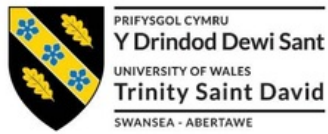


Testing Node.js Restful Web Services

by James Davies

FILE	17860_JAMES_DAVIES_TESTING_NODE.JS_RESTFUL_WEB_SERVICES_78196_288021497.DOCX (1.98M)		
TIME SUBMITTED	21-APR-2016 03:50PM	WORD COUNT	13335
SUBMISSION ID	56214046	CHARACTER COUNT	75025



Testing Node.js RESTful Web Services/Applications

By James Davies

Supervisor: Mr Michael Dacey

Second Marker: Dr Kemi Ademoye

Project submitted as part of the requirements for the award of
BSc(Hons) Web Development

April 2016

Declaration

I, ... declare that I am the sole author of this Project; that all references cited have been consulted; that I have conducted all work of which this is a record, and that the finished work lies within the prescribed word limits.

This work has not previously been accepted as part of any other degree submission.

Signed:

Date:

FORM OF CONSENT

I hereby consent that my Project, submitted in candidature for the B.Sc.(Hons) Web Development degree, if successful, may be made available for inter-library loan or photocopying (subject to the law of copyright), and that the title and abstract may be made available to outside organisations.

Signed :

Date :

Abstract

A Microservice is a small service with a limited range of functionality that can be built to an API (Application Protocol Interface). For example, it could be the calculation of an order or the maintaining of a set of customers' details. To deliver these types of services, developers use different tools such as Node.js, a JavaScript framework and MongoDB, a NoSQL database which the author has no previous experience of. The aim of this project is to evaluate the testing approaches for the development of Node.js Microservices. The objectives set were to analyse the techniques and technologies used in the development of Node.js Microservices, and to identify testing approaches and to apply appropriate tests and test frameworks during construction. Having had some practical experience an evaluation was made of the testing techniques.

An agile approach was chosen for the project using user stories and time boxed activities. A small system was developed with three services and a web front end. The requirement for this system was described in user stories and the design was documented in the form of sequence diagrams, wireframes, entity diagrams and a system overview. Behaviour driven testing techniques were employed during the development. This is where the tests are created to reflect the behaviour of the user story and the code is written by the developer to achieve this aim. The approach was a good one because the user story limited the amount of functionality that was being developed so that a well-defined set of tests could be developed. The conclusions of this report were that this is a good approach because it continually focuses you on the requirement. However, because each user story may identify additional functionality that needs to be added to a Microservice this does not necessarily produce a well-defined API. The scope of the user stories selected for this project included the user interface and the design did not thoroughly define the API. This created services that need to be tested and tidied up before they could be used as part of a working system. The recommendation is that the API is thoroughly specified so that when development is complete you can test it to its interface. The scope of your user story should break down the work to focus the developer on coding to the API. Microservices need to talk together and have a well-defined API that is thoroughly tested. This testing needs to be undertaken with other tools which can test the interfaces, for example, SoapUI. Recommendations include evaluating the different Behaviour Driven testing frameworks to identify strengths and weaknesses, performing contract and component interface testing, and using automated tools such as Docker to deploy an application for automated testing on a testing environment.

Contents

Declaration	1
Abstract	3
List of Tables	5
Table of Figures	5
Chapter 1 - Introduction	6
Chapter 2 - Literature Review	8
REST	8
REST & Node.js	9
NOSQL	10
Software Testing	11
Typical Service Architecture	12
Microservices	13
Test Approaches	16
Testing Microservices	20
Development Methodology	23
Chapter 3 - Design	25
Chapter 4 - Implementation	30
Database Creation and Testing	30
Node.js and the implementation of a basic web server	33
Node.js and the Express Framework	36
Jasmine Testing Framework	38
Testing Using SoapUI	41
Technical Problems	42
Chapter 5 - Evaluation	44
Chapter 6 - Conclusions and Recommendations	47
Chapter 7 - Reflection	49
References	50
Bibliography	52
Appendix 1	53
Appendix 2	55

List of Tables

TABLE 1 AIMS AND OBJECTIVES	6
TABLE 2 HTTP METHODS [2]	9
TABLE 3 DIFFERENCES BETWEEN MONOLITHIC AND MICROSERVICES ARCHITECTURE. [13]	15
TABLE 4 MICROSERVICES, RESOURCES AND HTTP VERBS USED.	29

Table of Figures

FIGURE 1 COMPARISON OF A TRADITIONAL WEB SERVER WITH NODE.JS [4]	10
FIGURE 2 THREE TIERED WEB SYSTEM [10]	13
FIGURE 3 KEY PRINCIPLES OF MICROSERVICES. [12]	13
FIGURE 4 REASONS WHY NODE IS A GOOD FIT FOR MICROSERVICES	15
FIGURE 5 TESTING APPROACHES TO MICROSERVICES	16
FIGURE 6 TEST DRIVEN DEVELOPMENT STEPS [18]	18
FIGURE 7 EXAMPLE OF A JASMINE UNIT TEST. [20]	19
FIGURE 8 BENEFITS OF CONTINUOUS TESTING [21]	21
FIGURE 9 BENEFITS OF CONTINUOUS DELIVERY [21]	22
FIGURE 10 USER STORIES	25
FIGURE 11 APPLICATION MICROSERVICE DIAGRAM.	26
FIGURE 12 ENTITY RELATIONSHIP DIAGRAM.	27
FIGURE 13 SEQUENCE DIAGRAM	28
FIGURE 14 STARTING UP A MONGO DATABASE	30
FIGURE 15 FINDING ONE RECORD IN A MONGODB TABLE.	31
FIGURE 16 DISPLAYING ALL CUSTOMERS IN A READABLE FORMAT	31
FIGURE 17 TURNING AN IMAGE INTO BINARY (JAVASCRIPT)	32
FIGURE 18 PULLING AN IMAGE FROM THE DATABASE AND CONVERTING IT TO BASE64	33
FIGURE 19 NODE.JS EVENT LOOP DIAGRAM	34
FIGURE 20 CALLBACK FUNCTION EXAMPLE	34
FIGURE 21 INITIALISING A NODE.JS PROJECT	35
FIGURE 22 NODE.JS STARTING UP A SERVER	35
FIGURE 23 HOW THE EXPRESS WEB SERVER WORKS	36
FIGURE 24 NODE.JS PACKAGES USED IN THE PROJECT	37
FIGURE 25 EXPRESS ROUTE TO ADD A NEW USER (CUSTOMER).	38
FIGURE 26 JASMINE UNIT TESTS FOR ADDING A NEW USER.	39
FIGURE 27 RUNNING JASMINE UNIT TESTS WITH FAILURES	40
FIGURE 28 EXPRESS ROUTE TO DELETE A USER (CUSTOMER)	40
FIGURE 29 USING SOAPUI TO TEST ADDING A NEW USER	41
FIGURE 30 ADD USER POST REQUEST FAILING	42

Chapter 1 - Introduction

Whilst doing background reading for this Web Development course some of the journals mentioned the terms, agile development, REST and NoSQL. The journals suggested that testing and development took place at the same time. The approach outlined the use of Microservices each of which has a single responsibility. For example, a service which handles only customer information. This service would handle the creation, updating and deletion of customers on a system. To develop a complete system multiple services would communicate with each other to deliver the functionality. Each of the services in a system were tested individually and as part of one application. This was a different approach to developing one large application which contained all the application logic which would have to be tested as a whole. To deliver these types of services, developers would use different tools such as Node.js, a JavaScript framework and MongoDB, a NoSQL database. As JavaScript was a third year topic on the course and testing is a skill that every developer needs to improve on, this was a good topic for investigation.

The aim and objectives are shown in table [1] below:

Aim

- To evaluate testing approaches for the development of node.js Microservices

Objectives

- Analyse techniques and technologies used in the development of Node.js Microservices.
- Identify testing approaches by developing Microservices and apply appropriate tests and test frameworks during construction.
- Critically evaluate the software testing techniques applied to Microservices.

Table 1 Aims and Objectives

The overall aim shown above was to evaluate the different testing approaches used for the development of Node.js Microservices. This would be achieved by three objectives. Firstly, the literature would be investigated to analyse the techniques and technologies used in the development of Microservices. This was a time boxed activity to be completed within nine weeks. At the same time as the literature review, testing approaches would be identified including a suitable testing framework to be used during construction for the second objective which would involve some development work. To prepare for this objective some training and prototyping were undertaken involving MongoDB and Node.js. This had to be completed within

a set period of time. The final objective would be to evaluate the software testing techniques used as applied to Microservices.

The format of the report is as follows. The literature review will analyse the techniques and technologies used in the development and testing of Node.js Microservices will be researched. This will be followed by a development methodology chapter which will explain the software development methodology chosen for the project and the hardware and software used. Design, implementation and evaluation will be covered by separate chapters. The report will be completed with chapters on conclusions, recommendations and reflection.

Chapter 2 - Literature Review

The literature review will cover, REST, Node.js, MongoDB and provides a description of each of these terms. The understanding of these technologies is important because they are mentioned in journals and books when reading about Node.js RESTful web services and applications. In addition, it will explain the importance of software testing and the differences between two and three tier monolithic applications, and Microservices. Microservices work in conjunction with each other when used as part of a REST based application. This will be followed by a description of the different types of testing that have been identified in the literature that are important in the testing of rest based services and applications.

REST

REST stands for Representational State Transfer; it is an architectural style for designing network applications. Instead of using complex mechanisms to connect machines together, HTTP is used instead. HTTP requests are used to create, read, update and delete data. This is an alternative approach to other methods which use CORBA, Remote Procedure Call or SOAP/XML to connect between machines. REST applications do not use cookies and are referred to as being stateless. Each request transfers all the information that the server needs in order to complete it. REST requests and responses are described as being lightweight because data is stored efficiently in the request [1]. An organization does not have to open up a large number of network ports because HTTP is being used. This means that it is easier to protect the network from attacks.

The key principles of REST are as follows:

- Represent everything with a unique ID. A URI is a uniform resource identifier. This is used to represent information for example, <http://test.com/users/123>: would represent the user with an ID of 123, or <http://test.com/users>: would represent all users in the system. All resources are accessed via this directory-like structure.
- Make use of standard HTTP methods such as GET, POST, DELETE, and PUT. (HTTP stands for Hyper Text Transfer Protocol) Browsers and software interact with resources using the verbs explained in the table [2].

HTTP Verb	Description
GET	Retrieves a resource identified by the URI. Can be compared to READ operation.
POST	Creates a new resource. Can be compared to the CREATE operation.
PUT	Updates resource with the provided data or creates the resource if it does not exist. Can be compared to the UPDATE operation.
DELETE	Deletes the resource identified by the URI. Can be compared to the DELETE operation.

Table 2 HTTP METHODS [2]

- Resources can be linked together provided they don't try to include lots of information.
- Resources can have multiple representations. For example, when following a URI the same information can be obtained in different formats such as, JSON, CSV, XML and HTML.
- Stateless communication. The server does not keep any information about the client allowing it to process requests efficiently.

[2]

REST & Node.js

A large number of developers use Node.js to build REST based systems. This is a JavaScript runtime built on Chrome's v8 JavaScript Engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient[3]. Node.js works on the basis of an event loop, HTTP requests are received on the server that trigger a function that has a call back to the event loop and the user when the processing is complete. This means that when you are writing code you have to be careful that you don't run a process that blocks the loop. For example, if you read a file from disk then you should not do it in sequential code because it could take several minutes to complete. Whilst this processing was taking place, nothing else could happen. This type of code should always be written in a call back function. This allows JavaScript to run on the server and in the browser. Prior to the development of Node.js, JavaScript was used in the browser to create dynamic web pages and manipulate pages on the web client. The advantage of Node.js is that it allows you to use data formatted in JavaScript Object Notation (JSON). Ryan Dahl, the creator of Node.js was aiming to "create real time websites with push capability"[4]. This is achieved by sending data via web sockets. This is important because both the client and the server can initiate communication and exchange data. This is different to the way websites are normally used. They normally use HTML, CSS and JavaScript in the browser and send data over port 80 for HTTP and port 443 for HTTPS. Java applets and Flash have been able to use PUSH technology but this is over nonstandard ports. To use them would require extra

permissions to be opened on firewalls. This is not possible in some organisations who want to keep their systems secure.

Examples of companies which use Node.js are the New York Times, Dow Jones and eBay. Node.js is capable of handling a huge number of simultaneous connections with high throughput which makes it highly scalable. Figure [1] below compares a traditional web server with Node.js.

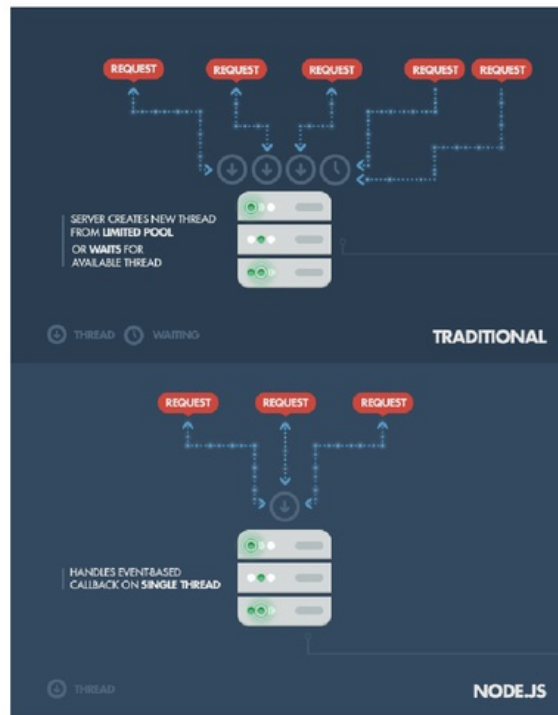


Figure 1 Comparison of a traditional web server with Node.js [4]

The traditional web server makes multiple threads to handle requests whereas Node.js uses a single thread.

Examples of applications where Node.js is used are Chat applications, Data streaming, Dashboards and server side web applications. Node.js allows you to write server side code in small pieces. These are referred to as Microservices. Instead of writing huge applications developers can break everything down into multiple smaller applications. Each individual application will be dedicated to performing one action[5].

NOSQL

Where server side web applications are used a JSON storage database such as MongoDB will ease development. Data can be passed from database to server and clients and back again

without having to convert it. MongoDB is a NOSQL database. This is different from an SQL database because you can use structured and un-structured data. Applications which need high performance, high availability and are scalable [6] should use NOSQL. As it uses key value pairs, document stores and large data tables. This type of database reduces the need for joins which are heavily used in relational databases. This provides high speed and better performance [7].

Software Testing

Software testing is important to discover defects, to ensure that the product works as expected and prove that the software has no faults. "This can only be achieved by executing the software on a computer. Its behaviour must be compared to the requirements. Thus, testing of software is an important and difficult task in software development"[8]. Testing software can be achieved in different ways:

- Executing a program to find failures
- Executing a program to measure quality
- Executing a program to provide confidence
- Analysing a program or its documentation to prevent failures
- Tests can also be performed to acquire information about the test object, which is then used as the basis for decision-making—for example, about whether one part of a system is appropriate for integration with other parts of the system.

[8]

Testing is not just something that is undertaken by one person. It is important that many skilled people take part in this process to ensure that there is quality. "We all agree that great test engineers need to have a strong grounding in engineering and ideally computer science skills. We also all agree that a great tester has something different in them than a developer. They have that tester DNA built into their core[9]."

The earlier a fault can be detected and corrected, the less costly it is to fix. "In testing, it is essential to find the important bugs as early as possible while also striving to design tests that will find bugs and verify functionality and correctness for the entire life of the application[9]."

Software testing can be affected by the requirement, the choice of tools, the experience of the developers and testers creating the software and the design of the system.

System requirements can also have an effect on how software is tested for example, the software needs to work on multiple system such as UNIX, Windows and Linux. Or with a particular browser for example, Chrome and Firefox.

Software systems can be developed by using code that is being compiled or interpreted. If you are using a language like C#(C-Sharp) when you go to compile the program, all errors will be flagged up and have to be corrected before it will finish compiling. This does not mean that the software has no defects it just means that there are no obvious errors. If you were to compare this to an interpreted language such as JavaScript which Node.js uses, then you will not be notified of any errors but once run they are apparent. These are only fixed by debugging. "Testing interpreted code raises additional concerns over testing compiled code, as there is no compiler to check the syntax of the program. Therefore, to find syntax problems, the code must be executed. You should test 100 percent of the code, that is, every line of the code. Testing every line of a script is laborious"[10].

Whilst you are undertaking software development, the quality of the finished product can be effected by the inexperience/experience of the people involved. For example, an inexperienced designer might design an overcomplicated solution which is difficult to test, or an inexperienced programmer might write a large number of lines of code where a small function would be better suited. These examples show that it is important to include reviews in the design and coding process so that the design and the code are as efficient as possible. Pete Goodliffe says, "To ensure you're crafting excellent code, you need people who are checking it at every step of the way. People who will make sure you're working to the best of your ability, and are keeping up to the quality standard of the project you're working on[11]. This encourages accountability, "If you know that someone else will read and comment on your code, you're more likely to write good code"[11].

Typical Service Architecture

The overall design of the system will also influence the way that you need to test it. Large monolithic systems built on two or three tiered architectures can be difficult to test because software in each of the tiers works together to deliver functionality. Figure [2] below shows a three tiered web-based system. This shows that to develop and test changes you need all the tiers available to prove its works.

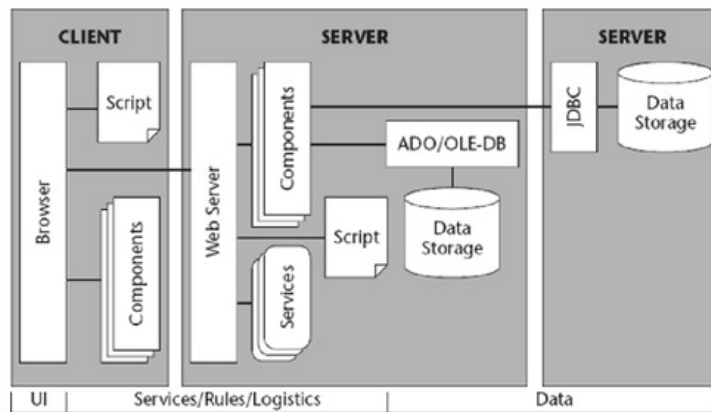


Figure 2 Three Tiered Web System [10]

Microservices

In order to make systems easier to develop, test and maintain software designers are now developing systems based on REST principles using Microservices, for example the Amazon homepage where they are using approximately 40 Microservices to deliver content. This means that if Amazon want to introduce a change it is less costly because they are changing one small bit of software as opposed to changing a monolithic system. The principles of micro services are shown in the Figure [3].

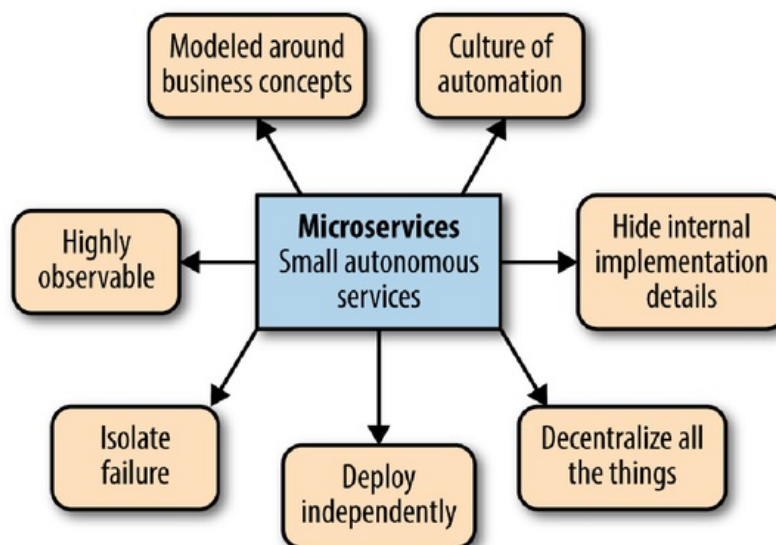


Figure 3 Key Principles of Microservices. [12]

Microservices “is an architecture style, in which large complex software applications are composed of one or more services. Microservices can be deployed independently of one another and are loosely coupled. Each of these Microservices focuses on completing one task only and does that one task really well. In all cases, that one task represents a small business capability”. [13] These small pieces of software perform different tasks. Amazon is an example of a company that had a large monolithic system in 2001. “If you go back to 2001,” stated Rob Brigham Amazon’s AWS Senior Manager for Product Management, “the Amazon.com retail website was a large architectural monolith[14]”. This was architected in multiple tiers, and those tiers had many components in them.”

This monolithic approach demonstrated that over time software “as it grows and the code base gets larger and the architecture gets more complex, that monolith is going to add overhead into your process, and that software development lifecycle is going to begin to slow down”. [14] In response to this software architects developed a concept of decoupling service architectures using Microservices. In order to do this they “went through the code, and pulled out functional units that served a single purpose, and we wrapped those with a web service interface”. [14] For example, the buy button and the calculate tax are Microservices. This means that if the buy button functionality changes it can be recoded and retested without affecting any other functionality on the page.

In the discussion above, Monolithic and Microservice applications have both been mentioned and explained. Table [3] shows the differences between the Monolithic and Microservices architecture.

Category	Monolithic architecture	Microservices architecture
Code	A single code base for the entire application.	Multiple code bases. Each microservice has its own code base.
Understandability	Often confusing and hard to maintain.	Much better readability and much easier to maintain.
Deployment	Complex deployments with maintenance windows and scheduled	Simple deployment as each microservice can be deployed individually, with minimal if

	downtimes.	not zero downtime.
Language	Typically entirely developed in one programming language.	Each microservice can be developed in a different programming language.
Scaling	Requires you to scale the entire application even though bottlenecks are localized.	Enables you to scale bottle-necked services without scaling the entire application.

Table 3 Differences between Monolithic and Microservices Architecture. [13]

As can be seen from this table a Microservices architecture has many advantages over a monolithic system because the code is in smaller chunks making them easier to deploy, maintain and scale.

“Node.js is a good fit for developing Microservices.” [15] Tony Pujals Node Summit presentation states the reasons for this in Figure [4]:

- Excellent package management system
- Minimal ceremony involved in publishing packages (package.JSON, npm publish)
- Encourages lightweight, narrowly-focused packages b/c creating a package is easy & publishing is cheap
- Designed from the beginning for end-to-end asynchronous I/O to promote highly scalable services
- Excellent performance (thanks to V8), which also contributes to scalability
- End-to-end (and isomorphic) JavaScript stack can significantly accelerate development

Figure 4 Reasons why node is a good fit for Microservices

With the use of Microservices updating and maintaining an application is a lot easier. This is because should a fault appear in any part of the main application, the service dedicated to handling the request can be easily found and edited. Testing is also easiest when an update is made to the system, only the application that was altered needs to be tested rather than having to test the entire system from top to bottom. When a system is first created the application can be more complicated to test overall as there are many small applications and services which are

linked together. This means that a range of methods need to be used in order to test the application. In Figure [5] Martin Fowler shows how Microservices link together and the types of testing that can be used.

Test Approaches

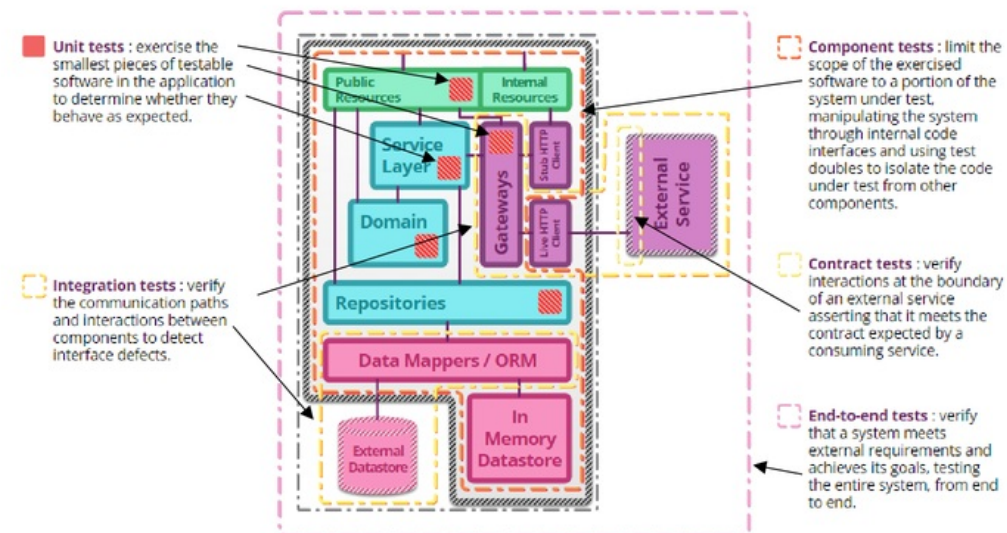


Figure 5 Testing Approaches to Microservices

Approaches to testing can vary, for example, Testers at Microsoft “have a holistic viewpoint on testing”[9]. At Microsoft three types of testing are referred to; Black box, White box and Grey box. White box testing is an approach where you test the software whilst the software is being built. Testing that takes place after the software is built is called black box testing. The white box testing approach “uses test design based on analysis of the underlying source code or schemas not visible to the end user. Test cases founded solely on a white box approach are typically thorough, but nearly always miss key end-user scenarios[9]”. Techniques that are used to do white box testing involve testers to put assertions in code, or use test driven/behaviour driven development. Microsoft testers use both black box and white box techniques ensuring that they test from a customer perspective whilst ensuring that an application works correctly. This is called grey box testing.

Before you start to design your software you should start by designing your unit tests. As a developer or tester, you need to ask yourself questions before you start. Microsoft testers “Ask how the software is supposed to work. Ask how it handles data. Ask how it handles errors. Asking questions about the software can help testers get a jumpstart on design before any of the

code is ever written[9]". By asking these questions you can come up with a test strategy which influences the design and coding of the solution. Testers and developers should also think about testability as early as they can "Testability is the degree to which software can be tested completely and efficiently. Making specific design choices, choosing simple algorithms, enabling test hooks (additional functionality written solely to make testing easier), and making internal variables visible are all examples of how to increase testability[9]."

Software developers put assertions at points in their programs to test whether their program is working. A test is "a condition that must be tested to confirm conformance to a requirement[16]". Lots of tests are required to meet customer requirements. An assertion is a Boolean expression which can either be true or false indicating whether a piece of functionality is working or not. For example, you might check a database. You would make a call to the database, if the connection is available then the assertion would be true and you can continue running the code. If the assertion is "false" then the program will crash and any errors will be presented. The benefits of doing this are that you can catch errors almost as soon as they occur and to catch subtle errors that are not easy to find. The disadvantages of assertions are that they could be coded incorrectly thus reporting an error that does not exist. They are also slow to execute and use up memory as well.

This type of approach has been developed into a process called Test Driven Development. Test Driven Development (TDD) "is a method of software development in which unit testing is repeatedly done on source code. The concept is to "get something working now and perfect it later." After each test, refactoring is done and then the same or a similar test is performed again. The process is iterated as many times as necessary until each unit is functioning according to the desired specifications[17]."

Test Driven Development has three steps which are repeated over and over again until the software development is complete. Shown in Figure [6].

Write a test that expresses how you'll use the code and what you need it to do. This test will fail, producing a red bar on many UI's.

Write enough code to get the test to pass, but no more. If you need more code, for example, to check for errors, first write another test to demonstrate that feature. For now, just write enough code to get the test to pass.

Clean up the code to remove redundancy and improve the design. Then re-run the tests to make sure you didn't break anything.

Figure 6 Test Driven Development Steps [18]

The biggest problem in TDD "is that there is a disconnection between what the product is supposed to do and what the test suite that the development team builds is testing. TDD does not explicitly say how to connect both worlds. This leads to a lot of teams doing TDD, but testing the wrong things. Yes, perhaps they were able to test all their classes, but they tested whether the classes behave as expected, not whether the product behaves as expected[19]." The company could have a very detailed test suite that covers a lot of tests that all pass. But this does not help them find out if the product itself will work as it should. Behaviour Driven Development (BDD) on the other hand "tries to fix these problems by making the test suite directly dependent of the feature set of the product. Basically, BDD is a test-first approach where a new coding task can be created only when a change in the product happens: a new requisite, a change in an existing one, or a new bug [19]."

When you first develop a program using behaviour driven development (BDD) you create a program that fails because the feature doesn't exist then enhance it to produce a small, simple piece of code that meets the requirement.

Jasmine is a framework for BDD. It uses the concept of user stories as a way of writing tests. Below is an example of a test which has been written to check if a song has been paused. Shown in Figure [7]

```
describe("Player", function() {  
  describe("when song has been paused", function() {  
    it("should indicate that the song is paused", function() {  
  
      });  
    });  
  });  
});
```

Figure 7 Example of a Jasmine Unit Test. [20]

Testing Microservices

Once a component or piece of software has been built it is usually tested by experienced system testers. Black box testing “is an approach based on testing an application without any knowledge of the underlying code or functionality of the application[9].” This approach relies on the fact that users of the product or application are only focused on whether the software works and are not concerned with how the software is designed or developed. In order to test web applications, they could either be tested interactively from screen to screen or via an automation tool, for example, Selenium. Web services can be tested by software tools such as SoapUI which also test REST functionality. With regard to Microservices, examples of black box testing are, contract testing, component testing, component interface testing and end-to-end testing.

Microservices are components which have their own API’s (Application Protocol Interface). To test them a tester would use component tests. Component tests limit “the scope of the exercised software to a portion of the system under test, manipulating the system through internal code interfaces, and using techniques to isolate the code under test from other components [13].” These tests are run against the individual service. SoapUI is a tool which can help test REST based APIs. “SoapUI provides us with a very useful option to extract the resource and the associated method automatically from the endpoint URL [2].” This provides a way of testing without having to write code to test the interface.

Component interface testing will verify that data being passed between Microservices in the system is done correctly and verify application flow, apart from simple integration between components. To make testing easier each service has a contract with the services that use it. “The contract consists of expected requirements for input and output data structures, performance, concurrency characteristics, and so on [13].” Once all Microservices have been developed and tested End-to-end testing tests the system “verifying that the system as a whole meets business requirements and achieves its goals [13].” One possible problem with Microservices is that if one component fails, the error message received might not be specific enough to identify the problem. This is sometimes only found during an end to end test.

IBM recommend in their book “Microservices from Theory to Practice” that continuous testing should take place. “Continuous testing reduces the cost of testing while helping development teams balance quality and speed [21].” Figure [8] below shows the benefits of continuous testing:

- Able to avoid unpredicted disruption of business by making sure that software is tested regularly.
- Applications are more reliable
- Proactive problem prevention
- More effective and quicker testing. The more
- Less rework as any problems with the software will be fixed almost as soon as they appear.
- More Efficient and happier workers
- Lower operational costs and improved topline

Figure 8 Benefits of continuous testing [21]

To help the continuous testing process Gartner suggest that testing is easier when the software works the same way in all environments. "Works on my machine → Works on QA machines → Works (the same) on production hardware → Stays working (securely) on the production environment [4]." To help ensure that time is not lost trying to fix environment problems new tools have been developed to make this easier. These tools allow the user to "wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in [5]." These file systems are called containers and are "fundamentally changing the way we develop, distribute and run software. Developers can build software locally, knowing that it will run identically regardless of host environment — be it a rack in the IT department, a user's laptop or a cluster in the cloud" [6]. Docker is an example of one of these tools. It "makes it possible to set up local development environments that are exactly like a live server, run multiple development environments from the same host that each have unique software, operating systems, and configurations, test projects on new or different servers, and allow anyone to work on the same project with the exact same settings, regardless of the local host environment [7]." From my own experience at college I discovered that software behaved differently on different machines because they were not built the same way. These problems take a lot of time to fix. The following figure [9] shows the benefits of continuous delivery.

Continuous delivery has the following benefits:

- The leading practice of deploying code rapidly and safely into production-like environments is followed.
- Deployments start automated tests to ensure that components perform business functions as expected.
- Every change is automatically deployed to staging.
- Deployments are on-demand and self-service.
- Applications can be deployed into production with the push of a button, when business is ready.

Figure 9 Benefits of Continuous Delivery [21]

The literature review has identified the technologies that are used in constructing Node.js web services and explained REST (Representational State Transfer), MongoDB and Node.js. This review has also explained Microservices and why Node.js is a good tool to use for RESTful applications. Having explained REST and the technologies it is used with, a number of different types of testing have been identified and outlined. During development, unit testing is essential. Three testing techniques have been explained these are Assertions, Test Driven Development (TDD) and Behaviour Driven Development (BDD). Usually when using TDD and BDD a framework is used. One example of a framework used for Node.js is Jasmine.

These tests need to be carried out frequently to ensure no bugs or errors are present in the code upon release. Once development is complete more testing takes place in the form of black and grey box testing. Where Microservices are concerned, Component Testing, Component Interface Testing, Contract Testing and End to End testing are carried out. In an agile project that delivers to cloud based systems the previously mentioned types of testing are carried out continuously. New tools and techniques are constantly appearing which allow the software tester to continuously test and deploy. This speeds up the process as multiple test suites can be run in parallel.

Development Methodology

Whilst researching and reviewing the literature it was found that projects which were developing Node.js RESTful applications/services used an approach where a small amount of functionality was built and tested at the same time until it met the requirements. This is different from the waterfall approach which follows set stages; the design stage, the coding phase, the testing phase etc. The literature mentioned that projects were using an agile approach and the Agile Methodology. For this reason, the chosen methodology for the project is the Agile Methodology. A detailed explanation of the agile methodology is explained below. Although for this project only several techniques from this methodology will be used.

Andrew Stellman and Jennifer Greene explain that the agile methodology “is a set of methods and methodologies that help your team to think more effectively, work more efficiently, and make better decisions. These methods and methodologies address all of the areas of traditional software engineering, including project management, software design and architecture, and process improvement. Each of those methods and methodologies consists of practices that are streamlined and optimized to make them as easy as possible to adopt.”[22]

As part of the planning process, agile projects are planned in a series of sprints. Each sprint involves the development of user stories. A user story is a high level requirement which contains just enough information to allow a developer to estimate for a piece of work and to be able to implement it. Typically, a sprint lasts one or two weeks. Each user story has acceptance criteria which ensures that what is delivered meets the requirements. This approach will be used on this project to make sure that it is delivered on time. The aim is to complete the user story and produce developed and tested code. If whilst developing the software a problem arises and the developer thinks that the code will not be completed, then the scope of the requirement in the user story is reduced so that a piece of working software is produced. For example, if the Microservice is being called by a webpage which was aesthetically pleasing but the development of it was taking too long then a more basic page will be created to make sure the functionality works. The advantage of using this approach for this project is that the developer will be able to plan to develop several user stories and will be able to stick to the project timescale. In addition, when developing and testing software the user stories could be used to help write the tests.

Because the developer will be developing and managing the project on their own, a disadvantage of using this approach might be that the project might not proceed to plan unless the developer regularly makes sure they are on task. The developer will also regularly provide feedback to his supervisor.

To make sure that the project is delivered on time, the design section will outline the requirement and include some design diagrams. These diagrams will document the functionality built and will include sequence diagrams, wireframes, entity diagrams and a system overview.

The software that will be used for this project will be Node.js. As explained in the literature review, this is a product that is used to create RESTful web applications and Microservices. Node.js is open source software. Packages which enhance the functionality can be downloaded from the internet using a package manager. For example, the Jasmine framework which is a widely used framework for Behavior Driven Development. SoapUI which has REST testing functionality will be used to test the Microservice. No particular hardware is required and the software developed will be portable.

Chapter 3 - Design

The literature review identified that when developing Node.js services developers test code mainly using assertions, TDD (Test Driven Development) and BDD (Behaviour Driven Development) techniques. They also use user stories which allow them to develop small bits of code. Whilst developing they also use test frameworks which allow them to automate their tests. In order to follow the same approach and allow suitable software tests to be created, a shop application was designed. This provides a platform to trial and test out the approaches to testing. The idea was to create an application in which users could log in and purchase goods. For the purposes of this application items are purchased individually. To use the application, the user would have to logon onto the application with a user ID and a password provided by an administrator. The reason for choosing this functionality was because it allows you to develop several Microservices that can be used by a front end UI (User Interface). This would be similar to a requirement that you might get to deliver a system when working on a customer's project.

These requirements were described as user stories. Figure [10] below shows the user stories that have been chosen for implementation.

User Stories Implemented
As an administrator I want to maintain a set of user/customers' details that would allow users to logon to the application. The success criteria for this is that the web page shows a full list of userids and allows the user to create, amend and delete a list of customers.
As a customer when I log onto the application I want to view a list of goods to purchase and be able to select an item for purchase. The success criteria for this is for the user to be presented with a list of products from which they can select an item
As a user/customer I want to create an order for an item. The success criteria for this is that the user selects and creates an order for an item

Figure 10 User Stories

These user stories allow the creation of several Microservices so that software can be created during the implementation and are called Users, Products, Orders. All the services will use JSON to handle data. The administrator user story is an example of single responsibility because the Microservice will deal with the administration of user data. This Microservice will allow an administrator to add, edit and delete customers/users. The other two user stories use the three Microservices to deliver functionality to the user.

The requirement was investigated in several different ways. Diagrams were created to show what the overall system would look like, how data would flow through the system, how it would be stored, how it would be accessed and what it would look like. These diagrams were created so that the design would be consistent with the requirement and the implementation.

For this project a number of small services were developed as shown in the following Figure [11]. The diagram shows that there is a main service and three Microservices.

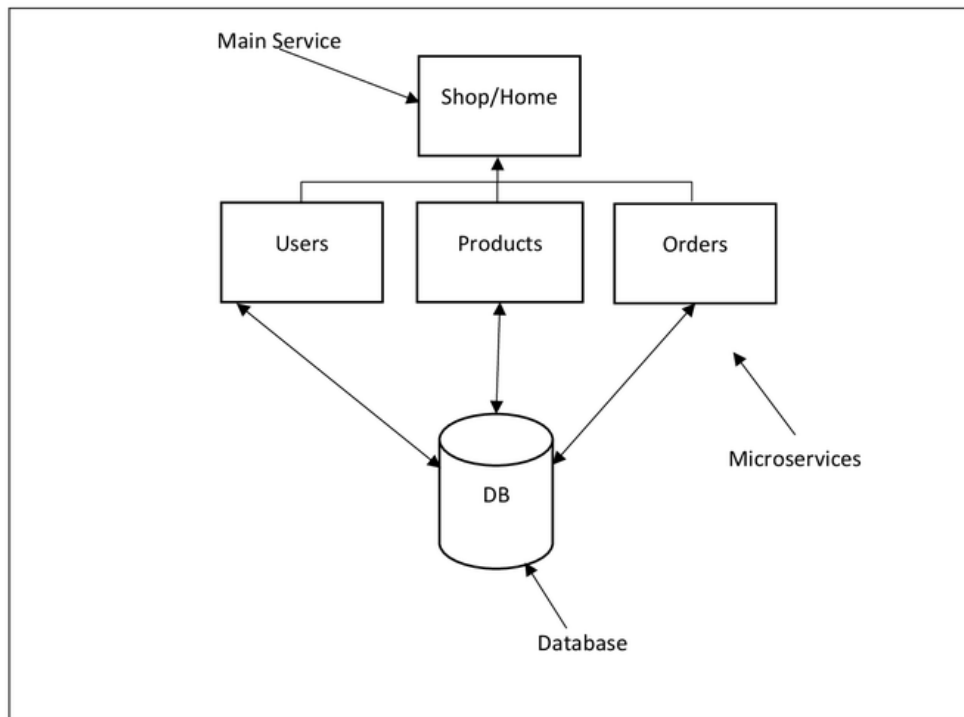


Figure 11 Application Microservice Diagram.

These three Microservices will store, retrieve and maintain data in the Mongo database. The fourth service is a controlling service that handles all user interaction. The Microservices provide a rest backend to the four service (Shop/Home) which is a web front end used by administrators and application users. Each of these services can be invoked and tested independently. The aim was for the Microservices to handle the following errors. Response code 200 for successful requests, response code 400 for Invalid data and 500 for system errors.

Each of the Microservices need to perform one or all of the CRUD (Create, Read, Update and Delete) operations. In order to do this a database was designed and this is shown in an entity relationship shown in the following Figure [12].

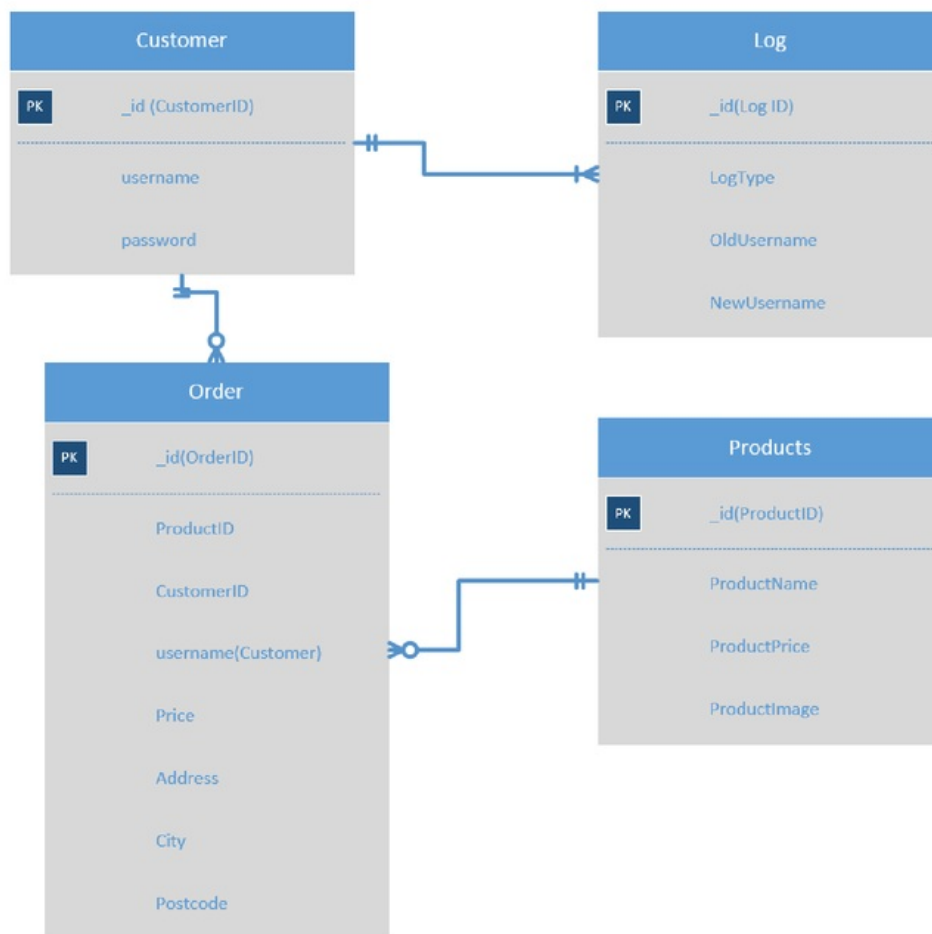


Figure 12 Entity Relationship Diagram.

Figure [12] above shows all the tables that are used in the application. These are, Customers (Users), Products, Orders and Logs (An audit entry is added to the log table when account information is changed).

The relationship between them are described below.

- Each Customer can have many orders and orders can only have one customer.
- Each change to Customer data can have many log entries and each log entry is for one change to Customer data.
- Each order can have many products and a product can have many orders.

For the purposes of this application the customer table will also include the user's password which will be set by the administrator.

Sequence diagrams were used in the design of the application to illustrate how the application should work. The sequence diagram below Figure [13] is for the administration functionality. Sequence diagrams show the steps involved and what order they are carried out when implementing a user story. When the administrator connects to the system, they request the user administration page from the Shop service. This page will use AJAX (Asynchronous JavaScript and XML) functionality to retrieve a list of users from the user service and display them on screen. Whilst using the page the administrator will have the ability to add, update and delete user information. This will be done by making AJAX calls to the user service via the global.js file. There will be a global.js file that handles all the functions relating to the application. The full diagram will be shown in Appendix 1. The extract below shows the update sequence.

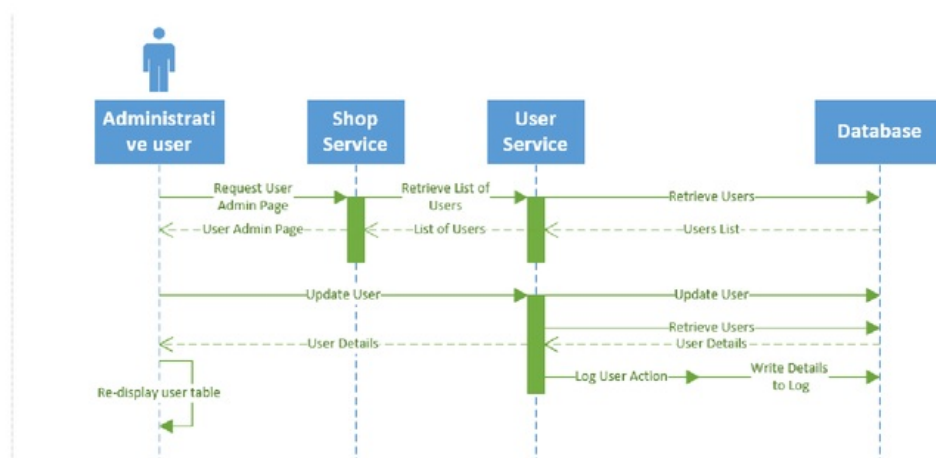


Figure 13 Sequence Diagram

Based on figure [11] and figure [13] the table [4] below provides a summary of what needs to be implemented for each Microservice. The table shows the name of the Microservice, the resources (URI – Uniform Resource Identifier) and the type of HTTP request that needs to be implemented.

Microservice	Resource (URI)	HTTP
User	/userlist	GET(list of users)
	/adduser	POST(Create User)
	/delete/id	DELETE(Delete individual user)
	/makechanges	POST(Update User Details)
Products	/products	GET(List of products)
Orders	/neworder	POST(New Order)

Table 4 Microservices, Resources and HTTP Verbs used.

Wireframe diagrams were also created as part of the design to provide an easy template to follow. Below is an example of the wireframe diagram produced for the administrator user interface. The rest are shown in Appendix 2.

Administration

CustomerID

Username

Delete User?

1234

User2

Delete?

4321

User1

Delete?

New User

Username...

Password...

Log In

Update User

Username...

Password...

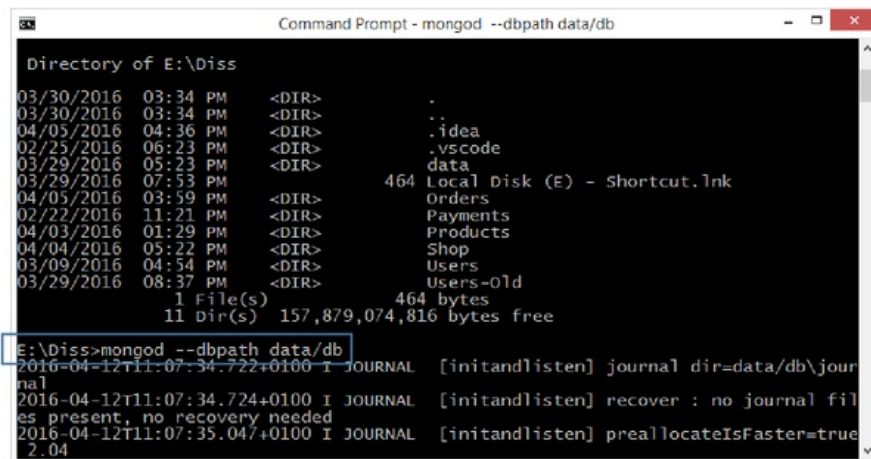
Log In

Chapter 4 - Implementation

In order to implement this project Node, the express framework for node and MongoDB were installed on a development PC. These are all required in order to develop the application. The implementation section will explain how the database was created and tested. This will include an explanation of why the images were stored in a binary format in the database. The section includes a description of how Node.js works and a basic web server is created. This will include reasons why this was not suitable for the requirement and why the express framework was used. Explanation of the express framework and how it was implemented for this project is followed by a discussion of how Jasmine was used for this project and the code was developed. This chapter is concluded with an explanation of how SoapUI was used to do some basic testing of the Microservices and the main technical problems experienced whilst doing the project.

Database Creation and Testing

Before the construction phase can begin the MongoDB server process has to be started and the database created. Starting the server is achieved by opening a command prompt and running the command "mongod --dbpath data/db". Figure [14] below shows the command being used.



```
Command Prompt - mongod --dbpath data/db

Directory of E:\Diss
03/30/2016 03:34 PM <DIR> .
03/30/2016 03:34 PM <DIR> ..
04/05/2016 04:36 PM <DIR> .idea
02/25/2016 06:23 PM <DIR> .vscode
03/29/2016 05:23 PM <DIR> data
03/29/2016 07:53 PM 464 Local Disk (E) - Shortcut.lnk
04/05/2016 03:59 PM <DIR> Orders
02/22/2016 11:21 PM <DIR> Payments
04/03/2016 01:29 PM <DIR> Products
04/04/2016 05:22 PM <DIR> Shop
03/09/2016 04:54 PM <DIR> Users
03/29/2016 08:37 PM <DIR> Users-Old
1 File(s) 464 bytes
11 Dir(s) 157,879,074,816 bytes free

E:\Diss>mongod --dbpath data/db
2016-04-12T11:07:34.722+0100 I JOURNAL [initandlisten] journal dir=data/db\jour
na1
2016-04-12T11:07:34.724+0100 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2016-04-12T11:07:35.047+0100 I JOURNAL [initandlisten] preallocateIsFaster=true
2.04
```

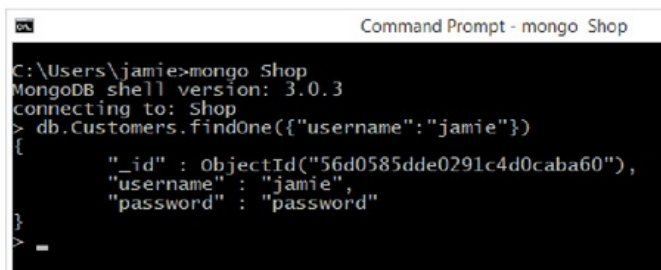
Figure 14 Starting up a Mongo database

This starts a server process. Whilst this is active you can connect to the database either by connection via an IP address or port or via a command shell. The database path in the example above relates to the directories which are relative to the place where you run the command.

MongoDB is a database which is document oriented. Tables are referred to as collections, rows are documents. Once the server is running, creating a database table (collection) is relatively simple. This can be done in two ways, either by using a command to insert a document into the

collection, `db.(Table/Collection Name here).insert({"Record Name": "Record"})`. Or by running the command `db.createCollection("CollectionName")`. This would create a collection based on the collection name. Each of the tables/collections in the entity diagram were created using the first method. For example, the customers collection was created using the following command. `db.Customers.insert({"username": "user1", "password": "password1"})`.

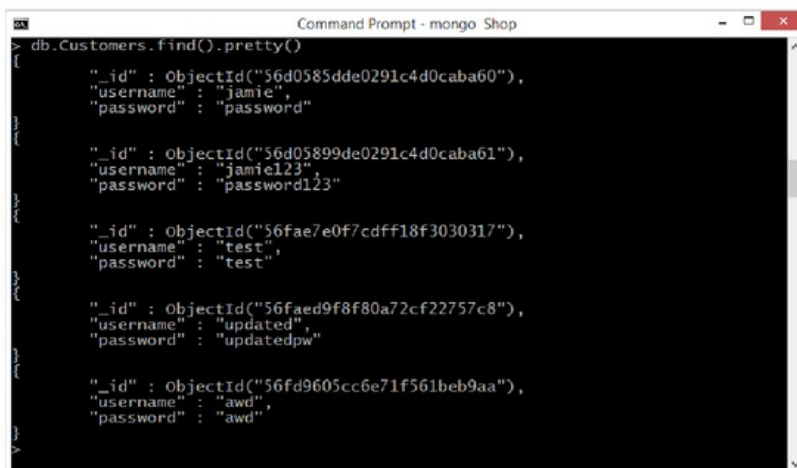
Once the database was set up it was tested to ensure that it was consistent with the design and that data could be selected, updated and deleted. To locate a record with the database, use a find command. For example, if you want to find a user's details from their username you would run the db find command shown in figure [15].



```
Command Prompt - mongo Shop
C:\Users\jamie>mongo Shop
MongoDB shell version: 3.0.3
connecting to: Shop
> db.Customers.findOne({"username": "jamie"})
{
  "_id" : ObjectId("56d0585dde0291c4d0caba60"),
  "username" : "jamie",
  "password" : "password"
}
```

Figure 15 Finding one record in a MongoDB Table.

If however, you want to see all the records in a database table you just need to use the following command `db.Customers.find()` which will find and print each record in that table. (Adding `.pretty()` to the command prints the records in a readable format.) See figure [16]



```
Command Prompt - mongo Shop
> db.Customers.find().pretty()
{
  "_id" : ObjectId("56d0585dde0291c4d0caba60"),
  "username" : "jamie",
  "password" : "password"

  "_id" : ObjectId("56d05899de0291c4d0caba61"),
  "username" : "jamie123",
  "password" : "password123"

  "_id" : ObjectId("56fae7e0f7cdff18f3030317"),
  "username" : "test",
  "password" : "test"

  "_id" : ObjectId("56faed9f8f80a72cf22757c8"),
  "username" : "updated",
  "password" : "updatedpw"

  "_id" : ObjectId("56fd9605cc6e71f561beb9aa"),
  "username" : "awd",
  "password" : "awd"
}
```

Figure 16 Displaying all Customers in a readable format

Records were inserted and deleted in the database to make sure that the database was consistent with the design.

A requirement of the application was for images to be displayed on one of the forms. This meant that as part of the implementation process research had to take place about how MongoDB stored images. There appeared to be two ways of doing this, the first was using GridFS which uses the database like a filing system where files can be stored and retrieved by name. Images could also be stored in a binary field. At the time the database was setup, the intention was to use GridFS, but as the project progressed the choice of database connection using the monk package meant that images could not be retrieved by the application code and displayed as part of a form. For this reason, the second method of using a binary field had to be used. A similar approach had been used when working on a multimedia web development assignment using local storage. Images were stored in base64 format and retrieved and shown on a webpage. The following screenshots in figure [17] show how an image is turned into binary and then stored inside the database.

```
MongoClient.connect(url).then(function (db) {  
  var collection = db.collection('Products');  
  return collection.insert({  
    productName: 'Wales Jersey',productDesc: 'Wales Senior International Rugby', productPrice:  
'25.99',  
    // the constructor takes a Buffer  
    productImage: new Binary(fs.readFileSync('/wales.jpg'))  
  }).then(function () {  
    //console.log('Welsh Jersey inserted');  
    db.close();  
  }).catch(function (err) {  
    console.log(err);  
  });  
});
```

Figure 17 Turning an image into binary (JavaScript)

The code below in figure [18] shows how the image is pulled from the database and the binary is converted into base64.

```
collection.find({},{}).function(e,docs){  
  docs.forEach(function(doc, index) {
```

```

base64Image = new Buffer(doc.productImage, 'binary').toString('base64');
base64Image = doc.productImage.toString('base64');
docs2.push( {
  "prod_id": doc._id,
  "productName": doc.productName,
  "productDesc": doc.productDesc,
  "productPrice": doc.productPrice,
  "base64Image": base64Image
});
});
res.JSON(docs2);
});

```

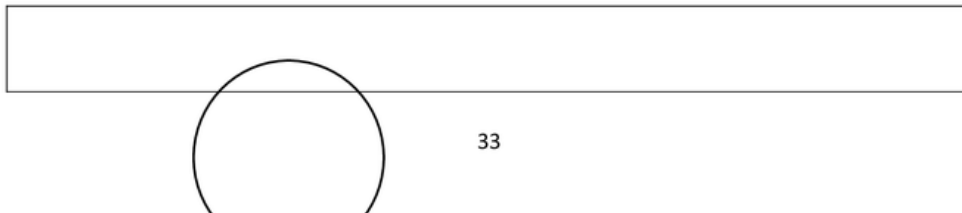
Figure 18 Pulling an image from the database and converting it to base64

The HTML below shows how the image is used on the website. Inside the image source tag a data image statement is added which defines a base64 binary image. The image is concatenated to the string prior to the width and height entries.

```
'
```

Node.js and the implementation of a basic web server

To enable the functionality to be built and tested it was necessary to understand how Node.js works and build a basic web server. "Node takes the approach that all I/O activities should be non-blocking. This means that HTTP requests, database queries, file I/O, and other things that require the program to wait do not halt execution until they return data. Instead, they run independently, and then emit an event when their data is available." Node.js has an event loop which runs continuously. Figure 19 shows how this loop takes in user requests and passes them on to other functions.



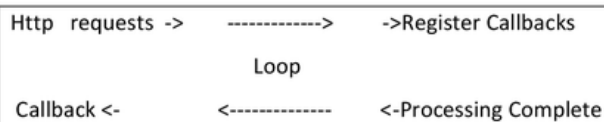


Figure 19 Node.js Event loop Diagram

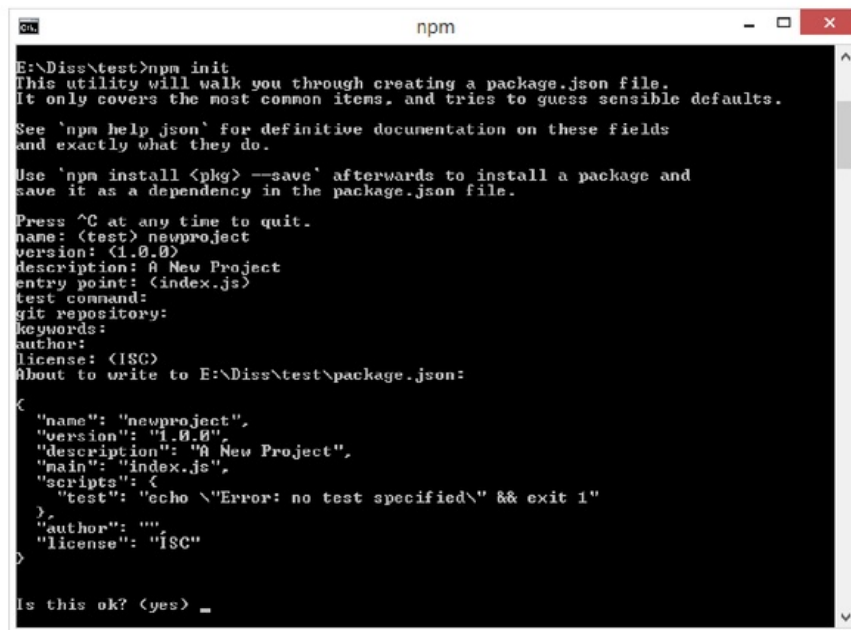
This means that programming in Node.js has lots of call backs for example. An example of a callback function is as follows shown in figure [20]

```
collection.find({},{}),function(error,docs){  
  if(error) throw error;  
  res.JSON(docs);  
});
```

Figure 20 Callback function example

In this example a request is being made to the function to return data from the database. This request will either be successful, in which case data will be returned to the requester, or unsuccessful in which case an error will be thrown. Whilst developing you have to be careful to follow this approach otherwise you can block the event loop and everything will stop working.

Hosting a standalone node.js web service is relatively simple. After creating a new folder for the project open up a command prompt and type “npm init”. This will initialise a new project. The screenshot below in figure [21] shows this process.



```
E:\Diss\test>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See 'npm help json' for definitive documentation on these fields
and exactly what they do.

Use 'npm install <pkg> --save' afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (test) newproject
version: (1.0.0)
description: A New Project
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to E:\Diss\test\package.json:
{
  "name": "newproject",
  "version": "1.0.0",
  "description": "A New Project",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this ok? (yes) _
```

Figure 21 Initialising a Node.js Project

This creates a package.JSON file containing all the necessary information about the new project.

To create a http server the package.JSON file has to be edited to add the http package.

Below is a code example of how a server would be coded in javascript by creating an app.js file.

Shown in figure [22]

```
var http = require('http');

const PORT=7000; //The port the server will listen to

function handleRequest(request, response){ //A function to handle requests and send
responses
  response.end('Server Running ' + request.url);
}

var server = http.createServer(handleRequest); //Create the server
server.listen(PORT, function(){ //Start the server
  //Callback triggered when server is successfully listening
  console.log("Server listening on: http://localhost:%s", PORT);
});
```

Figure 22 Node.js Starting up a Server

To start the web server, the command *"node app.js"* or *"npm start"* is executed. Although this is a simple and easy way to set up a server, for this project it did not have the code to handle basic

functionality such as, session management, dealing with requests and responses, error handling and routing. Routing refers to the creation of application endpoints (URI's) and a way of responding to a client's request. For these reasons the express framework was chosen.

Node.js and the Express Framework

As explained above if you just use Node without any framework you have to write a lot of additional code which is not a task for this project as it would take away focus from the testing aim. Fortunately, node has a framework which will handle all of this. The express framework has a generator tool which creates a skeleton application with this functionality which the developer has to customise. One of the first decisions a developer has to make is what middleware (Packages) they will want to use. This will be influenced by the user requirements.

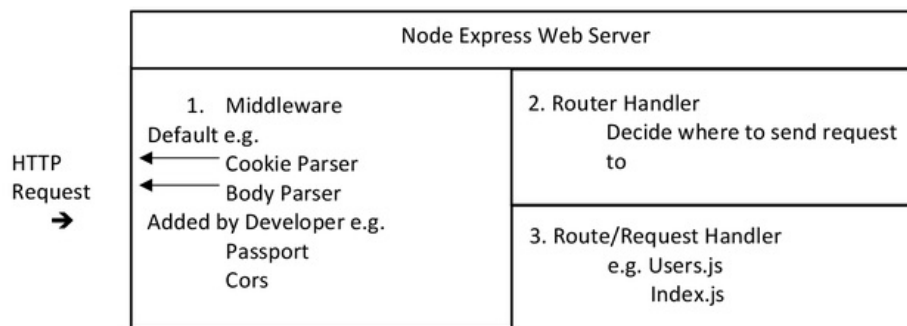


Figure 23 How the Express Web Server works

Figure [23] shows how the express framework works as a webserver. The middleware is shown on the left-hand side of the diagram. Middleware is often referred to as a package. Each package runs as a process against the HTTP request before it is sent to the route handler. Examples of additional middleware/packages that were installed for this project are, the MongoDB driver to allow the application to connect to the database and manipulate data, the Passport package that deals with user authentication, and the CORS package that fixes the "Content Origin Resource Sharing" error. Without the CORS package, content cannot be taken from the same domain. These packages are installed by using the command "*npm install (package name) --save*". This command installs the latest update of a package into the application also using "*--save*" to add the package and its version to the dependencies in the package.JSON file.

The example below in figure [24] shows the packages and their versions that are used in this project.

```
{
  "name": "Shop",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "cors": "^2.7.1",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "jade": "~1.11.0",
    "mongodb": "^2.1.7",
    "mongoose": "^4.4.5",
    "monk": "^1.0.1",
    "morgan": "~1.6.1",
    "passport": "^0.3.2",
    "passport-local": "^1.0.0",
    "serve-favicon": "~2.3.0"
  }
}
```

Figure 24 Node.js Packages used in the project

Although the package libraries have been added to the application directory, the application cannot use them until the developer includes and initialises the library. This is done by using a *require* (*var mongo = require("mongodb")*) statement and a use statement (*app.use(mongo())*). These are used inside the app.js file.

HTTP requests are first processed by the middleware. (Shown in figure 23), the application handles the request via a route handler that is defined in the app.js file. For example, for the customer administration functionality a route was added to handle requests for the resources shown in figure 4. This was done by adding the code "app.use('/users', routes)". A request for *http://127.0.0.1:8000/users* is processed by the app.js file which redirects requests to a request handler called "users.js". For this project code was created in the users.js files to handle all the user administration requests. The screenshot below in figure [25] illustrates this.

```

router.post('/adduser', function(req,res){
  var db = req.db;
  var collection = db.get('Customers');
  var collection2 = db.get('logs');
  var body = req.body;
  console.log(req.body);
  var username = req.body.username;
  console.log(username);
  if (isEmptyObject(req.body)) {
    res.status(400).send({ error: "Empty Object" });
  }
  if (username.length = 0) {
    res.status(400).send({ error: "Username is 0 chars" });
  } else {
    collection.insert(body, function(err,result){
      if (err) {
        res.status(500).send({ error: "Server Error" });
      } else {
        res.status(200).send("Record inserted " + username);
        collection2.insert({'type':'NewUser','username': username});
      }
    });
  }
});

```

Figure 25 Express Route to add a new User (Customer).

Jasmine Testing Framework

The literature review described a development and testing approach using the Jasmine framework, where a series of tests are designed to test the code before it is developed. These tests can be grouped together and used to test a user story. This approach is called behaviour driven development. These tests run asynchronously and use a call-back function. This waits for a response before either timing out and failing or carrying out the test. To use Jasmine, the developer has to install it from the command line with the command “npm install Jasmine-node --save” The tests are grouped together in a spec file.

Each Jasmine test is started with an “it statement” which explains what it is testing and has a series of expect statements. In this code the “expect” statements are checking the status code and the data which is returned within the response. The expect function has to evaluate to true to work. Test 2 in the example is testing the addition of a new user. In this case a 200 response is being returned. While tests 2a and 2b are testing for invalid data. In these cases, there is a 400 response returned and the body of the response contains the message invalid data. In test 2c not shown above test 2a is modified and the word JSON has been deleted. This causes a 500 response to be returned.

These tests are built up whilst the code is being created so they initially fail and as the code is added so they start to succeed. The screenshot below shows the output. For the complete user story there can be many tests. All those tests shown in green have succeeded. The Jasmine tests for the user stories are shown in Appendix 4.

For the user administration system, logic was written to handle the adding a user, deleting a user and updating a user functions. For example, on receiving a post request for adding a user the service validates the data received and inserts the data into the database. Should the data be wrong a 400 response is sent back to the calling application with a 400 status code. If the update fails, then a 500 message internal server error is returned.

This code was created at the same time as the Jasmine tests. For the User administration user story a set of tests were created to check that the user administrations functions work. An extract of the tests for the add user portion of the user story are shown below in figure [26].

```
it("\r\nTest 2 Should Add a user", function(done) {
  request.post(
    'http://localhost:8001/users/adduser',
    { json: { "username": "ADDUSERTEST", "password": "Password1" } },
    function (error, response, body) {
      if (!error && response.statusCode == 200) {
        console.log(body);
      }
      expect(response.statusCode).toBe(200);
    }
  );
  done();
}, 9000);

it("\r\nTest 2a Should Return a 400 response because the username is not supplied.", function(done) {
  request.post(
    'http://localhost:8001/users/adduser',
    { json: { "username": "ADDUSERTEST", "password": "Password1" } },
    function (error, response, body) {
      if (!error && response.statusCode == 400) {
        expect(response.statusCode).toBe(400);
        expect(body).toBe("Invalid Data");
      }
    }
  );
  done();
}, 9000);
```

Figure 26 Jasmine Unit tests for adding a new user.

```

As an administrator I want to maintain a set of users details that would allow users to login to the application - 2385 ms
test 1 Check Service Can Be Reached returns status code 200 - 100 ms
test 2 Should Add a user - 117 ms
test 2a Should Return a 400 response because the username is not supplied. - 111 ms
test 2a Should Return a 400 response because the username contains invalid characters. - 101 ms
test 2b Should Return a 500 response because the username is not supplied. (No JSON Data has been supplied causing server process to fail)
test 3 Should Not add a user but return a 400 response because no username sent - 100 ms
test 4 This should update a user record - 300 ms
test 5 This should delete a user via the service - 114 ms
test 6 This will not delete a userid because it does not exist - 101 ms
test 7 Should List Users - 100 ms
Failures:
1) As an administrator I want to maintain a set of users details that would allow users to login to the application
test 6 This will not delete a userid because it does not exist
Message:
Error: Expected 200 to be 300.
Stacktrace:
Error: Expected 200 to be 500.
at Request.callback (C:\Diss\Shop\spec\testsSpec.js:150:41)
at Request.self.callback (C:\Diss\Shop\node_modules\request\request.js:199:22)
at Request.emit (events.js:110:17)
at Request.<anonymous> (C:\Diss\Shop\node_modules\request\request.js:1036:10)
at Request.emit (events.js:129:20)
at IncomingMessage.<anonymous> (C:\Diss\Shop\node_modules\request\request.js:963:12)
at IncomingMessage.emit (events.js:129:20)
at _stream_readable.js:908:16
Finished in 2.574 seconds

```

Figure 27 Running Jasmine Unit tests with failures

The test shown in red has failed because the code is still being developed and the logic was not developed at the time the screenshot was taken.

```

-}):
/* Post to delete user */
router.delete('/deleteuser/:id', function(req,res){
  console.log(req.body);
  var db = req.db;
  var collection = db.get('Customers');
  var userToDelete = req.params.id;
  var collection2 = db.get('logs');
  var username = req.params.username;
  collection.remove({'_id':userToDelete}, function(err){
    res.send((err === null) ? {msg:'User Deleted - ' + userToDelete} : {msg:'error: ' + err});
  });
  collection2.insert({'type':'Deleted User','username':userToDelete, 'newUsername': 'DELETED'});
-}):

```

Figure 28 Express Route to delete a User (Customer)

There is no logic here to check that the deletion has failed and a 200 response code is returned every time the code is executed.

As you begin to develop and test using a TDD/BDD approach. You start off with a series of tests and then realise as you begin to code that extra tests are required and these need to be added as you go along. Although designs were produced the approach is good because you are continually thinking about the design as you develop and refine the tests.

Testing Using SoapUI

Having completed the development work although not strictly part of the objectives some further tests were undertaken using SoapUI. SoapUI is a tool which allows testing to be performed by accessing REST endpoints. Below is a screenshot of a test done in SoapUI shown in figure [29].

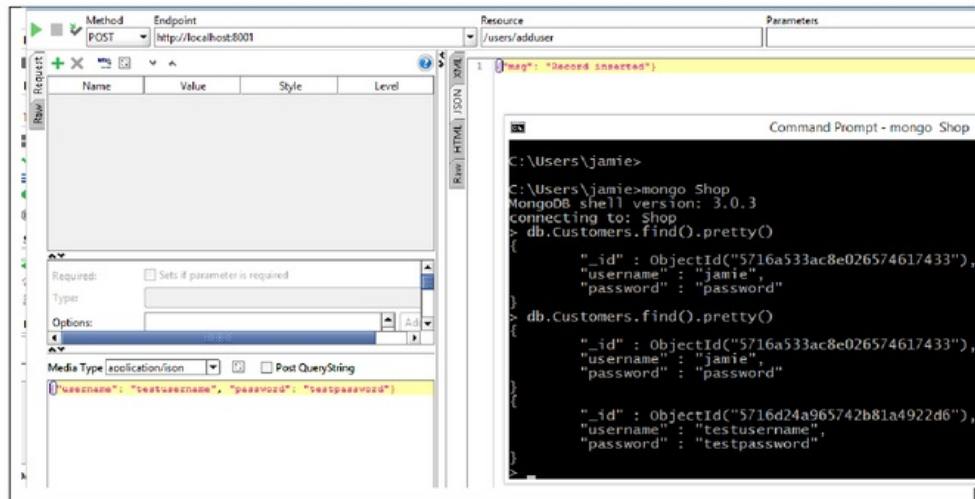


Figure 29 Using SoapUI to test adding a new user

Figure [29] shows a post request being sent to the “adduser” route (<http://localhost:8001/users/adduser>). The bottom left hand side of the figure shows the test data that is being sent with the post request and the right hand side displayed the record inserted message that was retrieved from the Jasmine test. A MongoDB shell also shows that the new record has been inserted.

Whilst undertaking this testing and trying to get the software working some of the errors did not appear to be consistent and this suggested that further code should be added to make the code handle more types of errors in a consistent way. Referring back to the literature this suggested that programmers classified errors this should have been identified prior to development, possibly at the design stage of the project or when creating the Jasmine tests. Figure [30] below shows an example of an error.

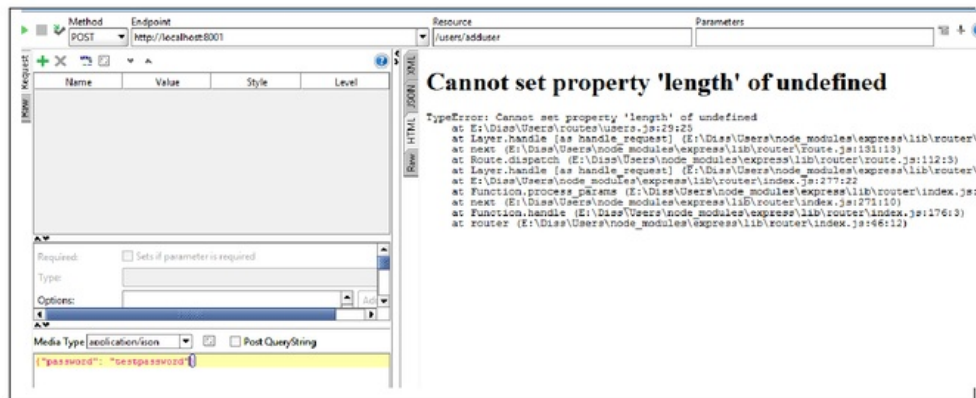


Figure 30 Add user post request failing

Figure [30] shows an example of an error message you get when the input data is not correct. The data that was included in the post request only included the password input and not the username input.

Technical Problems

Many problems were encountered during development. The Jasmine documentation is brief with few examples which makes initial development difficult. In addition, extra node.js packages are required to do specific testing. For example, `require.js` for web URL calls. These also have limited documentation which sometimes makes for a trial and error approach. When setting up the delete test it was not obvious how to code the Jasmine test. This was resolved by looking up the problem on web forums.

When starting Jasmine tests they are executed with the following command. "Jasmine-node spec" where spec is the folder the Jasmine tests are stored in. After a test is written and hangs it is not always obvious where the problem lies. In one case the test was always hanging. After investigation the problem was caused by the Microservice were running asynchronously. This meant that the tests had to inform Jasmine when they were complete. When testing first started the `"done()"` was left out. After considerable investigation additional flags were added to the Jasmine test command which displayed exceptions. This helped to identify the problem. The command was modified to be `"Jasmine-node --verbose --captureExceptions spec"`.

Whilst developing the code a lot of the debugging had to be done with `console.log` statements. These statements were added to the code and then removed once the code was working. This approach was essential because without it there was no easy way to resolve problems.

When attempting to communicate between services “Cross-Origin Resource Sharing” appeared as an error message in the browser when attempting to call the service. After investigating this was eventually resolved by adding in a CORS package and using “*npm install cors --save*” which automatically installs the latest version and updates the package JSON file. The application was then modified by adding the statement “*var cors=require(“cors”)*” into the app.js file.

Whilst undertaking this project there was quite a learning curve. In order to build a service, the developer has to understand how the express framework works and be able to modify it to their requirements. On occasions if you do not fully understand this the application will not work. One error encountered was to do with ordering the statements properly in the web server file. The code to access the web site routes was in the incorrect place above the database connection statement. This meant that none of the actions involving the database worked because the database connection had not been created.

When using JADE the HTML template engine minor spacing problems caused the HTML not to be created properly. The documentation was limited and it was a puzzle to get the HTML working correctly.

Chapter 5 - Evaluation

Initially the topic chosen for the first task required some training in JavaScript and MongoDB which was achieved by doing a self-study course called “MongoDB for Node.js Developers”. This course built a small blogging application and did not use the express framework. This built up some confidence in using the tools and went well. Having refined the dissertation and project aims it became apparent that to be productive Node.js developers were using the express framework. As the project scope had been refined to give a testing focus with the development of Microservices using Node.js and MongoDB some further prototyping with the express framework was undertaken. This was aided by a tutorial called “Creating a Simple RESTful Web App with Node.js, Express and MongoDB” [23]. Undertaking these activities was essential for the construction phase of the project.

The aim of the project was to evaluate testing approaches for the development of Microservices. Overall the aim of the project was met. The literature review identified the technologies that were being used in the development of these services and having understood them they were applied during the construction phase allowing this evaluation to take place. The project produced a working set of Jasmine tests and working software. Completion of testing after the development re-enforced the need for component and contract testing and a clearly specified API.

The literature review described and identified the approaches used for testing and the technologies used for creating Microservices which contributed to achieving the first objective and partly the second. The approach used in this project was white box testing. This approach suggests that as a tester you should keep asking yourself questions about how the software should work, how it handles data and how it handles errors. This project raised a lot of questions before starting testing regarding how Node.js works, how MongoDB works and how the express framework works. Having understood the basics of this, further questions were raised such as how you retrieve data from MongoDB, how Node.js should handle it, how session handling works, how to store and retrieve images, how routing works and how the Jade HTML framework works. Some of these questions had to be answered before work was commenced and some were answered through investigation, trial and error whilst the software was being developed. This disrupted the approach because the series of tests defined for the first user story were paused while issues were resolved. As expected, they all initially failed and then as code was developed some issues were resolved and the tests began to pass, while others led to investigation to answer the questions raised above. This was a bit discouraging in the beginning as things were very complicated and took days to complete. When trying to resolve problems it

was very difficult to figure things out as the documentation appeared to be quite limited. At the beginning a few packages were chosen to help meet the requirement. For example, the passport framework which was chosen to deal with session handling did not fully explain what was needed in terms of additional code to make it work correctly. This meant that you could bypass the login for the application and therefore data within sessions was not handled correctly. After much investigation the package was reinstalled and a few lines of code were added which enabled it to work.

Having overcome the problems, the second objective was met. The development work was carried out in a behaviour driven way using the Jasmine framework. However, the identification of the tests and the development of the code for the logon and shop functionality was rushed because of the problems discussed previously. The behaviour of the system and the expected outcomes of the user stories were achieved. Comparing this approach to creating a test plan for developing a piece of software which has been done for previous assignments this method appears more productive as you can keep running the tests until they pass and meet the requirement. This means that you are continually focused on the requirement and you can make minor changes and retest quickly with a series of automated tests. As discussed in the literature review Jasmine tests are useful because they allow you to test the application response to check if functionality is working. For example, if you add a user record to the database and expect it to be returned in the HTML then you can check that the record exists by use of an assertion. You can also check the HTTP response codes which enables you to catch errors as soon as they occur. Because the assertion is in the test and not in the developed code, this avoids the disadvantage mentioned in the literature that they may slow down program execution and occupy memory. After developing the code, the importance of developing a contract between two services as mentioned in the literature was understood. If the project were to be done again then the API for the web service would need to be specified in more detail before coding. This is because the services did not seem to cover all test paths. The Microservices had been coded whilst developing the user stories but did not cover all possible testing paths. Whilst testing using SoapUI there was no clear API to test against.

Although Jasmine was the framework chosen for this project, as development progressed this did not always have all the functionality that was needed to do the testing. For example, when authentication was implemented the Jasmine test did not work. This was because Jasmine did not support session handling. Additional packages can be installed which handle sessions. For example, Super Test was supposed to handle this functionality however this stopped Jasmine working altogether and had to be uninstalled. There are other frameworks all of which have

slightly differing functionality which handle this sort of problem in different ways. Examples are Mocha, Vows and Cucumber. Use of these frameworks might have given different results.

On reflection if the project were to be done again then a recommendation would be to have some additional training in Node.js and the Express framework before commencing work. This would have meant that the application could have been developed without having all the side issues. As MongoDB was a new tool for this developer, some training was undertaken prior to commencement of the project. This proved invaluable to the understanding of creating and querying collections.

Chapter 6 - Conclusions and Recommendations

The introduction outlined the main aim for this project which was to evaluate testing approaches for the development of node.js Microservices. Developers use different tools to develop these services including Node.js and MongoDB. Objectives were set to analyse the tools and technologies used in the development of Node.js Microservices, identify testing approaches and apply appropriate tests during the construction phase and to critically evaluate the software testing techniques.

During the literature review the technologies used to build these Microservices and the ways in which they were tested were discussed. Developers use an agile white box approach to testing. This takes the form of unit testing, assertions and behaviour driven development. The advantage of using a behaviour driven approach is that the software produced meets the user requirement because the developer focuses on the requirement. The review also explained how Microservices differed from large Monolithic systems. Monolithic systems require lots of testing when they are first created and each change requires a large amount of testing. Microservices require more specific testing when they are created and modified because they have a small amount of functionality. However, a large amount of testing has to take place when they are first put together to ensure data flows through the system and that they perform to their API. The review contributed to completing the objective to analyse techniques and technologies used in the development of Microservices and to complete the objective to identify suitable testing approaches.

A requirement was documented during the design phase using user stories and diagrams were produced to provide a system overview. These diagrams documented what the overall system would look like, how data would flow through the system, how it would be stored and how it would be accessed. These diagrams were created so that the design would be consistent with the requirement and provide some focus when developing the user stories. Because the approach was to develop user stories this did not document the API for the Microservices.

The implementation section explained how the database was created and tested and included an explanation of why the images were stored in a binary format in the database. This was followed by an explanation of why the Node.js express framework was used together with a discussion of how Jasmine was used for this project and the code was developed. This chapter is completed by explaining how SoapUI was used to do some basic testing of the Microservices and the main technical problems experienced whilst doing the project. Whilst undertaking the testing using SOAPUI there was realisation that the API needed to be more tightly specified to

ensure consistency. The design and implementation phases contributed to the objective to identify testing approaches and apply appropriate tests during construction.

The literature review, the implementation phase and evaluation phase contributed to the completion of the final objective to critically evaluate the software technique as applied to Microservices. Defining the tests before starting to code and following a behaviour driven approach focused development on the requirement and produced a series of automated tests. This approach improved understanding of the requirement. As this project was developing three Microservices and a user interface, the user stories chosen were too large and should have been broken down into smaller requirements. This needs to be considered during the development.

The tools and techniques used in the development were slightly different to previous experience. The database was non-relational and used collections, whereas the web server used an event loop which ran continuously. The programmer had to be careful when writing code not to block this loop. For this reason, most of the developed code uses the concept of a "callback" function. This approach starts a task running and upon completion sends either a response or an error to the calling function. Understanding how these work took some effort which took time away from the real aim of the project. Having got the base functionality working correctly, developing using behaviour driven development techniques was good because it focused on the requirement and produced a series of automated tests which were re-used over and over again.

Further work which could be undertaken includes, exploring the different testing frameworks for node.js comparing their functionality and recommending the best one, this work is necessary because while testing using the Jasmine framework there were some features that were missing from the framework for example, session handling. Post development testing is also an area for future work looking at the tools and techniques that could be used to test contracts and perform component interface testing. In addition, a large number of organisations are now automatically deploying their software using Docker, and using tools such as SoapUI to do automated testing on their testing environments.

Chapter 7 - Reflection

This project was chosen because it used different tools and techniques than I have previously used. These are, Node.js, MongoDB and the Jasmine testing framework. As this project would require me to use new tools and techniques some preparation was done prior to starting. For example, a MongoDB for Node.js Developers course was taken to start things off. However, when I started the project I found that there were lots of additional languages and tools to use. To deliver the project I made a decision to use the express framework because this had a lot of inbuilt functionality which I hoped would allow me time to concentrate on the testing. This meant that I had to take additional time to learn the Jade HTML pre-processing language and understanding the express framework and how it worked. In the future when undertaking project work it would be beneficial to keep the scope of the task smaller so that it is achievable. During the development of the application I wanted to focus on testing but I also wanted to have something to show and demonstrate that was not just a single box on a web page.

Doing this project has added to my skills as I now have experience of using MongoDB, Node.js and the Express framework. I have also looked into and used some aspects of the Agile Methodology. During development I also gained some experience of BDD (Behaviour Driven Development) using the Jasmine framework.

This has changed my way of looking at things. When I started the Web Development course I was just programming and then thinking about testing. Now I realise that you need to think about testing first. In order to do that you need to firmly understand the requirements and have a design. I now realise the difference between a developer and a programmer and I am now focussing more on testing.

References

- [1] Rest.Elksetin.org, "Learn REST: A Tutorial," 2015. [Online]. Available: <http://rest.elkstein.org/>. [Accessed: 05-Jun-2015].
- [2] Kankanamge, "Web Services Testing with soapUI," Packt Publishing, 2012, pp. 198, 203.
- [3] Node.js, "Node.js," 2015. [Online]. Available: <https://nodejs.org/en/>. [Accessed: 10-Nov-2015].
- [4] C. Tomislav, "Why The Hell Would I Use Node.js? A Case-by-Case Tutorial," 2015. [Online]. Available: <http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js>. [Accessed: 05-Jun-2015].
- [5] A. S. Ahmed, "NodeDay 2014: Micro-services in Node.js," 2014. [Online]. Available: <http://exponential.io/blog/nodeday-2014-you-only-need-100-lines-of-javascript/>. [Accessed: 05-Nov-2015].
- [6] G. Menegaz, "What is NoSQL, and why do you need it?," 2015. [Online]. Available: <http://www.zdnet.com/article/what-is-nosql-and-why-do-you-need-it/>. [Accessed: 05-May-2015].
- [7] MongoDB, "NoSQL Databases Explained," 2015. [Online]. Available: <http://www.mongodb.com/nosql-explained>. [Accessed: 05-Jun-2015].
- [8] H. Schaefer, "Software Testing Foundations, 4th Edition," Rocky Nook, 2014.
- [9] B. Rollison, "How We Test Software at Microsoft," in *Online*, 2008, pp. 27, 61, 60, 55, 57.
- [10] H. Q. Nguyen, B. Johnson, and M. Hackett, "Testing Applications on the WEb - Test Planing for Mobile and Internet-Based Systems," in *America*, vol. 1, Wiley, 2003, pp. 175, 144.
- [11] P. Goodliffe, "Becoming a Better Programmer," in *O'Reilly*, 2014, pp. 327, 328.
- [12] S. Newman, "Building Microservices," 2015.
- [13] S. Daya, N. Van Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, S. Narain, R. Vennam, and I. Redbooks, "Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach," 2015, pp. 13, 24, 36, 71, 44, 45.
- [14] S. Fulton, "What Led Amazon to its Own Microservices Architecture," 2015. [Online]. Available: <http://thenewstack.io/led-amazon-microservices-architecture/>. [Accessed: 10-Nov-2015].
- [15] T. Pujals, "Node.js Microservices," 2015. [Online]. Available:

<https://docs.google.com/presentation/d/1XUTR9macJ-vVknDbwxYvng3pwRD3DC5wg3AU3XtjP1w/edit>. [Accessed: 07-Jan-2016].

- [16] T. Team, "Test Assertions - How do they work?," 2010. [Online]. Available: <https://www.thoughtworks.com/insights/blog/test-assertions-how-do-they-work>. [Accessed: 05-Nov-2015].
- [17] M. Rouse, "Test-driven development (TDD) definition," 2015. [Online]. Available: <http://searchsoftwarequality.techtarget.com/definition/test-driven-development>.
- [18] T. King, "Twelve Benefits of Writing Unit Tests First," 2006. [Online]. Available: <http://sd.jtimothyking.com/2006/07/11/twelve-benefits-of-writing-unit-tests-first/>. [Accessed: 16-Nov-2015].
- [19] E. Amodeo, "Learning Behavior-driven Development with JavaScript," pp. 15, 358.
- [20] P. Ragonha, "Jasmine JavaScript Testing," 2015, p. 22,.
- [21] I. Redbooks, *Microservices from Theory to Practice*. 2015.
- [22] A. Stellman and J. Greene, *Learning Agile*. 2015.
- [23] C. Buecheler, "The Dead-Simple Step-By-Step Guide for Front-End Developers to Getting up and Running with Node.js, Express, Jade and MongoDB" 2015. [Online]. Available: <http://cwbuecheler.com/web/tutorials/2013/node-express-mongo/>

Bibliography

K. Banker, P.Bakkum, S. Verch, "MongoDB In Action, 2nd Edition". 2016.

R. Black, "Managing the Testing Process, 3rd Edition". 2009.

F. Bob, "Microservices, IOT and Azure". 2015.

Y. Eugene Liang, "JavaScript Testing". 2010.

E. Hahn, "JavaScript Testing With Jasmine". 2013.

L. Koskela, "Effective Unit Testing". 2013.

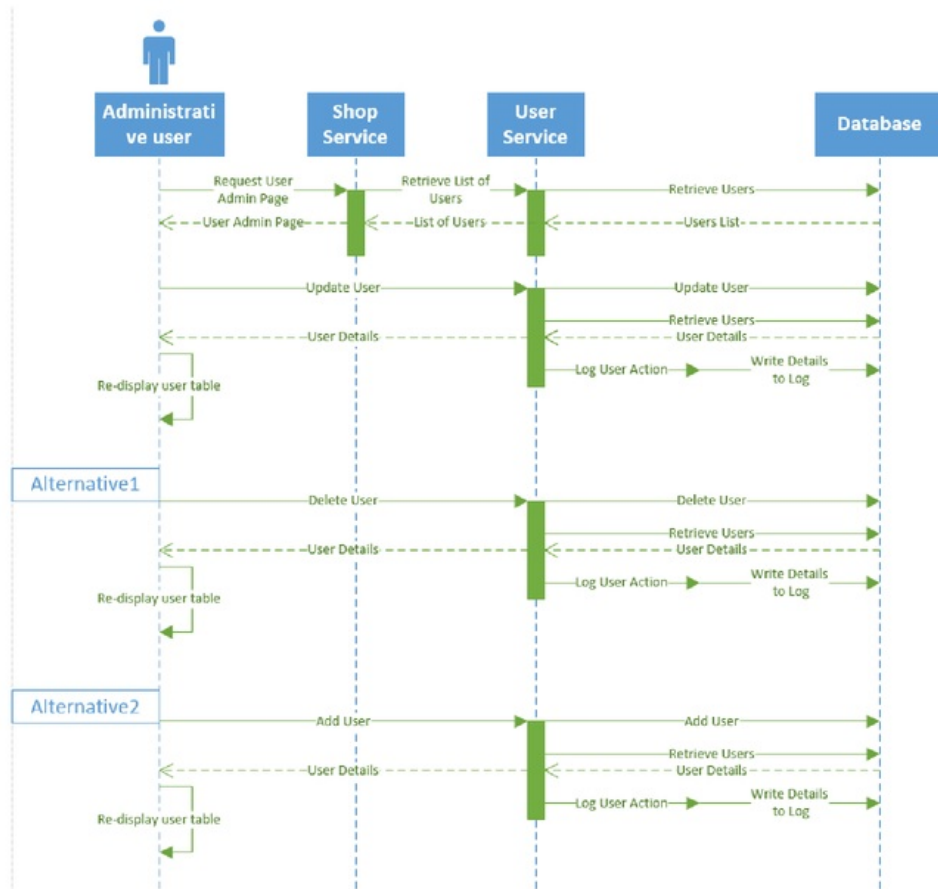
R. Osherove, "The Art of Unit Testing, 2nd Edition". 2013.

N. Sabharwal, S. Gupta Edward, "Practical MongoDB". 2014.

M. Sethi, "Jasmine Cookbook". 2015.

Appendix 1

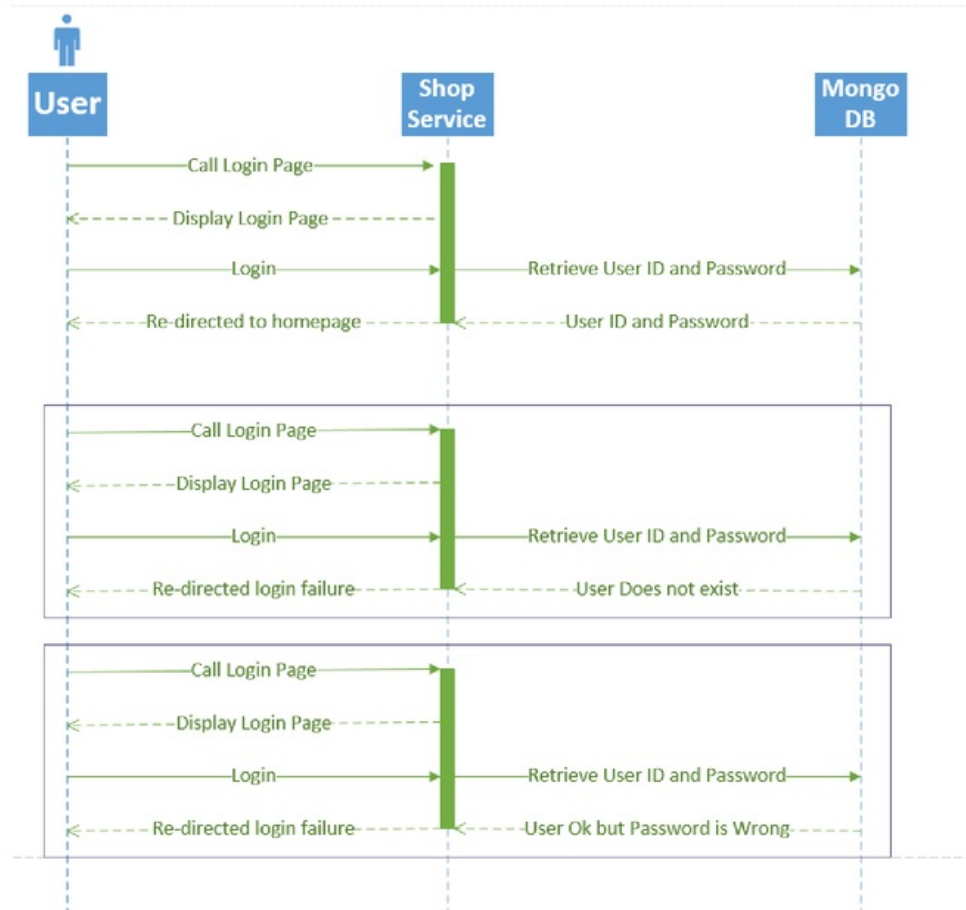
User Administration Sequence Diagram.



Placing an order sequence diagram.



User login sequence diagram.



Appendix 2

Login screen wireframe diagram.

A wireframe diagram of a login screen. It features a large rectangular container. Inside this container, there is a smaller rectangular box. Within this box, there are three elements: a text input field labeled "Username...", another text input field labeled "Password...", and a button labeled "Log In".

Administration screen wireframe diagram.

A wireframe diagram of an administration screen. It features a large rectangular container. At the top left of the container is the title "Administration". Below the title is a table with three columns: "CustomerID", "Username", and "Delete User?". The table contains two rows of data. Below the table, there are two side-by-side forms. The left form is titled "New User" and contains a "Username..." input field, a "Password..." input field, and a "Log In" button. The right form is titled "Update User" and contains a "Username..." input field, a "Password..." input field, and a "Log In" button.

CustomerID	Username	Delete User?
1234	User2	Delete?
4321	User1	Delete?

Product screen wireframe diagram.

