DONATION ANALYTICS

An interactive data visualisation web application

Ritchie Shekleton
N00143569

Abstract

The objective of the project was to learn new skills as a programmer and develop an application that could be used in a professional environment. Donation Analytics is an interactive data visualisation web application. The app includes a front and a back end. The app will include an interactive map of Dublin showing the distributions of blood donors by area. The user will also see the current situation regarding blood stocks separated by blood type, the location of clinics to donate blood and the spatial distribution of donors with their frequency of donating. The application is designed for blood bank staff although it was envisaged that some of the components of the app may be of interest to the general public.

Acknowledgements

First, I would like to thank my project supervisor Cyril Connolly, his advice and guidance helped the project become what it is today.

Second, I would like to thank my classmates in college for their help testing, their opinions on my ideas and their support throughout the last four years.

Lastly, I would like to thank my family for their love and support for the last twenty-two years.

Contents

1. Introduction

1.1 OVERVIEW OF PROJECT TECHNOLOGIES

Donation Analytics is an interactive data visualisation web application for viewing blood donation statistics. The application was made with the R programming language in RStudio and using packages such as shiny and leaflet.

1.2 PROJECT APPROACH

The work was split up into six sections. The first section involved creating some basic data visualisation with some geospatial functionality. Section two established a local login. Section three created the database. Section four developed registration functionality. Section five involved improving the UI while section six incorporated more content and visuals.

1.3 INITIAL CHALLENGES

The main challenges were learning the R programming language together with a wide range of R based packages

1.4 AREAS OF LEARNING

The aim of the project was to create a data visualisation app that could be used for monitoring blood donations. In addition to learning a new programming language, other skills involved working with data and creating a number of appropriate data visualisations.

2. Feasibility Study and Requirements

2.1 USER ROLES

During the requirements analysis development stage, the application's functionality is decided. The application is designed for blood bank staff although it was envisaged that some of the components of the app may be of interest to the general public.

2.1.1 Blood Bank Staff

A member of staff involved with the storage and safe keeping of blood donations. Someone who will be able to be trusted with sensitive information.

2.1.2 Member of General Public

An ordinary person who may be interested in donating blood.

2.2 USER REQUIREMENTS

2.2.1 Blood Bank Staff

As a member of blood bank staff, I require the following features:

- The ability to register accounts so I can provide logins for new staff members.
- The ability to view current blood stocks so I can alert the marketing department which blood types we require urgently.
- The ability the change in blood stocks over time so I can get in-depth analysis of the months progress in terms of donations.
- The ability to visualise donation statistics graphically.
- The ability to view the distribution of all clinics in Dublin so I can see which areas may need a clinic and which areas have too many.
- The ability to visualise donors represented geospatially on a map so I can see which areas are better at donating and which areas need more advertisements.
- The ability to receive notifications about how the time series of donations throughout the day so I can have up to date information quickly on evolving trends.

2.2.2 Member of General Public

As a member of the general public, I want to be able to view:

- Information on the blood donation so I can be well informed before I make a decision whether to donate blood.
- A map of Dublin with the clinic locations on it so I can see which clinic is closest to my location.
- News reports concerning blood donations so I am up to date with the current needs in terms of blood types.

2.3 USE CASE DIAGRAM

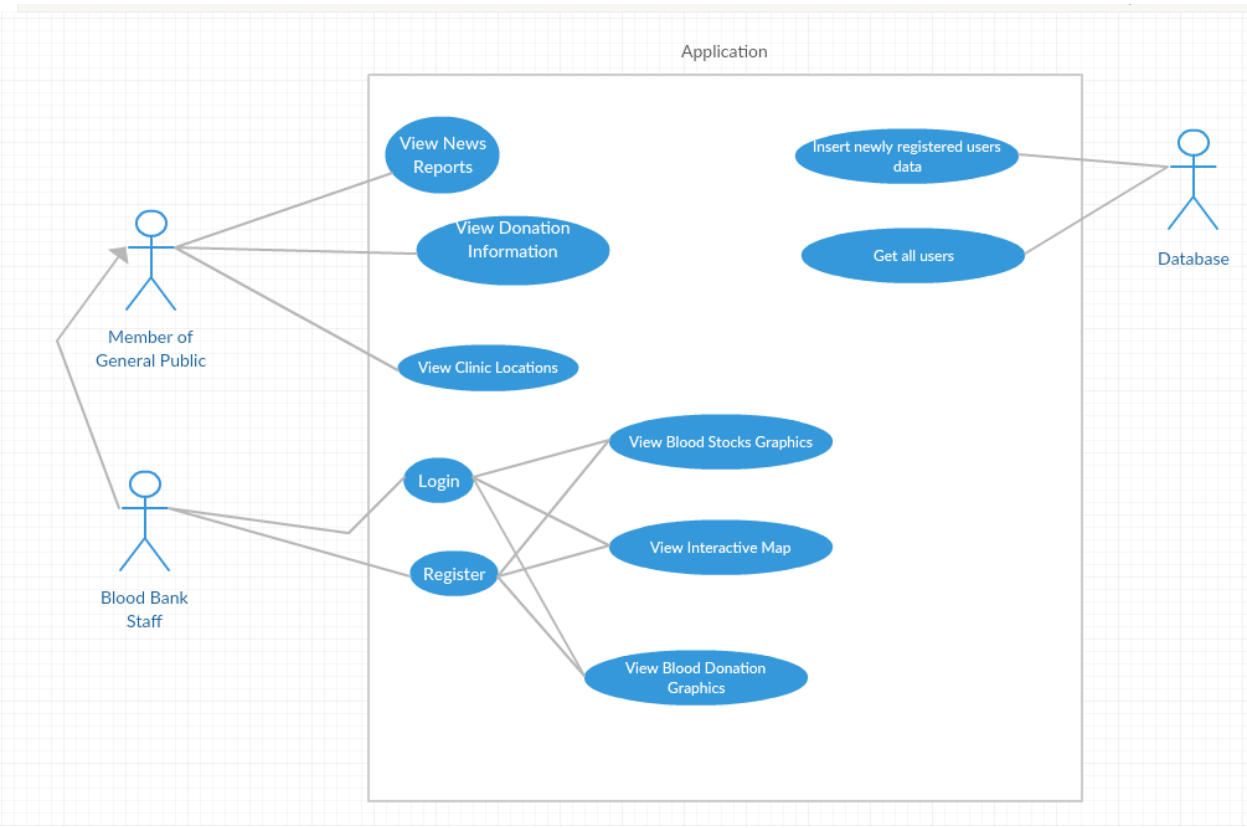


Figure 2.1: A use case diagram for the system

The application developed is an interactive data visualisation application which has two potential users as discussed above. The use case diagram in Figure 2.1, shows the different actions a user can do. A member of the general public can view news reports, donation information and clinic locations. The blood bank staff also have access to these features but in addition they are able to login and register which allows them to view statistics in

the form of graphs and plots on blood stocks and donations. They can also view an interactive map of Dublin which displays different visuals.

2.4 FUNCTIONAL REQUIREMENTS

The application will have the following features:

- a login screen which allows the user to sign in.
- a register button on the login screen which allows the user to register their account.
- a map of Dublin with information detailing each area's blood donation statistics.
- provide information about where and when the user will be able to donate.
- provide information on the stocks of blood in storage in the forms of charts/graphs.

2.5 NON-FUNCTIONAL REQUIREMENTS

The application will need to have minimal delay when extracting data from the database and when the user is interacting with the data being displayed.

When the user opens up the web application they will be greeted by a login screen where they can either sign into the application or register their account. After they have logged in or registered they will be transported to the home screen. It's envisaged that the home screen will contain the interactive map with widgets/buttons and links to view the other graphics.

2.6 SYSTEM MODEL

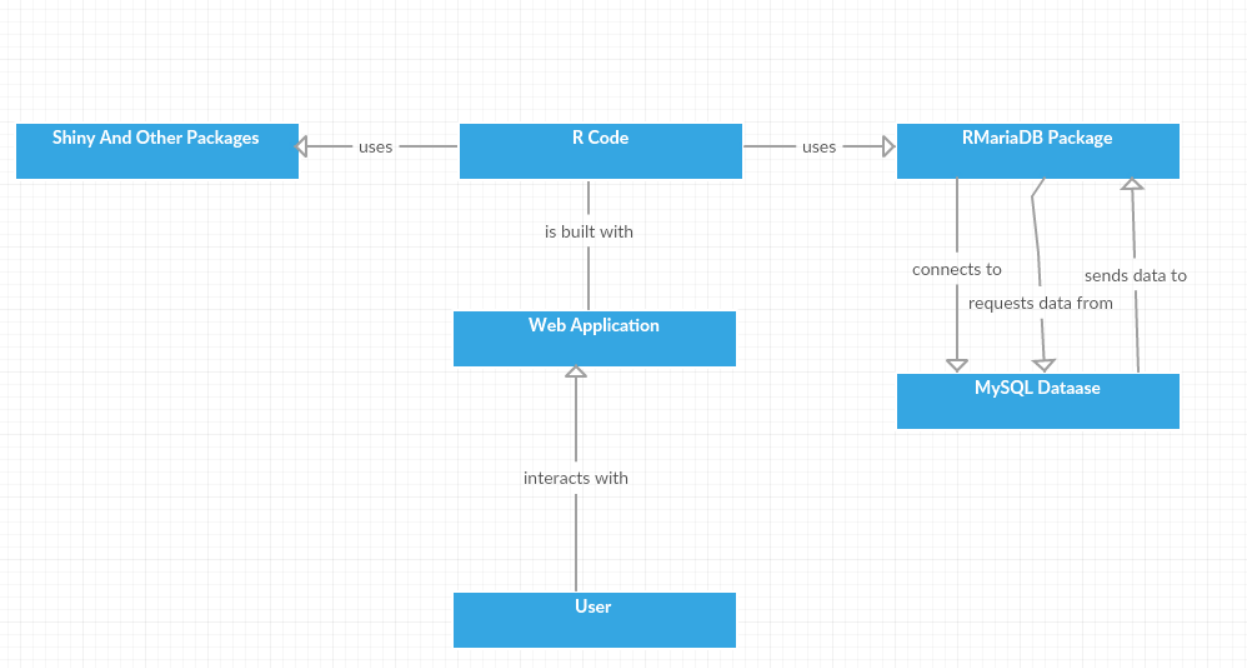


Figure 2.2: The system model for the application.

The model for Donation Helper, Figure 2.2, is a web application, which visualises data that is extracted from the MySQL database. The application is built using the R programming language with shiny and other R packages. The database is connected by using the RmariaDB package. This type of application could be made on multiple different platforms, Android, iOS and a web application each with their own advantages and disadvantages.

2.6.1 Android

Advantages

- Largest user base.
- Uses Java coding, androidplot and achartengine.
- Already have experience in coding.
- App does not have to be approved like in iOS.

Disadvantages

- No standardisation in products, multiple manufacturers making phones of different dimensions.
- Touch controls.

2.6.2 iOS

Advantages

- One manufacturer making the phones. Not a lot of differences in phone dimensions.
- Access to the iPhone market.

Disadvantages

- No experience in coding.
- Apps need to be approved in order to be released.

2.6.3 Web

Advantages

- Only needs a browser to run, can be on android and iOS platforms.
- Can use R programming language with shiny, html widgets and ggvis.
- App does not need to be downloaded.

Disadvantages

- App will need to be responsive in order to work with other platforms.

The web application was chosen as the preferable option after looking at the advantages and disadvantages of the development. It will be developed using the R programming language with one of shiny, html widgets or ggvis. The application will be connected to a MySQL database which will contain all of the data that is going to be displayed.

2.7 FEASIBILITY STUDY

The goal of the project is to develop an interactive data visualisation app which allows users to view and interact with data related to blood donations. The application will be mostly based around the main feature of an interactive map of Dublin. The application will show information on the frequency of blood donation in different areas of Dublin. The next section will discuss some of the probable risks associated with Donation Analytics.

Time Management:

Time management and planning is a significant feature of any project and needs to be closely monitored in the development of this application. In order to adhere to project deadlines there needs to be an achievable plan to help with continuing development of the application.

Code:

The programming language used in the development of this application is R which will be need to learnt by the writer. A MySQL database and a number of R packages like shiny will also need to be investigated. For the interactive map the information of each area will display as the user zooms in so as to limit the amount of resources consumed. This will need the database to be able to send data as soon as the user hovers over the area.

Data Retrieval:

This section is relating to getting the information needed in order to display it on the map, the data relating to the amount of people giving blood in the split-up areas. During the early stages of development an attempt was made to acquire any information or data from the Irish Blood Transfusion Service. Unfortunately, after an initial response no progress was made in receiving any information, although some of the content in the application was taken from their website.

3. Research and Analysis

Data visualisation is becoming more important in terms of displaying statistics and other information. It is constantly being improved with better technologies creating more detailed visuals and more interactive graphics. Interactive graphics have become more advanced with users able to filter and obtain details on demand. This literature review is going to provide information on data visualisation, its history and some types of data visualisation. The importance of the user interface and the usability of the visual graphics will also be discussed. The technical aspects of this review will discuss D3 with JavaScript and R with Shiny. Data is everywhere being collected by phones, social media, the census etc. All this data along with continuous improvements in data visualisation technology adds up to an interest in the usability and interactivity of data visualisation graphics.

3.1 DATA VISUALISATION

3.1.1 History of Data Visualisation

The beginning of modern graphics is said to have occurred from 1800-1850. During this time many of the most widely used visuals/graphics were invented. Pie and bar charts, scatterplots and histograms are some of the graphics invented at this time, Figure 3.1 is an example of old a graphic created around this time. Mapping had also improved allowing data on certain topics to be displayed instead of just showing places, William Smith introduced these geological maps in England in 1801. During the 1820s, Baron Charles Dupin was the first to use shading as a way to show the rates of areas of illiteracy in France (Dupin, 1826). These visualisation maps were then used by people like André-Michel Guerry and Adolphe Quetlet who would display statistics on these maps. Guerry and Quetlet are regarded as the founders of modern social science.

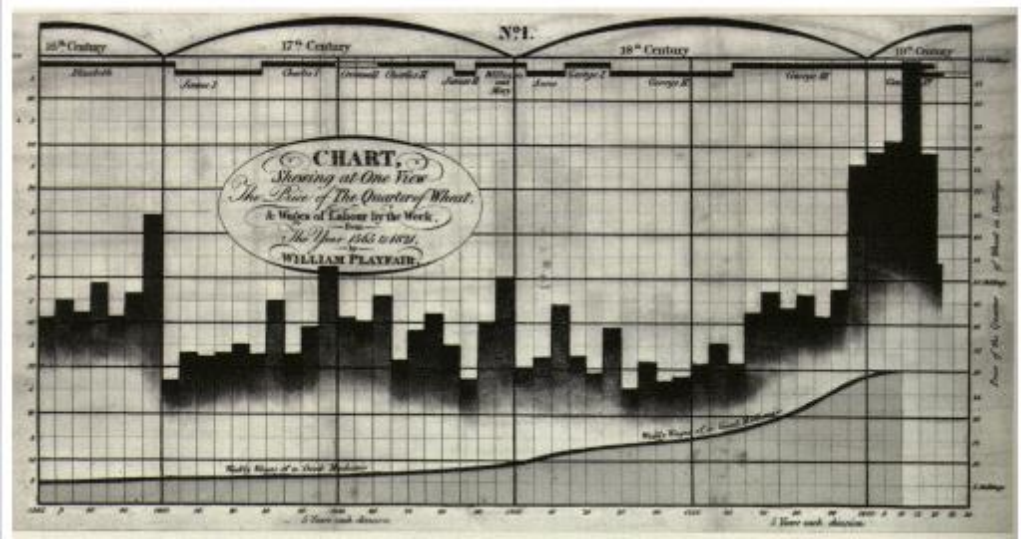


Figure 3.1: An example of old data visualisation, a time series graph by William Playfair in 1821, displaying prices, wages and ruling monarch over a period of 250 years. Playfair (1821), image from Tufte (1983, pg.34)

From 1850 up until 1900 is regarded as the golden age of statistical graphics. There were official state statistical offices being created around Europe in order to gather and visualise statistics to account for the growth of industry, economy, transport and residential areas. During this timeframe, Charles Joseph Minard's *Carte figurative des pertes successives en hommes de l'Armee Francais dans la campagne de Russe 1812-1813*, now known as "Napoleon's March on Moscow" was created (Figure 3.2). It is regarded as "the best graphic ever produced" (Tufte 1983). Polar area charts were invented by Florence Nightingale which she used to try and get better sanitary conditions when treating soldiers' wounds on the battlefield (Nightingale, 1857). Around the same time, Dr. John Snow used a dot map with the deaths from cholera during an outbreak in London in 1854 which helped him reach an outcome to where the source of the outbreak was.

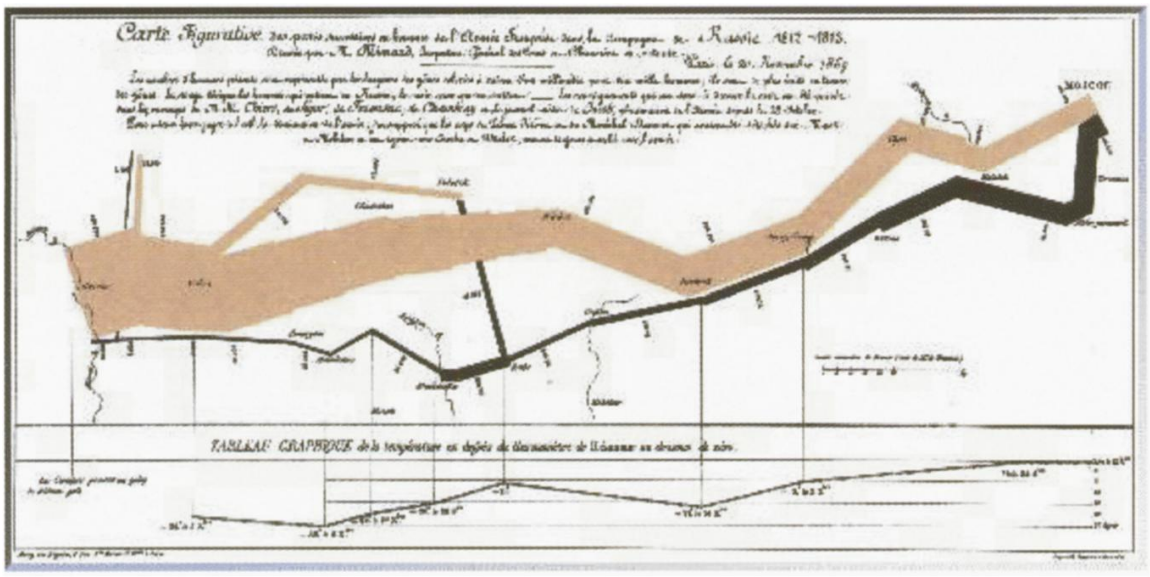


Figure 3.2: Charles Joseph Minard's Carte figurative des pertes successives en hommes de l'Armee Francais dans la campagne de Russe 1812-1813. Source Tufte 1983.

From 1900 up until 1950 is regarded as the dark ages of visualisation. There were very few advancements in displaying statistical graphics.

The resurgence of data visualisation was due to three major developments:

1. John W. Turkey wrote a paper called *The Future of Data Analysis* which called for a distinction between data analysis and mathematical statistics. Shortly afterwards under the Exploratory Data Analysis, new graphic displays were invented such as

stem-leaf and box plots. Turkey's work began to make graphical data analysis both interesting and respectable again (Friendly, 2008).

2. Jacques Bertin published *Semiologie Graphique* where he had organised the graphics by the features and how they were related to the data displayed.
3. In 1957, computer processing of statistics began after the creation of fortran, in a few years university computers were able to construct a number of new and old graphics by using computer programs.

The new innovations presently are, dynamic interactive software and visual animation. The new software allows the user to find patterns and structures inside of their datasets that might otherwise have remained undetected. Visual animation is useful when you want to display data progressing through periods of time.

3.1.2 Types of Data Visualisation

Graphics used for data visualisation can be separated into two groups those that are applicable for continuous data and those that are relevant for discrete data. When creating graphics, you must first decide which group your data belongs to. Continuous data has a large number of possible values whereas discrete data limited definite values. The next step would be to decide how many dimensions your data possesses, one dimensional, two dimensional or multidimensional.

Visualisation of One Dimensional Data

One dimensional (1D) or univariate data are datasets that only contain one variable, most of the graphical formats can also be used on higher dimensional data. There are less ways to display discrete data compared to continuous data (Saito, Miyamura et al, 2005). The most common ways to display 1D discrete data are bar charts, pareto charts and charts based on time. For continuous 1D data a histogram is the most widely used graphic.

Visualisation of Two-Dimensional Data

Two dimensional (2D) or bivariate data are datasets which contain only two variables, 2D data can be split up into three different categories in order to be displayed as graphics. The categories are both variables continuous, one variable discrete and one continuous and both variables discrete.

When both variables are continuous the graphics are visualising relationships the most common graphic to be used in this situation would be a scatterplot. Each variable would be one of the axes.

When one of the variables is discrete and the other is continuous the graphics would be used to display subgroups. Multiway box plots and trellis plots would be the most frequent graphics used.

When both of the variables are discrete, the data is most commonly displayed in tables. This data is displayed as stratified bar and trellis charts, mosaic charts and highlight tables.

Visualisation of Multidimensional Data

Multidimensional data or multivariate data are datasets which contain three or more variables. We can use multidimensional data on a 2D graphic by displaying the extra variables using shading, colours, labels, sizes and patterns (Peng, Ward, Rundensteiner, 2004). In Figure 3.3, the colour shading represents the percentage of pre-school families with the darker colour representing higher percentages. The circles represent the number of households rented from the local authorities with the larger circles corresponding to a higher number of rented households. The third variable displayed are the place labels.

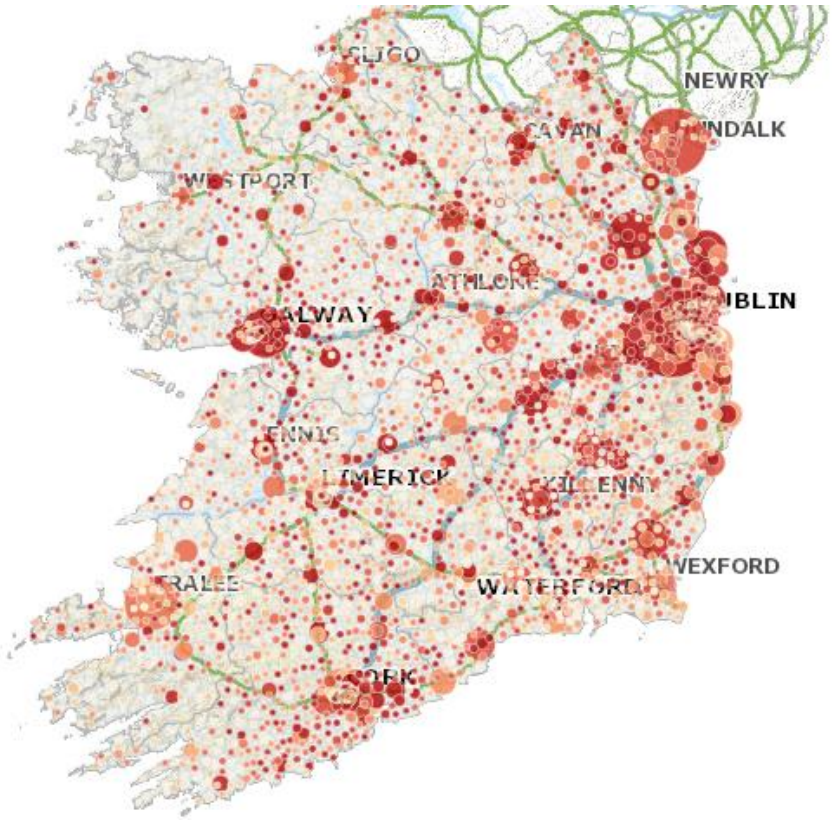


Figure 3.3: This map represents pre-school families and families living in accommodation rented from local authorities, 2016 at electoral division level. (taken from https://www.arcgis.com/apps/Cascade/index.html?appid=2ffa12e1c17c45eeb8acb06606b0d32b)

Examples of multidimensional data visualisation are 2D geospatial scatterplots, trellis plots and mosaic plots. Figure 3.3 is known as a bubble map. The x and y location values, colours and size of the circles are continuous variables while the electoral division areas can be regarded as discrete.

3.1.2 Importance of User Interface

An essential component for a good data visualisation graphic is an easily understandable user interface. When configuring a user interface, one important consideration is the input and output devices being manipulated by the user. Output devices are commonly described as displays because most outputs are visual but they also include auditory, haptic, tactile and olfactory channels. Some examples of output devices are monitors, mobile phone screens, headphones etc. Input devices are the physical devices used to interact with the system. Some examples are keyboards, mouse, microphones etc. (Dix, Finlay et al, 2005)

Ease of use and ease of learning, needs to be taken into account when designing a user interface. Ease of use is better when the application is going to be used often and ease of learning is critical when the user only needs to be on the application for a short amount of time. In terms of data visualisation if the data is constantly being added and updated ease of use might be the more important characteristic. When the graphics are static on a site or printed ease of learning is the more important consideration.

3.2 USABILITY

"It is important to realize that usability is not a single, one-dimensional property of a user interface. Usability has multiple components and is traditionally associated with these five usability attributes: learnability, efficiency, memorability, errors, satisfaction." (Nielsen, 1993).

Learnability is one of the most important aspects of usability, it is the study of how a user will be able to use a website or application. It is about the time it takes for a new user to get around and work with the software in front of them. When testing learnability, the user would be given tasks and seen how long it would take them and if they would need any help to achieve their goal (Jeng, 2005).

Efficiency is how confident the user was when completing tasks. The number of pages the user had to go on before they found what they were looking for, the times they clicked the mouse or pressed a key. All of this is taken into account when measuring the efficiency of software.

Memorability is a very simple concept as it is how well a user will remember the website/application. If a user comes back to the site after not using it for a while will they be able to achieve what they went there for easily or will they give up and switch to another software.

Error is seeing how well a user is able to adapt after making a mistake due to the design of the system. If the user notices a mistake after submitting a form, can they edit the form or to they have to start again etc.

Satisfaction can be subjective (Nielsen, 1993). It involves all of the previously discussed attributes. If the user is struggling to figure out how to manoeuvre around the website, they are not going to like it very much. If they like the website there is a higher chance of them remembering it (Lowry, Spaulding et al, 2006).

The usability and design of applications, websites etc. have been garnering more attention in relation to development. Usability has generally taken more of an engineering approach trying to establish a set of guidelines and universal practises that will guarantee is an outcome of system design (Nielsen 1993, Pearrow 2000, Shneiderman 1998). Some of these guidelines will be discussed in the next section.

3.2.1 Usability in Data Visualisation

There are multiple sets of guidelines for usability, that have been created and are being constantly revised. These guidelines are split into sections such as navigating the interface, organising the display, getting the user's attention and facilitating data entry (Shneiderman and Plaisant, 2010).

The parts of these guidelines that also relate to data visualisation are;

- Use unique and descriptive headings, the chart and axis titles need to accurately portray the information being visualised, any labels used must be distinct and not duplicated.
- Making sure that information visualised using colour can also be viewed without colour in case the user may have some variation of colour blindness.
- Keep displays consistent, do not have a colour representing one variable and then have variable represented by a different colour right after it. Also, headings titles in same locations etc.
- Do not make the graphic overly complicated. The user should be able to understand it quite easily even if they are unfamiliar with the topic.
- Do not have graphics with redundant data that does not add value to what is being displayed.

For data visualisation, considerable information could be given to users for time reliant data or for other conditions. However, the information must be presented to grab their attention (Wickens and Hollands, 2000). When getting the user's attention toward the graphic, the important factors are;

- Size: larger graphics are better but they need to be responsive in order to fit on different displays.
- Fonts: different fonts will help the user distinguish which information responds to the different aspects of the graphic, titles or labels.
- Hovering effects: sections of data may become highlighted or enlarged when the mouse cursor is over it, to help the user have a more accurate representation of where the areas are split.
- Colours: to separate or connect the information.
- Audio: to give feedback when the user is interacting with the graphic.

Users who are out and about for work or otherwise, may not want to have a laptop to look at or show data visuals. These users might favour paper because it is compact and manageable over the power of a laptop. This would be an example where the convenience of a smart phone would be useful but the design of data visualisation on these devices cannot follow the usual approach due to certain issues (Burigat and Chittaro, 2007).

The difference of size and resolution in the mobile displays means that the more detailed information can't be seen unless the user is zoomed in and when the users are zoomed in they lose the overall context of the graphic. So, users are constantly zooming in and out and panning over the information which can be confusing and tiresome. Weaker processing power used to be a big issue when producing more advanced visuals but with the newer mobiles this has become less of an issue. But due to the increasing number of people using mobile phones compared to computers means that usability for mobiles remains very important when considering constructing applications.

3.3 TECHNICAL ASPECTS

3.3.1 R and Shiny

R is a free software environment for statistical computing and graphics. The R environment possesses;

- A data handling and storage facility.
- Operators for calculations on arrays.
- A collection of tools and graphical facilities for data analysis.
- A programming language called S which includes loops, conditionals, input and output facilities and user defined recursive functions. (Venables, Smith and R Core Team, 2017).

R is a tool that can be used to create new approaches to interactive data analysis/visualisation. It possesses packages which are user created extensions and expansions to the base code.

Shiny is a package in R that makes it easy to build interactive web apps. It can be hosted on a webpage or it can be embedded using another package called R Markdown. Shiny can also be extended to accommodate htmlwidgets, CSS and JavaScript.

Shiny applications are contained in a single script called *app.R* which belongs in a user created directory. It contains three components, a UI object, a server function and a call to the *shinyApp* function. The UI object is in charge of the appearance and layout of the application. The server function has the instructions needed for the application to be built by the computer and the *shinyApp* function creates the objects used.

Shiny UI uses a function called *fluidPage* to construct a frame that automatically aligns to size of the user's screen/browser. The layout of the application is made by placing elements in the *fluidPage* function. Content can be added to a Shiny application by putting inside of a *Panel* function and using Shiny's equivalent of HTML tags. Example, *HTML Sample text , Shiny p("Sample text")*. Images are also easy to add in Shiny, just use the *img* function and have your file as a *src* e.g. *img(src = "image.png").* Any HTML friendly tags height, width etc. can also be added in the function. Images must be placed within a folder named *www* inside of the Shiny application's directory.

Shiny also comes with some pre-built widgets, made with R functions named similar to its role. For example, *radioButtons*() creates radio buttons etc. Widgets are added in the same way as content by using panels. Each widget needs several parameters the first always being a name for the widget together with a label.

To make a Shiny application reactive the following steps are required:

- Have an output function inside of the UI object which will place reactive objects.
- Have a render function inside of the server object which will inform the computer how to build the objects.
- Make sure the R expressions inside the render functions are surrounded by {}
- Have the render expressions in the output list, an entry for every reactive object in the application.
- Include an input value in one of the render expressions.

3.3.2 D3 and JavaScript

D3(Data-Driven-Documents).js is a JavaScript library that it is used to manipulate documents based on data. D3 uses HTML, SVG and CSS to visualise data. D3 allows data to be attached to the Document Object Model (DOM) and applies data driven transformations on the document. This data can also be used to create an interactive visualisation graphic in the form of SVGs. (Zhu, 2013)

Browsers have their own built in APIs used for manipulating the DOM but take more time and effort to do the simple aspects. JavaScript libraries were made to make the manipulation of the DOM more accommodating, jQuery being the most widely known. These libraries all use the same approach, selecting HTML elements and then applying changes to them. The idea was taken from CSS. The selection processes of a browser API, CSS, jQuery and D3 are shown in Figure 3.4 below. In order to be used for data visualisation they need to be able to create and remove HTML elements. This is not possible with CSS and jQuery as they bind the data individually. D3 uses enter and exit methods/selections in order to create new elements for data coming in and removing elements that are no longer needed in the project.

```
var ps = document.getElementsByTagName("p");
for (var i = 0; i < ps.length; i++) {
  var p = ps.item(i);
  p.style.setProperty("color", "white", null);
}
```
a

```
p { color: white; }
```
b

```
$("p").css("color", "white");
```
c

```
d3.selectAll("p").style("color", "white");
```
d

Figure 3.4: A simple document transformation that colours paragraphs white. (a) W3C DOM API; (b) CSS; (c) jQuery; (d) D3 (Bostock, Ogievetsky, Heer, 2011).

D3 uses dynamic properties, the value being attached to an element does not have to be a constant it can be represented by a function. The functions used can be very basic but provide a significant impact. Data in D3 is represented by an array of values and the user can choose the specific index in the array to manipulate. D3 does not possess its own visual graphics, it uses the already created ones from HTML, SVG and CSS. In coding you will use D3 to create SVG elements and use CSS to stylise them, any new additions to browsers will be able to work with D3 immediately. D3 is also easy to debug by using the browser's inspector/console. D3 also allows animations by their transition method/selection.

3.3.3 Comparison between D3 and R

D3 Advantages

- Visualise the data you want, create own classes.
- Large amount of data can be displayed
- Data bound to DOM, can use DOM functions.
- Loading data from different types of documents, json etc.
- Transforming and controlling SVGs is relatively simple.
- Compatible with JavaScript frameworks.
- Lots of examples and large community.
- Can use browser inspector to debug code.

D3 Disadvantages

- Not supported by IE8 and older browsers.
- Steep learning curve.
- Labels are not handled automatically.
- Takes longer to make compared to simple visualisation software.

Shiny Advantages

- Can work with other R packages.
- Can be integrated with D3.
- Can be embedded in HTML by using Rmarkdown.
- Not much extra to learn if already comfortable with R.
- Also has large community.

Shiny Disadvantages

- No actual tech support. Help is given mostly by the community via RStudio.
- If code is not optimised app could be slower than wanted.

3.5 CONCLUSION AND FURTHER RESEARCH

To conclude, usability is an integral part of modern data visualisation. It has produced guidelines to be loosely followed when creating software. These guidelines help when constructing visualisation graphics by giving the coder/creator a checklist of what may be added to their visuals. These graphics will then be able to attract the user's attention and will be easier for them to understand. Data visualisation has a long history intertwined with social sciences and its future will be heavily influenced by usability concerns either through social media or other areas. Only two possible ways of creating data visuals were discussed D3 (JavaScript library) and Shiny (a package for R). Both of them are attached to a separate coding language. D3 uses the DOM elements in the user's browser while Shiny needs Rmarkdown to be embedded inside HTML.

4. Design

4.1 SKETCHING THE UI

The first stage of designing the UI was creating paper prototypes/sketches. During this stage three main design prototypes were kept, a login screen, a registration menu and a basic solution for the main page. Originally the idea for the login screen was that it will be a pop-up window that appears when the user had opened the application. Figure 4.1, is a sketch of what the login screen would have looked like. The plan at this stage was that login was not going to be an option, the application was going to be solely for blood bank staff members. The login screen would have consisted of a text input for the username, a password input and two buttons one to log the user in and another which would have brought up the registration screen.

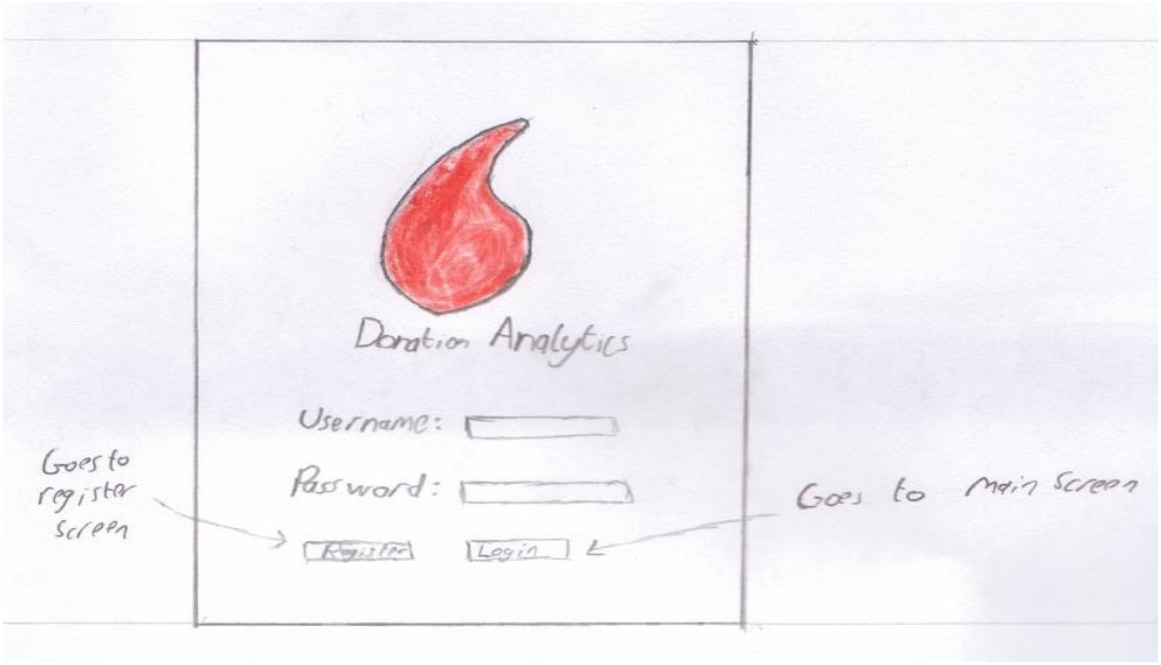


Figure 4.1: A sketch of the login page.

The registration screen in Figure 4.2, appears after the user clicks on the register button. It is very similar to the sketched login screen with inputs for username and a password. It has another password input this time for confirming the user's password. The last input is for an admin code. Any new staff member at the blood bank would be provide with an admin code and input this code when they are registering their account.

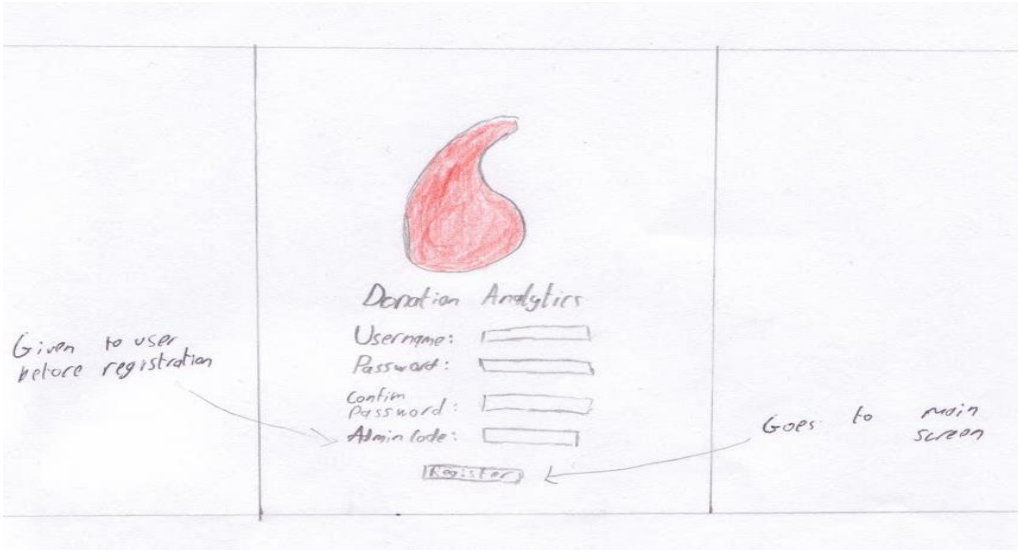


Figure 4.2: A sketch of the registration page.

The last sketch Figure 4.3 was the original idea for the main/home screen. The idea for the home screen was to have the interactive map the first content the user would see when they login. The navbar was planned to be put on the top of the page with links to stocks, locations and other information that was yet to be discussed. Also, at this stage it was thought that the map could be interacted with by using check boxes. The check boxes were later changed to radio buttons.

Figure 4.3: A sketch of the home page.

4.2 CREATING THE UI PROTOTYPE

The second stage to creating the UI was creating a functioning prototype in RStudio. The idea of the three sketches from the first stage were followed with the end results of this stage being somewhat similar to the original sketches. Before the sketches were followed basic graphs, plots and other pieces of code were experimented with to get an understanding of the coding language. The first step in creating the prototype was making a login screen as shown in Figure 4.4. The main difference between this login screen and the sketched one is the logo was not added. The usernames and passwords were stored on a local file during this stage of development as the database was not created. The validation at this stage checked if the username and password were on the same row in the file. If they weren't red text would pop up alerting the user.

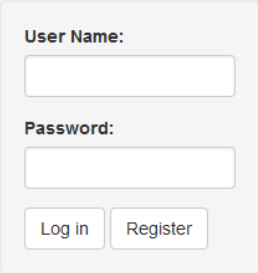


Figure 4.4: The prototype login screen.

After the login screen was created and its validation set up, the next step was creating a database and having the application connect to it. The back end of the login system was then altered to work with the database. After the database was connected and the login system fixed, the next step was creating a way for users to register. The prototype registration screen, Figure 4.5, is very similar to the prototype login screen. The only differences are the extra password input and the removal of the login button. At this stage of development, the admin code input was not added to the registration screen.

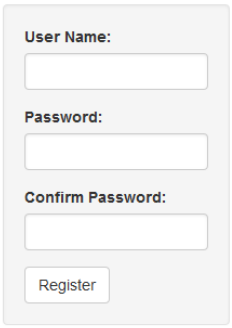


Figure 4.5: The prototype registration screen.

Not much thought was put into the design of the main/home screen as functionality was the goal at this stage. A number of graphs and maps were in separate R projects at this point. The main R project contains the database connection, the login and registration screens and a map similar to the ones showed in Figures 4.6 and 4.7. These images show the simplicity of the UI during this stage of development.

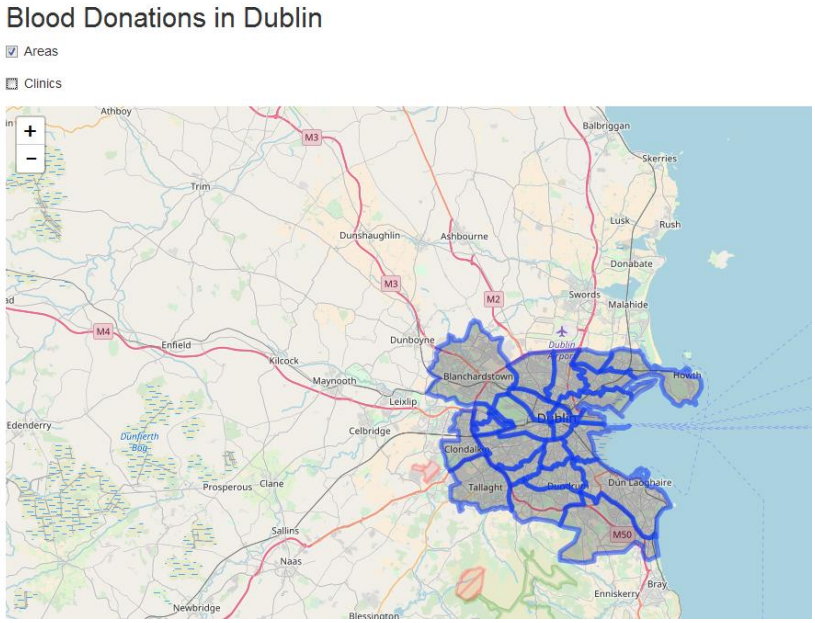


Figure 4.6: A highlighted area of Dublin activated by a checkbox.

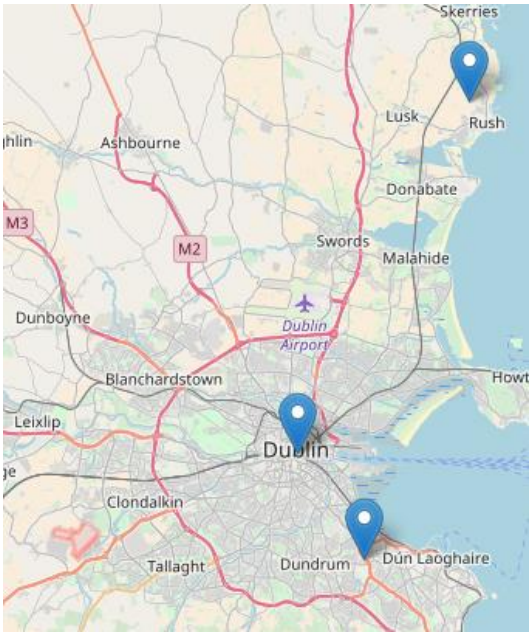


Figure 4.7: The location of three donation clinics in Dublin.

4.3 CREATING THE FINAL UI

The UI changes drastically from the previous version because of the introduction to the shiny dashboard R package. The shiny dashboard and the other R packages are explained in the implementation section of the report below. Shiny dashboard contains a *box* function and when used with Shiny's *fluid row* function can be used to create bootstrap like grids. The login screen in Figure 4.8, hasn't changed much in terms of coding but has changed a lot visually. The same can be said for the registration screen in Figure 4.9 below, the only exception being the admin code input was added.

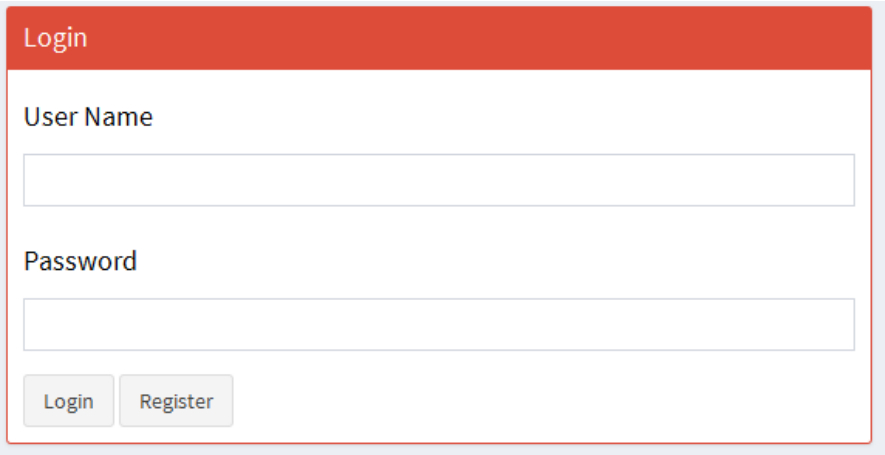


Figure 4.8: The finalised login screen.

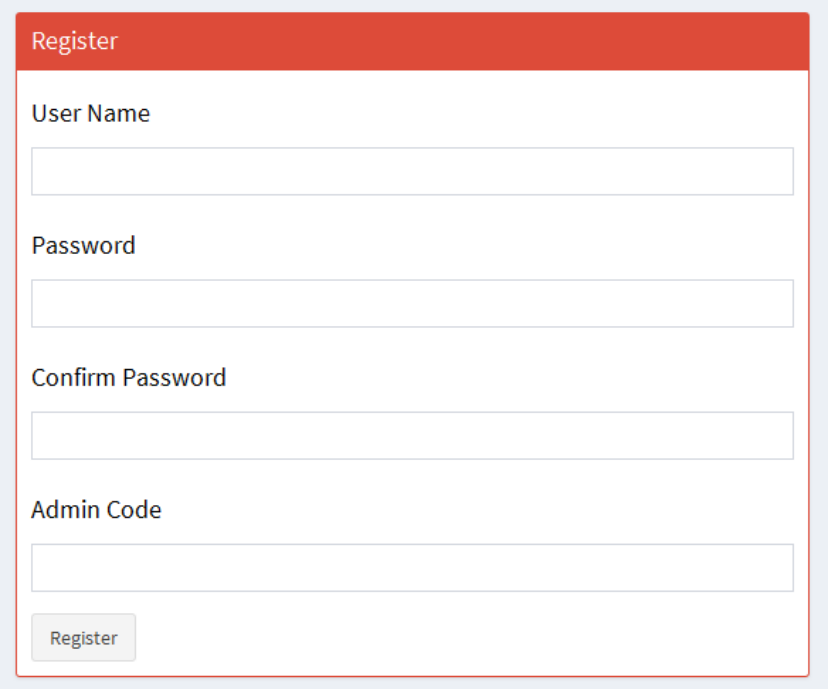


Figure 4.9: The finalised registration screen.

The UI for the rest of the application has also changed significantly. In Figure 4.10, is one of the pages visible to the user after they have logged in. The navbar design has been changed from originally being horizontal at the top of the page to being a vertical sidebar. Each of the graphs and other visuals are all placed inside boxes creating a gridded effect on each page. It is also optional whether the user logs in or not as all of the visualised statistics are behind the login. For example, information on the status of blood donations is visible to the general public before logging in.

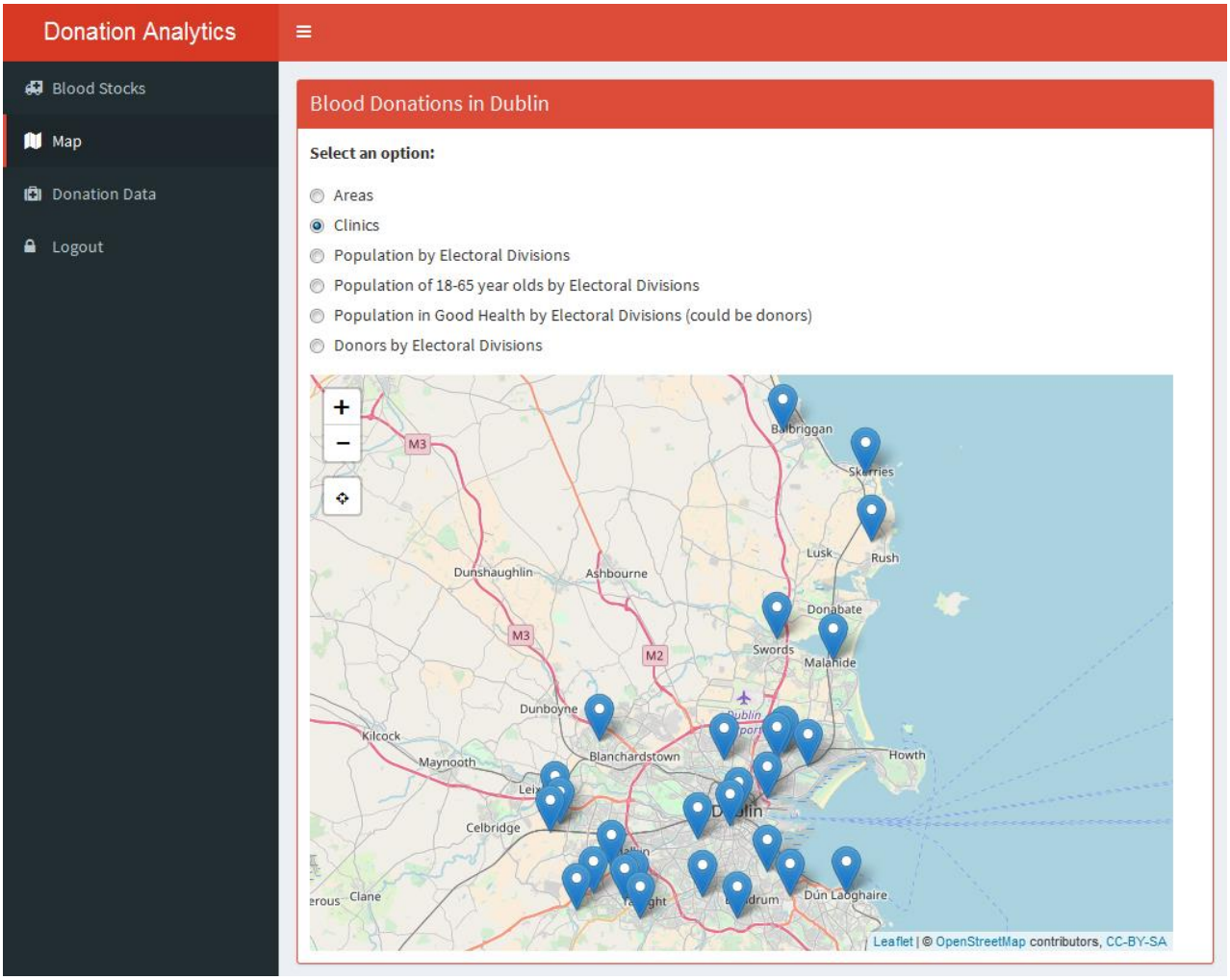


Figure 4.10: One of the screens viewable after a user has logged in.

5. Implementation

5.1 PACKAGES

Packages in R are used to expand the capabilities of R projects. To install these packages in RStudio go to the window in the bottom right of the screen. As shown in Figure 5.1. Click on the install button and a window will appear, Figure 5.2, type in the name of the package you want to install, make sure install dependencies is checked then click on the install button. To activate the packages in a project, have the code, *library("packagename"),* usually this piece of code is located at the start of the R file.

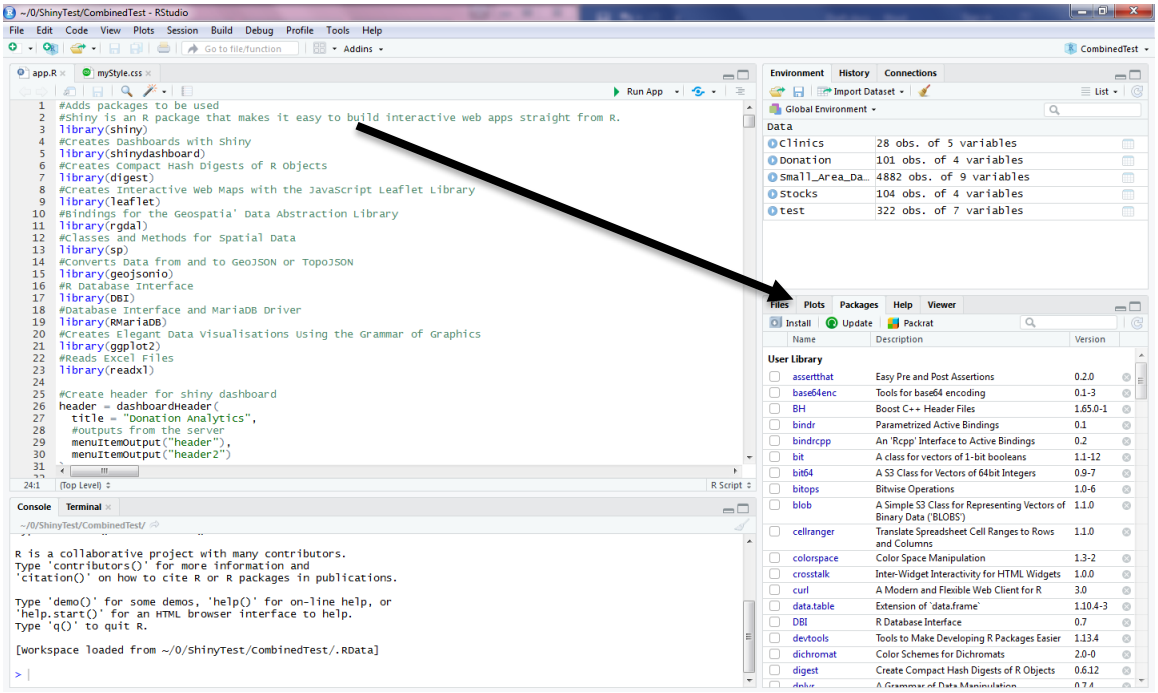


Figure 5.1: An arrow highlighting the install button in the packages tab in RStudio.

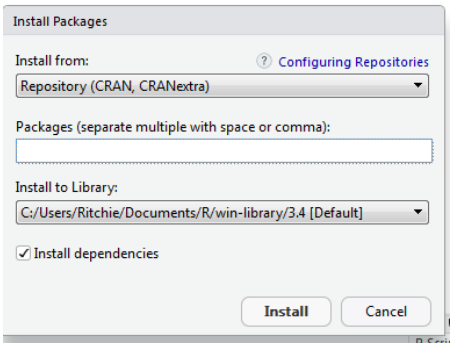


Figure 5.2: This window appears when the install button is clicked.

5.1.1 Shiny

Shiny is a R package that makes it easier to build interactive web applications using R code. Shiny applications are stored in an individual script called *app.R* or in server.*R* and *ui.R* scripts. This application was created in a single script. The script consists of an ui object, a server function and a call to the *shinyApp* function which is usually the last line of code in a Shiny app. The ui object contains the code that is used to display content to the user. The server object does all the calculations to create plots and graphs outputting the results to be shown in the ui object. Figure 5.3 is the script of a simple application made with Shiny. This code creates a basic application in shiny which shows a map of Ireland with the borders of the counties shown, Figure 5.4. This project uses the ShinyDashboard package for the ui. A more in-depth description of Shiny can be found in the research section of this report.

Figure 5.3: An example of a shiny application.

Figure 5.4: The result of the script shown in Figure 5.3.

5.1.2 Shinydashboard

ShinyDashboard is a package used to make dashboards an example which is shown in Figure 4.10. The dashboard is split into three parts, a header, a sidebar and a body. To create a dashboard the *dashboardPage*() function is used and it requires a header as shown in Figure 5.5, a sidebar in Figure 5.6, and a body shown in Figure 5.7. These components are created with the *dashboardHeader*(), *dashboardSidebar*(), and *dashboardBody*() functions respectively. The dashboard page is assigned to the ui object in the Shiny application. The header is used to create the title and some dropdowns for messages and alerts. The sidebar is for links to display different content on the body. The body displays content in *box*() functions and using *fluidRow*() functions.

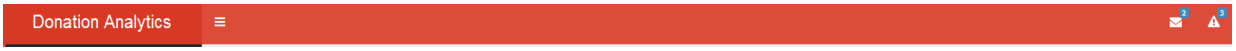


Figure 5.5: A ShinyDashboard header.

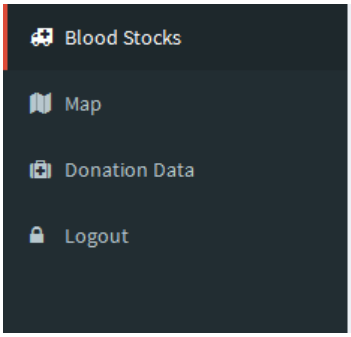


Figure 5.6: A ShinyDashboard sidebar.

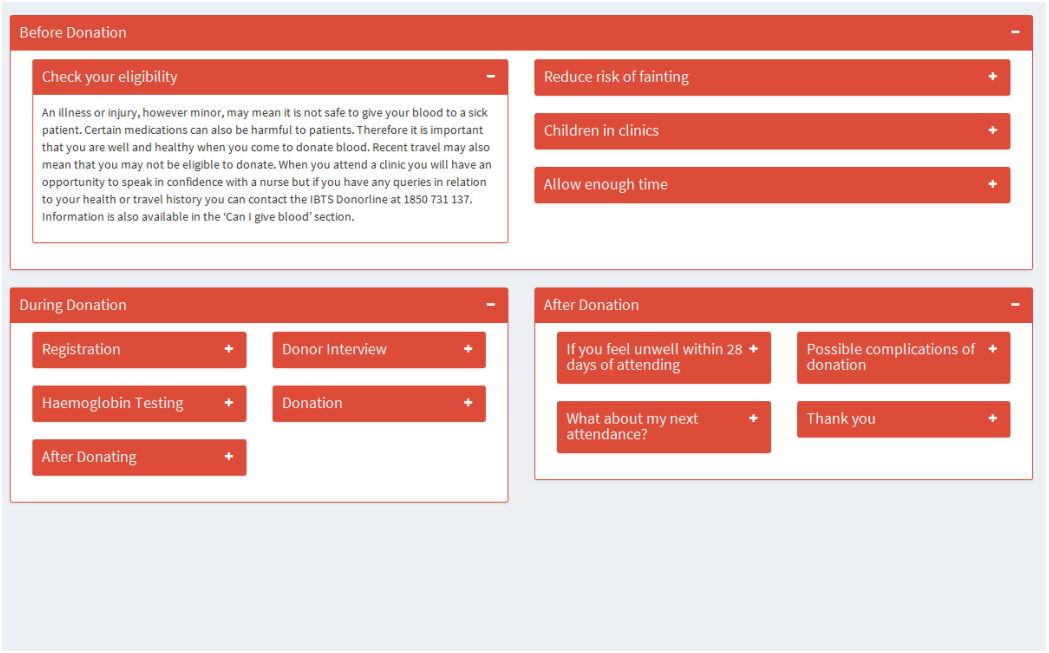


Figure 5.7: A ShinyDashboard body.

5.1.3 Digest

The digest package is used to create hash messages for authentication code. In this project passwords are hashed using digest before they are sent to the database.

5.1.4 Leaflet

The leaflet package is used to create maps in R. The map is created by using the *leaflet*() function and then by adding layers/features to it.

5.1.5 Rgdal

The R rgdal package is used for binding geospatial data. It is required when using the geojsonio package.

5.1.6 Sp

The R sp package consists of methods and classes for importing spatial data in the form of shape files. It is required when using the rgdal package.

5.1.7 Geojsonio

The geojsonio package is used to convert data into TopoJSON or GeoJSON from R classes. It is used to read the *dub.json* file in the form of sp.

5.1.8 DBI

DBI stands for database interface. It is used by R to communicate with database management systems.

5.1.9 RMariaDB

The RmariaDB package is used to connect to MariaDB and MySQL databases. This project uses a MySQL database for logins and registration.

5.1.10 Ggplot2

The ggplot2 package is used to create graphics and plots. All the plots in this project were made using ggplot2.

5.1.11 Readxl

The readxl package is used to read excel files into R.

5.2 DATABASE AND LOGIN

5.2.1 Connecting to the Database

The database first needs to be created in phpMyAdmin. The database is named shiny and has a table called users and columns named username and password. After the database is created the packages DBI and RmariaDB need to be installed. The code in Figure 5.8, creates the connection to the database. The *dbConnect* function is from the DBI package and it is responsible for the connection. It takes the host, user, password, database name, port and an object that inherits from the DBIdriver which in this case is from the RmariaDB package.

```
#Database Code
con <- dbConnect(RMariaDB::MariaDB(),
                 host="localhost", user="root", password="",
                 dbname="shiny",  port=3306)
```

Figure 5.8: Code required to connect to the database.

5.2.2 Login and Registration

As shown in Figure 4.8 in the design section of the report, the user requires a username and password to log in. Figure 5.9 is the code for the login ui. The *renderUI* function is used to generate html with shiny. The box function is from shiny dashboard and is used to display the content. Inside the box is a text input, a password input, two buttons, one to login and one to bring up the registration screen and a ui output which shows error messages.

```
#Login display
output$uiLogin <- renderUI({
  box(title = "Login", status = "danger", solidHeader = TRUE,
    textInput("username", label = h4("User Name", style = "color:black")),

    passwordInput("password", label = h4("Password", style = "color:black")),

    actionButton("login_button", "Login"),

    actionButton("regButton", "Register"),
    uiOutput("pass")
  )
})
```

Figure 5.9: Code used to create the login showed in Figure4.8.

After the login button is clicked a query is sent to the database to get all of the users and the results are put into a credentials variable. Row username and row password are then created. The *which* function checks to see if the database username matches the inputted username and then does the same for the password. The first if statement in Figure 5.10 checks to see if the username and password are on the same row in the database, if they

are valid credentials is set to true. Then the user is authenticated and logged in or if valid credentials is set to false authenticated is then set to false. If authenticated is false, the input status is changed to one of three options. Each of the input statuses refer to a different error message.

```
#When login button is pressed
observeEvent(input$login_button, {
  #Send mariadb query
  rs <- dbSendQuery(con, "select * from users where 1;")
  #Fetch results
  credentials <- dbFetch(rs)
  #Assign value for username
  rowUsername <- which(credentials$username == input$username)
  #Assign value for password
  rowPassword <- which(credentials$password == digest(input$password)) # digest() makes md5 h

  # if user name row and password name row are same, credentials are valid
  if (length(rowUsername) == 1 &&
      length(rowPassword) >= 1 &&  # more than one user may have same pw
      (rowUsername %in% rowPassword)) {
    userInput$validCredentials <- TRUE
    dbClearResult(rs)
  }


  # if a user has valid credentials he is authenticated
  if (userInput$validCredentials == TRUE) {
    userInput$authenticated <- TRUE
  } else {
    userInput$authenticated <- FALSE
  }

  # if user is not authenticated, set login status variable for error messages below
  if (userInput$authenticated == FALSE) {
    if (length(rowUsername) > 1) {#More than one username
      userInput$status <- "credentials_data_error"
    } else if (input$username == "" || length(rowUsername) == 0) {#username doesn't exist
      userInput$status <- "bad_user"
    } else if (input$password == "" || length(rowPassword) == 0) {#password doesn't exist
      userInput$status <- "bad_password"
    }
  }
})
```

Figure 5.10: The code that runs after the login button is clicked.

The code in Figure 5.11 is used to create the register screen in Figure 4.9. It appears after the user has clicked the register button on the login screen. It contains two text inputs, two password inputs a button and an ui output which will display errors to the user.

```
#Register display, first register button clicked
observeEvent(input$regButton, {
  output$uiRegister <-renderUI({
    box(title = "Register", status = "danger", solidHeader = TRUE,
      textInput("usernameR", label = h4("User Name", style = "color:black")),
      passwordInput("passwordR", label = h4("Password", style = "color:black")),
      passwordInput("cpassword", label = h4("Confirm Password", style = "color:black")),
      textInput("adminCode", label = h4("Admin Code", style = "color:black")),
      actionButton("regButton2", "Register"),
      uiOutput("pass2")
    )
  })
})
```

Figure 5.11: Code used to create the register showed in Figure 4.9: above.

After the register button is clicked a query is sent to the database to get all of the usernames. They are then assigned to a name check variable. The usernames are then compared to the user input. The code then uses if statements as shown in Figure 5.12 to check if the user has typed in a username, a password, confirmed the password and typed in a admin code. It then checks if the username is already taken if the passwords match and if the user submitted admin code matches the one in the code. If one of them is not true the user input status is changed and an error message is shown to the user corresponding to what needs to be changed. If everything is successful the username and the hashed password are assigned to x and y variables in order to be inserted into the database. The user is then authenticated and logged in.

```
#Register code, second register button clicked
observeEvent(input$regButton2, {
  #mariadb query
  rs4 <-dbSendQuery(con, "select username  from users where 1;")
  #Fetch results
  nameCheck <- dbFetch(rs4)
  #Assign value for existing usernames
  unameExist <- which(nameCheck$username == input$usernameR)
  #Check if username input is blank
  if (input$usernameR =="") {
    userInput$status <- "nameR"}
  #Check if username is taken
  else if (length(unameExist) >= 1){
    userInput$status <- "nameTaken"}
  #Check if password input is blank
  else if (input$passwordR =="") {
    userInput$status <- "passR"}
  #Check if confirm password input is blank
  else if (input$cpassword =="") {
    userInput$status <- "cpass"}
  #Check if Admin code input is blank
  else if (input$adminCode =="") {
    userInput$status <- "aCode"}
  #Check if Admin code input is blank
  else if (input$adminCode != adminCode) {
    userInput$status <- "aCode2"}
  #Check if passwords match
  else if (input$passwordR != input$cpassword){
    userInput$status <- "password"
  }else {
    userInput$status <- ""
    #X and y values set to be insered in mariadb
    x <- input$usernameR
    y <- digest(input$passwordR)
    #insert query
    rsReg <- dbSendQuery(con, sprintf("INSERT INTO users (username, password)
                                      VALUES ('%s', '%s');", x, y))
    dbClearResult(rsReg)
    dbDisconnect(con)
    userInput$authenticated <- TRUE
  }
})
```

Figure 5.12: The code that runs after the register button is clicked.

5.3 CREATING THE UI

5.3.1 Shiny Dashboard

The shiny dashboard package is used to create the ui in this application. In Figure 5.13 the ui is shown to be defined by a *dashboardPage* function. The function takes four variables known as skin, header, sidebar and body. The skin variable relates to the colour of the dashboard for this application and is set to red. The header variable was created using the *dashboardHeader* function that took a title and two outputs from the server which will be discussed later. The sidebar variable was created using the *dashboardSidebar* function that then uses the sidebar menu function. This function contains an output from the server which will also be discussed later. The body variable also has its own function called *dashboardBody*. In this application, it contains code used to import a CSS file and another output from the server to be discussed.

```
#Create header for shiny dashboard
header = dashboardHeader(
  title = "Donation Analytics",
  #outputs from the server
  menuItemOutput("header"),
  menuItemOutput("header2")
)
#Create sidebar for shiny dashboard
sidebar = dashboardSidebar(
  sidebarMenu(
    #outputs from the server
    menuItemOutput("menu")
  )
)
#Create body for shiny dashboard
body = dashboardBody(
  #Import CSS file
  tags$head(
    tags$link(rel = "stylesheet", type = "text/css", href = "myStyle.css")
  ),
  #outputs from the server
  menuItemOutput("ui")
)


# Define dashboard's UI
ui <- dashboardPage(skin="red",header, sidebar, body)
```

Figure 5.13: The code used to create a shiny dashboard.

5.3.2 Outputting UI from the Server

In order for the ui to be outputted, the server's logic needs to be defined. In Figure 5.14, the first line of code is setting up the server, it uses a function which contains the variables input, output and session. Input refers to radio buttons, text inputs and other widgets, output is what gets sent to the ui and session is the environment. Each output that is sent to the ui is defined by *output*$"*NameOfOutput"* which then uses the *renderUI* function to generate the HTML. The next step for each of the outputs is an if statement to check if the user is logged in or not. *Output$ui* is the main ui in the body as shown in Figure 5.14. *TabItems* is used in relation to *menuItem* in the sidebar. The *tabItem's* name

corresponds to a *menuItem* with the same name, when a user clicks on a *menuItem* the corresponding tab will be brought up. Inside the tab items the content is split into *fluid rows* and *boxes*. *Fluid rows* are used to create responsive rows and *boxes* are similar to shiny columns but they have extra features. An example of a box is shown in Figure 5.7. *Boxes* can contain a title, a width with a value of one to twelve, a collapsible variable which is either true or false and a status which applies a colour to the box. The danger status makes the box red, solid header decides if the whole box is the colour or just the header and collapsed decides if the box starts off showing the content. Inside the box can be HTML content equivalent tags, *p()* = <*p*>, any widget, plot outputs etc.

```
# Define server logic
server <- function(input, output, session) {

  #create ui output for body
  output$ui <- renderUI({
    #If user is not logged in
    if (userInput$authenticated == FALSE) {
      #tabitems are represented on the sidebar
      tabItems(
        #Content for non-administrative user
        tabItem(tabName = "home",
          fluidRow(
            box(title = "Before Donation", width = 12,  collapsible = TRUE, status = "danger", solidHeader = TRUE,
              box(title = "Check your eligibility",  collapsible = TRUE, status = "danger", solidHeader = TRUE, collapsed = TRUE,
                p("An illness or injury, however minor, may mean it is not safe to give your blood to a sick patient.  Certain medicati
              ),
              box(title = "Reduce risk of fainting", collapsible = TRUE, status = "danger", solidHeader = TRUE, collapsed = TRUE,
                p("Drinking plenty of cold, non-alcoholic fluids in the 24 hours prior to donating and eating savoury food and / or sal
              ),
```

Figure 5.14: The code used to output the ui.

5.3.3 Different UI Outputs

Output$ui is the main output and contains four tabs before and after the user has logged in. Before the user logs in there is a home page, a clinics page, a news reports page and a login page. The home page contains three main boxes, titled *before donation, during donation* and *after donation*. Inside of the three main boxes are more boxes that provide information to the user on a number of topics. The clinics page contains a map of Dublin that shows the locations where the user can go to donate blood. The news reports page has a variety of news reports in relation to blood donations. The login page allows the user to login or register.

After the user has logged in their options on the sidebar are blood stocks, map, donation data and logout. The blood stocks page contains multiple graphs and plots relative to the current stocks. The map page is a map of Dublin that shows the location of clinics, donations today, population of donors split into electoral districts etc. The donation data page contains more graphics, with donations classified by sex, which days have more donations etc. The logout page asks the user if they are sure they wish to exit and then logs them out.

Output$menu is the output for the sidebar menu. The script in Figure 5.15 checks if the user is logged in. It then outputs one of two menus, with the menu items correspond to the tabs discussed above. Each menu item has a title, which is what is displayed to the user, the tab name references the tab from the body and the icon which is taken from font awesome.

```
#Outputs sidebar menu
output$menu <- renderMenu({
  #if user logged in
  if (userInput$authenticated == TRUE) {
    sidebarMenu(
      menuItem("Blood Stocks", tabName = "stocks", icon = icon("ambulance")),
      menuItem("Map", tabName = "map", icon = icon("map")),
      menuItem("Donation Data", tabName = "donation", icon = icon("medkit")),
      menuItem("Logout", tabName = "logout", icon = icon("lock"))
    )
  }

  #user not logged in
  else{
    sidebarMenu(
      menuItem("Home", tabName = "home", icon = icon("h-square")),
      menuItem("Clinics", tabName = "clinics", icon = icon("map")),
      menuItem("News Reports", tabName = "news", icon = icon("user")),
      menuItem("Login", tabName = "login", icon = icon("lock"))
    )
  }
})
```

Figure 5.15: The output for the sidebar menu.

Output$header and header2 are both used only when the user is logged in. Figure 5.16 is the code used to output header2 and beside it is what it looks like in the application. As with the other outputs an if statement checks if the user is logged in. If the user is logged in both outputs are added to the dashboard header. These outputs are dropdown menus that display notifications and messages.

```
#Outputs dashboard header
output$header2 <- renderMenu({
  #If user logged in
  if (userInput$authenticated == TRUE) {
    dropdownMenu(type = "notifications",
      notificationItem(
        text = "5 new donors today",
        icon("users")
      ),
      notificationItem(
        text = "Donations have arrived at storage facility",
        icon("truck"),
        status = "success"
      ),
      notificationItem(
        text = "Stocks are low in Tallaght",
        icon = icon("exclamation-triangle"),
        status = "warning"
      )
    )
  }
  #If user not logged in
  else{
  }
})
```

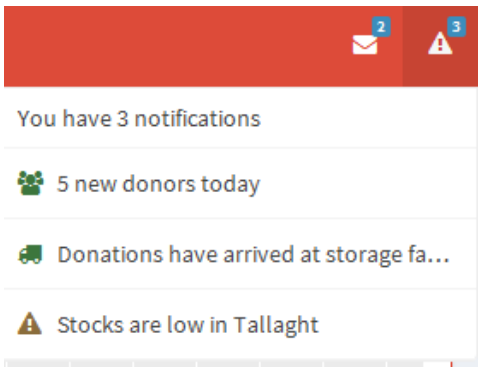


Figure 5.16: The code used to create the dropdown/header2.

5.4 CREATING THE GRAPHICS

5.4.1 Importing and Creating data

In order to create graphics using R, data needs to either be created or imported into R Studio. For this application the data used is in the form of Json files, csv, data frames and excel documents. The files to be imported are placed inside a data folder, which is inside the projects folder. To locate the folder when inside RStudio look under the files tab in the bottom left window, shown in Figure 5.17. In the same image the import dataset button is highlighted by the smaller arrow. This button is used to import the excel files. After the button is clicked there is a prompt asking for what type of file to import. A window showing a view of the data is there displayed as shown in Figure 5.18. Options at the bottom right of the window allow the user to name the variable that contains the data. Other options include how much of the data will be imported, if the row names are kept and if the data viewer will be brought up. In the bottom left is the code preview. The first line will tell the application the readxl package will be used, the second line is the created variable which will be used to create graphics while the third line will bring up the data in RStudio and does not need to be kept in the final application.

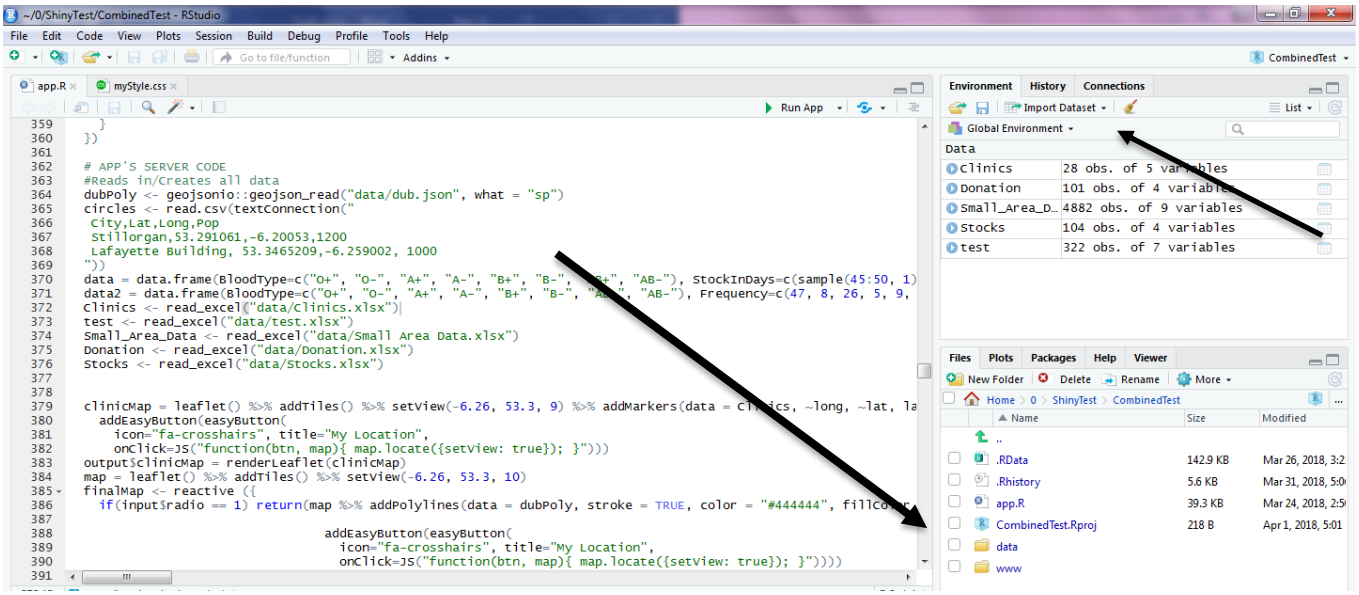


Figure 5.17: The files tab contains the data and www folders as well as the app and project files.

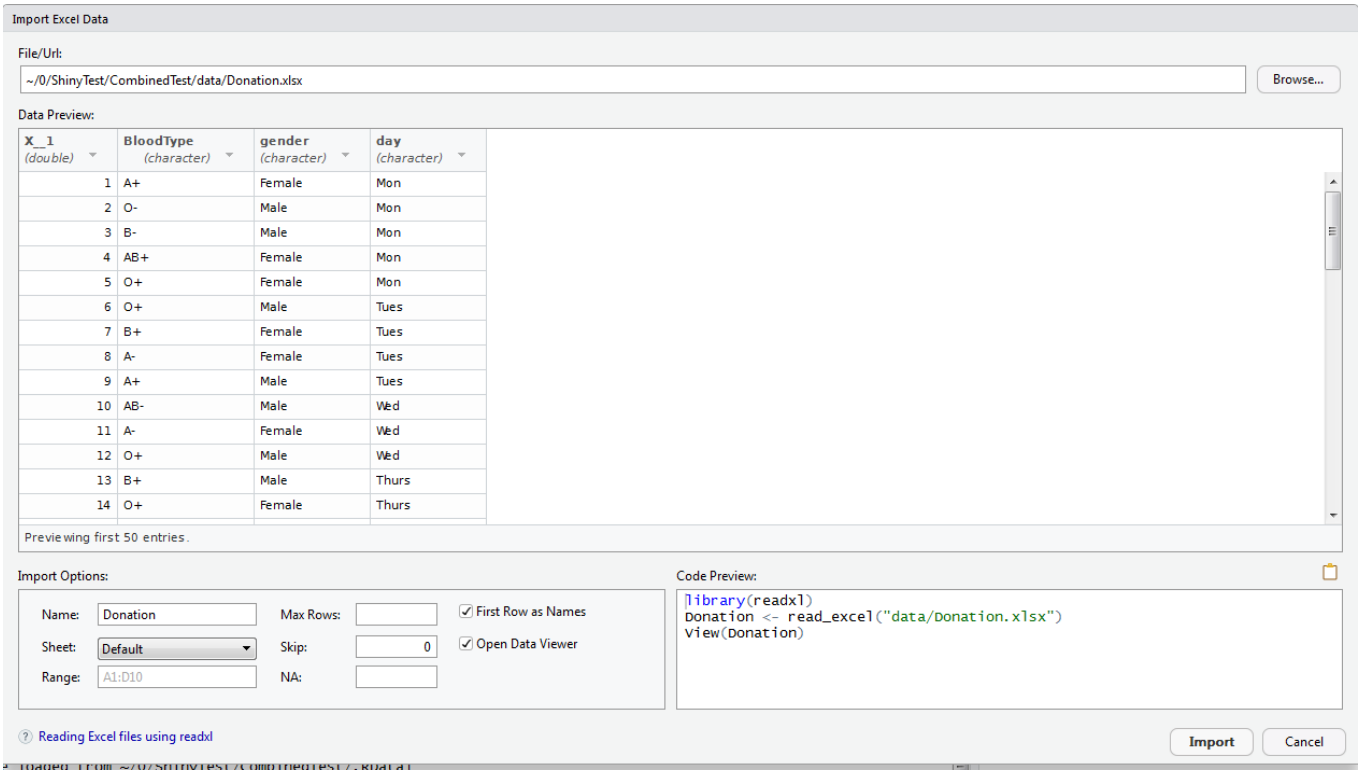


Figure 5.18: This window is displayed when importing an excel file to RStudio.

Figure 5.19 is the code used to either import data or create data that will be used to create graphs and plots. Using the geojsonio package the dubPoly variable is filled with spatial data of a selected area of Dublin. The circles variable's data was created inside RStudio, a csv file (otherwise known as comma separated files) was chosen for this variable because of its simplicity. If the circles variable contained a larger amount of data it would be more appropriate for it to be in a separate file. The *data* and *data2* variables are the basic way of creating data in R. These variables each have two rows of data and are used to create simple yet effective graphics. The other variables left to discuss include excel files, *small area data* and *test* were both created using data obtained from the Central Statistics Office. Other files were created using dummy data to illustrate proof of concept.

```
# APP'S SERVER CODE
#Reads in/Creates all data
dubPoly <- geojsonio::geojson_read("data/dub.json", what = "sp")
circles <- read.csv(textConnection("
 City,Lat,Long,Pop
 Stillorgan,53.291061,-6.20053,1200
 Lafayette Building, 53.3465209,-6.259002, 1000
"))
data = data.frame(BloodType=c("O+", "O-", "A+", "A-", "B+", "B-", "AB+", "AB-"), StockInDays=c(sample(45:50, 1),sample(8:12, 1),s
data2 = data.frame(BloodType=c("O+", "O-", "A+", "A-", "B+", "B-", "AB+", "AB-"), Frequency=c(47, 8, 26, 5, 9, 2, 2, 1))
Clinics <- read_excel("data/Clinics.xlsx")
test <- read_excel("data/test.xlsx")
Small_Area_Data <- read_excel("data/Small Area Data.xlsx")
Donation <- read_excel("data/Donation.xlsx")
Stocks <- read_excel("data/Stocks.xlsx")
```

Figure 5.19: The code used to import and create the data used in the application.

5.4.2 The Graphics and Maps

The graphics used in this application are created using ggplot2 and leaflet. Most of the visuals and graphics are not visible unless the user has logged in. The only visual that is available when the user has not logged in is a map of Dublin with markers at locations where donations can be made. Figure 5.20 is the code used to create this map. The variable *clinicMap* is created to store the information about the map. The leaflet function is used to create the map widget. *%>%* is a pipe used to chain together functions. The *addTiles* function is used to add the default map tiles from open street map. The *setView* function takes a longitude position, a latitude position and a zoom factor which is then used to configure the view. The *addMarkers* function is used to add markers on specified locations, in this case it uses the clinics dataset, takes the longitude, latitude positions and sets the markers labels to the corresponding names. The last thing added to the *clinicMap* is an easy button, it is used to grab the user's location and centre in on it. The map is then rendered and outputted. The output is shown in Figure 5.21.

```
clinicMap = leaflet() %>% addTiles() %>% setView(-6.26, 53.3, 9) %>% addMarkers(data = Clinics, ~long, ~lat, label = ~name) %>%
  addEasyButton(easyButton(
    icon="fa-crosshairs", title="My Location",
    onClick=JS("function(btn, map){ map.locate({setView: true}); }")))
output$clinicMap = renderLeaflet(clinicMap)
```

Figure 5.20: The code used to create the clinics map.

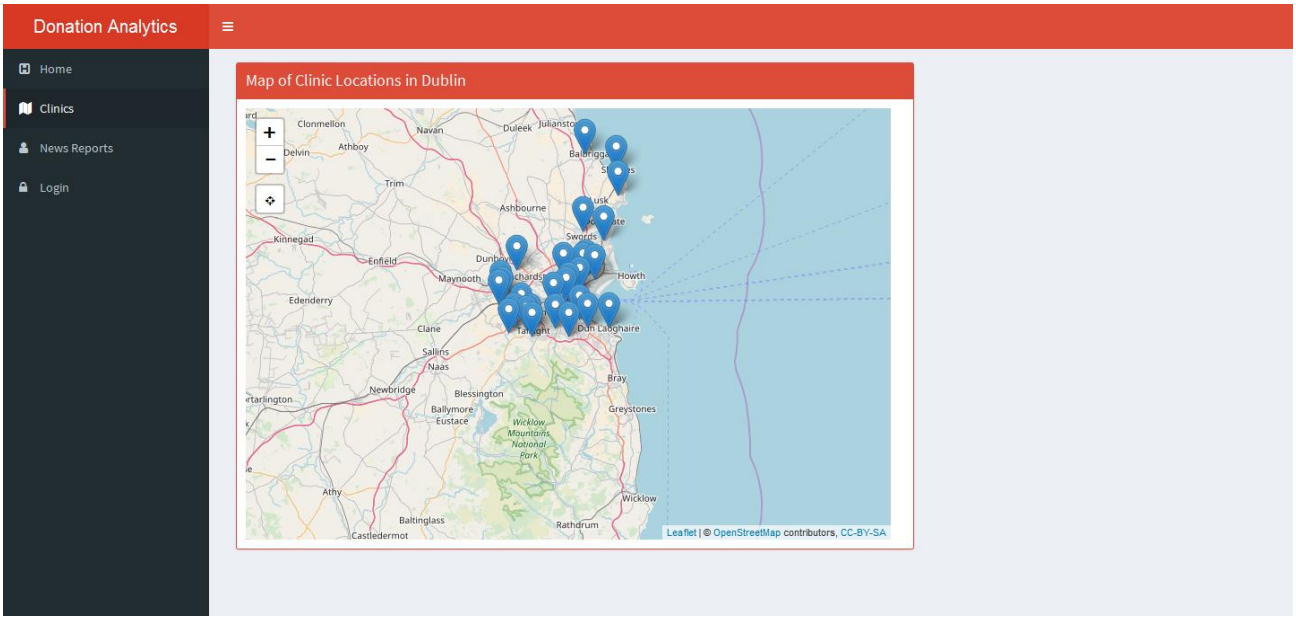


Figure 5.21: The clinics map from the script in Figure 5.20.

The map that can be viewed after the user has logged in is more complicated in terms of the code, as seen in Figure 5.22. However, it doesn't seem much different to the end user. The starting set up of the map is very similar, the map is created using the leaflet function

its tiles are added and the view is set. The next step is making the map interactive, this is done by creating a new variable and inserting the reactive function turning the old map into a reactive expression. The reactive expression will save the result when the application is run. When the expression is called it will check if the previously saved value is no longer up to date which usually depends on whether a widget has been altered. If it needs to be changed the value will be recalculated and the new result will be saved. In the case of this map the widgets used are radio buttons and whichever button is pressed will display a different visualisation on top of the map. If due to some error none of the buttons are selected an empty map will be returned. The map itself is shown in Figure 4.9 in the design section of this report.

```
map = leaflet() %>% addTiles() %>% setView(-6.26, 53.3, 10)
finalMap <- reactive ({
  if(input$radio == 1) return(map %>% addPolylines(data = dubPoly, stroke = TRUE, color = "#444444", fillColor = "red", fill=TRUE, label = ~name,
                              highlightOptions = highlightOptions(color = "white", weight = 2, bringToFront = TRUE)) %>%
                              addEasyButton(easyButton(
                                icon="fa-crosshairs", title="My Location",
                                onClick=JS("function(btn, map){ map.locate({setView: true}); }"))))
  if(input$radio == 2) return (map %>% addMarkers(data = Clinics, ~long, ~lat, popup = ~donations, label = ~name) %>%
                                addEasyButton(easyButton(
                                  icon="fa-crosshairs", title="My Location",
                                  onClick=JS("function(btn, map){ map.locate({setView: true}); }"))))
  if(input$radio == 5) return (leaflet(test) %>% addTiles() %>%
                                addCircles(lng = ~Longitude, lat = ~Latitude, weight = 1,
                                           radius = ~sqrt(Population) * 30, stroke = TRUE, color = "#444444", fillColor = "red", fill=TRUE,
                                           label = ~ElectoralDivision) %>%
                                addEasyButton(easyButton(
                                  icon="fa-crosshairs", title="My Location",
                                  onClick=JS("function(btn, map){ map.locate({setView: true}); }")))
                              )
  if(input$radio == 6) return (leaflet(test) %>% addTiles() %>% addCircles(lng = ~Longitude, lat = ~Latitude, weight = 1,
                                           radius = ~sqrt(PopulationAged18to65) * 30, stroke = TRUE, color = "#444444", fillColor = "red", fill=TRUE,
                                           label = ~ElectoralDivision) %>%
                                addEasyButton(easyButton(
                                  icon="fa-crosshairs", title="My Location",
                                  onClick=JS("function(btn, map){ map.locate({setView: true}); }")))
  )
  if(input$radio == 7) return (leaflet(test) %>% addTiles() %>% addCircles(lng = ~Longitude, lat = ~Latitude, weight = 1,radius = ~sqrt(NumberInGoodHealth) * 30,
                                           stroke = TRUE, color = "#444444", fillColor = "red", fill=TRUE, label = ~ElectoralDivision) %>%
                                addEasyButton(easyButton(
                                  icon="fa-crosshairs", title="My Location",
                                  onClick=JS("function(btn, map){ map.locate({setView: true}); }")))
  )
  if(input$radio == 8) return (leaflet(test) %>% addTiles() %>%addCircles(lng = ~Longitude, lat = ~Latitude, weight = 1,radius = ~sqrt(NumberInGoodHealth) * 5,
                             stroke = TRUE, color = "#444444", fillColor = "red", fill=TRUE, label = ~ElectoralDivision) %>%
                                addEasyButton(easyButton(
                                  icon="fa-crosshairs", title="My Location",
                                  onClick=JS("function(btn, map){ map.locate({setView: true}); }")))
  )
  else return (map)
})
output$myMap = renderLeaflet(finalMap())
```

Figure 5.22: The code used to create the interactive map.

The rest of the graphics and plots are separated into the pages of the application called blood stocks and donation data. For these visuals the code doesn't look as complex more effort was put into how the charts would look before they were coded. In Figure 5.23 the code used to create four different visuals for the blood stocks page. The first three of these graphs use data that was hard coded into the application with the remaining using data imported from the excel files. All the visuals were created using the ggplot2 package. A number of packages were tried including was plotly, which comes with interactive features for graphics such as zoom and a way of saving the graphics as images. However, these features couldn't be removed so the package was not included in this project. When creating graphics using ggplot2 data types must be categorized as discrete or continuous. Graphs coded to use data that the selected plot did not support could cause the application to either crash or show a blank graph. The different types of graphs created

for this application are bar charts, point plots, jitter plots, text plots, trellis plots by using the facet grid function and filled bar charts, as shown in Figure 5.24 and 5.25.

```
output$bloodStocks <- renderPlot(
  ggplot(data, aes(x=BloodType, y=StockInDays)) + geom_bar(color="black", fill=rgb(0.9,0.4,0.3,0.7), stat = "identity")
)
output$bloodTypeFreq <- renderPlot(
  ggplot(data2, aes(x=BloodType, y=Frequency)) + geom_bar(color="black", fill=rgb(0.9,0.4,0.3,0.7), stat = "identity")
)
output$bloodStocks2<- renderPlot(
  ggplot(data, aes(x=BloodType, y=StockInDays)) + geom_point()
)
stocks <- ggplot(Stocks, aes(x=BloodType, y=Stock)) + geom_bar(color="black", fill=rgb(0.9,0.4,0.3,0.7), stat = "identit
output$bloodStocks3 <- renderPlot(
  stocks + facet_grid(. ~ Date)
)
```

Figure 5.23: The code used to create four graphs using ggplot2.

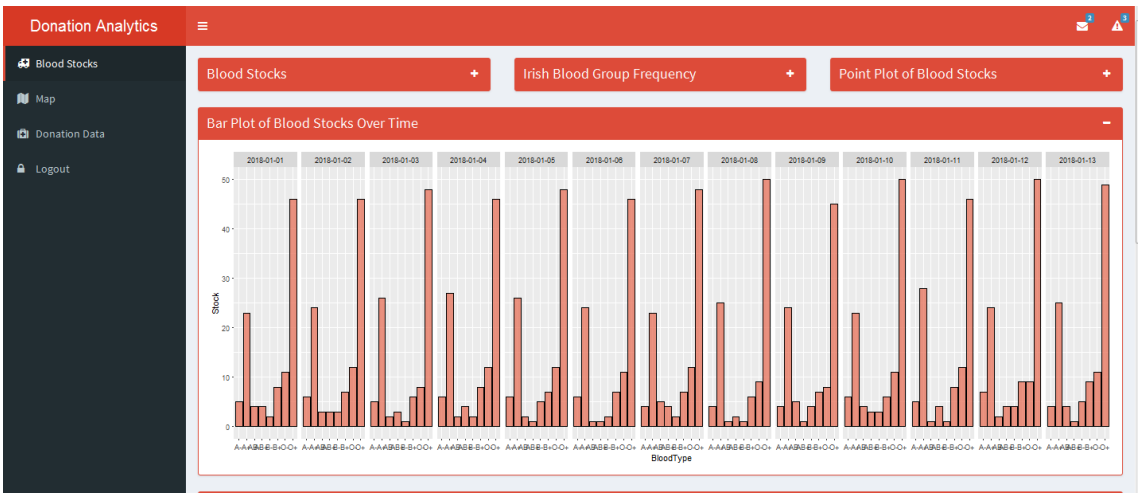


Figure 5.24: A bar plot of blood stocks over time from the blood stocks page.

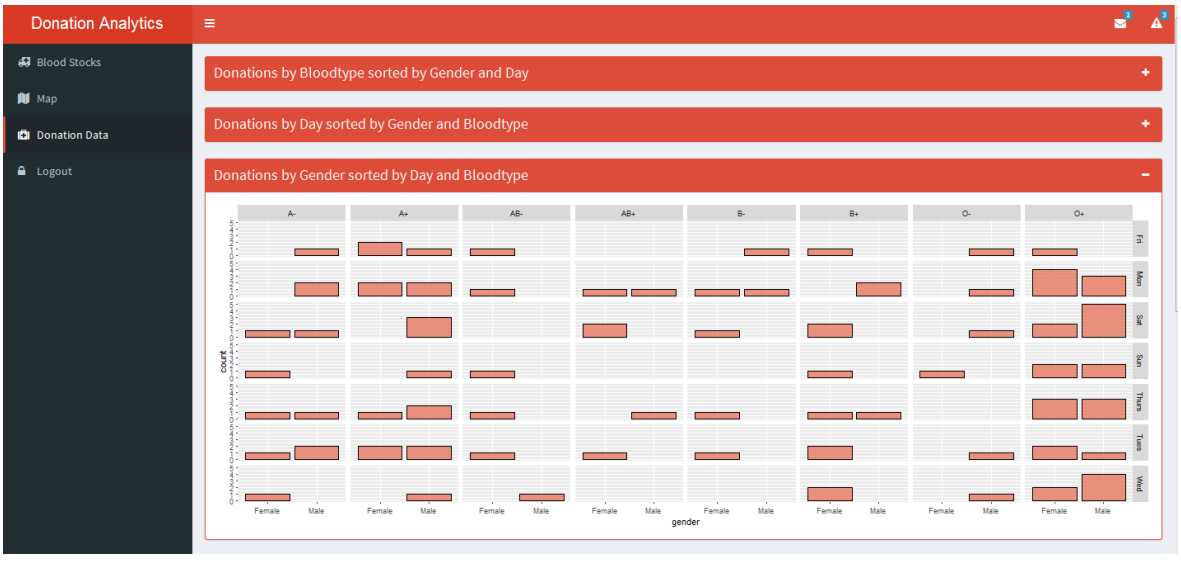


Figure 5.25: A bar plot of donations by gender sorted by day and bloodtype from the donation data page.

6. Testing and Results

6.1 UNIT TESTING

Each different piece of functionality in the application was first built inside its own RStudio project. The main projects made were for basic shiny, building a leaflet map, logging in, connecting a database, registration, using shiny dashboard and then a combined project when all parts were fused together.

When building the basic shiny application two different methods were used. The first attempt was creating the application with the UI and server variables as separate R files. At this stage of development there wasn't much code and having two separate files instead of one combined file seemed unnecessary. The app consisted of a few widgets, some html content and a simple graph.

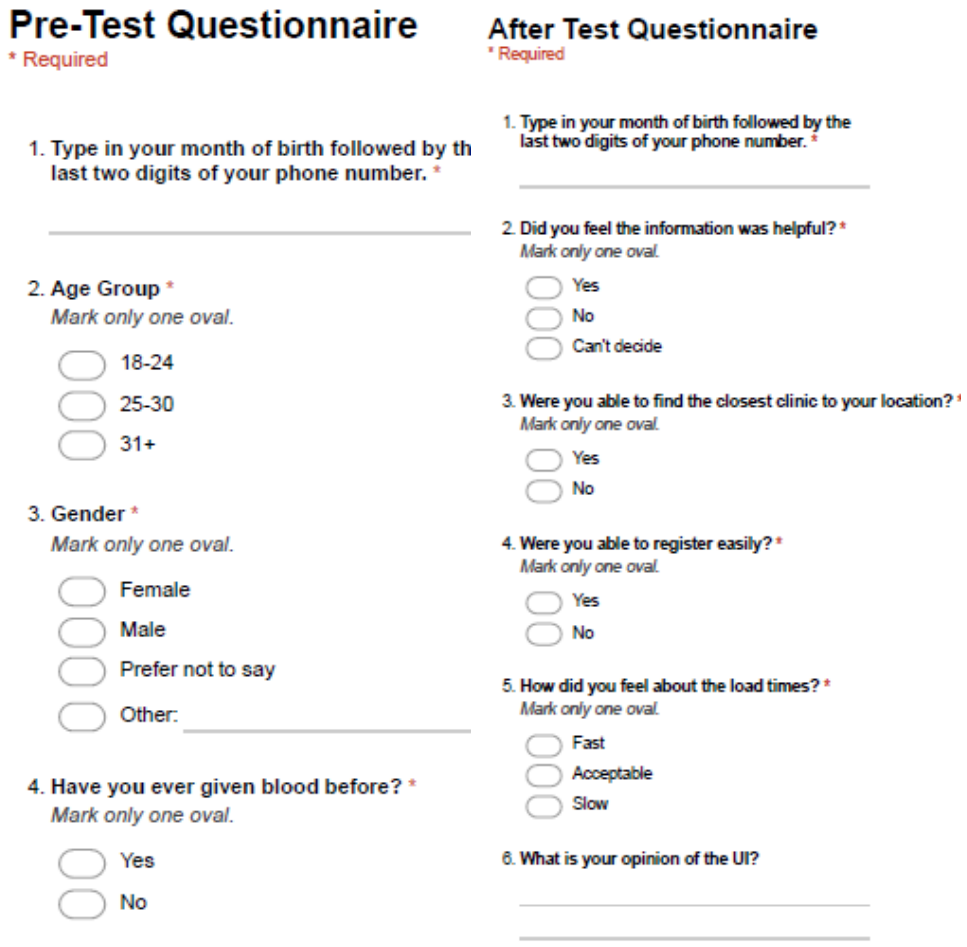
The leaflet map was built in stages, the first stage was creating a plain map and setting its view to Dublin. The JSON files were then imported into the project, this data was then used to layer polylines and markers on top of the map. The projects were combined and the shiny widgets were now used to interact with the map.

The login, database and registration projects were completed around the same time. The login was originally created to work with a local file. Luckily the code didn't need to be changed much with the introduction of a database. Multiple database packages were tried but the RMariaDB and the DBI packages were used. The main difficulty in coding the registration screen was getting the database to take the insert query. This problem was solved by adding two variables and using the *sprintf* function.

After all of the separate projects were combined and working, the big issue was how it looked. The plan at this point was to try and find a package that could be used to improve the application's look. The shiny dashboard package was discovered and another RStudio project was created to understand how to use it. A final project was then created and each of the separate pieces was brought into the project to create the finished application.

6.2 USER TESTING

The user testing done for this application was based around the user interface and the registration/login process. The users' responses were split up by gender, age groups and by if they had given blood before. The users were asked to fill out two questionnaires one before testing, shown in Figure 6.1, and one afterwards. The users were first instructed to get a feel of the application. After the user was comfortable they were asked to find the closest clinic to their location. Most users were able to go to the clinics section un aided and quickly found the closest clinic. The users were then asked to register. Some users got confused because the sidebar only had a login option. After getting to the registration menu users were given the admin code and had no trouble creating an account. The users were then allowed to explore the other side of the application. When they had finished exploring they were asked to complete the second questionnaire, Figure 6.2.



Pre-Test Questionnaire

* Required

1. Type in your month of birth followed by th last two digits of your phone number. *

2. Age Group *
Mark only one oval.
- 18-24
- 25-30
- 31+

3. Gender *
Mark only one oval.
- Female
- Male
- Prefer not to say
- Other:

4. Have you ever given blood before? *
Mark only one oval.
- Yes
- No

After Test Questionnaire

* Required

1. Type in your month of birth followed by the last two digits of your phone number. *

2. Did you feel the information was helpful? *
Mark only one oval.
- Yes
- No
- Can't decide

3. Were you able to find the closest clinic to your location? *
Mark only one oval.
- Yes
- No

4. Were you able to register easily? *
Mark only one oval.
- Yes
- No

5. How did you feel about the load times? *
Mark only one oval.
- Fast
- Acceptable
- Slow

6. What is your opinion of the UI?

Figure 6.1 and 6.2: The pre-test and after test questionnaires.

7. Discussion and Conclusion

The aim of this project was to create an interactive data visualisation application with a focus on blood donations that could be used in an actual blood bank. The applications planned functionality changed significantly from the initial idea to the end result. The functionality was changed multiple times along with the user roles. In the original idea active blood donors would have been able to use the application to keep track of their personnel statistics among other things. This idea wasn't kept because it could be seen as an invasion of privacy.

As mentioned before an attempt was made to acquire any information or data from the Irish Blood Transfusion Service. When no progress was made after the initial response the decision was made to use dummy data for parts of the project.

As discussed already in the testing section of the report each piece of functionality was built inside its own project. Backups were kept when new code was added into a project or when the projects were combined. The results of the user tests showed that the users were able to achieve the goals easily, finding their closest clinic and registering as an admin. However, the majority users thought the load times for the graphs were unsatisfactory. The result is unfortunate but it's not surprising, the reason the user was asked about the loading times because of existing concerns.

8. References

Bostock, M. Ogievetsky, V. and Heer, J. (2011) D3: Data-Driven Documents

Burigat S. and Chittaro L. (2008) Geographic Data Visualization on Mobile Devices for User's Navigation and Decision Support Activities

Dix A. Finlay J. Abowd G. Beale R. (2005) Human-Computer Interaction

Dupin, C. (1826). Carte figurativ de l'instruction populaire de la France.

Friendly, M. (2002) Visions and Re-Visions of Charles Joseph Minard

Friendly, M. (2008) A brief history of data visualisation

Jeng J. (2005) Usability Assessment of Academic Digital Libraries: Effectiveness, Efficiency, Satisfaction, and Learnability

Lowry PB. Spaulding T. Wells T. Moody G. Moffit K. Madariaga S. (2006) A theoretical model and empirical results linking website interactivity and usability satisfaction

Nielsen, J. (1993) Usability Engineering

Nightingale, F. (1857). Mortality of the British Army. London: Harrison and Sons

Pearrow, M. (2000) Web Site Usability

Peng W. Ward M. Rundensteiner, E. (2004) Clutter Reduction in Multi-Dimensional Data Visualisation Using Dimension Reordering

Playfair, W. (1821). Letter on our agricultural distresses, their causes and remedies; accompanied with tables and copperplate charts shewing and comparing the prices of wheat, bread and labour, from 1565 to 1821.

Saito T. Miyamura H. Yamamoto M. Saito H. Hoshiya Y. (2005) Two-Tone Pseudo Coloring: Compact Visualisation for One-Dimensional Data

Shneiderman, B. and Plaisant C. (2010) Designing the user interface: strategies for effective human-computer interaction

Tufte, E. R. (1983). The Visual Display of Quantitative Information. Cheshire, CT: Graphics Press.

Venables, W.N. Smith, D.M. and the R Core Team (2017) An Introduction to R

Wickens, C.D. and Hollands J.G. (2000) Engineering Psychology and Human Performance

Zhu N. (2013) Data Visualisation with D3.js Cookbook