers offer the option to remember login credentials so having one set of credentials for the **RPS** website and one for **RPDF** should not cause the user any great difficulties.

FR-17 is a ‘Won’t Do’ requirement because it is currently not known how to fetch the photos for the corresponding log files in the database. This would have to be implemented after the application has gone live and the live database is connected. As they are stored in an **FTP** server, the connection between the application server and the **FTP** server would need to be configured and thoroughly tested prior to being deployed. This would not be a feasible action to complete within the given timeframe.

3.3 Non-Functional Requirements

In contrast to a function requirement, a non-functional requirement is something that describes how the system should work and these requirements are commonly used to describe quality attributes of the system. Non-functional requirements are also used to 'judge the overall operation of a system, rather than specific behaviours' *(Eriksson, 2012)*. The non-functional requirements for this project are listed below, in **Table 2**.

Identifier	Requirement	Rationale
NFR-1	The system must be able to support at least 10 users running large queries simultaneously.	To allow multiple users to access the application without affecting waiting times for queries.
NFR-2	All queries should take no longer than 2 minutes to complete.	To maintain good loading times across the system and to not overload the server resources or database.
NFR-3	The system must be available 24/7, 365 days a year.	To allow for continuous use of the application regardless of which country the user is in.
NFR-4	The system must have a simple UI that is understandable for the older generation who may not be comfortable with technology.	To allow users of all ages to access and use the application, regardless of their technological ability.

***Table 2**: Non-Functional Requirements for RPDF*

NFR-1 has been selected as a non-functional requirement because it is vital for the system to be able to support multiple simultaneous users without noticeably affecting the performance of the system. If this is not met it could mean that once multiple users are using the system, it will become very slow very quickly and will discourage people from using it.

NFR-2 is a requirement because users will quite often become impatient if queries take a long time to complete. If a user becomes impatient and starts clicking buttons it could have the potential to crash the system and cause severe database problems for other users. Therefore, it is essential that the backend architecture is designed to be robust enough and quick enough to process queries from multiple users in a reasonable amount of time.

NFR-3 is essential because if the application is not available for lengthy periods of time then over time it will lose traffic until eventually it is not being used at all. The customers may want to access it at any time from any location, so it needs to be able to be kept live at all times.

NFR-4 is important to this project because the target demographic for this application is the older generation who may or may not be comfortable with technology. If the **UI** is not simple to navigate then it will take away from the whole point of this application, which is to upgrade the existing system to make it easier for users to build reports based on the results of their database queries. If this is not met, then it will result in the target demographic not using the application as it is too confusing to use. This will ultimately mean that the audience this application is aimed at will not use it so the project will effectively be rendered useless.

4 Design

After the requirements for this project had been gathered, analysed and prioritised, the application itself had to be designed. The application was designed so that in an ideal scenario all of the requirements listed above could be met.

4.1 Use Cases

There are two main Use Cases for this application, one that covers the entirety of the functionality of the application with the assumption that all users are already logged in, and one that covers the log in process of the application. **Figure 1** shows the Use Case Diagram that encapsulates the entire functionality of the **RPDF**.

4.1.1 RPDF Use Case Diagram

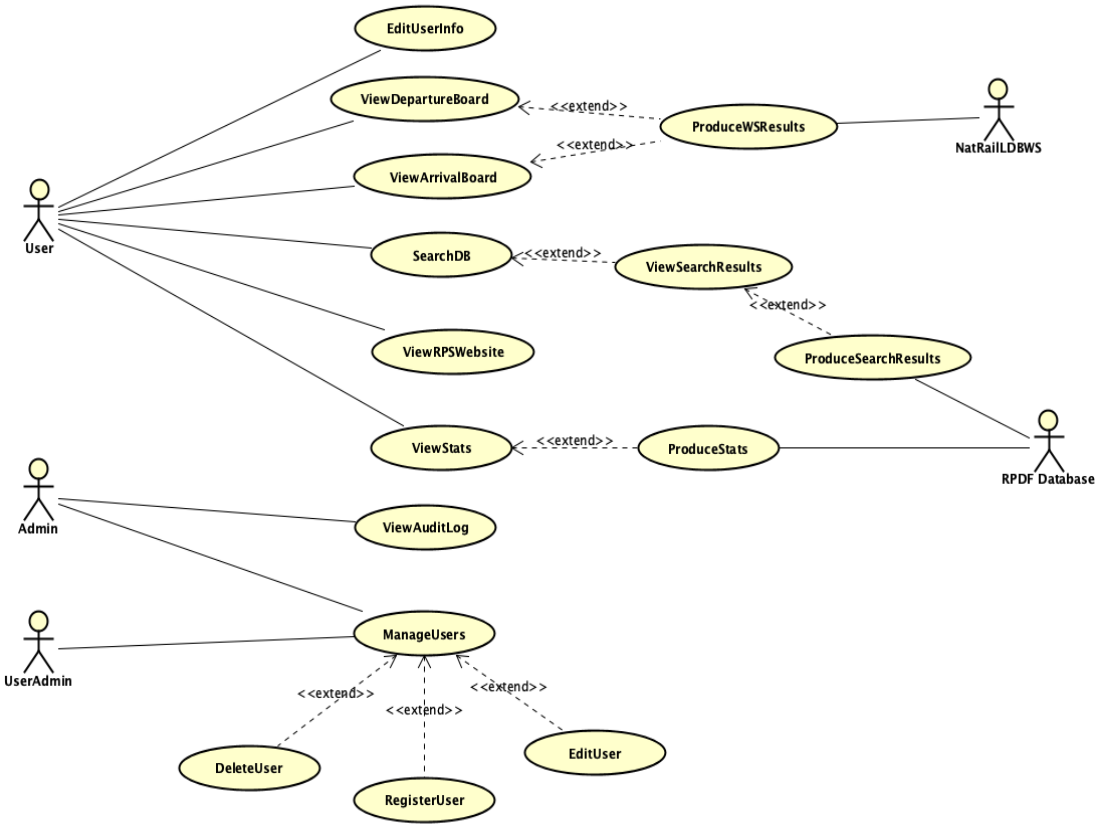

***Figure 1**: RPDF Use Case Diagram*

In this Use Case the most noticeable feature is that there are three different types of user; these are shown by the actors, ‘User’, ‘UserAdmin’ and ‘Admin’. These user types are in line with FR-3 which will prevent regular users from accessing areas that they are not entitled to access, such as the Audit Log or User Management areas of the application.

Users with the user type of ‘User’ will have access to all of the core functionality of the application, such as the Database Search function, Arrival and Departure Boards, the external **RPS** website and the Statistics page. Regular users will also have access to edit their own information, such as name, username and password.

The two administrative users, ‘UserAdmin’ and ‘Admin’ have the sole responsibility of administering the application. The ‘UserAdmin’ users will be responsible for

managing users, be it deleting them or editing them. They will also have the ability to register new users as the current process for registering new users does not allow them to register themselves. New users have to subscribe to the **RPDF** via the **RPS** website, upon subscription the 'UserAdmin' will receive an email and they will create the new user's account there and then.

The 'Admin' user has a very similar role to that of the 'User Admin', the main difference being that they have access to the audit log which provides full details of all activity that goes on within the application. The audit log covers everything from users logging in to the application, to users performing searches, deleting, editing and creating other users. Should all of the 'UserAdmin' users be unavailable the 'Admin' users also have permissions to manage users so that there will always be someone available to manage users at a moment's notice.

In this Use Case, the **RPDF** database is shown as an actor, this is because it is an external system that is access via **SSL** by the **RPDF** application. The primary roles of the **RPDF** database is to return the results to a user's query and to produce up to date statistics to be viewed on demand on the 'Statistics' page by a user.

Similarly, the National Rail Web Service is shown by the actor 'NatRailLDBWS', this is also because it is an external system that is only accessed when requested by the user. The main role of this system is to produce the results of the web service call whenever it is initiated by the user, it is responsible for passing the information to the controller to be presented to the viewer in a meaningful way.

4.1.2 Login Use Case Diagram

The Use Case above assumes that the user is already logged in, the Use Case that cover the log in process is below in **Figure 2**.

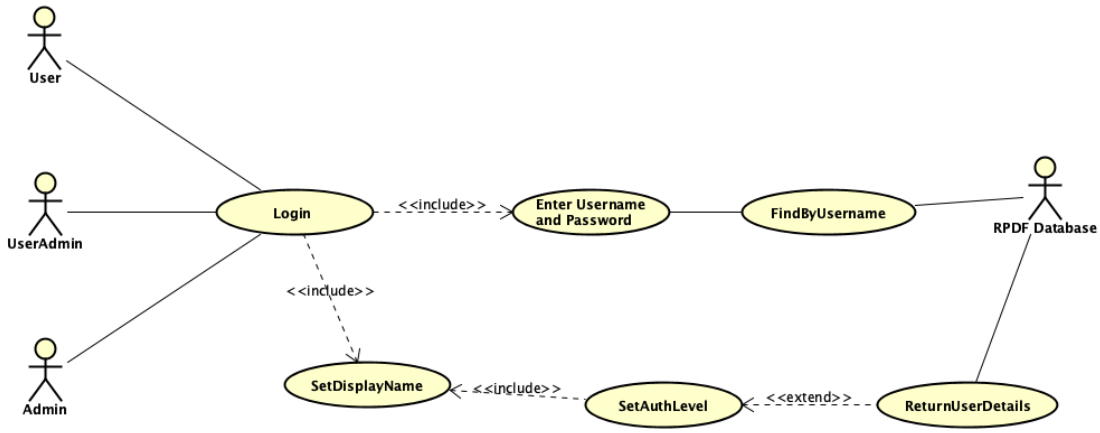

Figure 2: *Login Use Case Diagram*

This Use Case shows the process for logging in and setting a user's permissions for that session.

All users log in via the same login **UI**, by entering their unique credentials, as outlined by FR-2. The username and password entered are then passed to the **RPDF** Database which has a 'user' table that contains the details for every user, this includes their full name, username, password and user type. The details for the user that have been retrieved are then passed back to the controller which sets the permission level and display name for that user. Their permission level will dictate what they can and cannot see on their **UI** once they have logged in, as defined in FR-3.

4.2 Database Design

As this project is heavily reliant on the integrity of the database to function it was of vital importance that it was designed correctly to ensure it was as robust as possible so that FR-5 could be met successfully.

Before development began, a snapshot of the database was taken, and the data was used to produce the database that was to be used during the development and testing phase of production.

The live database is currently hosted externally so every effort was made to ensure that the database used in development and testing was as close to the live database as possible. **Figure 3** shows the structure of the database used in the development and testing phase.

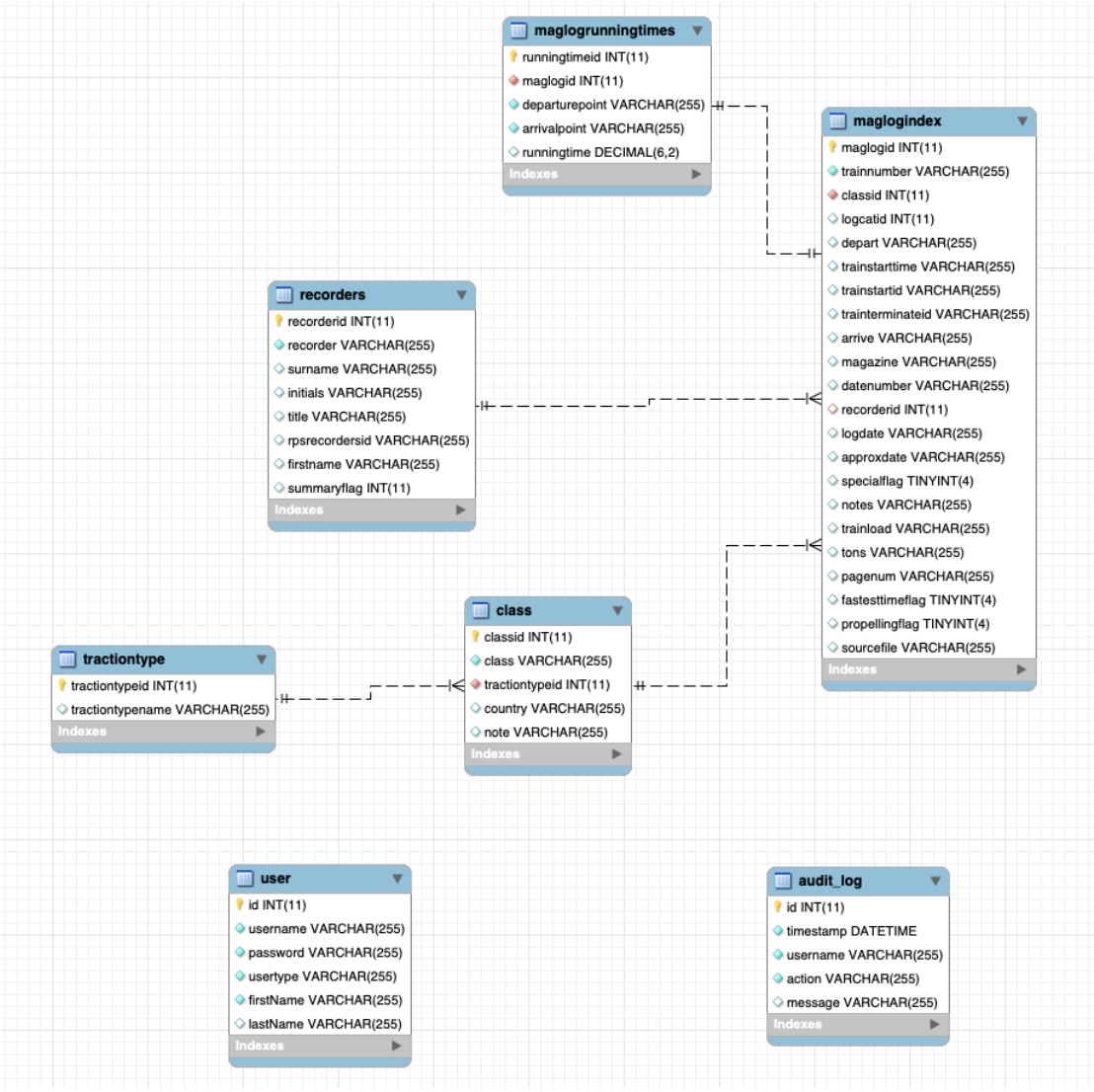

***Figure 3**: RPDF Relational Database Diagram*

The database for **RPDF** is fairly simple in the number of tables that it has, however, its complexity comes when trying to traverse the tables to extract information from the database.

The main table in the database is the '*maglogindex*' table, this table contains the vast majority of the data in the database, it currently contains approximately one million rows of data that has been collected by the **RPS**. The table forms the backbone of all queries performed on the database by the application as the rest of the tables are related to it via varying different relationships; these are detailed in the diagram above.

As the other tables contain data that is fairly unique, for example the recorder table which contains details of every person who has recorded data for the society, the design choice was made to keep those tables totally separate from the bulk of the data which is stored in the '*maglogindex*' table. This makes accessing the data from them a much simpler process.

The second most important table in the database is the '*maglogrunningtimes*' table. This table contains the run time for each log in the '*maglogindex*' table and contains the same amount of data as the '*maglogindex*' table. The '*maglogrunningtimes*' and '*maglogindex*' tables have a one-to-one relationship, this means that for every entry in the '*maglogindex*' table there is only one corresponding entry in the '*maglogrunningtimes*' table. They have this relationship because there can only be one run time for each entry in the '*maglogindex*' table, otherwise if multiple run times existed for the same entry there would be a lot of confusion as to which run time was correct for that entry. The one-to-one association also means that when querying the database, it is easy to obtain a set of totally unique results as the '*runningtimeid*' field is totally unique for each entry in the '*maglogrunningtimes*' table.

The '*recorders*' table is the next most important table in the database; this table contains data on every single person who has ever recorded data for the society. It has a one-to-many relationship with the '*maglogindex*' table. This means that for every entry in the '*recorders*' table, there can be multiple different entries in the '*maglogindex*' table. There are roughly 700 entries in the recorder table, some of the recorders have over 20,000 entries in the '*maglogindex*' table associated with them.

The final two tables that have relationships with the '*maglogindex*' table are the '*tractiontype*' and '*class*' tables. The '*tractiontype*' table does not have a direct relationship with the '*maglogindex*' table, however the information it provides is still vitally important to the queries that the **RPDF** will run. The '*tractiontype*' table contains 9 different types of train, ranging from Steam to **HST** and has a one-to-many relationship with the '*class*' table. The '*class*' table contains data on every class

of locomotive that the **RPS** has ever collected data on, it's relationship with the '*tractontype*' table allows us to associate certain classes of locomotive with their respective traction type; for example, class 50 may have a traction type of Diesel. The '*class*' table has a one-to-many relationship with the '*maglogindex*' table and therefore, by association, the '*tractiontype*' table also has a one-to-many relationship with the '*maglogindex*' table.

There are two tables in **Figure 3** that do not have a relationship with any other table, these are the '*user*' and '*audit_log*' tables. The '*user*' table contains the information on all of the users that are registered with the **RPDF**, including their name, username, password and their user type or permission level. The '*audit_log*' table is responsible for keeping a log of all of the activity that goes on within the **RPDF**, should the need of an audit arise. It collects data such as a record of every user that logs in, every time a search is carried out and other details such as a record of whenever a user is added, deleted or edited and the user who carried out those actions.

4.3 Sequence Diagrams

There are two main functions that have been covered by the sequence diagrams, these are the login process and the search process. **Figure 4** shows the login process.

4.3.1 Login Sequence Diagram

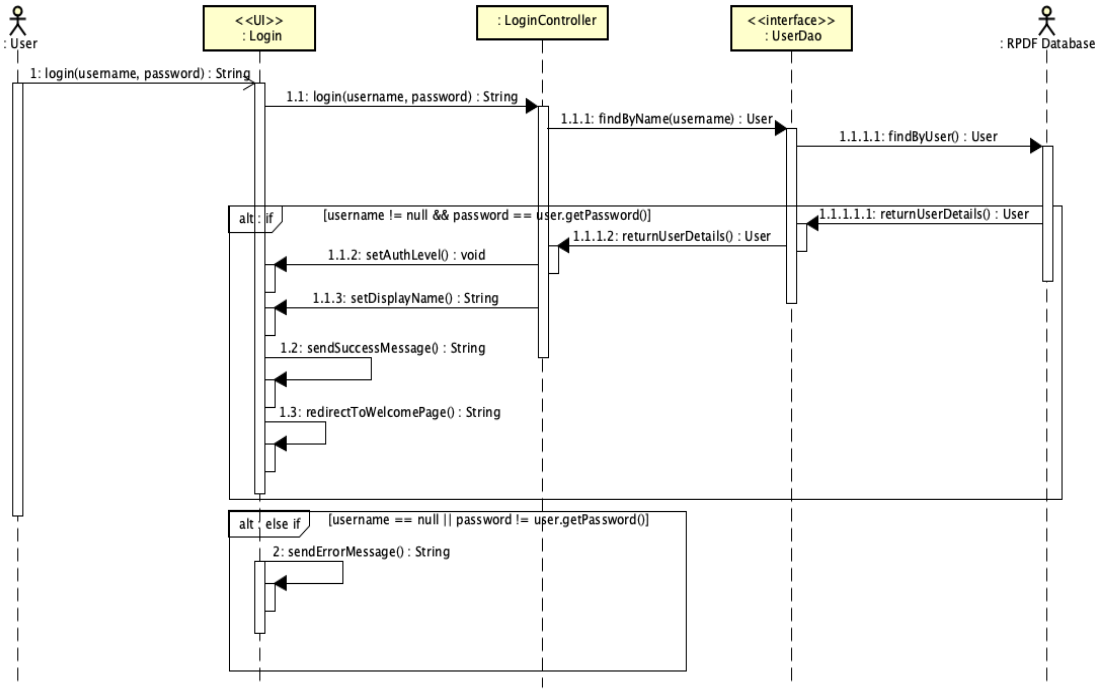

Figure 4: *RPDF Login Sequence Diagram*

The two actors in this sequence diagram are the User and the **RPDF** Database, the **RPDF** Database is listed as an actor because it is an external system. The first thing that should be noticed is the notation on the '*Login*' and '*UserDao*' timelines. The '*Login*' timeline has the notation of '<<UI>>', this is because the '*Login*' timeline represents the login webpage of the **UI** so is a user facing part of the system. The '*UserDao*' timeline has the notation of '<<interface>>', this is because it is an interface between the application and the underlying database.

In this sequence, the first part of the process is that the user enters their unique username and password, in accordance with FR-2. The details the user just entered are passed to the '*LoginController*', this controller is responsible for handling all login

and log out requests. To perform the login process, the '*LoginController*' calls a method in the '*UserDao*' called '*findByName*'. This method searches the '*user*' table in the database for the username entered by the user in the login **UI**.

The next part of the login process relies on the if statement detailed in the sequence diagram. If the username is not null, or matches the one in the database, and the password entered is the same as the one in the database, then the user's details will be passed from the database to the '*UserDao*' and then from the '*UserDao*' to the '*LoginController*'. Once the '*LoginController*' has the user's details, it will then set the permission level of the user and pass that back to the '*Login*' **UI**, the '*LoginController*' will then set the user's display name to their username. Once those values have been set a success message will be sent and the user will be redirected to the welcome page. The user's values that were set by the '*LoginController*' will then be retained for the remainder of the session, the **UI** will be different for each user dependent on their permission level, as detailed in FR-3.

If the username is null or is not found in the database or the password entered does not match the password stored for that user in the database, then an error message will be displayed. The error message will be different depending on what the error was, for example, if the username was not found the error message would say something like '*Invalid Username!*'.

The second sequence diagram is to show how the process of searching the database would work, **Figure 5** shows this process.

4.3.2 Search Sequence Diagram

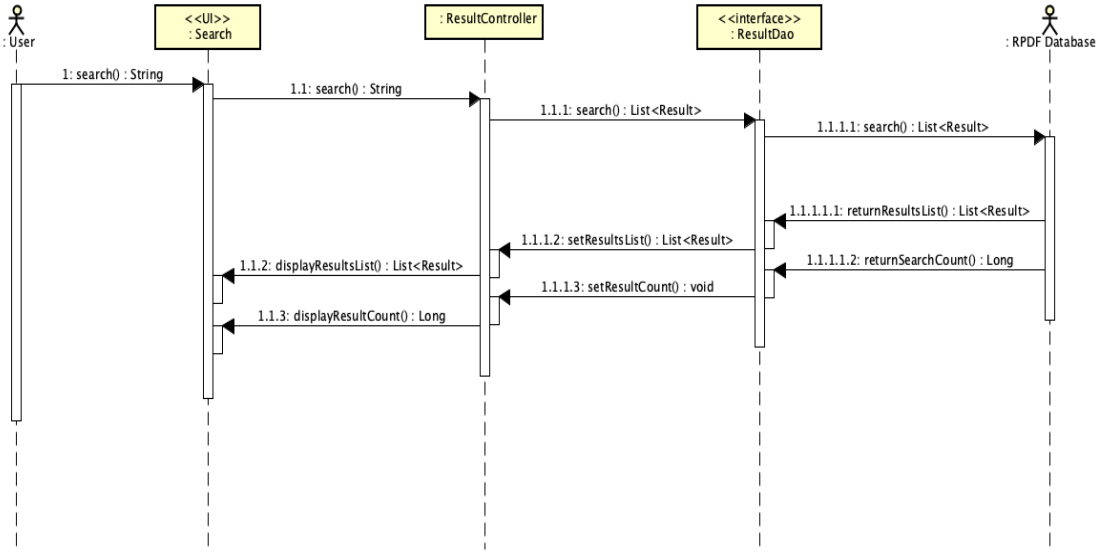

***Figure 5**: RPDF Database Search Sequence Diagram*

Similarly to **Figure 4**, the two actors in this sequence are the User and the **RPDF** Database, the '<<UI>>' is the '*Search*' timeline and the '<<interface>>' is the '*ResultDao*' timeline.

The search sequence starts when a user enters their search criteria into the search page. Those criteria are then passed to the '*ResultController*' which passes them to the '*ResultDao*'. The '*ResultDao*' then searches the database with the criteria given to it by the '*ResultController*', the database then performs two queries, one to get a list of the results and another to retrieve a count of those results for displaying on the results page in accordance with FR-9. The list of results and count of results are then passed back to the '*ResultDao*' which then returns them to the '*ResultController*'. After this the '*ResultController*' sets the result list and result count and displays them in the '*Search*' **UI** in a table with the count of results underneath in accordance with FR-6 and FR-9.

4.4 Class Diagrams

There are four class diagrams for this project, one for the Controller package, one for the **DAO** package, one for the Model package and one for the Services package. They are all interlinked so there are some shared classes across them, the shared classes have been included to show the interaction between the classes and packages.

4.4.1 Model Class Diagram

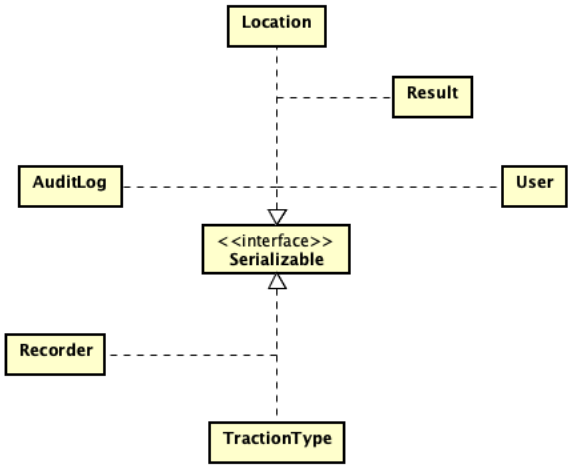

Figure 6: *RPDF Model Class Diagram*

As can be seen in **Figure 6**, the model package is fairly simple, this is because it only contains the basic Java classes for the application, or **POJO**s, for short. These model classes include all of the attributes to be used across the application, for example, the '*User*' model class contains the name, username and user type attributes for a User object. The diagram also shows that all of the model classes above have a realization relationship with the '*Serializable*' interface. This means that all of the models implement the operations and attributes of the interface. The reason that all of the model classes have this relationship is because the IDs for the database tables that are associated with each model class generate the IDs for each entry automatically. Therefore, the design choice was taken to generate them in a serialized manner, so the IDs would be 1, 2, 3, 4, 5, and so on.

4.4.2 Services Class Diagram

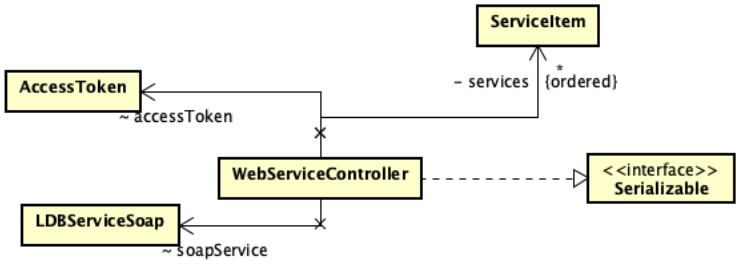

Figure 7: *RPDF Services Class Diagram*

As can be seen in **Figure 7**, the class diagram for the '*Services*' package is similar to the '*Model*' package in that the main class in the package, '*WebServiceController*' realizes the '*Serializable*' interface. The '*WebServiceController*' class also has an association with the '*AccessToken*', '*LDBServiceSoap*' and '*ServiceItem*' classes; these are all objects provided by the National Rail's **LDBWS** web service used to generate live arrival and departure boards needed to meet FR-13.

While the relationship between the '*WebServiceController*' and '*AccessToken*' and '*LDBServiceSoap*' classes is quite simple, the relationship between the '*WebServiceController*' class and '*ServiceItem*' is slightly more complex. The '*ServiceItem*' class produces an ordered list on '*service*' objects, which are then fed back to the '*WebServiceController*' class to be displayed on the webpage; with a multiplicity of *, there is also no limit as to how many '*service*' objects are fed back to the '*WebServiceController*' class.

In contrast with the '*Model*' and '*Services*' packages, the '*Dao*' and '*Controller*' packages are much more complex. This is because the majority of the application's processing capacity is delivered in these packages.

4.4.3 Dao Class Diagram

A **DAO**, or Data Access Object, is a Java Object that is used to provide an interface between the database and the controllers used in a typical **MVC** application to access the data contained in the database. As can be seen in **Figure 8**, the '*Dao*' package contains a lot of dependency, realization, and generalisation relationships.

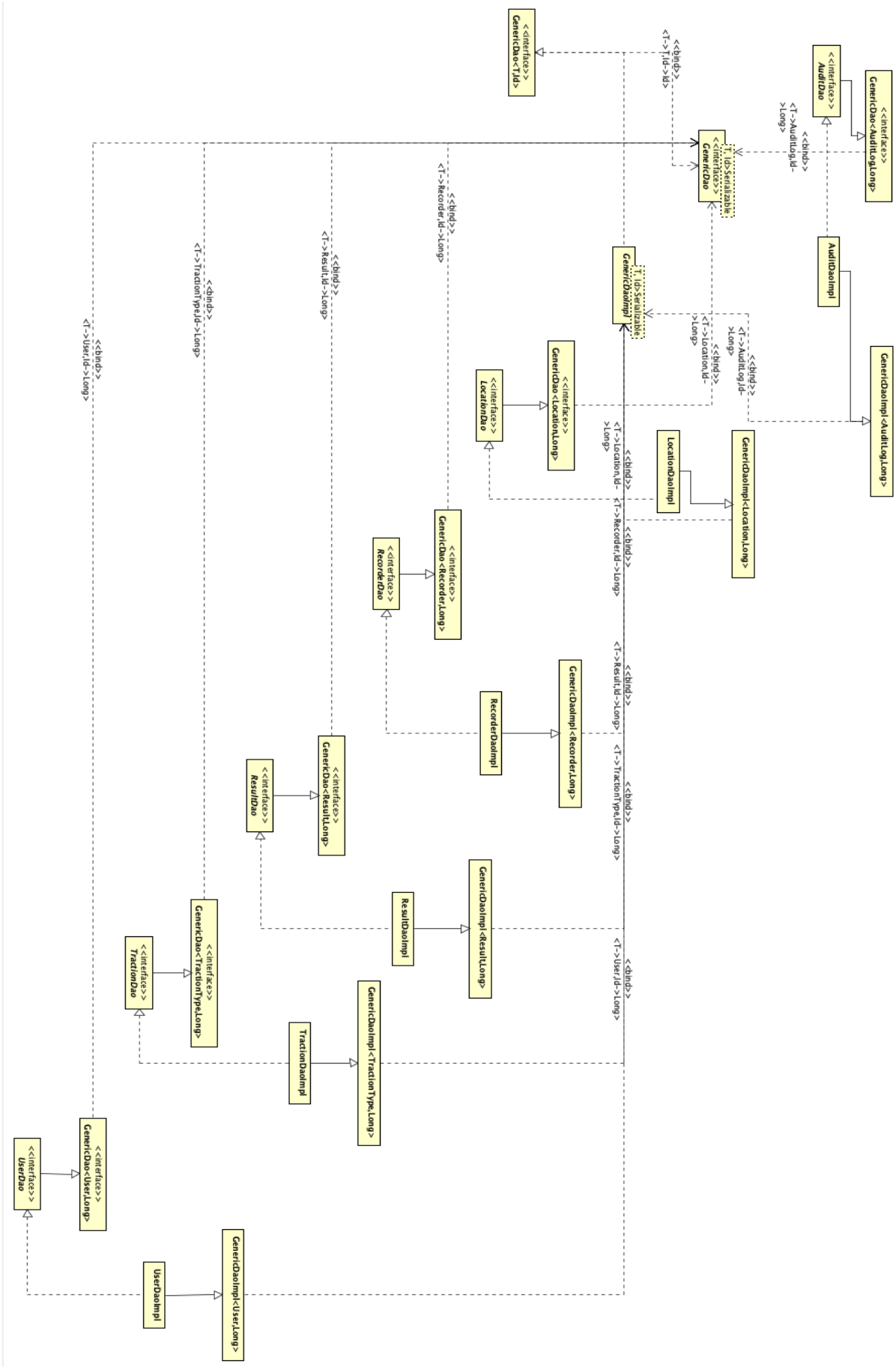

Figure 8: *RPDF Dao Class Diagram*

The central components of this package are the '*GenericDao*' and '*GenericDaoImpl*' classes, they contain the core functionality that is inherited by all of the classes in the '*Dao*' package. All of the classes in this package have a dependency on either the '*GenericDao*' or '*GenericDaoImpl*' classes. This means that if the '*GenericDao*' or '*GenericDaoImpl*' classes change, then the implementation of any of the dependent classes may also change. All of the specific '*GenericDao*' classes, for example the '*GenericDao<Location,Long>*' class, are dependent on the '*GenericDao*' class. The **DAO** classes associated with the Model classes, for example the '*LocationDao*' class, have a generalisation relationship with the specific '*GenericDao*' class for that Model. In this example the '*LocationDao*' class has a generalisation relationship with the '*GenericDao<Location,Long>*' class; a generalisation relationship means that the specific class inherits part of its definition from the generic class. Attributes, associations and operations are all inherited from the generic class by the specific class.

The implementation of the specific **DAO** class, in this example the '*LocationDaoImpl*' class, has a realization relationship to its interface counterpart, '*LocationDao*'. Just like the '*LocationDao*' class, the '*LocationDaoImpl*' class also has a generalization relationship with its specific '*GenericDao*' implementation, in this case it is the '*GenericDaoImpl<Location,Long>*' class.

4.4.4 Controller Class Diagram

Finally, the Controller class diagram shows the relationship between the Models, their respective **DAO**s and the Controllers used to control them. This can be seen in **Figure 9**.

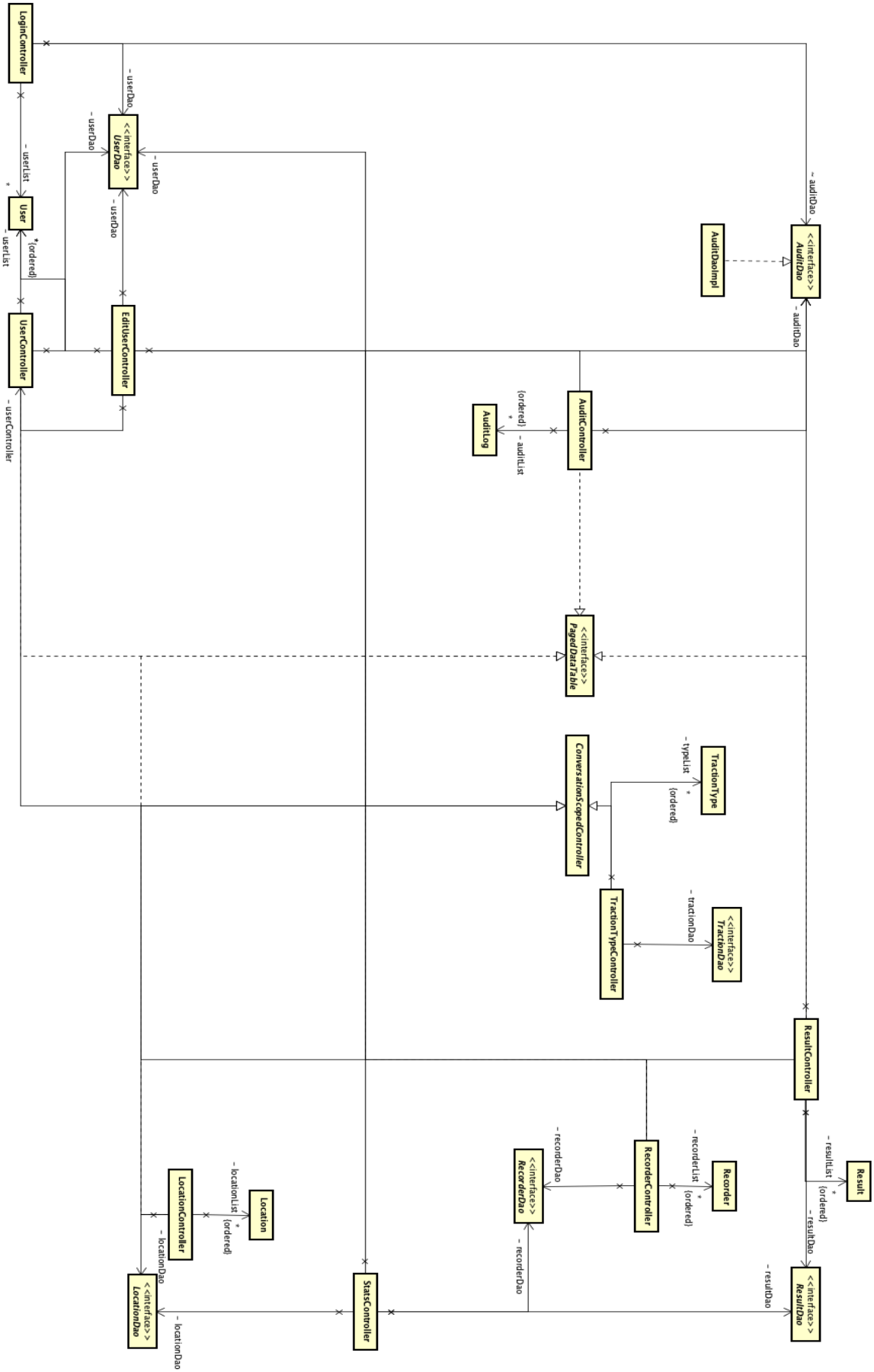

Figure 9: *RPDF Controller Class Diagram*

This class diagram is focused mainly around the '*ConversationScopedController*' class and the '*PagedDataTable*' interface, it assumes that it is known that all of the Model classes have a realization relationship with the '*Serializable*' interface. All of the controllers in this diagram have a generalisation relationship with the '*ConversationScopedController*' class. This class allows data that is produced by these controllers to be kept for the duration of the conversation, once the conversation is closed, the data is lost. All of the controllers in the diagram also have a realization relationship with the '*PagedDataTable*' interface. This interface was designed as a custom interface for displaying data in a data table to overcome problems that were encountered when displaying large datasets in a default data table. As all of the controllers will display data of some sort in a data table it was vitally important that this was implemented so that FR-6, FR-7 and FR-8 could be met.

The rest of the diagram is fairly similar to the other diagrams above, if we take the '*RecorderController*' class as an example, we can see that is associated with the '*RecorderDao*' interface and '*Recorder*' model class. The '*Recorder*' model class produces an ordered list of '*Recorder*' objects and has a multiplicity of many to one.

There is only one controller in this diagram that has associations with multiple **DAO**s, the '*StatsController*'. The '*StatsController*' has associations with multiple **DAO**s because it uses data from multiple sources in order to produce the live graphs that will be presented on the statistics page to meet FR-12. The **DAO**s that the '*StatsController*' has associations with include, '*RecorderDao*', '*ResultDao*', '*LocationDao*' and '*UserDao*'.

5 Implementation

Due to the nature of this project, and the fact that the development team consisted of one person, it was challenging to follow any specific methodology directly; therefore, different aspects of both the Agile and Waterfall methodologies were used when implementing the design for this project.

The aspects of the Agile Method that were used during the development of the application were that the end users were actively involved in making decisions on how the application should look, feel and function. The requirements evolved over time but the timeline for the project remained fixed and during development there was continual collaboration and cooperation between all stakeholders involved *(Anon, 2018)*.

The majority of the project was carried out using the Waterfall Method, more specifically, the '*V Model*'. The V Model is an extension of the Waterfall Method and is used to establish the association between each phase of development with each phase of testing, these are categorized as the Verification Phase and Validation Phase respectively *(Anon, 2019)*. The V Model is incredibly useful for projects that do not have a large timeframe and that have well defined, clearly documented requirements with technology that is well understood by the development and testing teams. **Figure 23** *(Anon, 2019)*, shows the V Model in diagrammatical form.

5.1 Login Implementation

The login function of the application was one of the most important functions. If it is not implemented correctly then some users could potentially gain access to areas of the application that they should not have access to, for example, the administration area of the application.

There are two main parts involved in the implementation of the login process, the first is the login method itself and the second is how the permission levels are set. The code for the login method can be seen in **Figure 10**.

```
public String login() {
    User user = userDao.findByName(username);// find all users by username

    if (user == null) {// if the username isn't in the database
        messageUtil.sendErrorMessage("Please enter a valid Username");// display error
        return "";// return null
    }

    if (!user.getPassword().equals(password)) {
        messageUtil.sendErrorMessage("Incorrect Password!");
        return "";
    }

    if (user.getUsertype().equals("ADMIN")) {
        authService.setSessionAttribute(AuthService.SESSION_AUTHLEVEL, AuthService.AUTHLEVEL_ADMIN);
    } else if (user.getUsertype().equals("USERADMIN")) {
        authService.setSessionAttribute(AuthService.SESSION_AUTHLEVEL, AuthService.AUTHLEVEL_USERADMIN);
    } else if (user.getUsertype().equals("USER")) {
        authService.setSessionAttribute(AuthService.SESSION_AUTHLEVEL, AuthService.AUTHLEVEL_USER);
    }

    authService.setSessionAttribute(AuthService.SESSION_USERNAME, user.getUsername());// set the username to that of
                                                                                       // the current user
    authService.setSessionAttribute(AuthService.SESSION_DISPLAYNAME,
            user.getFirstName() + " " + user.getLastName());// set the first and last name to that of the current
                                                            // user
    auditDao.addEntry(ActionType.LOGIN, "User " + username + " logged in");
    messageUtil.sendInfoMessage("Welcome " + user.getFirstName() + " " + user.getLastName() + "!");// display
                                                                                                   // success
    return "/secure/welcome?faces-redirect=true";// open welcome page
}
```

Figure 10: *RPDF Login Method*

When a user enters their username and password in the login screen, they are both passed to the login method in **Figure 10**. First a '*User*' Object is created by searching the User table in the database by the username entered via the '*UserDao*' interface. Then, if the User Object is null, or the username is not in the database, then an error message is returned, and the user is redirected to the login screen. The same check is also carried out for the password that has been entered.

With this method there are a lot of different attributes that are set, the first of these is the user's permissions. Once the user's details have been retrieved from the database, the '*usertype*' field is then checked and their permissions for that session are then set. This is all done through the '*AuthService*' class which has the sole responsibility of providing authorisation for the application, the code for the AuthService class can be seen in **Figure 11**.

```
package uwe.rpdf.controller;

import java.io.Serializable;

import javax.enterprise.context.SessionScoped;
import javax.faces.context.FacesContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.ws.rs.core.Context;

@SessionScoped
public class AuthService implements Serializable {

    private static final long serialVersionUID = 1L;

//  private static final Logger LOGGER = Logger.getLogger(AuthService.class.getName());

    // Session attribute constants
    public static final String SESSION_USERNAME = "username";
    public static final String SESSION_DISPLAYNAME = "displayname";

    public static final String SESSION_AUTHLEVEL = "authlevel"; // USER/USERADMIN/ADMIN

    public static final String AUTHLEVEL_ADMIN = "ADMIN";
    public static final String AUTHLEVEL_USERADMIN = "USERADMIN";
    public static final String AUTHLEVEL_USER = "USER";

    @Context
    private HttpServletRequest jaxrsRequest;

    protected HttpSession getSession() {
        FacesContext context = FacesContext.getCurrentInstance();
        HttpServletRequest request;

        if (context != null) { // Faces
            request = (HttpServletRequest) context.getExternalContext().getRequest();
        } else { // JAX-RS
            request = jaxrsRequest;
        }

        return request.getSession(false);
    }

    protected boolean isLoggedIn() {
        return (getSession() != null) && (getSessionAttribute(AuthService.SESSION_USERNAME) != null) ? true : false;
    }

    public void setSessionAttribute(String name, Object value) {
        getSession().setAttribute(name, value);
    }

    public Object getSessionAttribute(String attribute) {
        return getSession().getAttribute(attribute);
    }

}
```

***Figure 11**: RPDF AuthService Class*

Once the user's permissions have been set, their username and display name are then set for the session. All session attributes are set by the AuthService, a session will automatically time out after 20 minutes of inactivity unless the session is terminated by the user beforehand by logging out, for example. The username and

display name are set as session attributes so that they can be used to create a more personalised **UI** once the user has logged in.

After the session attributes have been set an entry is made in the audit log table in the database, a welcome message is displayed, and the user is redirected to the welcome page.

Once a user is logged in, the application needs to check if they have the required permissions to view certain parts of the application before rendering them in. This is done using the '*requiresAuthLevel*' method, the code for which can be seen in **Figure 12**.

```
public boolean requiresAuthLevel(String levelRequired) {
    boolean authorised = false;

    final String userAuthLevel = (String) authService.getSessionAttribute(AuthService.SESSION_AUTHLEVEL);

    if (userAuthLevel != null) {
        switch (levelRequired) {
        case AuthService.AUTHLEVEL_ADMIN:
            if (userAuthLevel.equals(AuthService.AUTHLEVEL_ADMIN))
                authorised = true;
            break;
        case AuthService.AUTHLEVEL_USERADMIN:
            if (userAuthLevel.equals(AuthService.AUTHLEVEL_ADMIN)
                    || userAuthLevel.equals(AuthService.AUTHLEVEL_USERADMIN))
                authorised = true;
            break;
        case AuthService.AUTHLEVEL_USER:
            if (userAuthLevel.equals(AuthService.AUTHLEVEL_ADMIN)
                    || userAuthLevel.equals(AuthService.AUTHLEVEL_USERADMIN)
                    || userAuthLevel.equals(AuthService.AUTHLEVEL_USER))
                authorised = true;
            break;
        default:
            break;
        }
    } else {
        FacesContext fc = FacesContext.getCurrentInstance();
        ConfigurableNavigationHandler nav = (ConfigurableNavigationHandler) fc.getApplication().getNavigationHandler();
        nav.performNavigation("/welcome?faces-redirect=true");
    }
    return authorised;
}
```

Figure 12: *RPDF requiresAuthLevel Method*

The way that the application checks the user's permissions is through the use of a Boolean switch-case statement. The user's permission level is retrieved from their session attributes and is put through this statement to decide which content they should be allowed to view. Firstly, the system checks to see if the user's permission level is not null, in other words, do they have a session open? If the answer is not null, then their permission level is run through the statement and the content is rendered appropriately. The permission levels for the **RPDF** are hierarchical, so the ADMIN user can do everything a USERADMIN user and everything a regular user can

do, for example. If the user does not have a session, then the system will automatically redirect them to the welcome page where they will have to log in.

This process is similar for the '*isLoggedIn*' method. This method checks whether or not the user is logged in, if they are then they will be automatically redirected to the post-login welcome page. This method is mainly used on the login page, just in case someone tries to access it whilst the still have a session open. The code for the isLoggedIn method can be seen in **Figure 13**.

```
public void isLoggedIn(ComponentSystemEvent event) {
    if (authService.isLoggedIn()) {
        FacesContext fc = FacesContext.getCurrentInstance();
        ConfigurableNavigationHandler nav = (ConfigurableNavigationHandler) fc.getApplication().getNavigationHandler();
        nav.performNavigation("/secure/welcome?faces-redirect=true");
    }
}
```

Figure 13: *RPDF isLoggedIn Method*

The implementation of the logout method is much simpler than that of the login method. The logout method simply invalidates the session, which clears all of the session attributes that were set by the login method and then redirects the user to the pre-login welcome page. The code for the logout method can be seen in **Figure 14**.

```
public String logOut() {
    authService.getSession().invalidate();// invalidate current session
    messageUtil.sendInfoMessage("Logged-Out Successfully!");// display success
    return "/welcome?faces-redirect=true";// open login page
}
```

Figure 14: *RPDF logout Method*

5.2 Search Implementation

As the main functionality provided by the application, the search function was the most complex part of the application to implement. As there are so many different interconnected fields in the **RPDF** Database, as can be seen in **Figure 3**, the first hurdle to overcome was the search query design and how that was to be implemented. Fortunately, Wildfly uses Hibernate which comes packaged with the ability to use Native **SQL** for database queries within applications; this made it significantly easier to implement the searches because once the search query had been created in MySQL, it can just be copied and pasted into the application. The code for retrieving the data from the database can be seen in **Figure 15**.

```
private String selectStatement() {
    return "SELECT " + "MLRT.runningtimeid RTID, " + "MLI.logdate LOGDATE, " + "R.recorder RECORDER, "
            + "MLI.trainstarttime STARTTIME, " + "MLI.trainstartid STARTID, " + "MLI.trainterminateid TERMINATEID, "
            + "MLRT.departurepoint DEPART, " + "MLRT.arrivalpoint ARRIVE, " + "MLI.trainnumber NUMBER, " + "C.class CLASS, "
            + "C.country COUNTRY, " + "MLI.trainload TRAINLOAD, " + "MLRT.runningtime RUNNINGTIME, "
            + "TT.tractiontypename TRACTIONTYPE " + "FROM class AS C "
            + "LEFT JOIN maglogindex AS MLI ON C.classid = MLI.classid "
            + "LEFT JOIN recorders AS R ON MLI.recorderid = R.recorderid "
            + "LEFT JOIN maglogrunningtimes AS MLRT ON MLI.maglogid = MLRT.maglogid "
            + "LEFT JOIN tractiontype AS TT ON C.tractiontypeid = TT.tractiontypeid ";
}
```

Figure 15: *Search Select Statement*

As can be seen above, the main problem with constructing the select statement for this query was the fact that there were so many different fields from various different tables that needed to be retrieved. The select statement above just assigns the String values to the '*selectStatement()*' method so it can be used in the search method, seen in **Figure 16**.

```
@SuppressWarnings("unchecked")
public List<Result> search(String recorder, String country, String trainClass, String number, String type,
        String start, String end, String time, String station1, String station2, long pageNumber, long pageSize) {
    String sql = selectStatement()
            + "WHERE MLRT.runningtimeid IS NOT NULL "
            + (recorder!=null && !recorder.isEmpty() ? "AND R.recorder LIKE :rec " : "")
            + (country!=null && !country.isEmpty() ? "AND C.country LIKE :country " : "")
            + (number!=null && !number.isEmpty() ? "AND MLI.trainnumber LIKE :num " : "")
            + (trainClass!=null && !trainClass.isEmpty() ? "AND C.class LIKE :cla " : "")
            + (type!=null && !type.isEmpty() ? "AND TT.tractiontypename = :type " : "")
            + (time!=null && !time.isEmpty() ? "AND MLI.trainstarttime = :time " : "")
            + (start!=null && !start.isEmpty() ? "AND MLI.trainstartid LIKE :start " : "")
            + (end!=null && !end.isEmpty() ? "AND MLI.trainterminateid LIKE :end " : "")
            + (station1!=null && !station1.isEmpty() ? "AND MLRT.departurepoint LIKE :station1 " : "")
            + (station2!=null && !station2.isEmpty() ? "AND MLRT.arrivalpoint LIKE :station2" : "");

    List<Result> entityList = null;
    try {
        Session ses = getSession();
        SQLQuery sq = ses.createSQLQuery(sql);
        if (recorder!=null && !recorder.isEmpty()) sq.setParameter("rec", "%" + recorder + "%");
        if (country!=null && !country.isEmpty()) sq.setParameter("country", "%" + country + "%");
        if (number!=null && !number.isEmpty()) sq.setParameter("num", "%" + number + "%");
        if (trainClass!=null && !trainClass.isEmpty()) sq.setParameter("cla", "%" + trainClass + "%");
        if (type!=null && !type.isEmpty()) sq.setParameter("type", type);
        if (time!=null && !time.isEmpty()) sq.setParameter("time", time);
        if (start!=null && !start.isEmpty()) sq.setParameter("start", "%" + start + "%");
        if (end!=null && !end.isEmpty()) sq.setParameter("end", "%" + end + "%");
        if (station1!=null && !station1.isEmpty()) sq.setParameter("station1", "%" + station1 + "%");
        if (station2!=null && !station2.isEmpty()) sq.setParameter("station2", "%" + station2 + "%");
        int firstResult = (((int) pageNumber - 1) * (int) pageSize);
        entityList = sq.addEntity(Result.class).setFirstResult(firstResult).setMaxResults((int) pageSize).list();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return entityList;
}
```

Figure 16: *RPDF search Method*

The way the search method is implemented allows for all fields to be used or empty, depending on the user's preference. This allows for a totally customisable experience for each user.

The search method uses conditional operators to decide whether or not the search query should contain certain values, for example, if a user wants to search by just the recorder field then that would be the only field that is run through the search query. Once the values to be run through the search query are set then an empty List is created. Then the query is run through the database and the results of the query are put in to the List.

When this was initially implemented, there were a lot of problems displaying the results if the list was over a certain size to the point where the whole server would crash, to overcome this, custom pagination of the results tables had to be implemented. The way this was implemented was by effectively creating a new results list for every page in the data table. Therefore, if the page size was set to 10

the search query would only return the first 10 results, then when the page was changed it would return the next 10 results and so on. This can be seen before the catch statement in **Figure 16**.

Due to this new implementation, a count of the results also had to be run at the same time as the main search query so that the total number of records returned by the query could be displayed. This can be seen in **Figure 17**.

```
private String selectCountStatement() {
    return "SELECT COUNT(*) AS count " + "FROM class AS C "
            + "LEFT JOIN maglogindex AS MLI ON C.classid = MLI.classid "
            + "LEFT JOIN recorders AS R ON MLI.recorderid = R.recorderid "
            + "LEFT JOIN maglogrunningtimes AS MLRT ON MLI.maglogid = MLRT.maglogid "
            + "LEFT JOIN tractiontype AS TT ON C.tractiontypeid = TT.tractiontypeid ";
}
```

Figure 17: *Search Count Statement*

This is a very much simpler method that the select statement in **Figure 15** as it is only responsible for getting the count of the results run in the query. The only difference between the search method and the searchCount method is that the searchCount method has a return type of Long instead of a List, this can be seen in **Figure 18**.

```
public Long searchCount(String recorder, String country, String trainClass, String number, String type,
        String start, String end, String time, String station1, String station2) {
    String sql = selectCountStatement()
            + "WHERE MLRT.runningtimeid IS NOT NULL "
            + (recorder!=null && !recorder.isEmpty() ? "AND R.recorder LIKE :rec " : "")
            + (country!=null && !country.isEmpty() ? "AND C.country LIKE :country " : "")
            + (number!=null && !number.isEmpty() ? "AND MLI.trainnumber LIKE :num " : "")
            + (trainClass!=null && !trainClass.isEmpty() ? "AND C.class LIKE :cla " : "")
            + (type!=null && !type.isEmpty() ? "AND TT.tractiontypename = :type " : "")
            + (time!=null && !time.isEmpty() ? "AND MLI.trainstarttime = :time " : "")
            + (start!=null && !start.isEmpty() ? "AND MLI.trainstartid LIKE :start " : "")
            + (end!=null && !end.isEmpty() ? "AND MLI.trainterminateid LIKE :end " : "")
            + (station1!=null && !station1.isEmpty() ? "AND MLRT.departurepoint LIKE :station1 " : "")
            + (station2!=null && !station2.isEmpty() ? "AND MLRT.arrivalpoint LIKE :station2" : "");

    Long entityCount = Long.valueOf(0);
    try {
        Session ses = getSession();
        SQLQuery sq = ses.createSQLQuery(sql).addScalar("count", LongType.INSTANCE);
        if (recorder!=null && !recorder.isEmpty()) sq.setParameter("rec", "%" + recorder + "%");
        if (country!=null && !country.isEmpty()) sq.setParameter("country", "%" + country + "%");
        if (number!=null && !number.isEmpty()) sq.setParameter("num", "%" + number + "%");
        if (trainClass!=null && !trainClass.isEmpty()) sq.setParameter("cla", "%" + trainClass + "%");
        if (type!=null && !type.isEmpty()) sq.setParameter("type", type);
        if (time!=null && !time.isEmpty()) sq.setParameter("time", time);
        if (start!=null && !start.isEmpty()) sq.setParameter("start", "%" + start + "%");
        if (end!=null && !end.isEmpty()) sq.setParameter("end", "%" + end + "%");
        if (station1!=null && !station1.isEmpty()) sq.setParameter("station1", "%" + station1 + "%");
        if (station2!=null && !station2.isEmpty()) sq.setParameter("station2", "%" + station2 + "%");
        entityCount = (Long) sq.uniqueResult();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return entityCount;
}
```

Figure 18: *searchCount Method*

5.3 Web Service Implementation

Implementing the National Rail web services was a fairly simple process once the code had been downloaded using the provided **WSDL** file. The departure board was the first web service implemented, the code for the web service can be seen in **Figure 19**.

```
public void departureBoard(String crs) {

    Ldb soap = new Ldb();
    soapService = soap.getLDBServiceSoap12();

    accessToken.setTokenValue(token);

    GetBoardRequestParams params = new GetBoardRequestParams();
    params.setCrs(crs.toUpperCase());

    StationBoardResponseType departureBoard = soapService.getDepartureBoard(params, accessToken);

    services = departureBoard.getGetStationBoardResult().getTrainServices().getService();

    if (services.isEmpty()) {

        messageUtil.sendErrorMessage(
                "No Departures Available for " + departureBoard.getGetStationBoardResult().getLocationName());

    }

    this.setLocationName(departureBoard.getGetStationBoardResult().getLocationName());
}
```

Figure 19: *Departure Board Code*

The most challenging part about implementing the departure board was discovering which methods from the ones provided by the National Rail to use. Once this had been figured out then it was just a case of accessing the methods and variables needed to produce the desired output.

The accessToken variable is set when the web service is initialised by passing the token that was provided by the National Rail as a String and assigning it to the accessToken variable. Both the arrivals and departures board have a return type of '*StationBoardResponseType*', once the departure board was returned it was just a case of adding the services to an ArrayList called '*services*', this was the ArrayList that was then going to be displayed on the **UI**. The location name was then set so that it could be displayed on the **UI** to allow the user to know which station they had got the departure or arrivals board for.

The code for the arrivals board is almost identical to that of the departure board, the only difference being the variable '*departureBoard*' has been changed to '*arrivalBoard*'. The code for the arrival board can be seen in **Figure 20**.

```
public void arrivalBoard(String crs) {

    Ldb soap = new Ldb();
    soapService = soap.getLDBServiceSoap12();

    accessToken.setTokenValue(token);

    GetBoardRequestParams params = new GetBoardRequestParams();
    params.setCrs(crs.toUpperCase());

    StationBoardResponseType arrivalBoard = soapService.getArrivalBoard(params, accessToken);

    services = arrivalBoard.getGetStationBoardResult().getTrainServices().getService();

    if (services.isEmpty()) {

        messageUtil.sendErrorMessage(
                "No Departures Available for " + arrivalBoard.getGetStationBoardResult().getLocationName());

    }

    this.setLocationName(arrivalBoard.getGetStationBoardResult().getLocationName());

}
```

Figure 20: *Arrival Board Code*

Both the arrival board and departure board get their results by using the user inputted '**CRS**', or station, code. The entered **CRS** code is then passed through the webservice and all services for that **CRS** code are added to the services ArrayList. By default, the web service returns all services for the entered **CRS** for the next two hours. However, to prevent display issues for larger stations, the timeframe will be unable to be changed by users.

6 Testing

To ensure that all of the highest priority Functional Requirements in **Table 1** were met and worked as expected, a thorough testing program had to be carried out. The details of all tests carried out on the system have been detailed in **Table 3** below.

Test Case	Case Purpose	Expected Result	Actual Result	Success/Failure
1. RPS Website	User can open RPS website from welcome page (FR-1)	RPS Website opens in new tab	RPS website opened in new tab	Success
2. Login	User can login using their unique credentials (FR-2)	Once correct credentials are entered the user is redirected to the welcome page	User was redirected to welcome page upon entering correct credentials	Success
3. Login failure	User cannot login using incorrect credentials	Entering incorrect credentials will return an error message	An error message was received when incorrect credentials were entered	Success

4.	Admin menu	Only admins can see the admin menu (FR-3)	When a regular user logs in they will not be able to see the admin menu	When logging in as a regular user the admin menu is not visible	Success
5.	Search visibility	The search page should only be visible to signed up users (FR-4)	When trying to access the search page when not logged in the user will be redirected to the welcome page	User was redirected to the welcome page when trying to access the search page	Success
6.	Search Results	The results of each search should be displayed in table (FR-6)	The search should return a list of completely unique results with a count of the results at the bottom of the page	The search returned the correct number of results, but they were all duplicates of the first result	Failure
7.	Search Refinement	An initial search can be refined by up to 7	If another filter is added the number of results	The number of results reduced; however, the results	Failure

	secondary filters (FR-5)	should reduce	were still all duplicates of the first result	
8. Results table can be exported and printed	Allow users to save a hard copy of their search results (FR-7, FR-8)	When the button is pressed, either the results are downloaded, or a print manager pop-up opens	The results download as expected and the print manager pop-up opens as expected	Success
9. Create new user	Admins should be able to create new users	User created correctly and added to the database	User was created successfully and was added to the database	Success
10. Delete user	Admins should be able to delete users	User is deleted from the database	User was deleted from the database	Success
11. Edit User	Admins should be able to edit user details	Changes to the user are saved in the database	The changes were saved in the database	Success
12. Edit own info	Users should be	All changes made by a	All changes the user	Success

	able to edit their own information	user should be saved to the database	made were saved to the database	
13. Arrivals Board	Users should be able to view an accurate live arrivals board (FR-13)	The arrivals board should match the one on the National Rail website	The arrivals board matched the one on the National Rail website	Success
14. Departure Board	Users should be able to view an accurate live departures board (FR-13)	The departures board should match the one on the National Rail website	The departures board matched the one on the National Rail website	Success

Table 3: *RPDF Testing Details*

As can be seen in **Table 3** above, the majority of the test cases passed successfully. There were only two test cases that failed, these were test cases 6 and 7; both of which were to do with the search function and how the results were displayed.

The reason that both of these tests failed is that instead of displaying a list of unique results, duplicates of the first result retrieved were displayed instead. For example, if a search retrieved 2000 results, the results that would be displayed would be the same record 2000 times over. This happened regardless of how many filters were being used, hence both test cases 6 and 7 failed for the exact same reason.

To resolve this issue, a deeper look into the search query had to be carried out. To return a list of totally unique results, one of the fields had to be totally unique

otherwise duplicate records would be returned. This issue stemmed from Hibernate, the query technology used by Wildfly, whereby it needs one field to be marked as an '*ID*' field so that it uniquely identifiable. Prior to this error occurring, the field in the database that was being used as the ID field was the '*maglogid*' field in the '*maglogindex*' table. However, this field does not have a one to one relationship with the rest of the data, therefore, it will return duplicate entries when it is being searched against.

After looking into the database structure further, as can be seen in **Figure 3**, it was discovered that the '*runningtimeid*' field in the '*maglogrunningtimes*' table had a one to one relationship with the data in the rest of the database; specifically, the maglogindex table where the majority of the data is contained. This means that upon searching with the runningtimeid field marked as an ID in Hibernate, it returned a list of totally unique results and the test case could be marked as passed.

7 Evaluation

When evaluating this project by looking deeper into the test results and comparing them to the requirements, it can generally be considered a successful outcome.

All of the highest priority functionality listed in **Table 1** were implemented successfully, the outcomes of the tests of the highest priority tests can be seen in **Table 3**.

The main features of the application, logging in, security and access to the external **RPS** website were all implemented successfully and when tested produced the expected outcome. Out of the six requirements prioritise as 'Must Do', only two of them failed upon initial testing. The thorough report of why the tests failed and what was done to overcome these problems is in section **6 Testing**; however, in short the tests were failing because the ID field in the query was mapped to the wrong field and wrong table in the database. This is the kind of problem that could have been easily prevented with prior knowledge of the data, however, in most cases this is not possible so this kind of problem was likely to occur and although it was frustrating at the time the fix was very simple and did not cause any other difficulties to the development process.

The requirements marked as 'Should Do' were surprisingly simple to implement. Part of the reason for choosing to use BootsFaces as a development framework was because it contains a lot of useful features that would enable the final product to become a much more polished application. The use of BootsFaces meant that implementing FR-7 and FR-8 was very simple; this is because there are methods built in to the data table provided by BootsFaces that enable the developer to add attributes in the **HTML** code that simply add the functionality without having to do any extra coding.

The only ‘Should Do’ requirement that took a lot of work was FR-9, this is because the way that the searches were carried out had to be totally changed. However, the effort was worth it because with this requirement being met it made the way the results are presented to the user look much more professional and made the application much more polished as a whole.

Of the ‘Could Do’ requirements, only two were not implemented, these were FR-11 and FR-14. FR-11 was not included as such, however it was implemented in a different way. It was not implemented as a ‘pop-up’ box but rather as a static welcome page, implementing a pop-up box that appeared every time a user logged on would be a challenge as it would involve checking if a user had just created a new session, if they had then it would display the pop-up and if they had not then it would not display. Using a static welcome page to display the terms and conditions was a much simpler workaround and allowed for this requirement to be mostly met, even if it was not exactly to specification.

FR-14 was not included directly because BootsFaces already accounts for this. One of the major benefits of using BootsFaces is that it has pre built in responsive design, this means that the majority of the components used by BootsFaces automatically adjust depending on the size of screen they are being displayed on. The components that do not have automatic responsiveness will more often than not have an attribute that allows them to become responsive, this usually comes in the form of a Boolean value that when set to ‘true’ will allow the component to become responsive. Therefore, FR-14 was automatically implemented when BootsFaces was implemented.

The tests that were not related to functional requirements were not as important, but without them being implemented the application would not be a completely functioning product. For example, without the ability to create, delete or edit users the user base would remain static which would defeat the whole point of an application that is designed to be used by an organisation with a continually evolving user base. Also, without the ability to edit their own information, users would not be

able to update their preferences and this could mean that their personal information becomes incorrect; for example, if a user gets married and their surname changes they will want to update it and without this functionality this would not be possible.

Finally, the last ‘Could Do’ requirements to be implemented were the arrivals and departures board, FR-13. These were implemented to add some individuality to this product and to tie it to the user base. As this product is aimed at railway enthusiasts it was a logical choice to include these web services so that users would not have to leave the application to check if their train was on time. The successful implementation of this requirement means that the product as a whole becomes totally unique to the target users and demographic that it is aimed at.

The success of the project was heavily reliant on the choice of technologies used. Had BootsFaces not been used as a framework for this project then the development time could have been considerably greater, and the finished product would not have looked at all as professional and as polished as it does now.

8 Conclusion

Upon reflection of the outcomes of this project it can be concluded with a fair degree of certainty that a successful data analytics tool can be created with no cost to the customer.

If this project was to be carried out again, perhaps for a different customer with different data, then the first thing that would need to be changed would be the way the data structures are introduced to the developer. In the context of this project, the majority of the time pre-development was spent analysing the database and trying to figure out the structure of the data. This analysis took almost two weeks, which put the whole project behind schedule and prevented any development from being started. In future projects it would be much more effective if the developer and someone who understood the data from the customer side were to sit down and map out the data structures rather than leaving it up to the developer to decipher. This would ultimately make query writing for the database much easier to develop and implement.

When it comes to querying the database, in future projects it would be greatly beneficial to write all queries in the code the same way. In this project both **JPQL** and Native **SQL** were used to query the database in multiple ways. Initially, **JPQL** was going to be used throughout the project as the one and only query language, this was because that it had been used before for similar projects. However, it was quickly discovered that the niches of **JPQL** made it far too complicated to use for the complex and lengthy queries that the application was going to use; therefore, a simple solution had to be found, this is where Native **SQL** was used. The majority of the queries were written in **SQL** as it was just a case of copying the **SQL** statements from MySQL straight into the application's code. **JPQL** was used in the application's simple queries, such as retrieving all of the users to display on the user management page; however, to make the application easier to maintain in the future it would be highly beneficial to use one query language instead of two.

If the demographic of the application changed at any point in the future this application could benefit from having some more unique **UI** features adding to it to make it have a more modern feel. An example of this could be responsive menu items, like the 'tiles' that can be seen on most modern windows systems. The use of this would add a certain uniqueness and individuality to the application whilst not taking anything away from its purpose. However, as the target demographic consists of people who are not technically adept, the **UI** needed to remain as simple as possible whilst still delivering on functionality.

During the development phase it was difficult to adhere to any developmental methodology, a hybrid of both Agile and Waterfall was used, this did have its benefits, such as the fact that there were no disagreements between members of the development team and decisions were easy to make in terms of developing the application. However, if this project was to be carried out in the future, it would be a positive addition if it was developed in a team it would mean that a methodology such as Agile could be followed. This would allow the development team to prioritise work, spread the workload and work in sprints, rather than doing all of the work in one go. Splitting up the workload in to two-week sprints would have made the volume of work associated with this project much more manageable.

However, despite these areas for change, the project can overall be considered a success. This is because it was able to deliver a fully functioning search engine that provided a report building functionality and operational live timetables, all whilst providing a simple to use interface for the target demographic.

Appendices

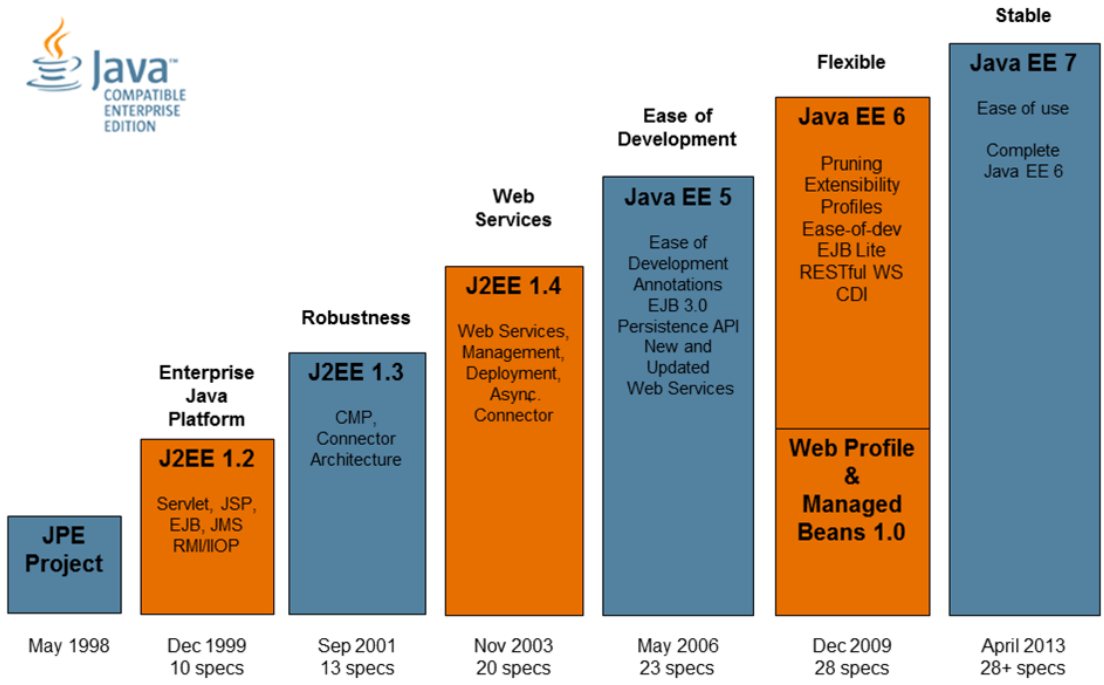

***Figure 21**: The Evolution of Java EE by Alex Theedom, 2017.*

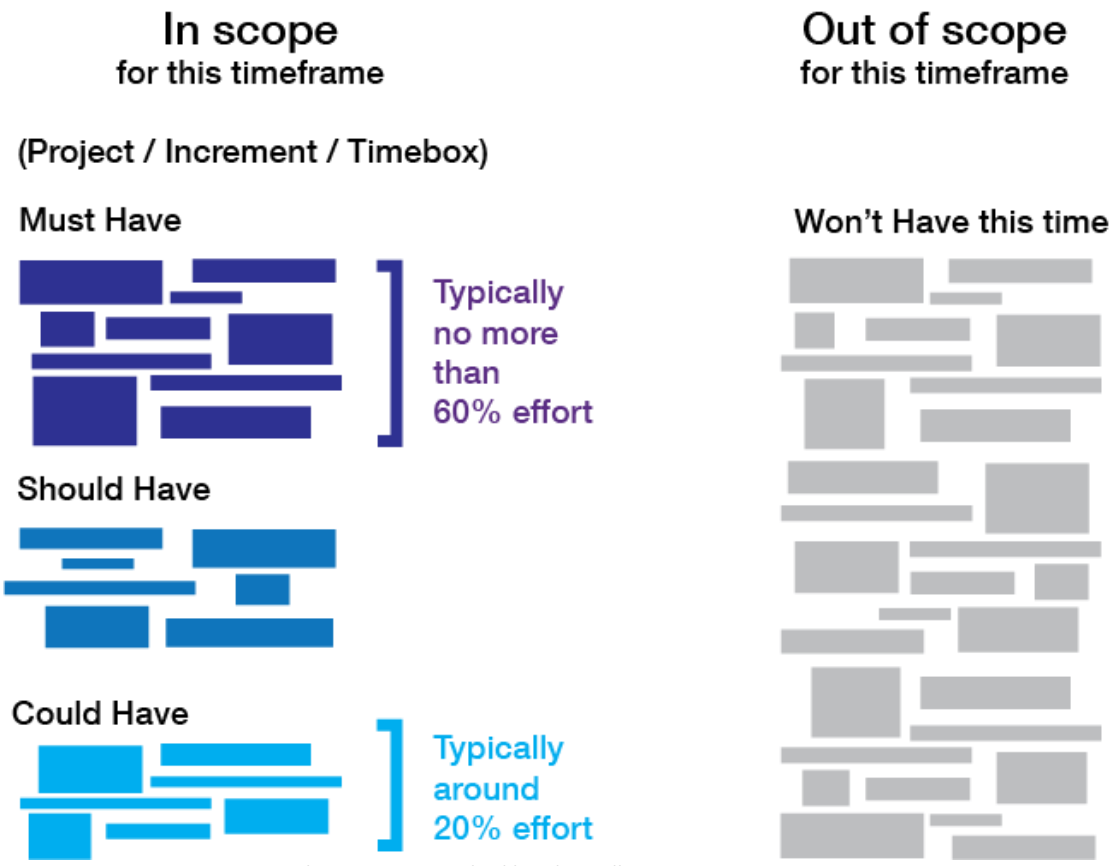

***Figure 22**: Prioritisation using the MoSCoW Method by Abi Walker, 2014.*

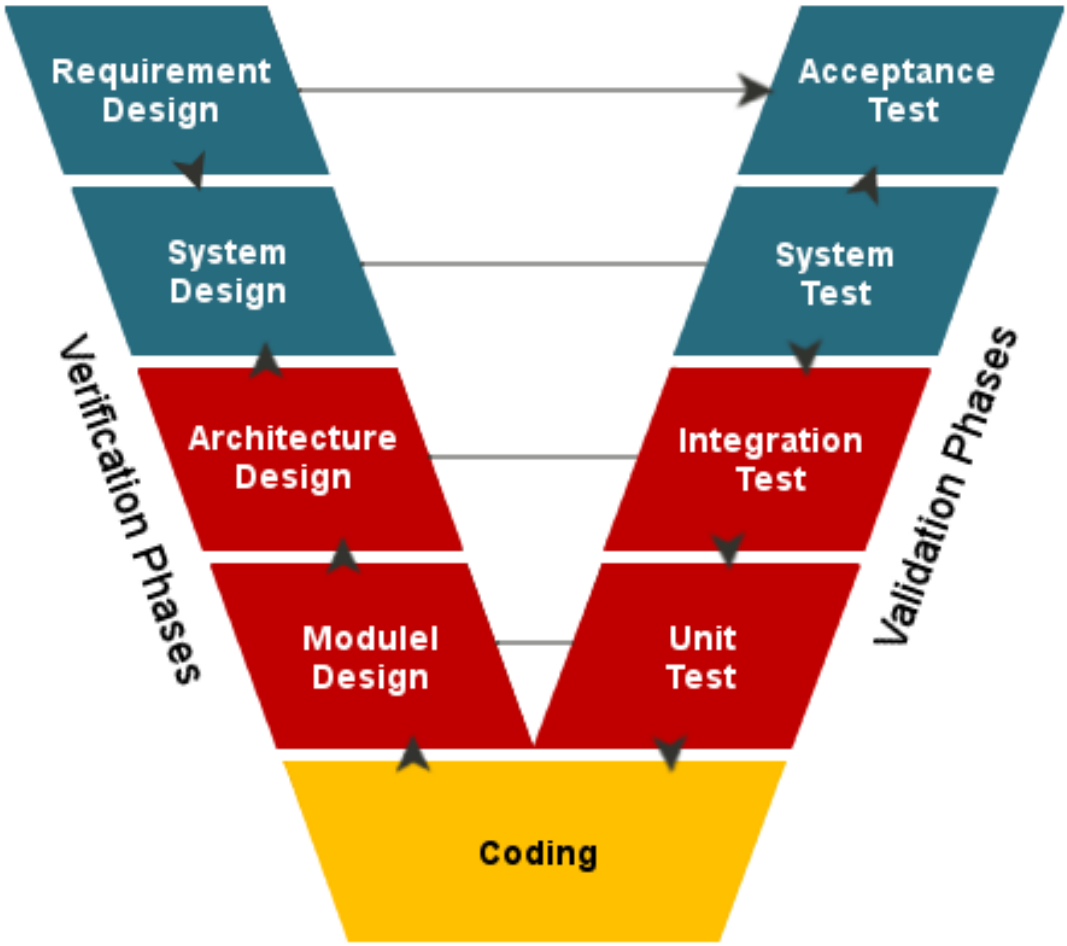

Figure 23: *The V Model by Anon, 2019*

All source code for this project can be found at:

https://gitlab.com/mattherring97/rpdf

Glossary

AJP – Apache JServ Protocol

API – Application Program Interface

CRS – Computer Reservation System

CSS – Cascading Style Sheet

DAO – Data Access Object

EJB – Enterprise JavaBeans

FTP – File Transfer Protocol

GUI – Graphical User Interface

HST – High Speed Train

HTML – Hypertext Markup Language

JAAS – Java Authentication and Authorization Service

JASPI – Java Authentication Service Provider Interface

Java EE – Java Enterprise Edition

Java EL – Java Expression Language

Java SE – Java Standard Edition

JAX-RS – Java API for RESTful Web Services

JAX-WS – Java API for XML Web Services

JBoss EAP – Jboss Enterprise Application Platform

JCA – Java EE Connector Architecture

JMS – Java Messaging Service

JMX – Java Management Extensions

JNDI – Java Naming and Directory Interface

JPA – Java Persistence API

JPQL – Java Persistence Query Language

JS – JavaScript

JSF – JavaServer Faces

JSP – JavaServer Pages

JTA – Java Transaction API

LDBWS – Live Departure Boards Web Service

MVC – Model View Controller

OS – Operating System

OSGi – Open Service Gateway Initiative

POJO – Plain Old Java Object

RMI – Remote Method Invocation

RPDF – Rail Performance Data Foundation

RPS – Rail Performance Society

SOAP – Simple Object Access Protocol

SQL – Standardised Query Language

SSL – Secure Socket Layer

UI – User Interface

WSDL – Web Services Description Language

XHTML – Extensible Hypertext Markup Language

XML – Extensible Markup Language

Bibliography

Anon (2018) *10 Key Principles of Agile Software Development*. Available from: https://project-management.com/10-key-principles-of-agile-software-development/ [Accessed 30 March 2019].

Anon (2019) *V Model-Verification and Validation Model |Professionalqa.com*. Available from: http://www.professionalqa.com/v-model [Accessed 30 March 2019].

Collins, F. (2008) *Rail Performance Society - RPS*. Available from: http://railperf.org.uk/index/index [Accessed 14 October 2018].

Eriksson, U. (2012) *Functional Requirements vs Non Functional Requirements*. Available from: https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/ [Accessed 15 March 2019].

Marchand, T. (2019). *Dundas BI Software - 2019 Reviews, Pricing & Demo*. Available from: https://www.softwareadvice.com/uk/bi/dundas-bi-profile/ [Accessed 3 Mar. 2019].

Marchand, T. (2019). *Dundas BI Pricing - Dundas Data Visualization*. Available from: https://www.dundas.com/dundas-bi/pricing [Accessed 3 Mar. 2019].

Marshall, A. (2015). *Top 10 Open Source Java and JavaEE Application Servers*. Available from: https://blog.idrsolutions.com/2015/04/top-10-open-source-java-and-javaee-application-servers/ [Accessed 7 Mar. 2019].

Nash, M. (2002). *Frameworks and components*. 1st edition. Cambridge: Cambridge University Press.

Pal, K. (2018) *Weighing the Pros and Cons of Real-Time Big Data Analytics*. Available from: https://www.techopedia.com/2/31245/technology-trends/big-data/weighing-the-pros-and-cons-of-real-time-big-data-analytics [Accessed 15 October 2018].

Picciano, R. (2019). *IBM Cognos Business Intelligence - 2019 Reviews & Pricing*. Available from: https://www.softwareadvice.com/uk/bi/ibm-bi-profile/ [Accessed 3 Mar. 2019].

Picciano, R. (2019). *Cognos Analytics - Pricing*. Available from: https://www.ibm.com/uk-en/products/cognos-analytics/pricing [Accessed 3 Mar. 2019].

Rauh, S. (2015). *Getting Started with BootsFaces: Responsive Design*. Available from: https://www.beyondjava.net/getting-started-with-bootsfaces-responsive-design [Accessed 7 Mar. 2019].

Theedom, A. (2017). *Java EE: Past, Present, and Future - DZone Java*. Available from: https://dzone.com/articles/java-ee-past-present-and-future [Accessed 3 Mar. 2019].

Theedom, A. (2017). *The Evolution of Java EE*. [online]. Available from: https://i1.wp.com/readlearncode.com/wp-content/uploads/2017/02/java-ee-histroy.png?ssl=1 [Accessed 3 Mar. 2019].

Tuan, N. (2019). *A brief history of Java EE*. Available from: http://sgdev-blog.blogspot.com/2014/02/a-brief-history-of-java-ee.html [Accessed 3 Mar. 2019].

Tyson, M. (2018). *What is JSF? Introducing JavaServer Faces*. Available from: https://www.javaworld.com/article/3322533/what-is-jsf-introducing-javaserver-faces.html [Accessed 6 Mar. 2019].

Walker, A. (2014) *The DSDM Agile Project Framework (2014 Onwards)*. Available from: https://www.agilebusiness.org/content/moscow-prioritisation [Accessed 12 March 2019].

Walker, A. (2017) *18 Best Open-Source and Free Database Software*. Available from: https://learn.g2crowd.com/free-database-software [Accessed 4 April 2019].

Wulff, N. (2018) *What's The Difference Between Data Analytics And Data Analysis?*. Available from: https://www.getsmarter.com/blog/career-advice/difference-data-analytics-data-analysis/ [Accessed 13 October 2018].

Zukanov, V. (2018). *Top Java Application Servers: Tomcat vs. Jetty vs. GlassFish vs. WildFly*. Available from: https://stackify.com/tomcat-vs-jetty-vs-glassfish-vs-wildfly/ [Accessed 7 Mar. 2019].