

Cloud Web Services

by Gareth Williams

Submission date: 16-Jan-2024 11:48PM (UTC+0000)

Submission ID: 222299136

File name: 17790_Gareth_Williams_Cloud_Web_Services_536492_603394933.docx (2.92M)

Word count: 4781

Character count: 27017

Assignment 1 & 2: Cloud Based Web Services

Gareth Williams (1910285)

16th January 23:59

Contents

1. Introduction	4
1.1 Scenario	4
2. Web Application - AJAX and REST Coding	6
2.1 Functional Requirements Overview	6
2.1.1 Application Scenario.....	6
2.1.2 Initial Use Cases.....	7
2.1.3 Functional Requirements Catalogue	8
2.1.4 UI/UX Requirements	9
2.1.5 Wireframes.....	9
2.1.6 UI/UX.....	11
2.2 Non-functional Requirements	12
2.2.1 Asynchronous Code – Client Side	12
2.2.2 Automatic Content Updates.....	12
2.2.3 Dynamic Webpage & Responsive Web Design.....	13
2.2.4 CSS Design Decisions	14
2.2.5 Custom Stylesheet.....	14
2.2.6 Development Framework.....	14
2.2.7 Package Management	14
2.2.8 EJS Templates	14
2.3 Story Map	16
2.4 Application Architecture	17
2.4.1 Component Overview	18
2.4.2 Technology	19
2.4.3 Containerisation – Docker	19
3. REST API	20
3.1 Web Framework - Express	21
3.2 REST API demo – Postman.....	22
3.3 Solution Codebase Overview	23
3.4 Patch implementation	23
4. Testing	24
4.1 Application Testing.....	24
4.2 Web Service - REST API Testing	26

4.2.1	API Tests - Chai	26
5.	Conclusion	28
6.	References.....	29
7.	Bibliography	30
	https://caniuse.com/es6-module	30
8.	Appendices.....	31
8.1	Appendix A – Functional Tests Log	31
8.2	Appendix B – REST API Example Tests	32

Figure 1 - Use Case Diagram - Iterations	7
Figure 2 - Initial Sketches	9
Figure 3 - User Journey Wireframe	10
Figure 4 - Initial Encounter Design	10
Figure 5 - Developer Tools Landscape View.....	13
Figure 6 - Developer Tools Portrait View.....	13
Figure 7 - Story Maps	16
Figure 8 - Solution Architecture	18
Figure 9 – Mohanram: REST constraints [6].....	20
Figure 10 - Express Philosophy.....	21
Figure 11 - Testing REST APIs.....	22
Figure 12 - Codebase Folder Structure	23
Figure 13 - Test Case Template [8]	24
Figure 14 - Example Mocha output.....	28

1. Introduction

This document describes the approaches to developing a responsive web application and examines and demonstrates the approaches taken during design and build.

This document is structured in to three main sections:

- Web Application – design and build of the user facing functionality.
- REST API – how back-end services are implemented.
- Testing – how both the application and backend APIs are tested.

Supporting Materials:

A Github repository for the code can be found here:

<https://github.com/Slartibartfast2000/AssignmentClean.git>

Unzip the file or clone and execute docker compose up from the root folder

A video demonstration of the app can be found here:

<https://youtu.be/HJIntlY0t0Y>

Note the database create script will need to be executed in postgresSQL before the app can be used, this can be found in the web-service folder - web-service/src/models/databaseadmin.sql

- The postgres db connection details can be changed in the .env file for the web-service
- The web-server also has its own .env file

1.1 Scenario

The high-level use case is an application for a health organisation to help nurses manage their caseload during nurse-led clinics. The application will allow the nurse to create patient records and manage the nurse-patient encounter by recording conditions and treatments.

The nurse will predominately use mobile and desktop devices so the application requires the ability to run on multiple operating systems and hardware platforms – to meet the flexibility, the application will be web based and accessed via a standard web browser.

The nurses work as a team and can share the caseload, this requires the application to ensure data is refreshed automatically when data is updated by a team member.

2. Web Application- AJAX and REST Coding

The following sections describe the first iteration of requirements and design steps for developing the web application and has the following structure:

- Outlines the main scenario and use-cases
- Functional and UI Requirements – User experiences
- Non-functional Requirements – describes technical approach to the solution

2.1 Functional Requirements Overview

This section provides a brief high-level overview of *use cases* and then breaks down the tasks identified into **functional** and **non-functional** requirements.

Section 2.2 provides a detailed log of the user requirements. These functional requirements will be used as the basis for the application testing.

Finally in this section story maps and wireframes are used to inform the application development and UI/UX development.

2.1.1 Application Scenario

There are three main entities in the solution use case: the nurse, the patient, and the encounter.

The **nurse** will require the ability register a new patient; this adds to the patient to a general pool, the nurse will select the **patient** to add to their specific caseload where they can record the **encounter** details



These clinics tend to treat **patients** who have already been diagnosed with either chronic conditions or require follow-up care following operations. Each time a patient presents at a clinic and meets a nurse is it known as an **encounter**.

The **patient** will be sent appointment details to attend the clinic. Their details will also be passed on to the **nurse** team to add to the patient pool. When the patient **presents** at the clinic, their conditions and treatments provided need to be recorded.



2.1.2 Initial Use Cases

To help elaborate requirements, a number of use cases are mapped out an example is shown in *Figure 1*. These use cases inform the functional requirements catalogue.

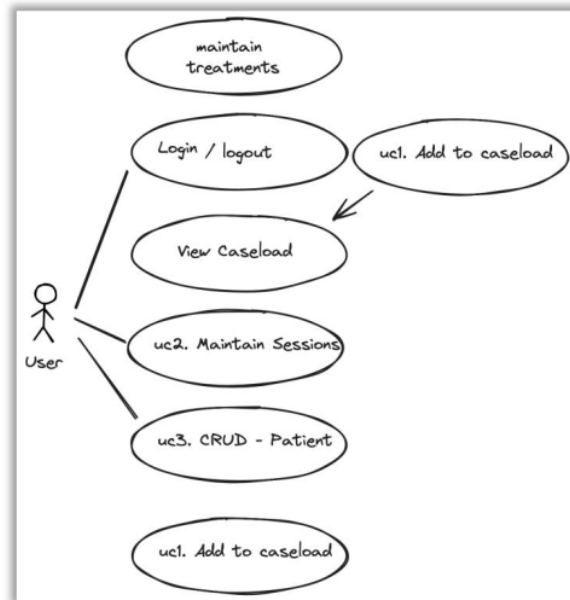


Figure 1 - Use Case Diagram - Iterations

2.1.3 Functional Requirements Catalogue

Table 1 – Requirements Log captures the first iteration of user requirements.

This section layouts a formal requirements table to allow measuring whether the developed application meets the user needs and to inform testing plans.

Table 1 - Requirements Log

No.	Title	Description
R1	View list of patients	
	Record, name, date of birth, address, sex, nhs number for patient	
R2	Amend patient record	
R3	Delete patient record	
R4	Assign patient to my caseload	
R5	Remove patient from my caseload	
R6	View and manage my caseload	
R7	View and manage other clinicians caseloads	
R8	Add Encounter Data	Date, Patient,
R9	Record multiple conditions	
R10	Record multiple treatments	
R11	Work on multiple device types	Mobile Device such as tablet
R12	Work quickly	
R13	Friendly error messages	
R14	Know if a patient record has changed	
R15	Search Patients	
R16	Details automatically updated	

These requirements will be mapped to development tasks using the story map approach.

2.1.4 UI/UX Requirements

Although the solution will be web based, user may find that the traditional multi page web design with post backs and page refreshes slightly jarring experience. There are cases where traditional multi-page design work well, such as a one of questionnaire style interface.

In the book SPA Design and Architecture, Emmet lists several reasons for the use of the SPA approach over traditional multipage web sites:

“Renders like a desktop application, but runs in a browser”, “decoupled presentation layer”, “Faster, lightweight transaction payloads”, “Less user wait time”, “Easier code maintenance” [1]

For these reasons, the Single Page Application approach will be taken to the UI and UX design.

2.1.5 Wireframes

2.1.5.1 Initial Sketches

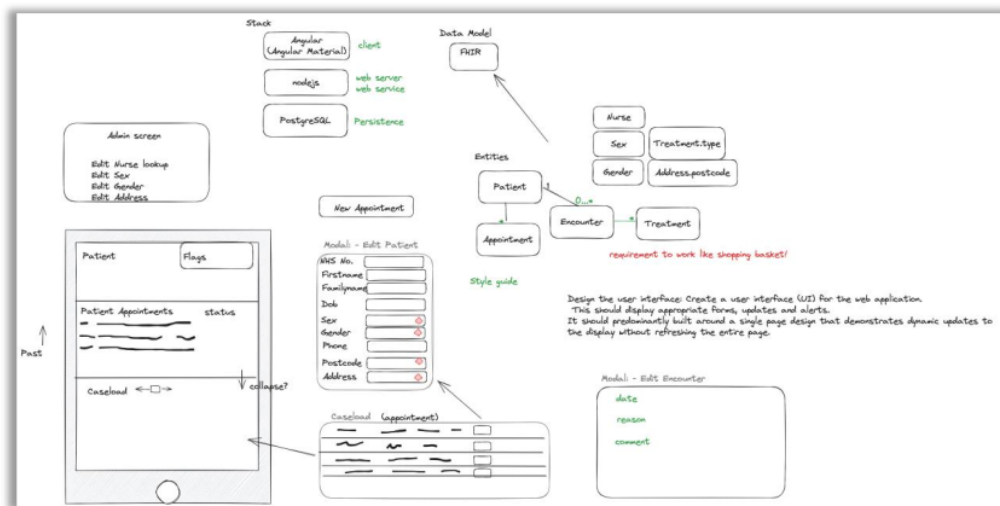


Figure 2 - Initial Sketches

2.1.5.2 User Journey

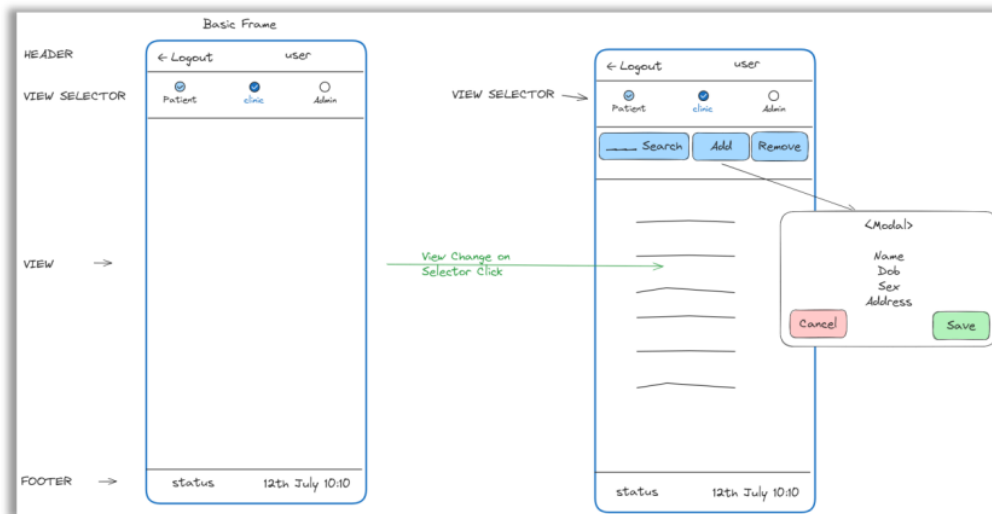


Figure 3 - User Journey Wireframe

2.1.5.3 Encounter

Figure 4 shows a wireframe with the initial design of the Encounter input form.

Figure 4 - Initial Encounter Design

2.1.6 UI/UX

To help with the smooth transitioning server-side rendering will be used to create dynamic content, this content will be loaded using AJAX techniques.

To achieve this EJS (Embedded JavaScript) Framework will be used. Table 2 – EJS and Dynamic HTML show the readability of the EJS code.

Table 2 - EJS and Dynamic HTML

EJS	Hand crafted HTML (patient.js)
<pre> <table id="encountersList" class="table table-striped"> <thead> <th>EncounterDate</th> <th>PatientId</th> <th>Clinician</th> <th>Status</th> </thead> <tbody> <% encounters.forEach(function(encounter) { %> <tr> <td><%= encounter.encounterDate %></td> <td><%= encounter.patientId %></td> <td><%= encounter.nameFamily %>, <%= encounter.nameGiven %> </td> <td><%= encounter.clinicianId %></td> <td><%= encounter.status %></td> </tr> <% %> </tbody> </table> </pre>	<pre> const modalBody = document.getElementById("modal-body"); // html prefix here is a special tag - it helps vscode to format modalBody.innerHTML = ` <div class="container mt-4"> <div class="row"> <div class="col-md-6"> <div class="form-group"> <label style="display: none;" for="id">id:</label> <input type="hidden" class="form-control" id="patientid" name="patientid" value="\${patient?.id ?? ""}" /> </div> <div class="form-group"> <label for="nhNumber">NH Number:</label> <input type="text" class="form-control" id="nhNumber" name="nhNumber" value="\${patient?.nhNumber ?? ""}" required /> </div> <div class="form-group"> <label for="lastName">Family Name:</label> <input type="text" class="form-control" id="familyName" name="familyName" value="\${patient?.nameFamily ?? ""}" required /> </div> </div> </div> </div> ` </pre>

2.2 Non-functional Requirements

This section outlines some of the elements required of the solution architecture.

Why didn't we use REACT or Angular?

Additional framework to understand and support

2.2.1 Asynchronous Code – Client Side

To further meet the responsive requirement the use of asynchronous code will be used. This will be achieved by using *promises*. In the book **JavaScript In Easy Steps**, McGrath explains that: "in JavaScript a Promise object represents the eventual completion, or failure, of an asynchronous operation and its resulting value." [2]

McGrath also explains that the JavaScript engine itself is a *synchronous* language and the ability to carry out asynchronous tasks relies on the web browser API. The *fetch* function is an example of this:

The javascript *fetch* function returns a promise (what is a promise)?

```
function refreshPatientList() {
  displayPatientsList();

  fetch("/getStatus")
    .then((response) => response.json())
    .then((data) => {
      // Update the status on the page with the received data
      document.getElementById("statusElement").innerText = data.status;
    })
    .catch((error) => {
      console.error("Error updating status:", error);
    });
}
```

The above can be simplified to the following by using try catch...



2.2.2 Automatic Content Updates

As the application will be used while treating patients it's essential the data displayed is accurate. Any lists displayed will auto update when the underlying database changes. This will be achieved by implementing simple *timer* and polling techniques to refresh the UI.

In Section 14.1 *Timers* of the book JavaScript the Definitive Guide says of timers:

"seeTimeout() and setInterval() allow you to register a function to be invoked once or repeatedly after a specified amount of time has elapsed" [3]

Using this timer technique and AJAX will allow data to be fetched asynchronously and the UI updated without user interaction.

The main `index.js` file includes the code for this that calls a REST API to refresh the patient list.

2.2.3 Dynamic Webpage & Responsive Web Design

In the book *Responsive Web Design with HTML5 and CSS3*, Ben Frain describes responsive web design as:

“... the presentation of web content in the most relevant format for the viewport and device accessing it.” [4]

Frain goes on to explain that the approach here is to design for the smallest device, in this case the device will be a tablet, and have flexibility for the content to expand.

To achieve this during development, browser developer tools will be used to simulate the screen size of a tablet. An example of this can be seen in figure X.

The browser *developer tools* allow the choosing of the view port size as well as emulating how a touch screen would react to button presses and scrolling.

Most popular web browsers have similar tools. The example here is using Microsoft Edge.

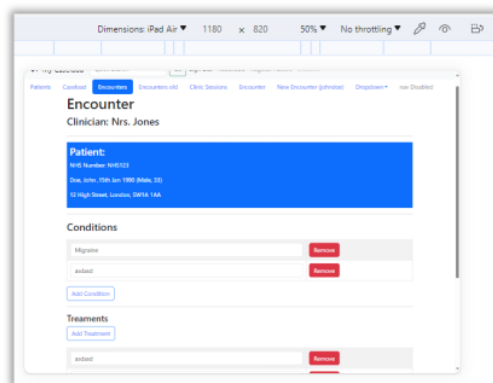


Figure 5 - Developer Tools Landscape View

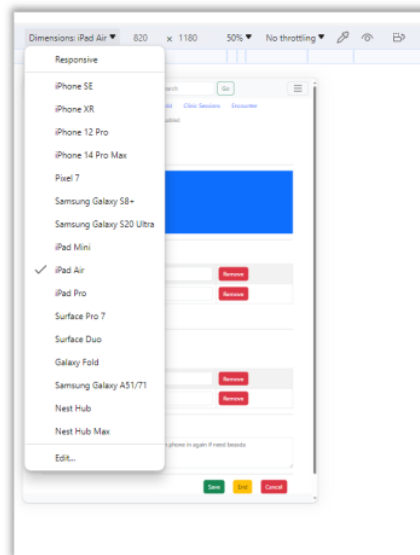


Figure 6 - Developer Tools Portrait View

As well as helping with UI design, the developer tools area of web browsers also help track down client side coding problems.

2.2.4 CSS Design Decisions

To meet the modern and responsive UI requirement, the Bootstrap a responsive and consistent a CSS framework is used. Bootstrap is a commonly used framework and can be manage as an npm package.

<https://getbootstrap.com>

The framework itself can be downloaded or simply referenced, in the case of this application, the software will be hosted behind a corporate firewall, so CDNs will not be used. Also, sending

2.2.5 Custom Stylesheet

The app.css file in the public folder

2.2.6 Development Framework

2.2.7 Package Management

Npm will be used to

2.2.8 EJS Templates

The application uses EJS [EJS -- Embedded JavaScript templates](#)

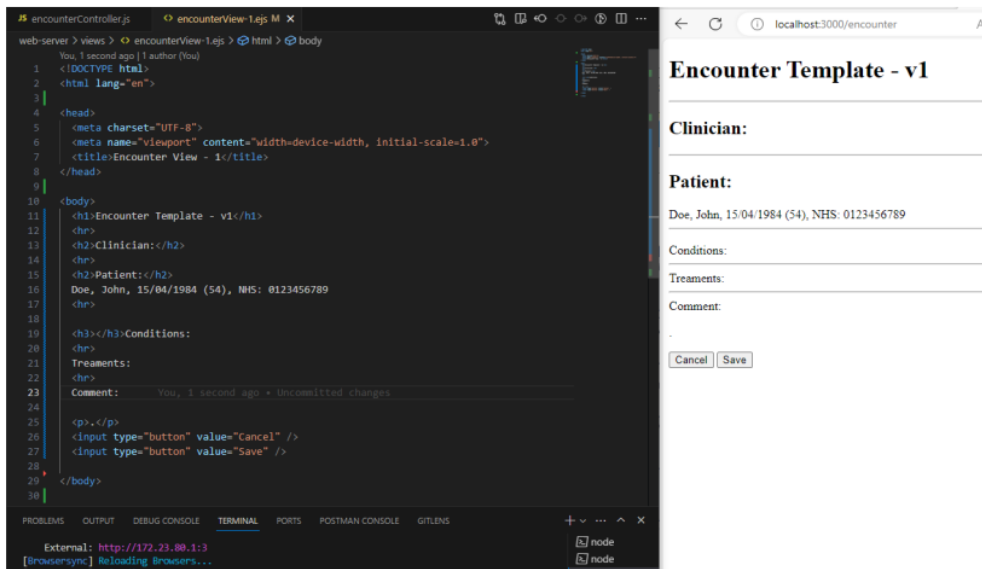
Why?

Content can be developed independently.

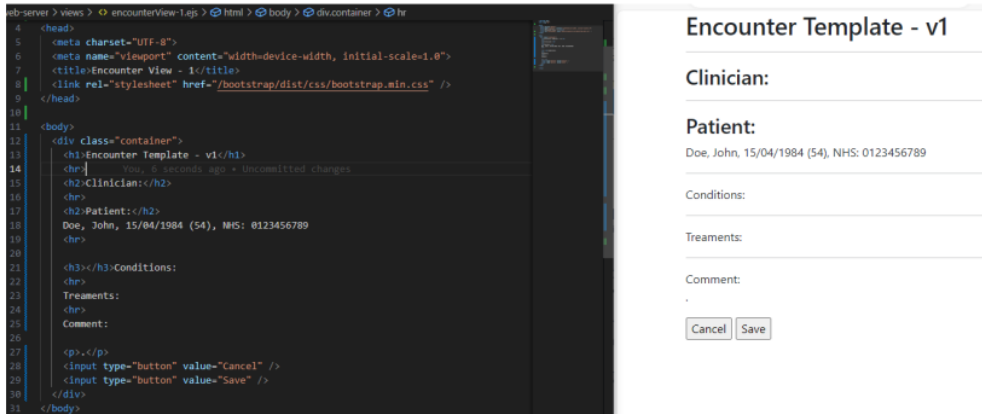
To demonstrate different approaches the patient related content is generated using dynamically HTML and the Encounter content using EJS. The following screen shot shows the difference between the two coding approaches.

Patient pages use dynamic HTML loaded in to the mainContent <div> container

Example:



Simple Bootstrap css



Finally Partial Template

This example is a Single developer EJS allows to scale up team

Composable screens

Partial EJS

2.3 Story Map

Using the first iteration of the requirements a Story Map is build up to help break down and keep track of development tasks.

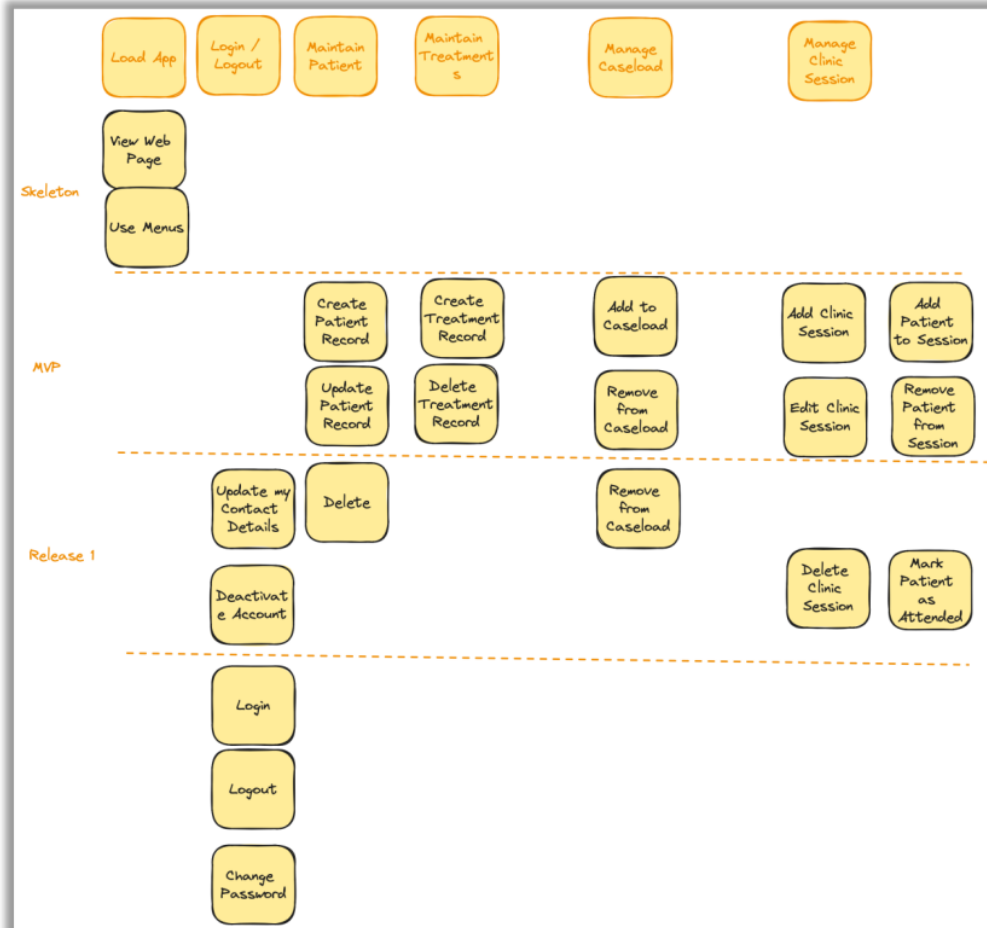


Figure 7 - Story Maps

2.4 Application Architecture

1. **Smooth Transitions:** Implement smooth transitions and animations when updating content with AJAX. This not only makes your application visually appealing but also provides a more polished user experience.
2. **Loading Spinners or Skeleton Screens:** When fetching data from a web service, consider displaying loading spinners or skeleton screens to indicate to the user that something is happening in the background. This gives a sense of responsiveness.
3. **Real-Time Updates:** If your web service supports real-time updates, consider implementing live updates using technologies like WebSockets. This can add an interactive and dynamic element to your application.
4. **Error Handling:** Implement meaningful error handling for AJAX requests. Display user-friendly error messages and consider providing suggestions on how to resolve the issue.
5. **Responsive Design:** Ensure that your application is responsive and works well on various devices. Consider using AJAX to load different sets of data or adjust the UI based on the screen size.
6. **Progressive Enhancement:** Implement progressive enhancement by providing basic functionality for users without JavaScript, and then enhance the experience for users with JavaScript enabled. This ensures accessibility and a wider user base.
7. **User Feedback:** Provide feedback to the user when an AJAX request is successful or encounters an error. This could be in the form of notifications, toast messages, or changes in the UI.
8. **Asynchronous Form Submissions:** Use AJAX to submit forms asynchronously, without refreshing the entire page. Provide visual feedback on successful form submissions **or display validation errors** without reloading the page.
9. **Dynamic Content Loading:** Load additional content dynamically as the user scrolls down the page. This technique is often referred to as infinite scrolling and can improve the user experience by reducing page load times.
10. **User-Centric Design:** Think about how AJAX can improve the overall user experience. For example, use AJAX to create a more interactive and seamless navigation experience, reducing the need for full page reloads.

Why?

Weak typing

Javascript client and server

Jquery = none – bootstrap dependent

Gareth Williams (1910285)

17

Why not react

DOM Manipulation

Promises?

Ajax

<https://nodejs.org/en/download/>

2.4.1 Component Overview

The application architecture is designed to be decoupled REF and therefore will use

The use of REST and AJAX decouples the application stack as they are standards rather than a technology e.g. C# or PHP. The aim is to modularise and make reusable components throughout the stack.

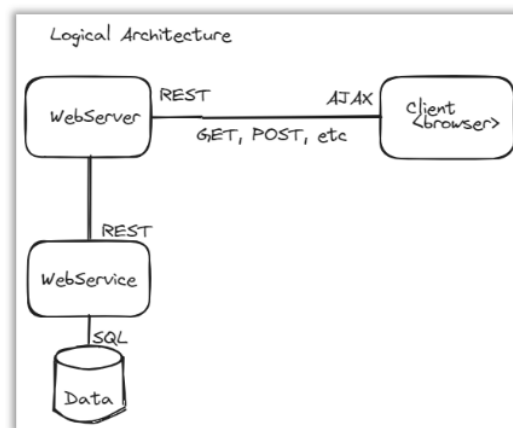


Figure 8 - Solution Architecture

The sketch in Figure 5 shows the four logical components of the solution and how they are related.

- **Client** – this is a web-based application so the client will be a web browser
- **WebServer** – responsible for serving the web pages and interacting with the client based on requests. The webserver will serve both static and dynamic content
- **WebService** – private web service that is essentially APIs to deal with data and database interactions
- **Data** – the persistent store of application data

The client will be classic web browser such as Safari, Edge, Google Chrome to render HTML/CSS and application JavaScript.

The WebServer and WebService will use Nodejs to serve the Application pages and APIs

The database will use PostgreSQL which is a widely supported relational database engine.

2.4.2 Technology

Building a website using node

The solution will use nodejs REF for both the web server and web service

The database will be postgresQL REF

Why use these?

In the introduction to the LinkedIn Learning video, Daniel Khan describes the widespread use of node:

“Node.js is a major platform for the web, used by companies like Uber, Netflix, and Intuit. Express.js is a fast, minimalist framework that sits on top of Node.js and allows you to build powerful single and multi-page web applications and websites” [5]

2.4.3 Containerisation – Docker

In the article, *Containerization, microservices and serverless cloud computing*, Georgios L. Stavrinides and Helen D. Karatza say of containerisation:

“In recent years, container-based virtualization (or containerization) has gained prominence, especially in cloud platforms. A container is a lightweight unit of software that contains all the necessary code, system libraries and runtime environment required to run the application, providing process-level isolation.” [5]

Tools such as Docker allow development of containerised solutions locally that can be easily packaged and deployed to cloud service providers where they can be scaled for large user base.

3. REST API

In chapter one of the book *RESTful Java Web Services - Third Edition*, Mohanram says of the REST approach:

“REST is not an architecture; rather, it is a set of constraints that creates a software architectural style, which can be used for building distributed applications.” [6]

Mohanram then summarises the constraints as shown in *Figure 6*

Client-server: This constraint keeps the client and server loosely coupled.

Stateless: There should be no need for the service to keep user sessions.

Cacheable: This constraint has to support a caching system.

Uniform interface: This constraint indicates a generic interface to manage all the interactions between the client and server in a unified way, which simplifies and decouples the architecture.

Layered system: The server can have multiple layers for implementation. This layered architecture helps to improve scalability by enabling load balancing. It also improves the performance by providing shared caches at different levels. Being the door to the system, the top layer can enforce security policies as well.

Code on demand: This constraint is optional. This constraint indicates that the functionality of the client applications can be extended at runtime by allowing a code download from the server and executing the code. Some examples are the applets and the JavaScript code that get transferred and executed at the client side at runtime.

Figure 9 – Mohanram: REST constraints [6]

To support these constraints, RESTful services use HTTP as it's protocol.

3.1 Web Framework- Express

The npm package *express* provides a simple web server framework, it is ideal for SPA and API applications.

The following caption is take from the express web page, and describes their philosophy:

Philosophy

The Express philosophy is to provide small, robust tooling for HTTP servers, making it a great solution for single page applications, websites, hybrids, or public HTTP APIs.

Figure 10 - Express Philosophy

In the book SPA Design and Architecture, Scott Jr. says of JSON:

“most modern SPAs use AJAX techniques but use JSON as the preferred data exchange format. It’s an ideal format for the SPA because it’s lightweight and compact, and it’s syntax is well-suited for describing object structure.” [1]

In Appendix B of SPA Design and Architecture, Scott JR states that “... AJAX is short for Asynchronuous JavaScript and Xml and generally refers to using JavaScript to dynamically update a web page with the results of an XHR (XMLHttpRequest).” [1]

3.2 REST API demo – Postman

To demonstrate the decoupling of the REST API from the main application ² the screen shot below shows an example POST of a patient record. The example in figure x shows the post and the success response of 201, in addition the API in this case echo's back the record along with the new patient resource id to be used for subsequent actions.

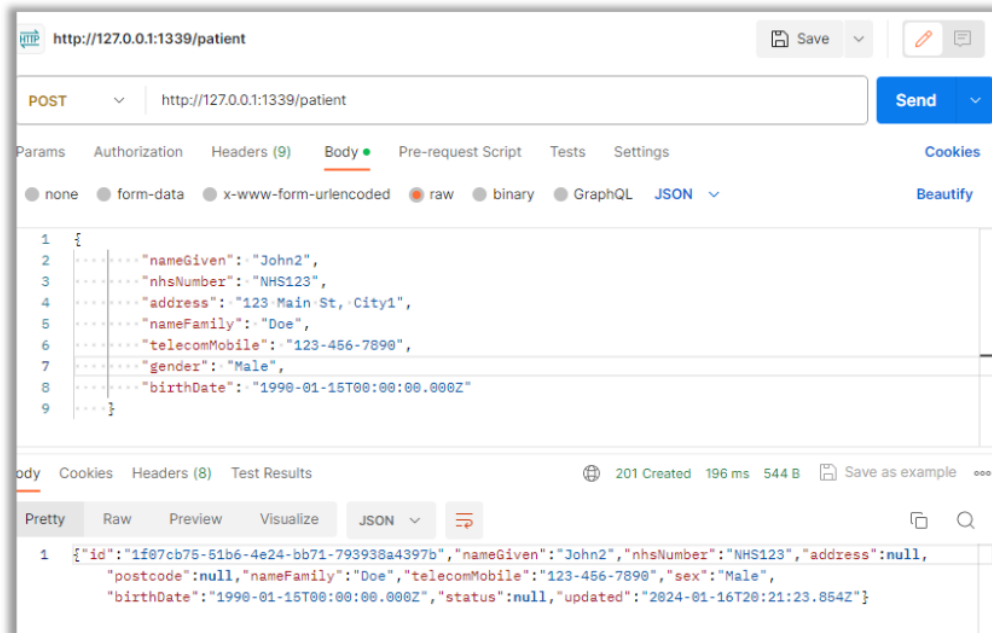


Figure 11 - Testing REST APIs

3.3 Solution Codebase Overview

The main solution is kept in a single folder, this includes the web-server, web-service and docker information

Both the web-server and web-service contain nodejs express based projects.

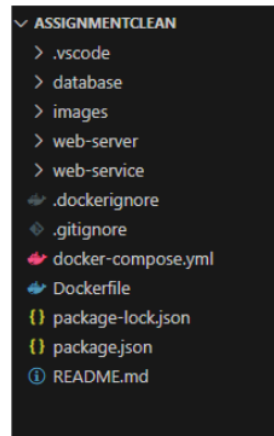


Figure 12 - Codebase Folder Structure

3.4 Patch implementation

The use of PATCH can depend on use case. It is particularly useful when the requirement to maintain changes within a given record are important, for example, knowing which user changed which field. Where this information could be lost with a simple, delete and replace of the whole record.

How PATCH is implemented server side depends on the need for concurrency, for example, a record locking mechanism may need to be in place for a medication app to avoid two users attempting to update at the same time.

Another approach to avoid inconsistent records is to check the state of a record before updating:

- Retrieve current state
- Compare the current state with desired state
- Create patch json
- Send patch request

4. Testing

The testing has been broken down in to two areas:

- Application Testing - Client side / black box testing
- REST API Testing (WebService)

4.1 Application Testing

In the book Software testing an ISTQB-ISEB foundation guide, P Morgan and B Hambling describe “Specification-based (Black-box)” [7] techniques:

“The main thing about specification-based techniques is that they derive test cases directly from the specification or from some other kind of model of what the system should do.” [7]

This section describes the functional testing of the Application. The purpose of the testing documentation will allow the same manual tests to be reproduced as and when a new release or bug fix is made.

The test documentation can be written as part of the requirements process and not retrospectively once a release has been built, this ensures that test plans and steps are in line with requirements.

The ebook Practical-Software-Testing by Chindam Damodar provides an example “test case template” [8] an example of which is shown in *Figure 7*.

TC #	TS#	Test Design/ Steps	Input Data	Expected Result	Actual Result	Pass	Fail	Comments	Critical	Severity Major Minimal	Moderate
1		Launch the browser, Enter the valid URL and click on [GO] or Press [Enter]	www.wspr.com	Should display the WAPR Home Page							
2		Enter valid Customer ID and password Click on [Submit]		Should display the Customer Home Page							
3		Verify Spellings, Fonts, Alignments of Customer Home Page		Should not be any Spelling Mistakes. Should be a same Fonts, same Alignment							
4		Focus on 'Logout'and Press tab Continuously		Tab order should go with all the fields according to their occurrences							
5		Click on 'View Profile'		Should display a View Profile Page							

Figure 13 - Test Case Template [8]

The testing documentation and steps will follow the example in *Figure 7* to describe the test steps for each of the requirements from the functional requirements catalogue.

The following tables *Table 1 – Test Case – launch Application* and *Add Patient* are **samples** of the functional test scripts used, *Appendix A – Functional Test Scripts Log* contains the full list of test cases.

Table 3 - Test Case - Launch Application

Test Case					
Test Id:	1	Test Title:	Launch Application	Date:	23-12-2023 09.30
Test Description:			User opens a web browser and navigates to the homepage		
Test Scope: (Requirement Numbers):					
Preconditions/ Assumptions:			Services are running, user environment and device working		
Test Steps...					
No.	Test Step	Input Data	Expected Result	Actual Result	Pass/Fail
1	Open the browser and input url and press enter	http://localhost:81	Display Homepage (index.html)		
2	Save page as favourite and create desktop icon				
Test Case					
Test Id:	2	Test Title:	Add Patient	Date:	24-12-2023 13.00
Test Description:			Add a new patient record		
Test Scope: (Requirement Numbers):					
Preconditions/ Assumptions:			Services are running, user environment and device working. Home page is loaded		
Test Steps...					
No.	Test Step	Input Data	Expected Result	Actual Result	Pass/Fail
1	Open registration modal	Click the “Register Patient” button on the home page	Register Patient Modal to Appear	As expected	P
2	Enter data	NHS Number: 01234567 Surname: Jones GivenName: Tom DOB: 15/7/2021	Data can be entered and stays in each field	As Expected	P
3	Save and close patient Record	Click “save patient” button	Modal window closes. Newly created record appears at top of patient list	As Expected	P

4.2 Web Service- REST API Testing

As the **web-service** component is essentially a distinct component of the solution, it can be tested independently. In the book Practical DevOps, J.Verona states that:

“Automated testing of backend functionality, such as SOAP and REST endpoints, is normally quite cost effective. Backend interfaces tend to be fairly stable, so the corresponding tests will also require less maintenance effort than GUI tests, for instance.” [9]

The automated testing can perform part of Continuous Integration and Continuous Deployment (CI/CD) pipelines and stop deployment and report on failures to developers.

4.2.1 API Tests - Chai

Jon Peck provides a brief survey of Nodejs testing frameworks and states:

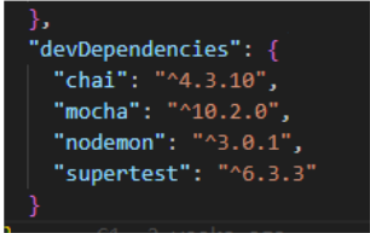
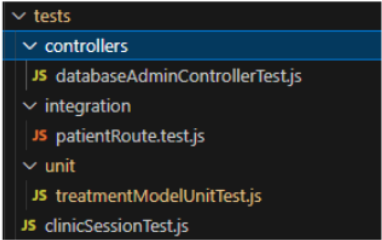
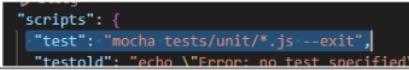
“Some of the most popular Node.js testing frameworks in alphabetical order include, Ava, Jasmine, Jest, Mocha, and tape.” [10]

Following the guidance the solution will use Mocha along with Chai for asserting the tests.

Appendix C shows code for a test of the web-service *patientRoute* REST API.

The following table briefly describes some of the aspects of Mocha and Chai

Table 4 - Mocha and Chai Overview

Description	Example
In this solution the web-service codebase contains <i>Mocha</i> and <i>Chai</i> Chai (chaijs.com) frameworks. These packages work together and are included as <i>devDependencies</i> so they don't become part of the production build.	 <pre>}, "devDependencies": { "chai": "^4.3.10", "mocha": "^10.2.0", "nodemon": "^3.0.1", "supertest": "^6.3.3" }</pre>
The web-service contains a test folder which contains a number of test files that contains small functions that executed buy the frameworks.	 <pre> └─ tests └─ controllers JS databaseAdminControllerTest.js └─ integration JS patientRoute.test.js └─ unit JS treatmentModelUnitTest.js JS clinicSessionTest.js</pre>
Using Node package.json start up scripts can be configured to execute the tests. In this example,	 <pre>"scripts": { "test": "mocha tests/unit/*.js --exit", "testold": "echo \"Error: no test specified\""</pre>

<i>mocha</i> will execute any files found in the tests/unit folder and matching the pattern *.js	
---	--

4.2.1.1 Example Unit Test with Chai

The simple example in figure x, describes a single Chai unit test. In this example the patient route is expected to return a successfully including a valid json string that is expected to be an object.

```
const { expect } = require('chai');
const TreatmentModel = require('../src/models/treatmentModel');

describe('Treatment Model', () => {
  it('Return a list of 10 treatments', async () => {
    const treatmentModel = new TreatmentModel();
    const treatments = await treatmentModel.getTreatments();

    expect(treatments).to.exist;
    expect(treatments).to.have.lengthOf(10);
  });
});
```

For illustration the followig table shows a success, and emulating a failure by searching for a non existent resource id.

<pre>✓ GET: should return a single patient (247ms) 1 passing (252ms)</pre>	
<pre>1 failing 1) patientRoute Tests GET: should return a single patient: Uncaught AssertionError: expected 404 to equal 200 + expected - actual -404 +200</pre>	

The following screen shows the output of the testPatientRoute script that tests the POST, GET, PATCH and DELETE functionality of the REST API.

```
garethwilliams@Gareths-MBP-3 web-service % npm run testPatientRoute
> web-service@1.0.0 testPatientRoute
> mocha tests/integration/patientRoute.test.js --exit

Note: the value of PORT is set in both docker-compose.yml and .env
they MUST match!
[Web-service] listening at http://localhost:1339

patientRoute Tests, POST, GET, PATCH, DELETE
  ✓ POST: should create a new patient (64ms)
  ✓ PATCH: should patch patient record
  ✓ GET: should return a single patient
  ✓ DELETE: should delete a single patient

4 passing (107ms)
garethwilliams@Gareths-MBP-3 web-service %
```

Figure 14 - Example Mocha output

5. Conclusion

Designing and developing a responsive cloud ready web application in Node.js, Express, REST, and EJS demonstrated the power of the frameworks. The use of JavaScript based Node.js and Express demonstrates how quickly and easily a web app can be setup, handling routes, and building APIs.

The use of EJS for templating dynamic pages showed how such tools can simplify dynamic html making it easier to code and support, in addition it increases modularisation and reusability.

Express middleware made the development of REST APIs and rendering of dynamic content a straightforward task.

The use of Docker also helped demonstrate the power of containerisation to help both development and infrastructure to develop and deploy scalable solutions.

The next steps would be to focus on security, for example introducing security tokens to the public facing APIs, such as adding Jason Web Tokens to REST calls. From a front-end perspective, the user interface could be enhanced with tools like React or Vue.js.

6. References

- [1] J. Emmit A. Scott, SPA Design and Architecture, New York: Manning, 2016.
- [2] M. McGrath, JavaScript In Easy Steps, Leamington Spa, Warwickshire, United Kingdom: Easy Steps Ltd, 2020.
- [3] D. Flanigan, Javascript: The Definitive Guide, Sixth Edition, CA: O'Reilly Media, 2011.
- [4] B. Frain, Responsive Web Design with HTML5 and CSS3 Second Edition, Birmingham UK: Packt Publishing Ltd., 2015.
- [5] D. Khan, "building a website with node js and expres js," LinkedIn, 2020. [Online]. Available: <https://www.linkedin.com/learning/building-a-website-with-node-js-and-express-js-3/template-variables-in-more-detail?autoSkip=true&resume=false&u=77966930>. [Accessed 08 12 2023].
- [6] B. M. Balachandar, RESTful Java Web Services - Third Edition : Master Core REST Concepts and Create RESTful Web Services in Java, Birmingham: Packt Publishing Ltd, 2017.
- [7] P. Morgan and B. Hambling, Software Testing an ISTQB-ISEB Foundation Guide 2nd Edition, Swindon: British Informatics Society Ltd., 2010.
- [8] C. Damodar, "Practical Software Testing," 01 01 2024. [Online]. Available: <https://www.softwaretestinghelp.com/practical-software-testing-new-free-ebook-download/>.
- [9] J. Verona, Practical DevOps, Birmingham: Packt, 2018.
- [10] J. Peck, "LinkedIn.com," 06 07 2021. [Online]. Available: <https://www.linkedin.com/learning/instructors/jon-peck?u=77966930>. [Accessed 10 12 2023].

7. Bibliography

S. Deshmukh, D. Mane and A. Retawade, "Building a Single Page Application Web Front-end for E-Learning site," 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC), Erode, India, 2019, pp. 985-987, doi: 10.1109/ICCMC.2019.8819703.

PACKT, P. (Producer), & . (2018). Building a Single Page Application with JavaScript. [Video/DVD] PACKT Publishing. Retrieved from <https://video.alexanderstreet.com/watch/building-a-single-page-application-with-javascript>

V. GAVRILĂ, L. BĂJENARU and C. DOBRE, "Modern Single Page Application Architecture: A Case Study," Studies in Informatics and Control, vol. 28, (2), pp. 231-238, 2019. Available: <https://www.proquest.com/scholarly-journals/modern-single-page-application-architecture-case/docview/2694620086/se-2>. DOI: <https://doi.org/10.24846/v28i2y201911>.

<https://getbootstrap.com>

Node.js (nodejs.org) <https://nodejs.org/en>

<https://www.w3schools.com/nodejs/default.asp>

Docker Desktop - <https://www.docker.com/products/docker-desktop/>

<https://caniuse.com/es6-module>

https://moodle.uwtsd.ac.uk/course/view.php?id=2536 Postgres toilet99 kyrbuw-9sijci-fUnmen – chatgpt password?
Demo Video 1. Introduction and overview 2. What is the app for 3. Build and deploy the application from source code 4. Database postgres 5. Web Services testing = private network 6. Web Server = public facing App
Cloud Based Web Services.pptx
Entity Model Excalidraw User map and wireframes: https://excalidraw.com/#room=6f89464f42c34bb2f7ca,tuBS7SAGugbvRFoOc6DrkA

8. Appendices

8.1 Appendix A – Functional Tests Log

Test Case					
Test Id:	1	Test Title:	Launch Application	Date:	23-12-2023 09.30
Test Description:			User opens a web browser and navigates to the homepage		
Test Scope: (Requirement Numbers):					
Preconditions/ Assumptions:			Services are running, user environment and device working		
Test Steps...					
No.	Test Step	Input Data	Expected Result	Actual Result	Pass/Fail
1	Open the browser and input url and press enter	http://localhost:81	Display Homepage (index.html)		
2	Save page as favourite and create desktop icon				

8.2 Appendix B – REST API Example Tests

These tests can be repeated by executing `# npm run testPatientRoute`

```
const chai = require("chai");
const expect = chai.expect;
const request = require("request");
const app = require("../../service");

describe("patientRoute Tests, POST, GET, PATCH, DELETE ", () => {
  let patientId = "";

  it("POST: should create a new patient", (done) => {
    const patientData = {
      "nameGiven": "John2",
      "nhsNumber": "NHS123",
      "address": "123 Main St City1",
      "nameFamily": "Doe",
      "telecomMobile": "123-456-7890",
      "gender": "Male",
      "birthDate": "1990-01-15T00:00:00.000Z"
    };

    request.post(
      {
        url: "http://localhost:1339/patient",
        json: patientData,
      },
      (err, res, body) => {
        if (err) {
          console.error(err);
          return done(err);
        }

        expect(res.statusCode).to.equal(201);
        expect(body).to.be.an("object");
        patientId = body.id;

        done();
      }
    );
  });

  it("PATCH: should patch patient record", (done) => {
    const patientData = {
      "nameGiven": "John2",
      "nhsNumber": "NHS123",
      "address": "123 Main St City1",
    };
  });
});
```

```

    "nameFamily": "PATCHED",
    "telecomMobile": "123-456-7890",
    "gender": "Male",
    "birthDate": "1990-01-15T00:00:00.000Z"
  }

  request.patch(
    {
      url: `http://localhost:1339/patient/${patientId}`,
      json: patientData,
    },
    (err, res, body) => {
      if (err) {
        console.error(err);
        return done(err);
      }

      expect(res.statusCode).toEqual(200);
      expect(body).toBe.an("object");

      done();
    }
  );
});

it("GET: should return a single patient", (done) => {
  request.get(
    {
      url: `http://localhost:1339/patient/${patientId}`,
    },
    (err, res, body) => {
      if (err) {
        console.error(err);
        return done(err);
      }

      expect(res.statusCode).toEqual(200);
      expect(body).toBe.a("string");

      const parsedData = JSON.parse(body);

      expect(parsedData).toBe.an("object");

      done();
    }
  );
});

```

```

it("DELETE: should delete a single patient", (done) => {
  request.delete(
    {
      url: `http://localhost:1339/patient/${patientId}`
    },
    (err, res, body) => {
      if (err) {
        console.error(err);
        return done(err);
      }

      expect(res.statusCode).to.equal(200);
      expect(body).to.be.a("string");

      const parsedData = JSON.parse(body);
      expect(parsedData).to.be.an("object");

      done();
    }
  );
});
});

```

API Test outcome

```

[Web-service] listening at http://localhost:1339

patientRoute Tests, POST, GET, PATCH, DELETE
✓ POST: should create a new patient (174ms)
✓ PATCH: should patch patient record
✓ GET: should return a single patient (83ms)
✓ DELETE: should delete a single patient

4 passing (286ms)

PS C:\inchware\vscode\AssignmentClean\AssignmentClean\web-service>

```

Cloud Web Services

GRADEMARK REPORT

FINAL GRADE

91 / 100

GENERAL COMMENTS

Assignment 1:

=====

Rest & Ajax (45): 41

Requirements (15): 15

Design & Docs (30): 30

Demo (10): 6

Overall: **92%**

Assignment 2:

=====

Rest API (40): 40

Patch (10): 9.5

Test doc (30): 24.5

Testing (10): 10

Demo (10): 6

Overall: **90%**

Feedback:

=====

You could have added a little more explanation with the requirements, but they are clear anyway.

Really liked the design and how you have presented it.

Application is quite concise but performs all the REST operations using AJAX as required.

I was hoping for a little more documentation of tests, but there are a good number of tests in the code. I have been a bit generous with the mark because you have some good software testing.

The demo was a bit short, I was anticipating some discussion of how the application worked.

Very well structure application code, with controllers, routers and models.

Excellent

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12



Comment 1

I think you are missing a bit of code here!!

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22



Comment 2

figure 11

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34
