Patient Handling System: Final Report

[1 Introduction 3](#_Toc447750226)

[1.1 Current Practices 3](#_Toc447750227)

[1.2 Automating the Process 4](#_Toc447750228)

[2 Technical Review, Development Process and Previous Work 5](#_Toc447750229)

[2.1 Source Control 5](#_Toc447750230)

[2.1.1 Benefits 5](#_Toc447750231)

[2.2 Application Architecture 5](#_Toc447750232)

[2.2.1 Facility Architecture Possibilities 6](#_Toc447750233)

[2.3 Product Backlog 6](#_Toc447750234)

[2.4 Weekly Meetings 6](#_Toc447750235)

[2.5 Previous Solutions 7](#_Toc447750236)

[2.5.1 Impact on Technology Choice 7](#_Toc447750237)

[3 Technical Issues 8](#_Toc447750238)

[3.1 Database Design 8](#_Toc447750239)

[3.1.1 Webpages Tables 8](#_Toc447750240)

[3.1.2 Decision Tree Storage 9](#_Toc447750241)

[3.2 Server Instances 9](#_Toc447750242)

[3.3 Decision Tree Creator 10](#_Toc447750243)

[3.3.1 Displaying the Trees 10](#_Toc447750244)

[3.3.2 Updating the Tree 11](#_Toc447750245)

[3.3.3 Entering Solutions 11](#_Toc447750246)

[3.4 User Accounts 13](#_Toc447750247)

[3.5 Unit Tests 14](#_Toc447750248)

[3.5.1 The Mocking Framework 14](#_Toc447750249)

[3.5.2 Unit Test Integration 14](#_Toc447750250)

[3.6 Deployment Issues 15](#_Toc447750251)

[3.6.1 Initial Deployment 15](#_Toc447750252)

[3.6.2 Move to Azure 15](#_Toc447750253)

# Introduction

Handling patients who have mobility problems, is not surprisingly, an intricate task. It involves the consideration of a number of different variables, and the use of many different movement tasks, with different processes required for each. The patients themselves, for one thing, can vary significantly in a number of ways, such as in weight, co-operation level, weight bearing capacity, whether they are injured or not, and more. Other factors that also need to be included are the equipment available to the patient handler, and the features of each one. With different movement tasks, the status of some or all of these attributes need to be included in the final action taken to move the patient. Neglecting any could result in poor outcomes, ranging from injury to the patient or patient handler, to undue stress for both parties.

## Current Practices

To aid patient handlers with the above process, a series of algorithms were designed by [Name of company?], which guide the patient handler through an algorithm, in which the path taken through the algorithm, and the eventual solution, or handling plan generated by the algorithm is based on the value of various different patient and equipment attributes. Figure 1.1 shows an example of such an algorithm.

Image of Algorithm 4

Figure 1.1 – A sample patient handling algorithm

In total, there are sixteen algorithms that a patient handler has to either learn off or have on their person at all times when on duty. They also may need to have on hand the various patient/equipment attributes, such as weight, co-operation level, etc. in order to iterate through the algorithm. This results in a process that, if due care was not taken, could produce many errors - for example if the patient handler was under time constraints and guessed some of the patients attributes, or if they read a logical operator incorrectly. The purpose of creating an application for this process is to avoid these errors, which would lead to a better quality of service, and ultimately more satisfied patients and patient handlers.

## Automating the Process

With all of the above in mind, it is clear that there is a need for an application that would aid in this process. This application could be beneficial in different ways, with the main one being that the application would iterate through each algorithm and could produce handling plans for a set of patients. How this is integrated into the patient handler’s day to day job could vary, however a likely scenario is that they would print off the required handling plans for each patient, and store them in an easily accessible place next to the patient. Other implementations would also be possible, such as using a tablet to access the handling plan for a patient. This would be a more expensive option however, but could be more effective if the patient or equipment attributes changed regularly.

This information led to a decision that a web application would be the best type of implementation for these particular requirements. Given the sensitive data that may be stored in this application it would be best if the handling facility had its own internal network where application could be hosted. However, this may not be possible in some facilities, and in that case, the application and associated database could then be hosted on cloud based services, such as Microsoft Azure, which, on a basic subscription plan, consisting of a web application and a database, would cost €600 annually. The web application itself would store all of the patients’ details, along with the various handling plans, which would both be editable.

# Technical Review, Development Process and Previous Work

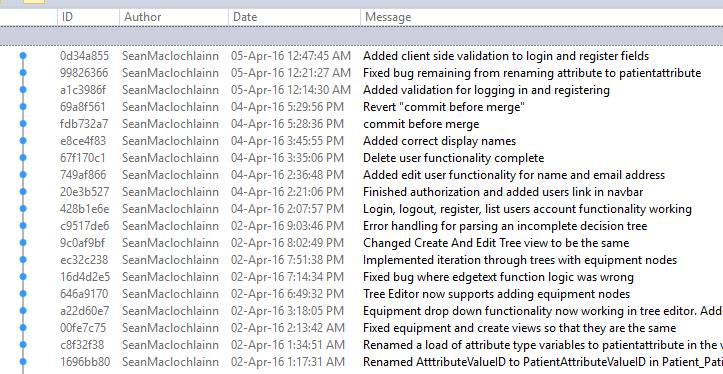
This section will expand upon the technologies suggested during the project definition document, and will detail which of these were chosen. It will also describe the development process used during the completion of this project.

## Source Control

The source control eventually chosen was git, which was then hosted on GitHub. A visual studio extension for GitHub was used to connect to the repository and complete all source control operations.

### Benefits

Even though there was only one developer working on the application, the use of source control still provided some key benefits to the development process. Firstly, it was very useful to be able to revert changes made to a file. An example this was, when trying different packages for the document editor, changes were made to some of the configuration files, and it was unclear which changes were added from installing the package, and which were there originally. With the benefits of source control however, the file could be reverted back to the previous state before the package was installed, which removed the additional changes made to the file. Secondly, committing each change to source control provided documentation and timestamps for what type of changes were made, and when.



A selection of the recent commits made to the project

## Application Architecture

The application itself is a distrusted system, albeit a relatively simple one currently. It is hosted on a Microsoft Azure server. This then communicates with the database, which is an NUIG server, located on campus. Ideally the application and database would be both on the same network, as this would give an improved response time, however the current solution is still satisfactory. If the application was taken to a further stage, an e-mail server would also need to be communicated with, which would most likely not be within the same network.

### Facility Architecture Possibilities

If the application was to be used by a facility, an ideal scenario would be to host the application on the facilities LAN (Local Area Network). This would be optimal for performance, and also potentially security. However, it would require the maintenance of the servers if there weren’t any on site beforehand. As mentioned previously, an alternate option would be to host the application on the cloud.

## Product Backlog

New features and bugs found in the project were created and found regularly during the project development. These ranged from minor stylistic changes, to major feature requests. A product backlog was maintained to store all of these development tasks in one place. This was in the form of an excel spreadsheet. The details of each feature or bug, mainly the description, priority, and status were all documented within this spreadsheet.



The product backlog for the application

When the product backlog was started to be maintained in a more serious manner, towards the end of the project development span, there was a noticeable improvement in the productivity of the development of the project. It was also much easier to see how close the project was to being completed, by looking at how much work was outstanding.

## Weekly Meetings

Every week or two, a meeting was had with the project supervisor, where feedback was given on any features added since the last meeting, and problems encountered and their potential solutions, were discussed. This helped to keep the application development focused on the important areas, and also ensured that the product was being built in a way in which the end users would be satisfied.

## Previous Solutions

There have been two previous attempts at building an application for the company for this specific process. Both attempts were done by final year students, and while being functioning software, were never brought into a working environment by the company. Having access to the reports for both projects (however, not the actual software itself), the previous work done provided a good resource for the work on this application. Mainly, they provided an excellent introduction to the problem domain. The software developed for this report was quite different than the software developed by the other two developers, so for this reason, the previous work was not as beneficial as could have been hoped.

### Impact on Technology Choice

From analysing the previous work done, it was clear there were some advantages and disadvantages to the approaches taken by both. Ultimately, however, for this application it was decided to go with the proven environment of ASP.NET MVC. This is a popular choice when building applications with many CRUD (Create, Read, Update and Delete) type operations, which make up a significant portion of this one – i.e. the storing and editing of patient, user, and equipment data.

# Technical Issues

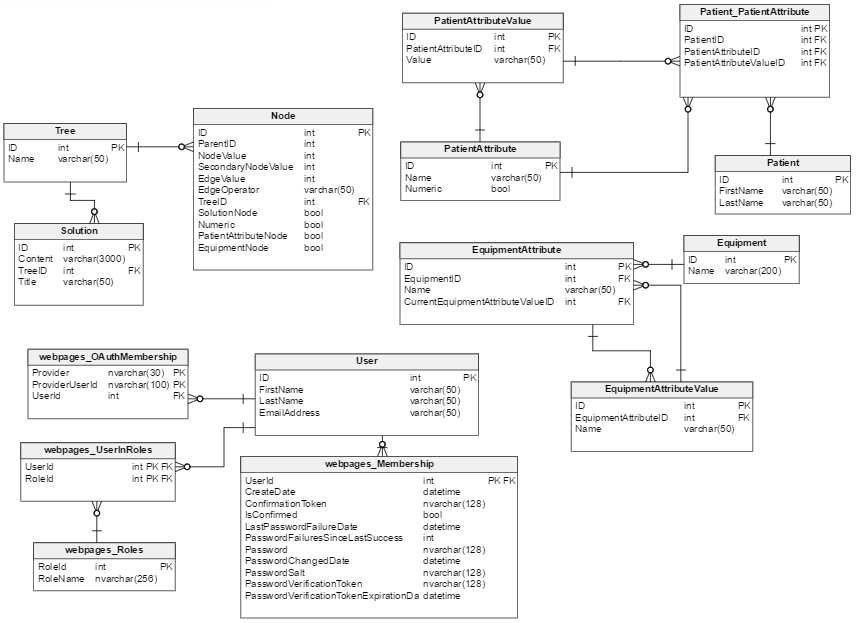
This section will detail some of the more technical aspects of the project that were encountered, and how they were dealt with – ranging from the designing the architecture of the application, to describing how different front end plugins were used to solve various problems.

## Database Design

The goal when designing the database for this application was to create a flexible and easily understood database schema. This was achieved using database normalization techniques, which among other things, eliminated data redundancy, and also ensured that making changes to the schema and performing CRUD operations did not involve much complexity. For example, initially it would seem to make sense to store the attributes for each patient (weight bearing capacity, co-operation level, etc.) in the Patient table. Instead however, they are put in a table of their own (PatientAttribute), making adding new patient attributes at any point a trivial operation.

### Webpages Tables

The tables listed below whose names begin with webpages\_ have been automatically created by a .NET Membership provider called SimpleMembership. This is a tool that helps to streamline the process of integrating user accounts and user access into ASP.NET applications, of which more will be discussed about later. It is also worth noting that the Password attribute of the webpages\_Membership table created by SimpleMembership is encrypted. The process of encrypting and unencrypting these values when they need to be accessed is handled by SimpleMembership.



The entity relationship diagram for the database

### Decision Tree Storage

One of the more complex parts of the database is the Node table, which is essentially where each decision tree is stored. There are many different ways to store decision trees, from storing them in an XML document to, the different relational or graph database models that can be used. After some research into the advantages and disadvantages of each, it was decided to use an adjacency list model to store each tree.

A brief explanation of this model is as follows: Each row in the table represents a node in the tree, and also contains a pointer to some other row that represents the parent node of the current node. Edge values and any other values that the node has are then represented as attributes in the table.

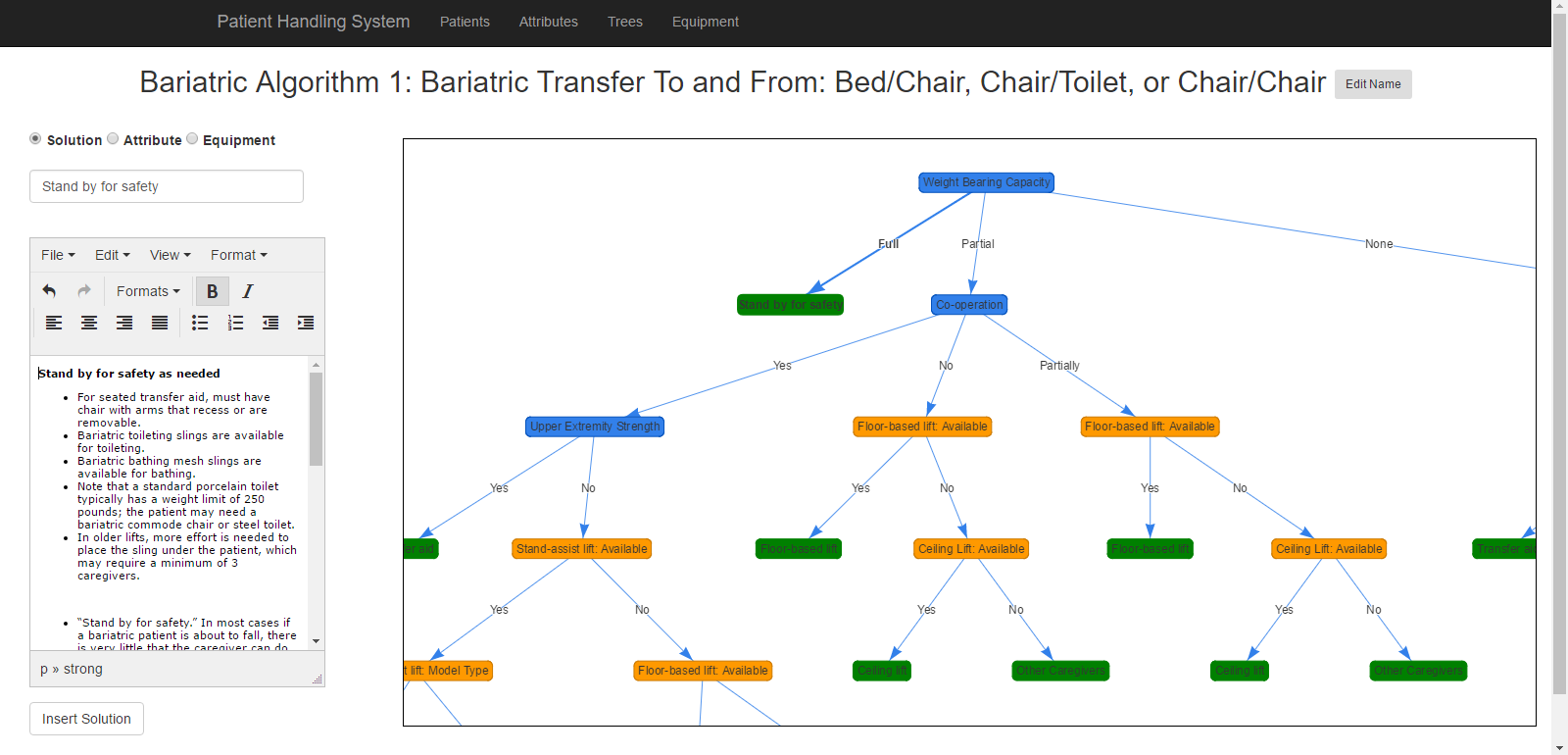
## Server Instances

There were two different database server instances created, one stored locally, on the computer used to develop the application, and one on the NUIG server. The NUIG server was treated as a UAT server, so that anybody wishing to view/test the application during its development could do so. To ensure that both databases performed the same as each other, the schema of each one was scripted, and then these scripts were compared using text comparison software. This ensured that each database was an exact replica of the other.

If the database was to be moved to a server at one of the company’s facilities at some point in the future, a script of either database could be generated and then executed on said server.

## Decision Tree Creator

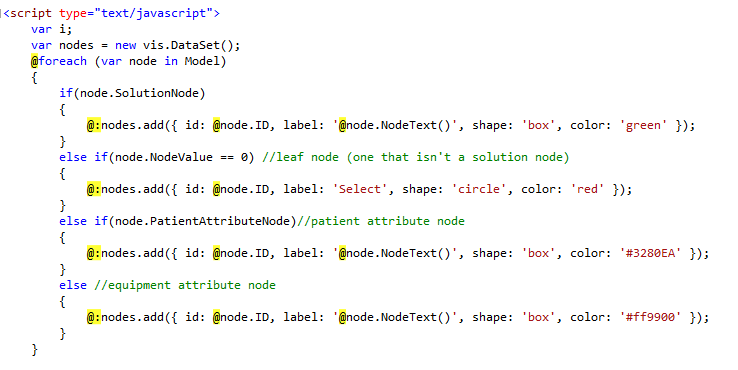
Designing an interface for creating the decision trees was the most technically challenging aspect of this application. Various different JavaScript plugins and other front end technologies were used to achieve this. The goal was to design an interface that was both easy to use and effective.



The decision tree editor

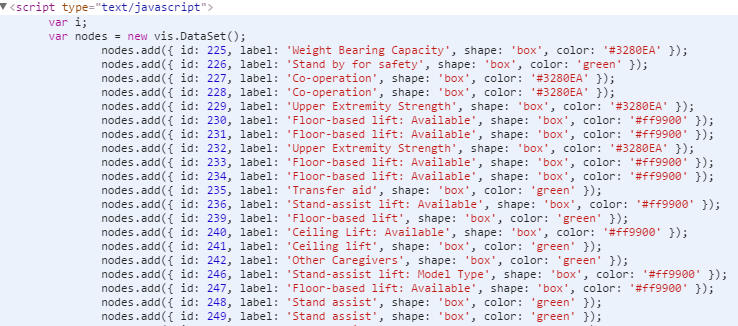
### Displaying the Trees

Research on how to display the trees was done during semester one, where different front end tools were examined too see which was the most suitable to this application. Eventually a decision was made to use Vis.js, a JavaScript visualization library, of which one of its applications is that of displaying decision trees. Trees are created by specifying the nodes, elements and various configuration settings within the JavaScript script eventually executed within the HTML document created.



A mixture of Razor and JavaScript is used to render each tree

To do this a mixture of Razor and JavaScript was used. Razor is a programming syntax used to retrieve data from the model. Generally it is used to render HTML markup, however in this case its use is extended to generate data for the Vis.js script passed to the view. In this case, the model object contains the list of node objects that are to be displayed in the tree. An example of the final output of this process is shown below. The tree edges are also generated in a similar fashion, further down in the script.



This was a surprisingly flexible approach to adding the data to the script; as you can see in [FIRST FIGURE], the shape, colour and other attributes of the node can be specified based on the values of node object.

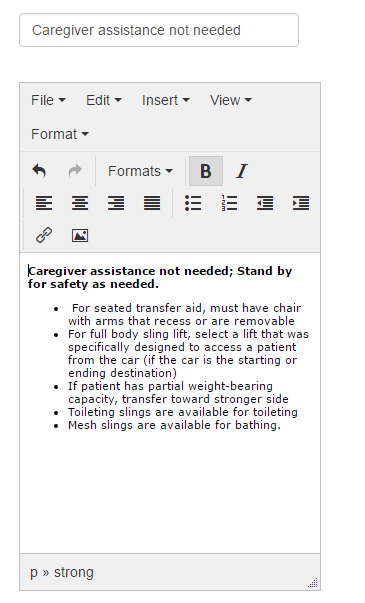
### Updating the Tree

To update the tree, Ajax and partial views were used. Each time the tree was updated, a node and the relevant stub nodes were added to the tree. The parent node for the node to be inserted is specified by an onclick() method that adds a hidden element containing the parent nodes id to the form. The left hand side of the screen acts as a form, containing the details of the node that is to be entered into the tree. [TODO you can add more about adding equipment, about the tabs and how other parts of the form are hidden]

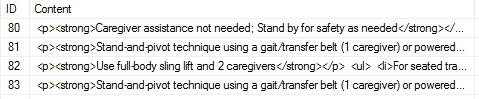
This Ajax POST request is then submitted, the node is added to the tree in the database, and the partial view, which contains the tree is then updated upon this submission. Other features, such as deleting nodes, and error handling (if a non-stub node is selected) are also implemented.

### Entering Solutions

The leaf node on each tree contains a solution, or handling plan. This is a set of instructions for the patient handler, detailing what to do for this specific path. These set of instructions can typically be quite lengthy, and also may require additional images/video content to be added. As such it was decided to use a text editor plugin, TinyMCE. TinyMCE is a JavaScript plugin, designed to create HTML documents, formatted in a similar style to a word document. It is used instead of a textarea element. When a form containing a TinyMCE plugin is submitted to the server, the output will be a HTML document in the format of a string.



The TinyMCE Editor



A handling plan, stored as a string with HTML formatting created by the TinyMCE plugin

It supports many typical document editing functionalities, such as bullet points, italics, numbered lists, tables and much more.

Images and videos are also supported on TinyMCE. These are displayed by using <img> and <iframe> tags, which can be seen by inspecting the HTML markup. This means that all that is required is a source for these images and videos. This could be a local file, stored on the server itself, or else a link to an image/video hosting website. Either method would be a valid approach. Image and video hosting websites also usually support private uploads, where the content is not listed anywhere on the site (aside from the content uploaders account), and a direct link is the only way in which the content can be viewed. This could potentially be an option if the web application was used in an actual handling facility.

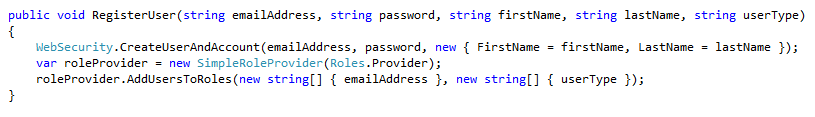


A handling plan containing an embedded image

As in [IMAGE NO] the final format of the handling plan is similar to that of a word document. The patient handlers could then print out this document to have on hand. Using CSS it was possible to add page breaks to this document, thus enabling each handling plan to have its own page.

## User Accounts

User accounts are an important part of many web facing applications, and this one is no exception. Accounts were required for different users to be able to log on, and to provide different users with different levels of access to parts of the application. The addition of user accounts to a web application brings with it many time intensive development overheads, such as registering, logging in, password resetting, and user roles. This would take a lot of time to successfully implement if it was to be built from scratch. Fortunately there are providers that help with this process, and in this application a provider called SimpleMembership was used. SimpleMembership allowed the quick creation of these above functionalities, saving a lot of development time.



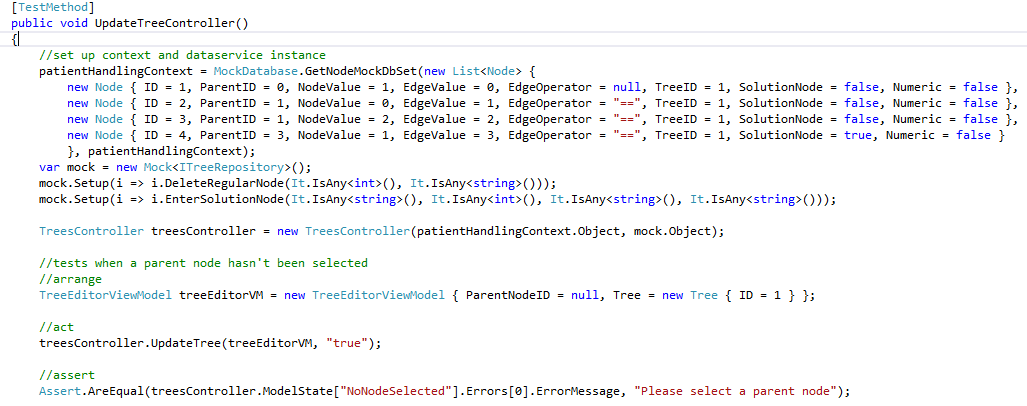
The SimpleMembership provider in action

## Unit Tests

The concept of integrating unit tests and test driven development into the development process of an application is one that is widely used in industry. This is due to a number of reasons; firstly, when any changes are made to the application (including major rewrites that could have effects on other sections of code), it ensures that the change made does not affect the functionality of the rest of the application. Each unit test runs when the application compiles – so any functionality that has been broken will be immediately found. Secondly, they are an effective way of proving that a section of code written does what it was supposed to do. Thirdly, they are a useful way for developers not familiar with the code base to understand what certain sections of are supposed to do.

### The Mocking Framework

Much of unit testing in ASP.NET MVC is performed using a concept called dependency injection. Essentially, a mock repository is created that is a replica of the real database, and unit tests are then performed on this repository. This is done by using constructors in the controller classes, where an instance of the object containing the database tables is passed in. Usually, this object would be the database itself, however when unit testing, this object is mocked, and contains test data specified in the unit test itself. The framework used to accomplish this overall process was called Moq, an open source .NET mocking framework.



Mocking the node database for an error handling unit test

Of course it would have been possible to use an actual database to perform unit tests, however this would mean that the data would not be in main memory, and the tests would take longer to execute, thus slowing down the process.

### Unit Test Integration

Overall it was a useful exercise to integrate the unit tests into the application, a lot was learned during the process, and the groundwork has been built to fully implement test driven development into the project. However, as the process of implementing unit tests was not as successfully applied as had been hoped for. To successfully implement unit tests, a specific type of code layout has to be implemented to all parts of the code base that need to be tested. It is a modular layout, where different parts of the application are separated, allowing for easier testing. For example, the data access layer and the business logic layer are put into two different sections, allowing each to be tested on their own. This process, and how to implement it, was discovered too late into the project for it to be integrated successfully, however if more time was had, this would have been implemented.

## Deployment Issues

Deployment of the application was an important task that had to be undertaken during the development of this application. Successful deployment would mean that the application users, or anybody else concerned, could view the progress made.

### Initial Deployment

Firstly, on deploying to the NUIG IIS (Internet Information Services) server, an error message was received. After researching the error, it was discovered that the .NET version running on the server was version 4.5. This meant that the only applications that could be ran on the server were those using the .NET 4.5 framework. The application was then downgraded to .NET 4.5 (from .NET 4.5.2) and then successfully deployed. Fortunately, this did not have any impact on any of the applications functionality.

### Move to Azure

Towards the end of the application development, the application suddenly stopped deploying to the NUIG IIS server, for an unknown reason. It was determined however, based on the evidence that other applications had also stopped deploying, that it was an issue with the server and not the application itself. A ticket was raised with the IT support team to fix this issue, however at the time of writing it has not been resolved. This meant that the application would have to be deployed onto a different server, or not at all. It was then decided to use Microsoft Azure, a cloud based platform, to deploy the application. Students are able to use this service for free, allowing it to be successfully deployed to Microsoft Azure for no additional cost.

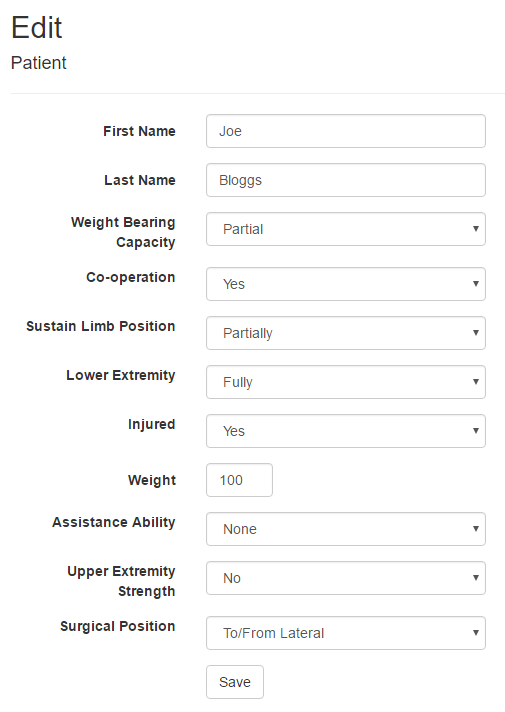
# Results

This section will describe the main features of the application, analyse the work that was expected to be completed versus the work that was completed, and speculate as to how the application would perform in a real life environment.

## Application Overview

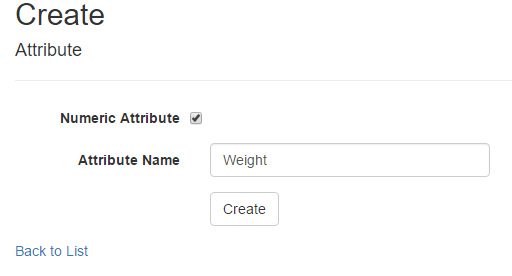
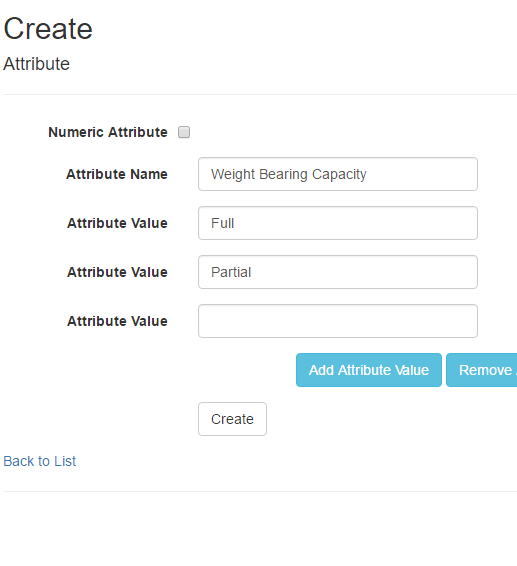
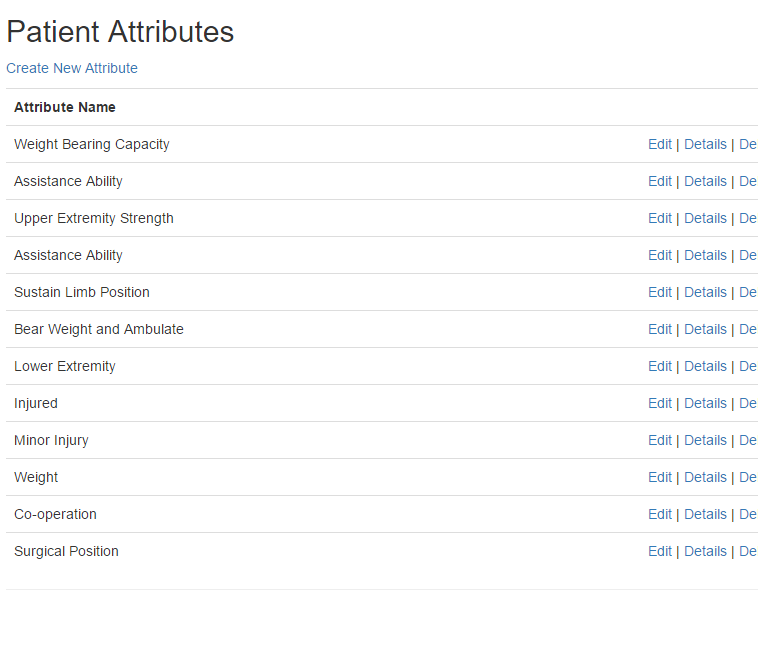
This section will detail each part the application that was developed.

### Patients



Basic CRUD functionality is supported for patients. Patients all have the same amount of attributes (defined in the next section), and a value must be selected for each attribute.

### Attributes



CRUD functionality is also in place for patient attributes. These can have a variable amount of attribute values, or else can be numeric.

