



NUI Galway
OÉ Gaillimh

End to End Development & Documentation of an AngularJS & NodeJS App, with Full Continuous Integration & Test Automation.

A Software Engineering Project

JAMES O'MEARA

13715519

BACHELOR OF SCIENCE

COMPUTRE SCIENCE & INFORMATON TECHNOLOGY

(4BCT1)

MARCH 2017

SUPERVISOR: MR ENDA BARRETT

1 Table of Contents

1 Table of Contents.....	2
2 Table of Figures.....	5
3 Acknowledgments.....	7
4 Introduction	8
4.1 Brief	8
4.2 Single Page Web App.....	8
4.3 Software engineering Processes'	9
4.3.1 Requirements	9
4.3.2 Design	9
4.3.3 Implementation	9
4.3.4 Acceptance, installation, deployment	9
4.3.5 Maintenance.....	9
4.4 Key challenges	9
4.5 Timeline	10
4.6 Design considerations.....	11
4.7 User requirements.....	12
4.8 System architecture.....	12
5 Technologies	13
5.1 Frameworks	13
5.1.1 AngularJS	13
5.1.2 NodeJS	14
5.1.3 D3.js	15
5.2 Package Managers.....	16
5.2.1 NPM (Node Package Manager)	16
5.2.2 Bower (Client Side Package Manager).....	17
5.3 Automation Tools	17
5.3.1 Grunt (Automation Tool)	17
5.3.2 Jenkins	17
5.4 Testing Frameworks & Tools	18
5.4.1 Jasmine	18
5.4.2 Karma.....	18

5.4.3 Jasmine_node	18
5.4.4 Selenium & Protractor	19
5.5 Source Control	19
5.5.1 GitHub.....	19
5.6 Servers (VMS)	19
5.6.1 Amazon EC2.....	20
5.7 Database	20
5.7.1 MongoDB.....	20
5.9 Languages	20
5.9.1 JavaScript.....	20
5.9.2 Html	20
5.9.3 CSS	20
6 Implementation	21
6.1 Prerequisites	21
6.2 Project structure	21
6.3 Main package structure.....	22
6.4 Setting up Node Server	23
6.5 Setting up Angular	24
6.6 Setting up testing Frameworks.....	25
6.6.1 Client testing Library (Unit Testing).....	25
6.6.2 Client Side Test Runner (Unit Testing).....	25
6.6.3 Client Tests	27
6.6.4 Server Side Testing (Unit Testing).....	28
6.6.5 Integration Testing (End to End Testing)	29
6.6.6 Coverage.....	30
6.6.7 Writing Tests.....	31
6.7 Git integration.....	33
6.7.1 Creating git project	33
6.7.2 Linking with Jenkins	34
6.8 Automation.....	35
6.8.1 Grunt build tasks.....	35
6.8.2 Grunt Test Automation.....	38
6.8.3 Npm	39

6.9 Creating Build/Deployment & Hosting Server	40
6.10 Setting up VM for Node App.....	41
6.11 Setting up VM for Jenkins	41
6.11.1 Creating slave machine.....	42
6.11.2 Creating tasks (build & test task).....	43
6.11.3 Creating tasks (deploy & restart task)	45
6.11.3 Creating tasks (Scheduled Task)	46
6.12 Tutorials.....	47
6.13 D3 Integration.....	48
6.14 Setting up Database Connection	53
6.15 Working on the App.....	54
6.15.1 UI	54
6.15.2 Simple examples.....	56
6.15.3 Embedding plunker and jsfiddle	57
7 Issues.....	58
8 Conclusion.....	59
9 Evaluation	61
10 Future Work	62
11 Bibliography	63

2 Table of Figures

Figure 1 High level View Example	12
Figure 2 D3.js examples	15
Figure 3 Module count : http://www.modulecounts.com/	16
Figure 4 Server.js file snippet.....	23
Figure 5 High level app folder layout	23
Figure 6 Setting up angular dependencies	24
Figure 7 Jasmine Test.....	25
Figure 8 Output from Karma testing.....	26
Figure 9 Karma configuration snippet	26
Figure 10 Controller test snippet	27
Figure 11 Basic Jasmine_nodejs console output	28
Figure 12 Protractor config.....	29
Figure 13 Karma coverage config snippet.....	30
Figure 14 Example output of coverage html	30
Figure 15 Example of actual coverage, showing 1 of 2 paths tested.....	30
Figure 16 Sample setup for a jasmine test.....	31
Figure 17 Example client side unit test	31
Figure 18 Server side jasmine test example	32
Figure 19 Jasmine Integration test example.....	32
Figure 20 Screenshot from Github Showing project Repository	33
Figure 21 Github Repo of This Project	33
Figure 22 GitHub Webhook Configuration	34
Figure 23 Browserify example	36
Figure 24 Grunt watch automation task.....	37
Figure 25: Grunt Karma config.....	38
Figure 26 NPM command to start app	39
Figure 27 Amazon EC2 instances	40
Figure 28 EC2 dashboard snippet	40
Figure 29 Jenkins login screen	41
Figure 30 Jenkins options panel.....	42
Figure 31 Jenkins Manage nodes option	42
Figure 32 Labelling Jenkins slave	42
Figure 33 Jenkins pane showing slave machines.....	42
Figure 34 Jenkins job trigger via github	43
Figure 35 Jenkins config: clone git repo.....	43
Figure 36 Jenkins config, delete workspace and enable nodejs	44
Figure 37 Jenkins build commands.....	44
Figure 38 Jenkins Build task select slave to run task on	45
Figure 39 Jenkins Build trigger from other successful job	45
Figure 40 Jenkins Build commands for hosting vm.....	45
Figure 41 Setting built triggers for periodic build	46

Figure 42 Examples of times for peroidic builds config	46
Figure 43 Example view of tutorials loaded in browser	47
Figure 44 D3 Service	48
Figure 45 D3 directive collecting data passed in	49
Figure 46 D3.js Directive watcher	49
Figure 47 Defining force on the graph.....	50
Figure 48 Creating SVG Container to page	50
Figure 49 Drawing the links for the graph	50
Figure 50 Initial config of D3.js Force graph nodes & links.....	50
Figure 51 D3 Graph screenshot	51
Figure 52 Graphs showing more detail about topics, top left: Testing, top right: Git, bottom: Continuous Integration	52
Figure 53 Retrieving something from MongoDB database with Node.js	53
Figure 54 Inserting something into MongoDB database with Node.js.....	53
Figure 55 Creating angular tabs.....	54
Figure 56 Angular function ng-if to make sense of dynamic content.....	54
Figure 57 Angular iframe taking scope variable to display tutorial in iframe.....	55
Figure 58 Angular ng-if: will display d3 graph with current scope data	55
Figure 59 Example Directives home page.....	56
Figure 60 Angular Directive for jsFiddle.....	57
Figure 61 Angular directive html for jsFiddle.....	57

3 Acknowledgments

I would like to thank IBM, while I was on placement I approached my manager and supervisor regarding Ideas for projects. As this was linked to what I was working on during placement I thought this would be a good idea, since I would have a basis to work on and look more into.

I would also like to thank my supervisor for taking on me and my project and helping me in anything I needed.

4 Introduction

4.1 Brief

“End to End processes & development of a single page web app using AngularJS & Node.js. documenting/demonstrating the process and any necessary tutorials, from developing an app, utilizing full testing frameworks and Continuous integration and deployment of an app for an enterprise level app.”

The app should provide tutorials and as a walkthrough, to show new developers, students or anyone interested in the full development cycle of an enterprise level app. This project will concentrate on the developer side, it will give an introduction into AngularJS, Node.js, testing frameworks for unit and integration testing, Continuous Integration with a build and hosting server. It will also provide additional tasks to provide local development build and test automation.

Aims:

- Build a single page application using NodeJS/AngularJS to display tutorials on the relevant technologies for new developers/students.
- Configure and setup a build system for NodeJS/AngularJS application.
- Deploy and configure a continuous integration platform capable of automating the build process.
- Host Build server and app on separate servers (amazon web services, Microsoft azure etc.).
- Evaluate several test frameworks and rigorously test the application, from unit tests, through to integration tests etc.
- Allow the user to test out their code using the application whilst following a tutorial (they won't have to clone a project etc.).
- Generate coverage reports/metrics and ensure that the code meets high test coverage.
- Configure tasks to run test automation locally and as build jobs on build server.
- Configure automation tasks to run locally for development purposes, should detect file changes and run appropriate tasks such as building app and appropriate testing.
- Create a visualization of the main topics and components covered by the tutorials in the app.

4.2 Single Page Web App

A Single page web app, is simple that, A single web page which doubles up as an App, rather than creating a conventional website where the user would be directed & redirected to several pages and must wait for each individual page to load. A single page web app is an app that can be accessed through the web, and will provide a single access point for simple interaction for the user.

Similarly, to an app on a mobile device this web app will provide the same level of functionality, but through a different medium (the Web).

4.3 Software engineering Processes'

Here are the general processes in software engineering, whether creating an app for your own use or up to an enterprise app for a large company, you should follow these. The way in which they may be ordered can be somewhat different if you use an Agile, Waterfall or any other approach. But the concept is the same.

4.3.1 Requirements

Application requirements are gathered here.

4.3.2 Design

Describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudocode and other documentation.

4.3.3 Implementation

When actual code is written

4.3.4 Acceptance, installation, deployment

The final stage of initial development, where the software is put into production and runs actual business.

4.3.5 Maintenance

After the product, has launched, this is when additional changes are made, corrections/bugs are fixed and any other modifications that are required to keep it in shape.

4.4 Key challenges

Key Challenges for this Project is the level of detail for the tutorials for taking the user from nothing, to having a simple single page web app created, complete CI integration, building and testing. It must appeal to developers that have little to no knowledge of AngularJS and NodeJS but also be specific enough to give the developer an insight into the complete end to end process of a product.

4.5 Timeline

(Semester 1)

Weeks 6-8

- Node & angular set up
- Database connected

Week 9-10

- Automated testing frameworks added
- Portion of instructions done

Week 11-12

- Study for exams...

(Semester 2)

Week 1-3

- Set up VM for Jenkins and for hosting
- Continue tutorials
- Continuous integration (Jenkins setup)
- Git push requests

Week 4-6

- Set up Jenkins to build on hosting VM
- Research into D3

Week 7-9

- Create data and force graphs for high level views
- Start report

Week 10-12

- Finish site
- Finish Report

4.6 Design considerations

How would my tutorials look like?

The tutorials should have clear descriptions, screenshots and examples of code. It should be at a level of detail not to overwhelm who is reading it, but still detailed enough and separated out to make it easy to find and follow certain things.

Who/What is my audience?

My audience will be developers/software engineers getting their hands on with AngularJS and NodeJS for the first time. And help give an insight into the whole process from creating the code, to testing it, how it is built/bundled for a website. Then how Continuous integration takes its part for nightly builds and tests and deploys it to the hosting server.

How will my tutorials be accessed?

They will be done in word/files or PDF's and I may combine them to possibly make a small book. But for the fact of having them easily accessible I will make them accessible on my sample app. They will also be included in with the project if the user wants to clone the repository and play around the code themselves, and read the tutorials alongside the source code.

How am I going to set up Continuous integration?

I would like to have some VMS accessible at any time, ideally free since money may be an issue to continue with this project. Ideally I would like two, one for a build server and one for a hosting server, and ideally something I can configure/install and start from scratch so I can document this.

What Testing frameworks would I use?

For this I, would have to look at what there is available, Ideally I would like to cover all aspects of testing, black/white box testing methods and incorporate code coverage metrics. Testing should include server-side, client-side and End-to-End (automated click through the site)

4.7 User requirements

User should be able to look at some sort of poster or graph at first to see the overall scale of the topic, i.e. some sort of high level diagram to get a feel of what keywords/key items that are included and what they are related to. This could be done with word, paint etc., something that could give the user some interactivity would be the best solution to see what is linked to what.

E.g. something along the lines of this very simple graph, that could demonstrate all the links between the different components and processes.

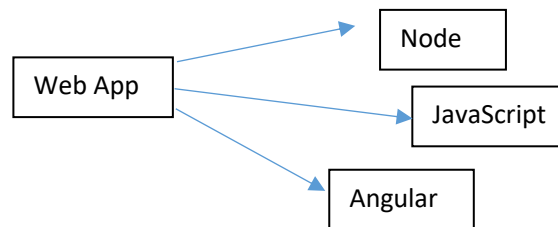


Figure 1 High level View Example

The App should also display some example elements in it, that could be used solely for examples or tutorials, as some parts may start to take on a bit of complexity. The ideal situation would be for the user to clone the repository look at these tutorials, examples on the site and then begin to play about, see how it works, and make their own.

4.8 System architecture

This app will have a NodeJS server side and AngularJS client side. This app will not require much support from the server, as Angular JS allows for a responsive dynamic content. It will use a plugin architecture to make use of any modules available by NPM (Node Package Manager) to control all the packages that may be needed.

5 Technologies

Below are the technologies that will be used in this project. How they may compare to similar technologies and any they are used.

5.1 Frameworks

These will be the main frameworks I will be using in my project; they are Angular and NodeJS which will deal with the front end and the back end of the app.

5.1.1 AngularJS

AngularJS is a JavaScript framework to create responsive Dynamic Web Pages. AngularJS lets you use html and JavaScript to create and extend custom elements. AngularJS is also very powerful in a way which makes html dynamic and responsive, without any page reloads since with html alone, element changes or Dom changes must require commutation back and forth between the server and client.

AngularJS is based of JavaScript and s loaded into the browser, allowing for your computer to do the work rather than waiting on a server to respond with what view to load or what type of page. This ensures the page runs faster and more responsive since the user is not waiting for a refresh, or a response from the server. AngularJS does not completely cut out requests and responses to and from the server but removes the unnecessary unneeded requests.

For example, AngularJS 2-way data binding, allows the developer to insert a text box in one position and a display of what is in the text box elsewhere, is something is added to the first, then it will appear in the second. Although this is a very simple example, this means there could be some data loaded into the DOM and the use could manipulate it with functions or buttons available dynamically and responsively without having to request the server to do so.

5.1.2 NodeJS

NodeJS is a JavaScript Framework for the server side. NodeJS is “event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, Npm, is the largest ecosystem of open source libraries in the world.”

NodeJS is Asynchronous event driven, this means there is a process in the background polling for events, and if certain events fire then certain parts of code or functions would execute, In JavaScript polling is done by the runtime engine, this allows NodeJS and AngularJS to be more dynamic and allow certain things to happen as events fire. Say you click this button and fire off an event, this means that the underlying engine will pick it up and execute what needs to be done, instead of waiting for a loop. This can prove to be hard to debug as you cannot always tell what event may fire off first or respond first.

NodeJS advantage also lies within it being based on JavaScript many developers may work solely on the client side with JavaScript and may not ever touch the backend, with NodeJS and AngularJS working so well together it stands to no reason why anyone working on front cannot work on the backend and vice-versa. JavaScript and NodeJS is single threaded and since JavaScript is widely known it makes sense to use it as a backend as to not create that new layer of learning on top of the developers currently working on a project.

Synchronous and **asynchronous** transmissions are two different methods of transmission synchronization. Synchronous transmissions are synchronized by an external clock, while asynchronous transmissions are synchronized by special signals along the transmission medium.

5.1.3 D3.js

D3.js is JavaScript Library that provides data visualization tool that uses html, CSS and JavaScript to manipulate documents based on data. D3 allows you to create unique, interactive and dynamic SVG components in the browser. D3.js main goal is to change something based on data. D3 is extremely fast, it supports large datasets and allows code reuse through diverse plugins.

D3 provides functionality to give dynamic properties to objects on screen based on the data provided, D3 provides functionality to select certain data, track mouse movement and click events. This makes D3 very versatile in what it can do. It can create unique graphs of nodes and links, pie charts with animations, tree structures, all in a way that will visually describe the data and look cool while its doing it.



Figure 2 D3.js examples

5.2 Package Managers

A Package manager is something that will keep track of the dependencies installed their version and you can also customize what dependencies you want to install and when. Example, install all dependencies for development, since this may require many more dependencies for testing and building, you could also set up it in a way it will only install the dependencies it will need to run on the host server. Making the app take up less room and be more efficient in memory usage.

5.2.1 NPM (Node Package Manager)

“Npm is the package manager for JavaScript. Find, share, and reuse packages of code from hundreds of thousands of developers — and assemble them in powerful new ways.”

Npm is there to install and manage dependencies of your project. You can install dependencies at certain versions and save it in a file called “package.json”, in this file you will find all the dependencies you have installed and the versions In which you have chosen.

Npm is now quite a large package manager that has more than 6 million users a month, and is growing fast compared to many other repository (maven, CPAN, PyPI etc)

Module Counts

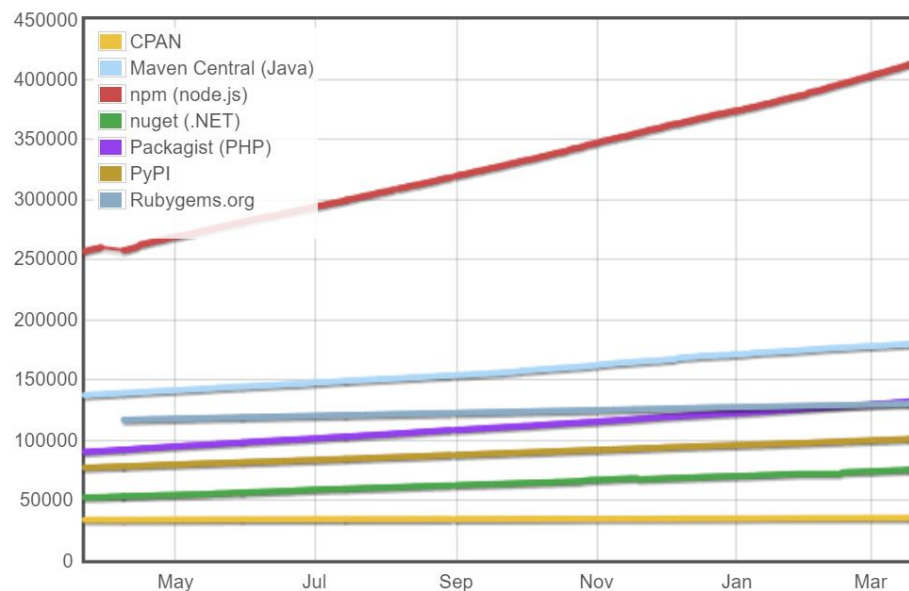


Figure 3 Module count : <http://www.modulecounts.com/>

5.2.2 Bower (Client Side Package Manager)

Compared to NPM, Bower concentrates on client side repository's and handling client side dependencies'.

“Bower can manage components that contain HTML, CSS, JavaScript, fonts or even image files. Bower doesn't concatenate or minify code or do anything else - it just installs the right versions of the packages you need and their dependencies.”

Bower is optimized for the front end of an application compared to NPM which would be an overall package manager, Bower also keeps tracks of the dependencies installed and the versions picked. NPM can be used to install front end packages as well, but bower is more accustomed to installing html, css and other front end packages that can be used in development

5.3 Automation Tools

Automation tools are there to take the extraneous tasks away and automate them, they can range from simple tools that will recognize a file being saved and then run a command or it could be all the way to recognizing a git push, re cloning the repo, re installing dependencies re building testing in every way and then finally deploying and restarting the server.

5.3.1 Grunt (Automation Tool)

Grunt is a great tool, that can be installed via NPM. Grunt is a task runner and can be used to control any tasks that may need run locally for said app. It can be used to re-run builds once a source file has been saved, it can be used to control many other task runner or test runners like karma, jasmine_node or selenium and protractor, which we'll find out about later.

5.3.2 Jenkins

Jenkins is an open source automation server, and is a pretty powerful tool. You can setup a variety of tasks, whether it's working on the local machine it is running on, set up slave machines to run tasks and processes on or pull/copy/deploy to/from many other places.

Jenkins is a great continuous integration tool used for automating any project no matter the size. You can set up multiple jobs/tasks to clone your repository, build and install it, run tests and if and only if they all run successfully you can run another job to deploy your app to another server.

5.4 Testing Frameworks & Tools

Multiple Frameworks were used for testing, as each method of testing would have to be carried out and documented. Ranging from client side testing, server side testing and end to end click through testing.

Black box testing

Is the testing of a piece of software, where the expected output is testing without having to know what the internals that the system does. Example End to end testing of a web app: you will test clicking through buttons to see if the expected output shows up, and not worry about how it does it.

Uses: Acceptance Testing, System Testing

White box testing

This type of testing will test the internal structure/software of any app. This testing can also be considered unit testing, where each method and statement is tested. With white box testing the implementation must be known and knowledge of the language the source code is written in.

Uses: Unit Testing, Integration Testing

5.4.1 Jasmine

Jasmine is the language used for writing the tests, it is a version of JavaScript called behavior driven JavaScript. Jasmine allows the developer to create different suits for each test if needed. Jasmine has a simple way of creating tests “it” function for the actual test and then using “expect” something to be something function.

5.4.2 Karma

Karma is a Test runner for client side unit tests. Karma will launch a browser instance of your choice (headless, Firefox, Chrome etc.) and run the tests on that. It will display the output on the command line window.

Karma can also be integrated into Grunt, Jenkins, Travis and many more. Karma can be configured to run continuously, this means as you are editing code, save a file it will pick this up and re-run all the tests and provide the output. Output can also be contained in a html file to show coverage; output can be configured for output type also.

5.4.3 Jasmine_node

Jasmine Node is the server side equivalent of Karma for server side unit testing, this framework will run the server side tests and output to console. Jasmine Node also provides coverage. Although Jasmine was initially made for client side testing, it is quite powerful at mocking, and this can be of great advantage for server side testing.

5.4.4 Selenium & Protractor

Protractor is the test runner for this type of test. These tests are used to do click thoughts of the app. For example, go to URL, click a button, and then expect text to change colour and something to pop up etc. These are called End to End tests.

Selenium here is used as the webserver, while Protractor is the test runner. Like karma, selenium can create different instances for each test, Chrome, Firefox IE11 etc.

5.5 Source Control

Source control is going to be the home of all the code. Where each developer will access the code, and update it. Source controls provide great control over who can edit what and when, the history of each edit and structure of the source files.

5.5.1 GitHub

GitHub is a great example of a common source control, it has taken off over the last few years and now has over 20million users, and is comparable to subversion, although GitHub is decentralized and each developer, when they make a clone of a repository will have it stored locally. This allows developers to create different branches in their own local branch and then merge it back with the master branch (root branch).

With this local copy of the repository it also comes with all the history, all the previous commits etc. And on top of that since you have it stored locally, you may work on it offline and do not need a constant connection to the centralized version stored in GitHub.

5.6 Servers (VMS)

There were many options open to me, in regards to acquiring a virtual machine or server. I had a choice of:

- NUIG to provide me one
- Amazon web services
- Microsoft Azure

Although these are all good services, the one I could obtain by NUIG, would be free, but I would not have root access. This was crucial to me since this project would need at least 1 or 2 servers and access to install any necessary programs and use of ports.

As a student Amazon Web services allows students a year's free access to VMs, databases etc. although only micro instances, enough for a project so this was perfect. Microsoft Azure was also an option, but this would only provide me with 3 months of free services.

5.6.1 Amazon EC2

Amazon EC2 is amazon's web services platform for controlling virtual machines, it is super easy to spin up and shut down as much machines as you would like. Under the free tier that I got allowed me to have 1 t2 micro instance for a full month for 12 months, this was good but I could use 2 instances, as long as I did not go over the prescribed usage then I would not have to pay for them, although it is very cheap for a t2 micro instance this exactly what I was looking for.

When you create an instance, you are presented with all various operating systems and you may choose which type you want which was also a plus.

5.7 Database

Somewhere that all data will be stored, allowing different access to different users is also great.

5.7.1 MongoDB

I used MongoDB provided by NUIG as a student I was eligible to get a database and use it for this project. I could have used Amazon Web Services, but I had already this set up before I found out about Amazon EC2.

MongoDB is an open source document oriented database. This allows you to store objects within it, it is termed NoSQL, meaning that this database is not a traditional relational database.

5.9 Languages

These are the languages that will be used during this project.

5.9.1 JavaScript

JavaScript is a high-level object oriented programming language; it is widely used in web browsers. In this project, it is used in almost everything, from client side to server side since both angular and Node both use JavaScript. Also, all the testing frameworks use jasmine, which is a JavaScript testing language. And finally, all the test runners, task runners, grunt, Jenkins, karma, protractor etc. use JavaScript.

5.9.2 Html

Html is hypertext Markup Language which is used for web pages. It uses a standardized tagging system which can take multiple attributes to change things like colour, size, placement etc.

Html will be used to create the web pages.

5.9.3 CSS

CSS is a styling language that ties in with JavaScript and Html, to give the web pages a style, colours, placement of elements etc.

6 Implementation

6.1 Prerequisites

Must have prior knowledge of JavaScript at a beginner's level, and an understanding of MVC (Modal View Controller)

As JavaScript is widely used throughout this project for creating the back end, front end of the app, the testing and most of the continuous integration.

Also, some knowledge of Linux and have used command line before. This will be needed to SSH into the various virtual machines and there will be no display.

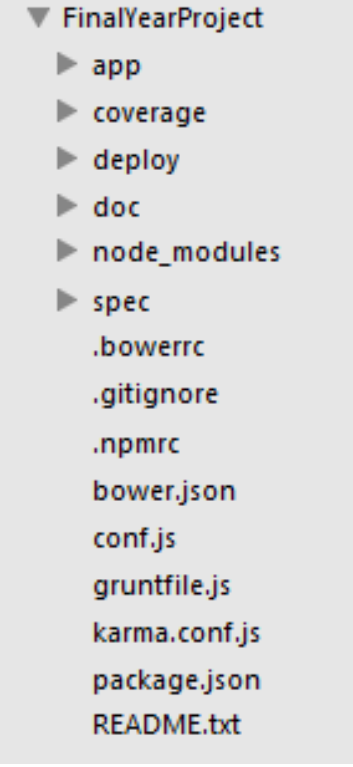
6.2 Project structure

Order:

- Setting up Node Server.
- Setting up Angular Web App.
- Create Git Repository, link with app.
- Set up automation for development
 - o Grunt
- Configure building tasks for app (browserify)
- Start on tutorials, this will be continued as more work is done.
- Set up client side test framework, karma
- Set up server side test framework, jasmine node
- Set up end to end test framework, protractor
- Integrate test automation into grunt
- Boot up 2 Linux VM instances
- Set up Jenkins on one VM
- set up Jenkins slave (hosting VM)
- Set up Node and host app on other VM
- Create Jenkins tasks:
 - o Build
 - o Test
 - o Deploy
 - o Nightly
- UI for app
 - o Simple tutorial directives
 - o Load in tutorials with nav bar
 - o Embedded plunker & jsfiddle
- Set up D3.js
- Creating data and force graphs
- New force graph for each

6.3 Main package structure

This is the file structure of the app, there are many folders, for the app source code, output from coverage, test spec files, and dependencies for the app.

	<ul style="list-style-type: none">- <u>App</u> In here all the source code can be found, there will be a public folder for anything that will be publicly available for the app. Along with any publicly available libraries. - <u>Coverage</u> This is the output from running the tests, This can be in the form on a .json file, html file or otherwise. Generally, it will be outputted as .html which will give an intractable page, in which the developer can see what is being tested and how often. - <u>Deploy</u> This folder will store any files that are for deployment. As there will be a Jenkins server which can handle this, it may be necessary to deploy the app locally. - <u>Doc</u> This folder will store all the tutorials; this will be there for anyone wanting to clone this repository and browse the tutorials
--	--

- Node Modules

This Folder will store all the dependencies that NPM will install

- Spec

This folder will house all the test files

- .bowerrc, .npmrc

These are the config files for Npm and bower, these files will detonate where each manager will install its packages and any other config

- .gitignore

Config file for Git, useful to outline what files should be ignored for committing, since we do not want to add the dependency files and folders into the git repository and force the user to install them themselves a gitignore can be used.

- Bower.json, package.json

These 2 files are the .json config files in which the dependencies are specified for Npm and bower. These files also keep track of version of each dependency and type.

- Gruntfile.js

This is the config file for grunt, all tasks will be found inside this file. Grunt will take place of the main task runner of the app, from grunt we will call our other test runners via grunt plugins. This will streamline how tasks will be run as only one task runner will be used/called.

- Conf.js

This is the config file for protractor and selenium, this file will store locations of all the test files that are for End to End tests among any other relevant config.

- Karma.conf.js

Karma config for all client side tests. This config will store locations of all client side tests/specs and will handle any config for karma test runner.

6.4 Setting up Node Server

First off a new empty folder was created. Node was installed and then a node project was created via nodes “npm init” command. This will create the file “package.json” and ask a few questions on the configuration of the app.

Next express was installed via Npm and a “server.js” file was created and the code to start a node server was added. Once the URL was accesses it would return an index file.

```
app.set('view engine', 'ejs');
app.set('views', __dirname + '/../client/src');

app.use(express.static(path.join(__dirname, '/../client/public')));

|
app.get('/', function(req, res) {
  res.render('index');
});
```

Figure 4 Server.js file snippet

Folder structure then began taking shape, creating separate folders for anything related to server and anything related to the client.

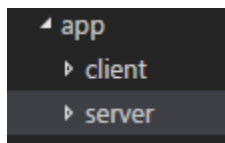


Figure 5 High level app folder layout

6.5 Setting up Angular

Since Node had been set up, there now had to be a file to be rendered, this is where the Angular side kicks in, Bower was installed via Npm, as this could then be used to keep track of client side dependencies and then angular was installed via Bower. I configured bower to install its dependencies into the public folder for the app.

A “ejs” file was made and called “index.ejs” this will be the root file that will be rendered when the node server is requested to render the app.

Since we are using angular, angular must be included into the file so it will be loaded into the browser. Next I created the starting point of an angular module app. This will be the entry point for the angular within the browser.

The app is now ready to start with a minimal node server and angular app.



Figure 6 Setting up angular dependencies

6.6 Setting up testing Frameworks

To set up testing frameworks I would have to consider what is available, what advantages and disadvantages there were for each. I would need something that could provide proper unit testing, mocking and black & white box testing.

6.6.1 Client testing Library (Unit Testing)

First of I considered client side unit testing, there were 2 packages that stood out to me, these were Jasmine & Mocha, which are both JavaScript testing libraries, they are both very similar in many ways. Mocha is a test runner, but mocha does not come built in with an assertion libraries, this means you must install additional packages to get the same functionality as Jasmine.

Both testing frameworks use the same style/approach when it comes to writing the tests, each library with use the same syntax for setting up test suites and tests. Example below:

```
describe("Suite name", function() {  
  it('Test name', function() {  
    //test code  
  });  
});
```

Figure 7 Jasmine Test

Both libraries can be used with Karma, a test runner in which will allow the tester to test on chrome, Firefox any many other browsers.

Jasmine provides an Assertion library which means having to load less libraries when testing, thus decreasing the number of dependencies on your app. Regarding Mocking, Jasmine wins again with providing a mocking library "jasmine spies" while mocha does not, although there are many libraries out there in which mocha could make use of. It just seems like a few steps too many to have to keep track of all the dependencies mocha will require, and if in the future will they all work together with new updates. It seems like the smarter idea to go with jasmine since it is more streamlined and less complex.

Mocha is more useful for very refined testing, and if you do not mind the additional complexity involved in finding libraries and setting up testing, then jasmine is the way to go.

I chose to install jasmine, as this could be easily integrated into all different types of testing, client side, server side support and can be used for integration testing as well as unit testing. It provides good mocking, spying and assertion libraries.

6.6.2 Client Side Test Runner (Unit Testing)

As I have picked Jasmine, this led me to Karma, which is a test runner, Karma is widely used with jasmine, which makes a nice all round setup for client side tests. Karma can be configured to run what tests you wish to include, it allows to run tests manually or continuously, which means that if it notices any files that it has included it will re-run tests specified.

Karma allows for multiple browser testing, this can be edited in the configuration file to include Chrome, Firefox among others. Karma also supports headless Brower testing, this means that when karma instance

launches it will not create a browser UI instance, but will run the tests in the background. It also allows for concurrent browser testing.

I installed karma, configured it to run all my tests on chrome, as I believed this may be the most used browser and would be a good example to show testing being run on this app. I also set it up to run continuously, so that I could have the tests run in the background as I was editing the source code, to ensure I did not break anything unintentionally.

Karma also has a configuration for reporting, this means the console output can be customized to fit the tests or the developer testing, you can disable it, which means it could potentially run faster, or customize it to give a read out of every test name, suite name and whether it passed or not. It can also be configured to show errors/warnings messages.

```
$ 10 0.0.0) - mainController Tests:
$ 10 0.0.0)    Expect the section to be defaulted to 0 PASSED
$ 10 0.0.0)
$ 10 0.0.0) - NavBarController Tests:
$ 10 0.0.0)    Expect the section to be defaulted to 0 PASSED
$ 10 0.0.0)    change the tab to be home PASSED
$ 10 0.0.0)    check is tab is active when not selected PASSED
$ 10 0.0.0)    check is tab is active when not selected PASSED
$ 10 0.0.0)
$ 10 0.0.0) - Hello world:
$ 10 0.0.0)    says hello PASSED
$ 10 0.0.0): Executed 6 of 6 SUCCESS (0.012 secs / 0.041 secs)
```

Figure 8 Output from Karma testing

```
colors: true,
logLevel: config.LOG_DEBUG,
autoWatch: true,
browsers: ['Chrome'],
singleRun: true,
```

Figure 9 Karma configuration snippet

colours: this will determine whether the console output will display colours or not, this is useful to determine if the test passes or not at a quick glimpse.

logLevel: this can be customized to display the amount of logging for console output options range from:

```
LEVEL OF LOGGING
config.LOG_DISABLE
config.LOG_ERROR
config.LOG_WARN
config.LOG_INFO
config.LOG_DEBUG
```

Browsers: This option determines what browsers it will run the tests against.

singleRun: will determine if the tests run once or will wait in the background for any changes made and then run the tests again.

6.6.3 Client Tests

I began with simple tests, for each angular component, so I could integrate this into some tutorials and walkthroughs to get a basic knowledge of how to test and setup angular client side tests.

Since there are different methods of injecting, and obtaining different components such as controllers, factories and directives.

I started off with testing my controllers, this is what gives my app the functionality or business logic if you wish, this small snippet below is almost like other angular components. This Code will acquire the actual app itself, without this I cannot access any of the apps modules or functions. It then will inject into the app to obtain important modules, like below I am creating a scope to test on, getting hold of the http module and the controller service, this will allow me to test my functions I have within my controller and any variables that may be access by the scope. (code snippet below)

```
describe("test name", function() {
  beforeEach( module('app') );
  beforeEach(inject(function ($injector, _$controller_) {
    $location = $injector.get('$location');
    $location.path('/');
    $httpBackend = $injector.get('$httpBackend');
    $rootScope = $injector.get('$rootScope');
    $scope = $rootScope.$new();
  }));
  it('Expect the section to be defaulted to 0', function() {
    expect( $scope.currentPage() ).toBe(1);
  });
});
```

Figure 10 Controller test snippet

I continued with testing other modules and angular components, to increase the coverage and to show the different ways in which they can be tested.

6.6.4 Server Side Testing (Unit Testing)

I found it much harder to find testing frameworks for node, compared to testing Angular and client side JavaScript. I came across a library called “jasmine_nodejs” this is an extension to Jasmine, allowing it to test node.js.

Since this uses the same syntax and style of tests as my client side does, as they both use Jasmine this was perfect for me.

I was also able to install a grunt plugin, meaning I could tie this in with my task manager and extend my testing automation further. This gives similar output to karma, as it displays the number of tests, whether they passed or failed and the description of test suites and specs. (as example can be seen below)

```
>> Executing 1 defined specs...  
  
Test Suites & Specs:  
  
1) Hello world  
   ✓ says hello  
  
>> Done!  
  
Summary:  
  
Suites: 1 of 1  
Specs: 1 of 1  
Expectations: 1 (0 failures)  
Finished in 0.004 seconds  
  
>> Successful!  
  
Done.
```

Figure 11 Basic Jasmine_nodejs console output

6.6.5 Integration Testing (End to End Testing)

This type of testing will test the app/system as a whole, as opposed to the client and server unit tests only testing functions. No knowledge of the inner system or function is needed to be known.

This type of testing must be able to create a webserver instance and direct itself to the apps URL, (the app may be hosted locally, or this tests can be pointed towards an app that may be hosted elsewhere).

I found and used Protractor, “Protractor is a Node.js program, and runs end-to-end tests that are also written in JavaScript and run with node. Protractor uses WebDriver to control browsers and simulate user actions.”

(Taken from: <https://docs.angularjs.org/guide/e2e-testing>).

Protractor also uses Jasmine which was a plus, as all testing frameworks use the same testing library, this means that for industry and for developers working on projects would have to learn the one framework, and not multiple testing frameworks.

I installed and set up protractor, protractor uses a config file that provides additional configuration, and also a grunt plugin which also adds extra functionality for test automation with integration with grunt.

Example snippet below of config file for protractor. This demonstrates setting the browser I used to launch the web driver with, and test the app on. You can also see here that jasmine is the selected testing framework, it may be possible that chai or other frameworks may work as I came across when considering client side testing frameworks. But in this case I went with Jasmine since I am using it for the other types of testing. Finally, I included a configuration option for protractor to locate the tests to run. Any additional specs are added, (as this is a simplified version)

```
// An example configuration file.
exports.config = {
  // directConnect: true,
  capabilities: {
    'browserName': 'chrome'
  },
  // Framework to use. Jasmine is recommended.
  framework: 'jasmine',
  // Spec patterns are relative to the current working directory
  when
  // protractor is called.
  specs: ['spec/e2e/test.js'],
  // Options to be passed to Jasmine.
  jasmineNodeOpts: {
    defaultTimeoutInterval: 30000
  }
};
```

Figure 12 Protractor config

6.6.6 Coverage

Coverage is a nice quick and easy way to provide metrics and reports, of what has been tested, what needs to be tested, how many times each line of code is hit when the app is launched, and the complexity of the app via the number of branches and lines of code.

This method is good for industry and this can help teams, and managers keep track of what is going on during development. It is also a great way to ensure complete testing as it may show paths and routes through functions that may be unthought off.

I configured karma to provide HTML coverage output, it could also provide a .json or other types if required, but for the most part I thought html was the most relevant, for developers and supervisors alike, since it will give them a great view of how much is tested, the complexity and what is missing.

I did this by adding the configuration in the karma.conf file. And installing a pre-processor, the pre-processor will go throughout the code base and tests and process it so it may be ready to create output for code coverage. (below is a code snippet to configure karma to provide coverage output)

```
preprocessors: {
  'app/client/public/app.js' : ['coverage'],
},
coverageReporter : {
  type : 'html', //or json
  dir : 'coverage/client'
},
```

Figure 13 Karma coverage config snippet

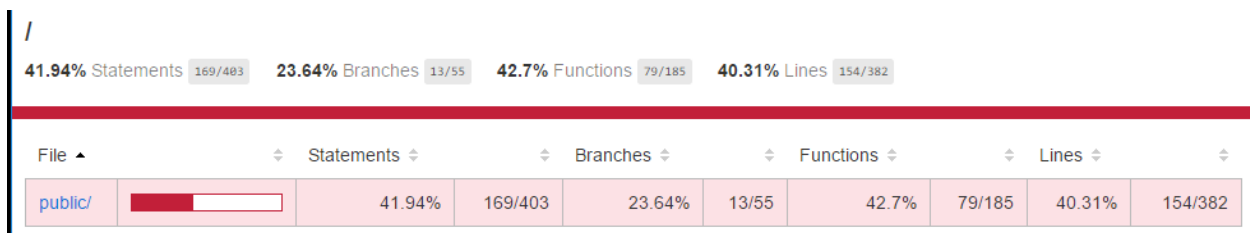


Figure 14 Example output of coverage html

```
$scope.showTab = function(num){
  E if($scope.currentPage() == num){
    return true;
  }else{
    return false
  }
}
```

Figure 15 Example of actual coverage, showing 1 of 2 paths tested

6.6.7 Writing Tests

All test frameworks used in this app are written in jasmine, each type of test, client, server or integration use different test managers, although they are based on the same testing framework. Each test starts out with declaring a test suite. Jasmine is a variation of JavaScript that is used for testing Angular and Node.

Jasmine structures its tests like a folder on a computer, you have 1 folder which may have other folders within and may also have tests within. In jasmine, we relate these “folders” to “describe” blocks and the files to “it” statements.

Then our Expect Statement will be the actual test, we may have multiple expect statements in one “it” test, or we may also multiple “it” in our “describe” also it is possible to have multiple describes inside “Describes”.

```
describe("suite name", function() {
  | it('testName', function() {
    //test
    expect( $scope.showTab(1) ).toBe(true);
  });
});
```

Figure 16 Sample setup for a jasmine test

Client-side unit tests

Client side tests are done differently from server-side, the tests are run on a node instance (karma) of the app, so we must use the service \$injector to get hold of what service we would like to test. As you can see in the example below, there is a “beforeEach” block, this means that this part of code will be executed before each test. This is a good practice, although it may take a little bit longer to execute, but since the test may be changing scope variables or mocking various functions, then we do not want to use a used function that another test used to reset variables and so on.

```
beforeEach( module('app') );

beforeEach(inject(function ($injector, _controller_) {

  $location = $injector.get('$location');
  $location.path('/');

  $httpBackend = $injector.get('$httpBackend');
  $rootScope = $injector.get('$rootScope');
  $scope = $rootScope.$new();

  $controller = _controller_;
  $controller = $controller('navController', {$scope: $scope});

}));

it('Expect the section to be defaulted to 0', function() {
  expect( $scope.selected() ).toBe(1);
});
```

Figure 17 Example client side unit test

Server-side Unit tests

Server side tests are run differently to client side tests. But follow almost similar setup in a test. It will use the same layout, describe, it beforeEach etc.

With node.js all files are “require” by each other, this is like testing the files. Instead of using services and injecting to get certain angular components. You must include the file at the top of each test file with require(filename)

```
var file = rewire("path_to_file");

describe("test suite", function() {
  it("the test", function() {
    expect(file.doSomething).toBe("cool");
  });
});
```

Figure 18 Server side jasmine test example

Integration testing

With integration tests, Jasmine is used also, this overlap of jasmine being used for all kinds of tests, is a great addition, this means that developers do not have to learn multiple frameworks for testing, which in turn will mean faster development cycles and more flexible developers.

```
describe('test suite name', function() {

  beforeAll(function() {
    browser.get('localhost:8080');
  });

  it('simple test', function() {
    expect("something").toEqual("something");
  });
});
```

Figure 19 Jasmine Integration test example

Above is an example of a simple integration test, the test must first request the URL of the page before it can even begin to do integration tests.

6.7 Git integration

I needed somewhere to upload my code base to, so that my build server and hosting server could access the source code, and with Github becoming ever so much more popular I decided on it. A source control was also an integral part in the end to end development of an enterprise software application, since the source code must be hosted somewhere and allow for many developers to work on it at the same time.

6.7.1 Creating git project

I created a git project once I had the initial start to my Angular app up and running, this will allow me to keep track of commits, timeframe and share my code. As this project is meant to be an end to end development and a starter app in angular and node. This would allow any future developers to clone my repo and experiment for themselves, and merge additional changes and functionality.

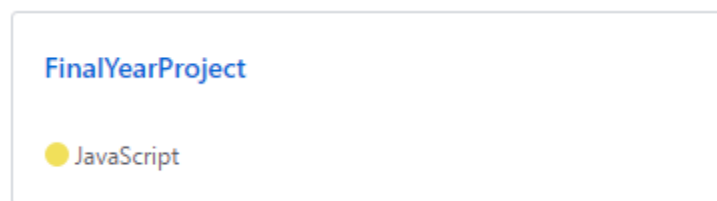


Figure 20 Screenshot from Github Showing project Repository

61 commits		1 branch		0 releases		1 contributor	
Branch: master ▾		New pull request		Create new file	Upload files	Find file	Clone or download ▾
JamesOMeara adding report so far				Latest commit 10d7efb 4 days ago			
app	adding report so far			4 days ago			
deploy	testing deployment on jenkins			29 days ago			
doc	adding report so far			4 days ago			
spec	corrected tests			2 months ago			
.bowerrc	added bootstrap			5 months ago			
.gitignore	edited gitignore and working on build script			a month ago			
.npmrc	tweaked browserify, corrected issue with building the 3 files - app -...			2 months ago			
README.txt	updated force graph with additional nodes and mouseover to show neigh...			18 days ago			
bower.json	edited gitignore and working on build script			a month ago			
conf.js	added e2e testing framework'			2 months ago			
gruntfile.js	added ability for force graph for each tutorial topic on click menu an...			12 days ago			
karma.conf.js	tweaked karma and tested both controllers			2 months ago			
package.json	added karma headless testing task			28 days ago			

Figure 21 Github Repo of This Project

6.7.2 Linking with Jenkins

An additional feature Github provides, is that once a commit is made, it can create push requests to supported applications. Using this I could configure build tasks on my build server to automatically start.

I did this by Github webhooks, available via settings, I then created a new webhook. I had to supply the URL to my Jenkins instance followed by “/github-webhook/” so that Jenkins could recognize this as a push request. Then Choose what time of trigger I would like for this webhook to activate, I chose just a push event, since I only wanted my Jenkins instance to build on new code pushes.

Webhooks / Manage webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL *

Content type

Secret

Which events would you like to trigger this webhook?

☒ Just the push event.

☐ Send me everything.

☐ Let me select individual events.

☒ **Active**
We will deliver event details when this hook is triggered.

Figure 22 GitHub Webhook Configuration

6.8 Automation

Automation is key component to this application, as this is what links one component to the next, how one event can affect the next and so on.

To include automation and task management in this app I had to consider what options there were, I came across two popular task managers, Grunt and Gulp.

Both are automation tools that use node.js and use plugins to increase their versatility. Gulp is designed that each task will complete each one task at a time and complete that task very well, while on the other hand, Grunt is more suited to accomplish multiple tasks at the same time. Counting the number of plugins available for each, Grunt seems to trump Gulp by almost 2x number of plugins.

I decided to use grunt as it seems to be supported more as it has a lot more plugins available to it. It also uses a .json type like to create and define tasks, which means it is easy to keep track of tasks, keep tasks separate and ultimately build upon them.

So, I installed Grunt via NPM, and any additional plugins that I needed to interface with my various other automation tools (karma/protractor etc.)

6.8.1 Grunt build tasks

Grunt Build Task

With Grunt installed I could start automating tasks, I began with a task that would build the app, since the app is using angular it is spread out over many files, different modules and so on. To run the app, you must include each file for the app in the index.js. To avoid this I bundled my app into one single file. This means that in my index.js I would only need to specify one file “app.js”.

I used a grunt plugin called “browserify”, browserify would allow me to bundle whatever modules/packages and files I wished into a single JavaScript file. This was super useful and exactly what I was looking for. First off I installed it and bundled everything into one file, the source code all 3rd party dependencies and all the templates.

I found that when I used this approach later it would slow down certain build tasks, for example if one line was changed on a source file then I would have to rebundle every dependency and template. A way I thought to get around this was to bundle 3 files instead of one.

This would mean that I could bundle:

- Source code
- Templates
- Dependencies

Into 3 separate files. This then meant that when building I was not forced to recreate each file, and if something was changed in the app source code then I was only required to rebuild that one file. This meant much faster building times for development.

Below is a view of all the files that would have to be specified into the index.js, on the right is the bundled app.js that has been built via grunt browserify.

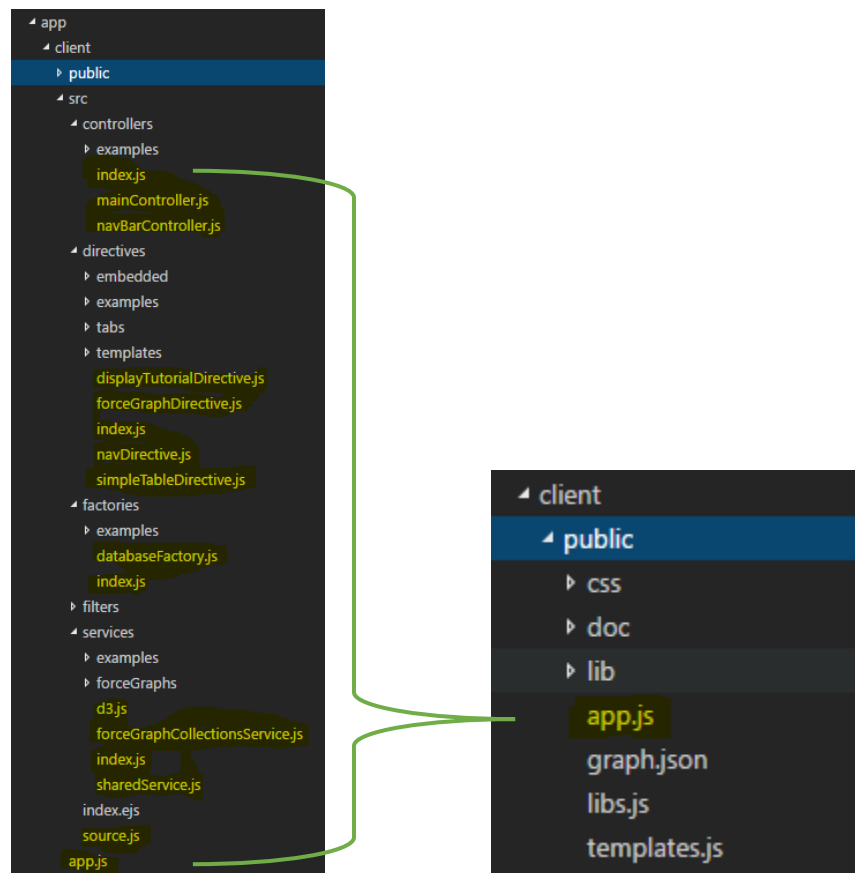


Figure 23 Browserify example

Grunt Watch Task

Another task I added to grunt was, a task that could fire off certain tasks when certain file(s) changed, (this task would run continually in the background/separate command window). This meant that I could implement different tasks for each type of file saved.

For Example: saving a source JavaScript file, I could call upon the build task for bundling the app.js file, then continue to run all the unit tests again. Compare this with saving a template, I could run the bundle task for creating the templates.js file and re run the end to end integration tasks to ensure that no other html or functionality was broken.

```
watch: {  
  build_app:{  
    files: ['./app/client/src/**/*.js'],  
    tasks: ['build_source']  
  },  
  build_dependency:{  
    files: ['./app/client/public/lib/**/*.js'],  
    tasks: ['build_dependency']  
  },  
  build_template:{  
    files: ['./app/client/src/**/*.ejs'],  
    tasks: ['build_template']  
  },  
  test_client:{  
    files: ['./app/client/src/**/*.js'],  
    tasks: ['test_unit']  
  },  
  test_server:{  
    files: ['./app/server/**/*.js'],  
    tasks: ['test_server']  
  },  
  test_integration:{  
    files: ['./app/client/src/**/*.ejs'],  
    tasks: ['test_e2e']  
  },  
},
```

Figure 24 Grunt watch automation task

The Figure above shows the grunt watch task, it shows each task that will watch for change within specific files, and each task will have its own task to run, if it detects change in said file(s).

6.8.2 Grunt Test Automation

Karma

Karma is test automation tool, I Briefly described earlier how it can be used to automate unit testing, although in the previous section I have the test task along with the “grunt watch”, I did originally have less grunt watch tasks, and have karma running continuously in the background in a separate window. This did prove useful however when working with the app, but a lot of the time it was unnecessary as you would require a second or third command window and must have a karma instance running continuously.

The upside of adding this unit testing to grunt watch, meant that if you were working on the source code this works well as you only should have one thing running, but on the other hand if you were to concentrate solely on testing then running karma continuously via grunt with a “grunt test_unit” command I have set up will ensure a karma instance is created and will continually look out for changes in all tests or source code of the app.

```
karma: {
  unit: {
    configFile: 'karma.conf.js',
    port: 9999,
    singleRun: true,
    browsers: ['Chrome'],
    logLevel: 'ERROR'
  },
  headless: {
    configFile: 'karma.conf.js',
    port: 9999,
    singleRun: true,
    browsers: ['PhantomJS'],
    logLevel: 'DEBUG'
  },
  dev: {
    configFile: 'karma.conf.js',
    port: 9999,
    singleRun: false,
    browsers: ['Chrome'],
    // logLevel: 'config.LOG_DEBUG'
  }
},
```

Figure 25: Grunt Karma config

With the above Figure, this shows the different levels of karma instances that can be created, and that I have created for automation in my app, I have 3 options. A single run to constitute a single unit test with chrome, a single run with a headless Brower, and a mode for development, in which it will run continuously with chrome (it will re-run all tests once it detects change in any test/spec file or source file)

6.8.3 Npm

Npm is the Node package manager, the highest level of managers within the app you could say, anything installed is done through npm. Although I have grunt set up as the apps task manager, NPM also provides task manager functionality, it has commands to install packages among others, and allowing you to create your own commands like grunt.

I set up a “start” command meaning, once you have “npm install” completed you could call “npm start” which would start running the server. I thought this was fitting since it is a node package manager and it will boot up the node instance for the app, rather than using grunt. This means that grunt can be used for any other build, test, deploy task etc. This start task does call a grunt task, meaning it can be accessed at the projects highest level or with grunt which is the projects task manager.

```
"name": "final_year_project",
"version": "1.0.0",
"description": "James O'Meara Final Year Project Computer Science ID:13715519",
"main": "app/server/server.js",
"scripts": {
  "start": "node app/server/server.js",
  "startcontinuous": "forever start app/server/server.js",
  "dev": "grunt watch",
  "test": "grunt test"
},
"author": "James O'Meara 13715519",
"license": "MIT"
```

Figure 26 NPM command to start app

Above figure shows the npm configuration for setting up tasks, you can see two commands to start the app, one will create an instance running in the command window, and the other “startcontinuous” will create background process running the app locally with “Forever”, Forever is a command line interface tool that can be installed via npm that enable you to run scripts continuously.

6.9 Creating Build/Deployment & Hosting Server

I created an amazon web services account, I picked this service because, as a student I could avail of a year's free hosting service and database among other services if I so wished. Even though they were microservices, they would suffice for my project.

Initially I was seeking to host my website somewhere, and was thinking I could install Jenkins locally and launch it when I needed, but this great opportunity provided me with a chance to create as much instances as I wished.

Therefore, with that in mind I set up 2 Linux virtual machines, one for a build server and one for hosting server. This means that I could have my app running constantly on one machine, while having my build server constantly running, enabling me to demonstrate more build tasks, i.e. building on git push command, scheduled builds and tests etc.

<input type="checkbox"/>	Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP
<input type="checkbox"/>	node	i-07f560e03e9092ec5	t2.micro	eu-west-1c	running	2/2 checks ...	None	ec2-34-252-6-135.eu-we...	34.252.6.135
<input type="checkbox"/>	Jenkins	i-0c8b93fcb613ec69b	t2.micro	eu-west-1c	running	2/2 checks ...	None	ec2-34-252-7-193.eu-we...	34.252.7.193

Figure 27 Amazon EC2 instances

With having 2 fresh blank VM's (Virtual Machines) this allowed me to configure each for its purpose from scratch, I could load in images if I wished, but some cost more money, a plus this would be a great learning experience and being able to document the process for the End to End processes & development.

Creating the instances were easy to do, with the free tier I was made aware which ones that I could avail of, also Amazon provide great analytics for how much usage you are consuming. With no cost to me I could run 2 instances instead of 1 for half a month at a time to stay within my free tier usage. Towards the end I was getting tired off restarting each instance each time I set about working on the project, and soon realized running 2 instances for the full month only cost a few cents (super cheap I thought), so this allowed me to leave my instances up running 24/7.

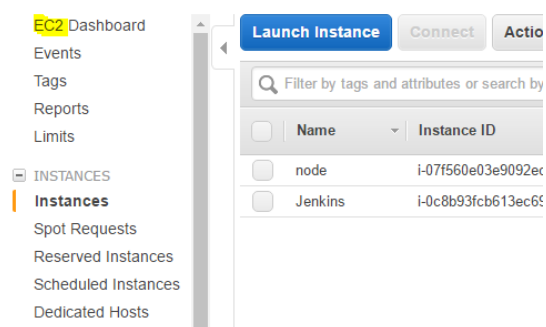


Figure 28 EC2 dashboard snippet

Amazon EC2 (VM instances section of amazon web services are named) makes it easy to configure some portions of your VM, here you can change the security groups and open ports easily. This enables me to run my Jenkins app and Node/Angular app of port 80.

6.10 Setting up VM for Node App

To set up the VM to host my app, I had to do an initial setup of file structure, creating a folder in which to store my app. I then installed Node.js (to host the app, and use npm to install app dependencies), Gti (to clone the repo), and then grunt command line globally (to be able to use my grunt commands when the repo is cloned), next I set up port forwarding and set my VM to listen on port 8080.

The basics are now setup so this would allow me to access this server through my Jenkins build machine and then deploy/build the app from there.

6.11 Setting up VM for Jenkins

My next task was to install Jenkins on linux, I followed the commands for Ubuntu outlined on the Jenkins site, and then began configuring it. The steps went from deploying Jenkins on Ubuntu, using iptables for port 80 -> 8080.

Once initial setup was complete, I logged in with my new user I created.

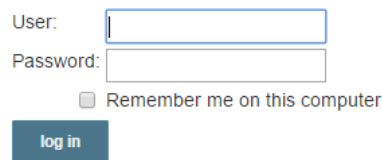
The image shows the Jenkins login interface. It consists of two text input fields: the first is labeled 'User:' and the second is labeled 'Password:'. Below these fields is a checkbox with the text 'Remember me on this computer'. At the bottom of the form is a blue button with the text 'log in' in white.

Figure 29 Jenkins login screen

6.11.1 Creating slave machine

I was originally planning to make a script that would SSH into my hosting server to deploy this app, but then with more reading on Jenkins I configured it to setup my hosting server as a slave machine. This meant that my Jenkins instance could SSH into my hosting server and have control over it.

First step to create a slave machine is to go into manage Jenkins settings and setup a new node for Jenkins to work off

-> Manage Jenkins -> Manage Nodes



Figure 30 Jenkins options panel



[Manage Nodes](#)

Add, remove, control and monitor the various nodes that Jenkins runs jobs on.

[About Jenkins](#)

Figure 31 Jenkins Manage nodes option

I then selected “New Node” and then gave the node a name to be used later. The next step for me was to fill in the description as this is what I will use to distinguish between nodes later (if I need to create any more).

Next, we will label this slave, so other jobs and tasks can select this slave to run its job on.

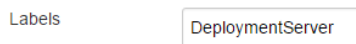


Figure 32 Labelling Jenkins slave

The final step for our slave machine now is to select the launch method, provide the Host details and credentials. These are the URL of my hosting server and the location of the public key. Once we have all this complete it will show up on the home page under “Build Executor Status” pane.



Figure 33 Jenkins pane showing slave machines

6.11.2 Creating tasks (build & test task)

After setting up the hosting server as a slave machine for Jenkins, I could potentially just go straight ahead and create a job that would use this VM and clone the git repository and then build and start the app. But this would not incorporate any testing, and would not ensure that the app would run once it is installed.

To combat this I created several tasks to clone the repository locally on the Jenkins VM instance, build it run it and then do full unit testing and then integration testing.

If all these tasks are successful, then if and only if they are all successful will the task fire off to clone and build.

I had to then install a Jenkins plugin for Git, NodeJS, this would allow me to configure specific tasks to build on push request (I covered this in the Github section setting up webhooks).

To create the tasks, since they are cloning building and testing, I kept it to 2 tasks, rather than making numerous amount of tasks/jobs in Jenkins, this means I could include everything to do with building and testing it locally on the Jenkins VM, while the other task would deal with everything related to deploying it to the hosting server and restarting the app.

To do this I began creating tasks, the first one I created was the local build and test, in which I needed the task to interface with Git, so that this build trigger would be connected to the previously mentioned webhooks,

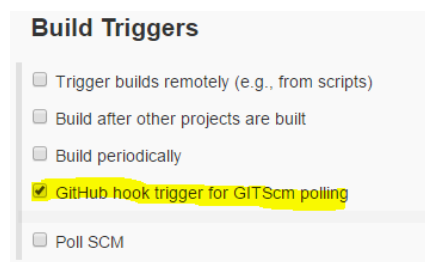


Figure 34 Jenkins job trigger via github

Next part of configuration was to configure the job to clone my git repository and then build it, and then run all necessary tests.

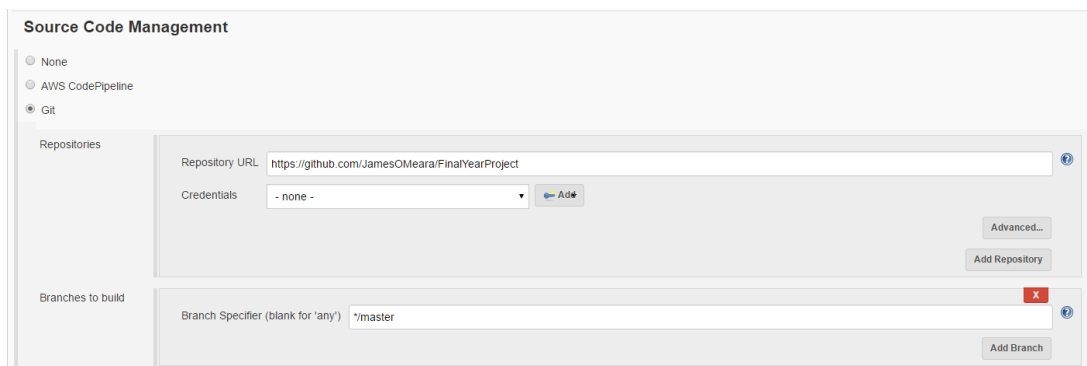
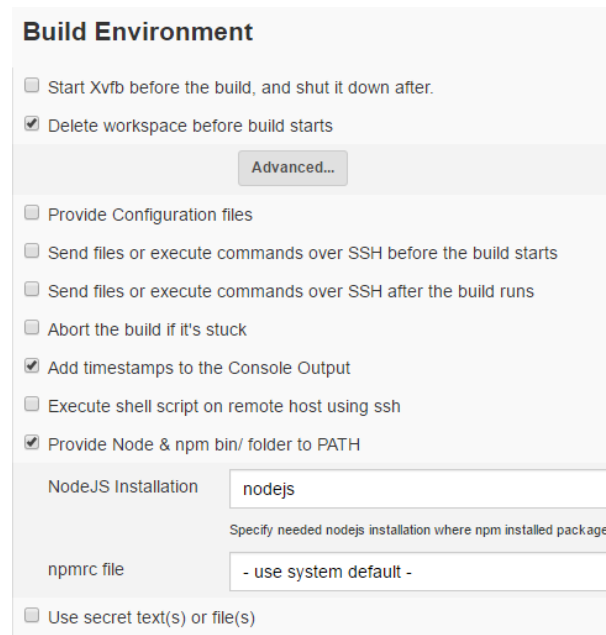


Figure 35 Jenkins config: clone git repo

Some additional configuration, I wanted to delete the old workspace, so this meant every time it cloned and installed it was a fresh install each time. This was done by: selecting “delete workspace before build starts” and enabling node for this task.



The image shows the 'Build Environment' configuration page in Jenkins. It features a list of checkboxes for various build options. The 'Delete workspace before build starts' checkbox is checked. Below this is an 'Advanced...' button. Further down, there are more checkboxes, including 'Add timestamps to the Console Output' which is checked. At the bottom, there are two dropdown menus: 'NodeJS Installation' set to 'nodejs' and 'npmrc file' set to '- use system default -'. A 'Use secret text(s) or file(s)' checkbox is at the very bottom.

Build Environment

- ☐ Start Xvfb before the build, and shut it down after.
- ☒ Delete workspace before build starts

Advanced...

- ☐ Provide Configuration files
- ☐ Send files or execute commands over SSH before the build starts
- ☐ Send files or execute commands over SSH after the build runs
- ☐ Abort the build if it's stuck
- ☒ Add timestamps to the Console Output
- ☐ Execute shell script on remote host using ssh
- ☒ Provide Node & npm bin/ folder to PATH

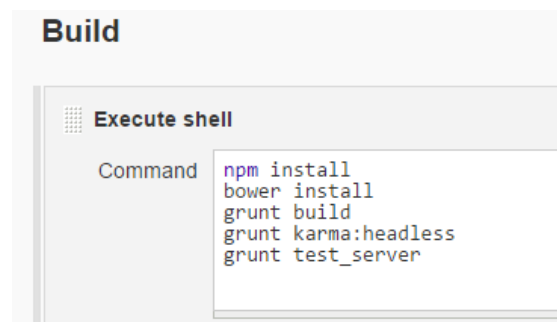
NodeJS Installation: nodejs
Specify needed nodejs installation where npm installed package

npmrc file: - use system default -

☐ Use secret text(s) or file(s)

Figure 36 Jenkins config, delete workspace and enable nodejs

Then finally issuing the commands.



The image shows the 'Build' configuration page in Jenkins, specifically the 'Execute shell' section. It contains a text area with the following commands: 'npm install', 'bower install', 'grunt build', 'grunt karma:headless', and 'grunt test_server'.

Build

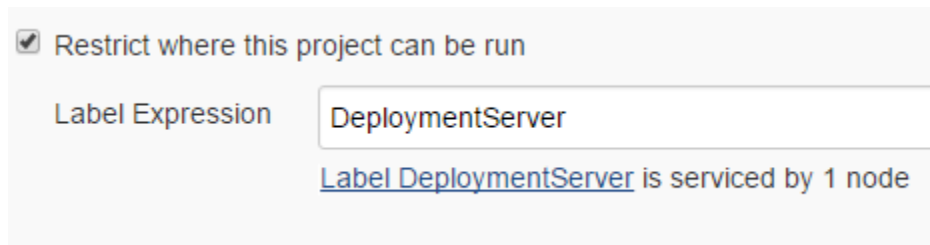
Execute shell

Command: `npm install
bower install
grunt build
grunt karma:headless
grunt test_server`

Figure 37 Jenkins build commands

6.11.3 Creating tasks (deploy & restart task)

This task was pretty like the build and test task, as this task will be restricted to run on the hosting server. This was done by selecting my slave machine I had previously setup.



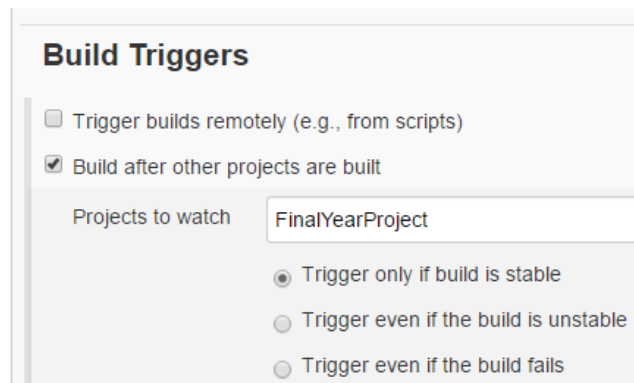
☒ Restrict where this project can be run

Label Expression

[Label DeploymentServer](#) is serviced by 1 node

Figure 38 Jenkins Build task select slave to run task on

The next step is also different than the previous job, this job will trigger once the build and test task has completed, just like the build task triggered when git sent a push request via webhook. To do this I had to simply select the option to trigger “build after other projects are built”.



Build Triggers

☐ Trigger builds remotely (e.g., from scripts)

☒ Build after other projects are built

Projects to watch

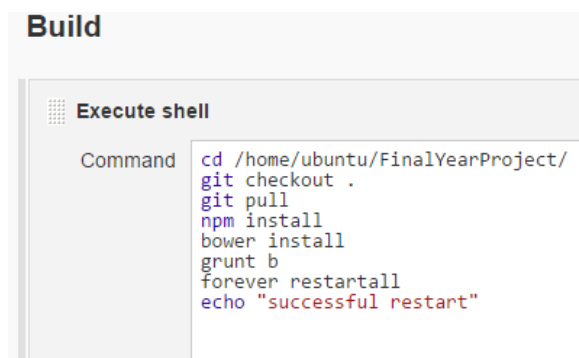
☒ Trigger only if build is stable

☐ Trigger even if the build is unstable

☐ Trigger even if the build fails

Figure 39 Jenkins Build trigger from other successful job

Then the tasks that will be run on the slave machine.



Build

☒ Execute shell

Command

```
cd /home/ubuntu/FinalYearProject/  
git checkout .  
git pull  
npm install  
bower install  
grunt b  
forever restartall  
echo "successful restart"
```

Figure 40 Jenkins Build commands for hosting vm

6.11.3 Creating tasks (Scheduled Task)

The aim of running a scheduled build is to, run some task or job periodically, although if you run unit tests and integration tests every push or commit, then you may wonder why do you need more tests, but you can never have enough tests, if you run the tests periodically and if it keeps track of the load generated, time, how much processes were running it could give you more of an idea of problems that may occur running something continually over a certain time period.

This is done by marking a build trigger with the build periodically button.

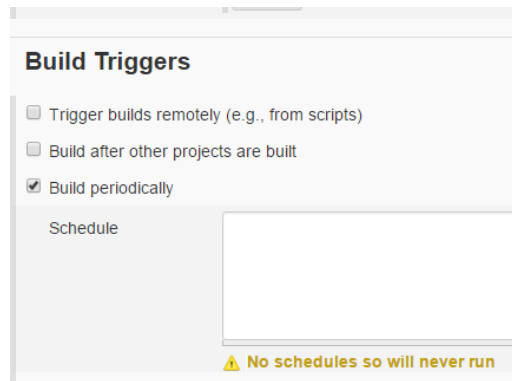


Figure 41 Setting built triggers for periodic build

Examples:

```
# every fifteen minutes (perhaps at :07, :22, :37, :52)
H/15 * * * *
# every ten minutes in the first half of every hour (three times, perhaps at :04, :14, :24)
H(0-29)/10 * * * *
# once every two hours at 45 minutes past the hour starting at 9:45 AM and finishing at 3:45 PM every weekday.
45 9-16/2 * * 1-5
# once in every two hours slot between 9 AM and 5 PM every weekday (perhaps at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)
H H(9-16)/2 * * 1-5
# once a day on the 1st and 15th of every month except December
H H 1,15 1-11 *
```

Figure 42 Examples of times for peroidic builds config

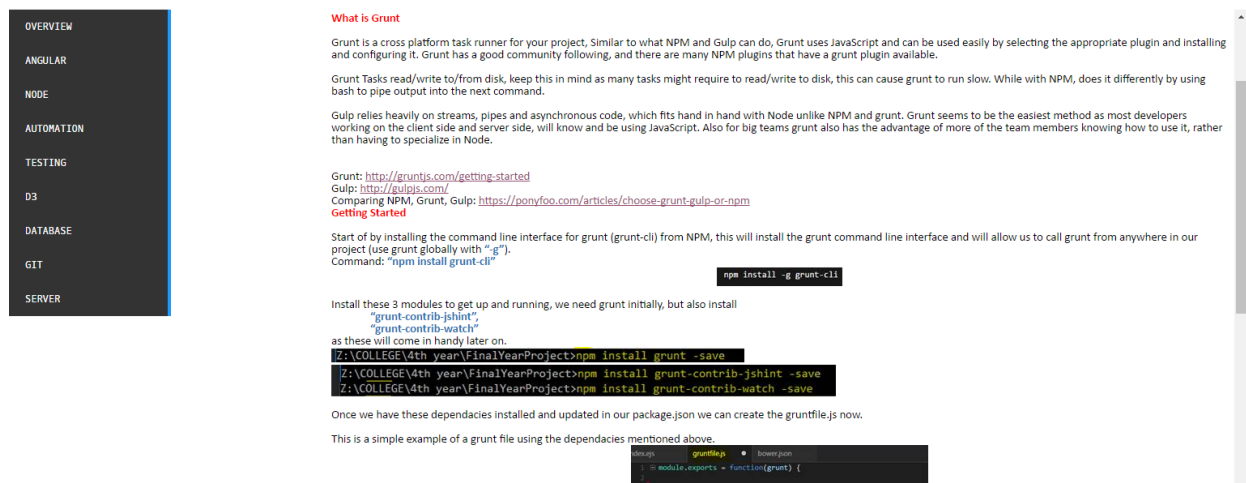
6.12 Tutorials

Part of the aim of this App/project was to create some tutorials that would demonstrate how to set up an app to a basic level in regards to full end to end development. These tutorials should be short and concise, and easy to read.

I used Microsoft Word to write them up, and then as I added more functionality and set up more elements, I had to figure out a way to embed these tutorials into the app, for ease of access and readability. Although an aspect of this app is that it is meant to be downloaded/cloned from Git, the tutorials will be accessible within the project, but also available online with the sample app.

I did this in a way that I could save the word files as html files, this meant I could place these files in the public folder of the app and use angular directive to dynamically load each file on request by what tab may be clicked.

The tutorials will include any links that were found of interest and included some screenshots and code snippets to help following the instruction that bit easier.



What is Grunt

Grunt is a cross platform task runner for your project, Similar to what NPM and Gulp can do, Grunt uses JavaScript and can be used easily by selecting the appropriate plugin and installing and configuring it. Grunt has a good community following, and there are many NPM plugins that have a grunt plugin available.

Grunt Tasks read/write to/from disk, keep this in mind as many tasks might require to read/write to disk, this can cause grunt to run slow. While with NPM, does it differently by using bash to pipe output into the next command.

Gulp relies heavily on streams, pipes and asynchronous code, which fits hand in hand with Node unlike NPM and grunt. Grunt seems to be the easiest method as most developers working on the client side and server side, will know and be using JavaScript. Also for big teams grunt also has the advantage of more of the team members knowing how to use it, rather than having to specialize in Node.

Grunt: <http://gruntjs.com/getting-started>
Gulp: <http://gulpjs.com/>
Comparing NPM, Grunt, Gulp: <https://ponyfoo.com/articles/choose-grunt-gulp-or-npm>
Getting Started

Start off by installing the command line interface for grunt (grunt-cli) from NPM, this will install the grunt command line interface and will allow us to call grunt from anywhere in our project (use grunt globally with "-g").
Command: `npm install grunt-cli`

Install these 3 modules to get up and running, we need grunt initially, but also install "grunt-contrib-jshint", "grunt-contrib-watch" as these will come in handy later on.

```
z:\COLLEGE\4th year\FinalYearProject>npm install grunt -save
z:\COLLEGE\4th year\FinalYearProject>npm install grunt-contrib-jshint -save
z:\COLLEGE\4th year\FinalYearProject>npm install grunt-contrib-watch -save
```

Once we have these dependencies installed and updated in our package.json we can create the gruntfile.js now.

This is a simple example of a grunt file using the dependencies mentioned above.

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    // Task configuration.
  });
};
```

Figure 43 Example view of tutorials loaded in browser

6.13 D3 Integration

I chose D3 to give my project and tutorials a visual representation of all the components, what is linked to what, and to get an easier way to view a higher-level view of all components included. With this I could create multiple graphs that in turn would give the user a more in depth look at each component and what it is “linked” to. I picked this framework, because I was aiming for a unique intractable and easy to understand way of showing a diagram for a software architecture diagram.

i.e. Angular → JavaScript

Continuous integration → Task Manage

D3 provides great functionality for data visualization, and it can achieve this in many ways, the method that I chose, that I thought would reflect this project and to potentially help the user/developer understand more clearly by looking at a graph. In particular, a force graph with nodes. This type of graph takes on a sort of tree like structure, one node leads on to multiple other nodes, and it can even create closed graphs (e.g. single page app - angular – JavaScript – node – single page app). Another bit of functionality that these force graphs provide is user interaction. The user can potentially grab any node and drag it about the screen, and since it is connected to another node, the whole graph will be dragged. This gives it a sense of something fun, which is what I’m looking for, something the user can interact with and learn. As well, if one node may be blocking or too close to others, then this feature allows the user to take it to one side to investigate further.

To use D3.js in my Angular app. I created an angular service that I could use in any Angular component that may need it. This service will be used in the directive I Created to handle the data passed in and then create the angular force graph for said data. This means that D3 will not have to be installed as a project/app dependency. As this means D3 will only be loaded with it is required.

To create this force graph as a reusable custom element, meaning I could pass any data (nodes and links) into it and it will produce a Graph.

```
app.factory('d3Service', ['$document', '$q', '$rootScope',
function($document, $q, $rootScope) {
  var d = $q.defer();

  function onScriptLoad() {
    // Load client in the browser
    $rootScope.$apply(function() {
      d.resolve(window.d3);
    });
  }

  // Create a script tag with d3 as the source
  // and call our onScriptLoad callback when it
  // has been loaded
  var scriptTag = $document[0].createElement('script');
  scriptTag.type = 'text/javascript';
  scriptTag.async = true;
  scriptTag.src = 'http://d3js.org/d3.v3.min.js';
  scriptTag.onreadystatechange = function() {
    if (this.readyState == 'complete') onScriptLoad();
  }
  scriptTag.onload = onScriptLoad;

  var s = $document[0].getElementsByTagName('body')[0];
  s.appendChild(scriptTag);

  return {
    d3: function() {
      return d.promise;
    }
  };
}]);
```

Figure 44 D3 Service

Above is the service that will create a script tag and fill in the attributes for loading d3 into the browser and then returning the d3 object, so that we can access its functions later.

For the directive, I started out to produce a force graph for a high-level view showing key words like angular, directive, testing, continuous integration. By doing this I will be able to create a clean view of what exactly is involved, what things may be similar/connected to, and what process may involve.

Inside the directive it will access the scope variables passed into it, and use these to set the nodes and links of the graph.

```
//setup data, if no data passed in create an empty array/object
nodes = (scope.data !== undefined ? scope.data.elements : {} );
links = (scope.data !== undefined ? scope.data.links : []);
```

Figure 45 D3 directive collecting data passed in

Then I set up an angular watcher, this will update the element if any data that has been passed into it has changed.

```
//setup a watch for new data
scope.$watch('data', function(newValue, oldValue) {
  if (newValue){
    restart(newValue)
  }else{
    console.log("no new value")
  }
}, true);
```

Figure 46 D3.js Directive watcher

To then create the graph with the nodes and links I configured D3 to do this for me. The first step was to create an SVG (graphics container within html, this will be used to draw our shapes. Next up was to add a sense of force to the graph. This would make all the nodes and links revolve around the center of the screen, and have less nodes floating off screen out of reach.

```
function initForce(){
  force = d3.layout.force()
    .nodes(d3.values(nodes))
    .links(links)
    .size([width, height])
    .linkDistance(70)
    .charge(-1000)
    .on("tick", tick)
    .start();
}
```

Figure 47 Defining force on the graph.

```
function initSVG(){
  svg = d3.select("#graph").append("svg")
    .attr("width", '100%')
    .attr("height", height);
}
```

Figure 48 Creating SVG Container to page

D3 works by appending different html components into the container and each other, as with the example above you can see “circle” are appended to nodes, and “line” is appended to links. This will draw each shape on the canvas, and its position will be determined by the gravity set up, along with the link distance and charge.

```
function drawLinks(){
  link = svg.selectAll(".link")
    .data(force.links())
    .enter().append("line")
    .attr("class", "link");
}
```

Figure 49 Drawing the links for the graph

```
function drawNodes(){
  node = svg.selectAll(".node")
    .data(force.nodes())
    .enter()
    .append("g")
    .attr("class", "node")
    .on("mouseover", mouseover)
    .on("mouseout", mouseout)
    .on("mousemove", mousemove)
    .call(force.drag);

  node.append("circle")
    .style('fill', function(d) { return d.group; })
    .attr('r', function(d) { return d.size; });

  node.append("text")
    .attr("x", 12)
    .attr("dy", ".35em")
    .text(function(d) { return d.name; });
}
```

Figure 50 Initial config of D3.js Force graph nodes & links

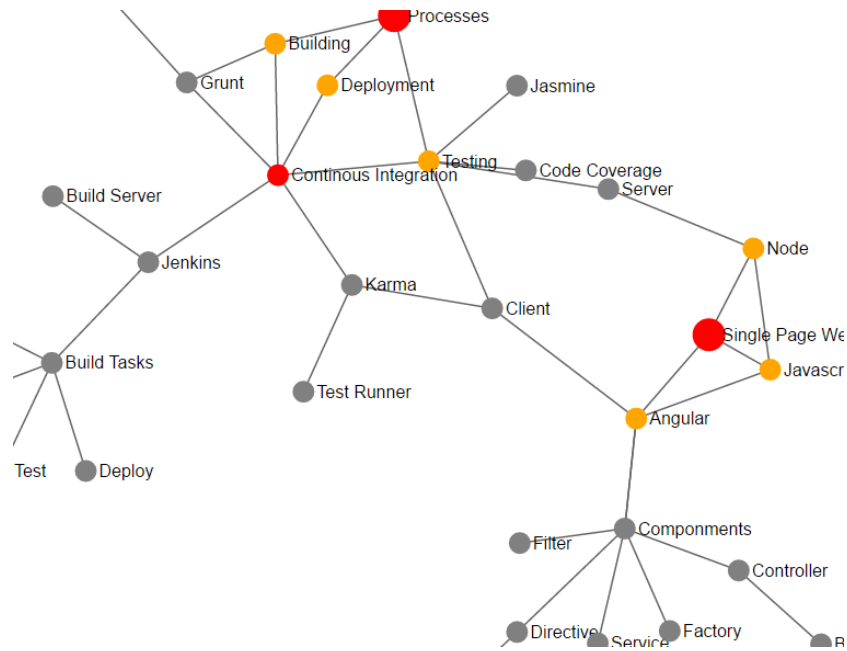
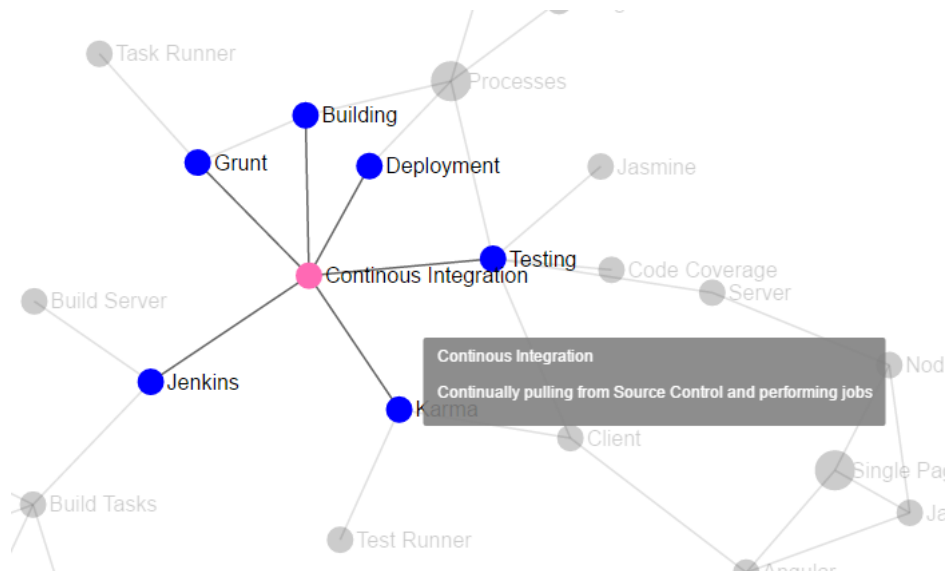


Figure 51 D3 Graph screenshot

Above is a screenshot of the overview (high level) graph. It is intractable and the user can drag nodes about. But I wanted to take this a step further and as more nodes are added the user should be able to use this graph easily, I did this by, adding functionality to hover over nodes, and only highlighting the neighbors of the nodes, while also display a text box with the name and description of the hovered node. (see below)



This feature allows the user to direct their attention to certain nodes and see what is connected to them. Along with this a text box is displayed with a short description about the selected node.

I then continued to use this for each topic I have covered for tutorials, as this could provide an example of digging down from the original high level diagram to a more informative one for each topic. (examples below)



Figure 52 Graphs showing more detail about topics, top left: Testing, top right: Git, bottom: Continuous Integration

6.14 Setting up Database Connection

For my project, I needed to setup a database, so as any data that may be required to be stored or updated should be stored here.

There are many types of databases' available and many different types. I have chosen MongoDB, since it is a document oriented type of storage, this means any JavaScript object that I have I can store them easily and quickly. Since NoSQL database it does not have any complicated joins or SQL to access the data, these types of databases are becoming hugely popular, so this was a design idea to keep in mind of growing projects in any sort of application, industry, down to a creating an application with huge data as a hobby. MongoDB is widely used with the MEAN Stack and Node.js.

With that in mind It was easy to set up a connection to the database via installing a package via Npm

"npm install mongodb --save"

Then in the server side code create functions to POST and GET from server and database. (I could secure a database supplied by the college, so I did not need to install and create my own.)

Retrieving something from the database: Must provide a get request back, as It must send the data back to the client, otherwise getting the data in the first place would have been a waste of time.

```
exports.findDocuments = function(req, res) {
  MongoClient.connect(dburl, function(err, db) {
    console.log("Connected correctly to server")

    db.authenticate(user, pass, function(err, result) {
      console.log("Authenticated correctly to server")
      var collection = db.collection('collectionName');

      collection.find({}).toArray(function(err, docs) {
        db.close();
        res.send( { data: docs } );
      });
    });
  });
};
```

Figure 53 Retrieving something from MongoDB database with Node.js

Inserting something to database: with inserting, deleting and updating the database the functions are quite similar, since there is collection.doSomething method. Each function to insert delete or update must first connect, then authenticate with the database and then it can issue its command. In the post request, the data for what collection and what record should be included, just like the type of data that is being sought for in the previous finding documents example

```
exports.insertDocuments = function(record) {
  MongoClient.connect(dburl, function(err, db) {
    console.log("Connected correctly to server")

    db.authenticate(user, pass, function(err, result) {
      console.log("Authenticated correctly to server")
      var collection = db.collection('collectionName');
      collection.insertMany( [ record ], function(err, result) {
        console.log("Inserted document into the document collection");
        db.close();
      });
    });
  });
};
```

Figure 54 Inserting something into MongoDB database with Node.js

6.15 Working on the App

While working on the app. I tried to Document as much as possible, while at the same time keep to a good practice on developing in angular. As angular is big into templating, creating custom elements and splitting the business logic into controllers, services, scopes and embedded in directives. I kept all this in mind, as this app should be easy to read, to test and ultimately understand for developers and students new to angular.

6.15.1 UI

Bootstrap

When I first got starting working on the app itself, I installed bootstrap dependency into the app, and used one of Bootstraps free usage demo templates. With this CSS, I could then begin starting on the layout of the app and how it would interact from one option/tab to the next, bearing in mind this is a single angular app, I was going to try show off some of the basic-intermediate capabilities. By creating a responsive dynamic app with minimal server intervention.

Page tabs

I designed my layout for the app, since this is a single page app, it will have no reloads or request pages from the server, that means it could get quite a lot of content to show in one view, and I wanted to be able to use angular and its way it uses data binding in the browser to show how tabs could be made.

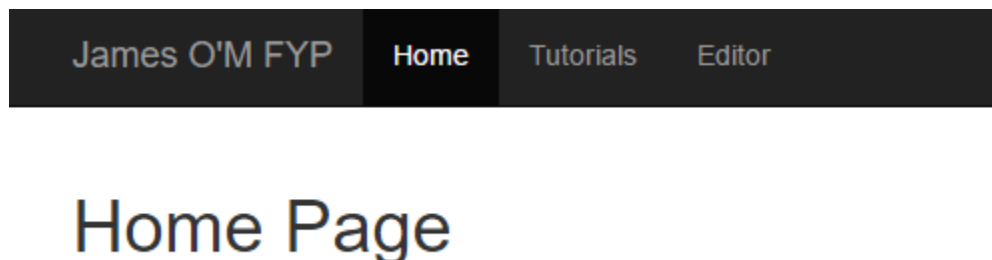


Figure 55 Creating angular tabs

The business logic behind this is that, if a certain tab is selected a function will execute and change a variable, this variable will be designated to determine what tab is being shown. Angular provides two great ways of showing and hiding elements. One way is “ng-show”/”ng-hide” This will do exactly what it says, if a variable is true it will display what elements it has stored within. Whether these elements are being shown or not, the html will still be there for view in the browser. While on the other hand Angular provides an additional way with “ng-if” this means that if a certain variable is true, then this will load the elements into the browser for view and if not then they will not be there at all unlike ng-show and hide, where they are still in the browser but not displayed.

```
<div ng-if="showTab(1)">
  <tab1-directive>
</div>
```

Figure 56 Angular function ng-if to make sense of dynamic content

Display tutorials Menu

I created another menu to select what topic and tutorial the user would like to view: again, with this menu the selected item would change the display inside a iFrame, since this was how I am displaying my tutorials **(As described in 6.12)**, I made a function that would change the URL to the public file of the tutorial, so that once you clicked on a certain topic or tutorial then the relevant file would be displayed in the browser.

When an option is selected, an angular function will activate and change a scope variable within the directive, this value will be an object that will hold information about that topic, (tutorial URL, data for D3 graph etc.)

```
<div ng-if="showDoc(selectedTutorial)">
  <div class="container">
    <iframe style="width: 100%; height: 800px" frameborder="0" src="{{selectedTutorial.url}}"></iframe>
  </div>
</div>
```

Figure 57 Angular iframe taking scope variable to display tutorial in iframe

Above code: will render an iframe showing relevant document if relevant topic name is selected.

```
<div ng-if=selectedTutorial.graphData >
  <force-graph-directive data = selectedTutorial.graphData >
</div>
```

Figure 58 Angular ng-if: will display d3 graph with current scope data

Above code: will display a d3 force graph when overview for a certain topic is selected.

6.15.2 Simple examples

Another key part of my brief and topic, was to demonstrate simple and intuitive angular components. Although I have a small app built, and developers, students and others will have access to this repository to clone it and learn for themselves.

These directives will take a beginner – intermediate approach in creating directives, controllers, services and factories (all angular components). These will be included in the app's front page, as a proof of concept. These examples will touch in different details, using hardcoded html/templating, different scopes, angular two way data binding, functions and how angular components interact with each other.

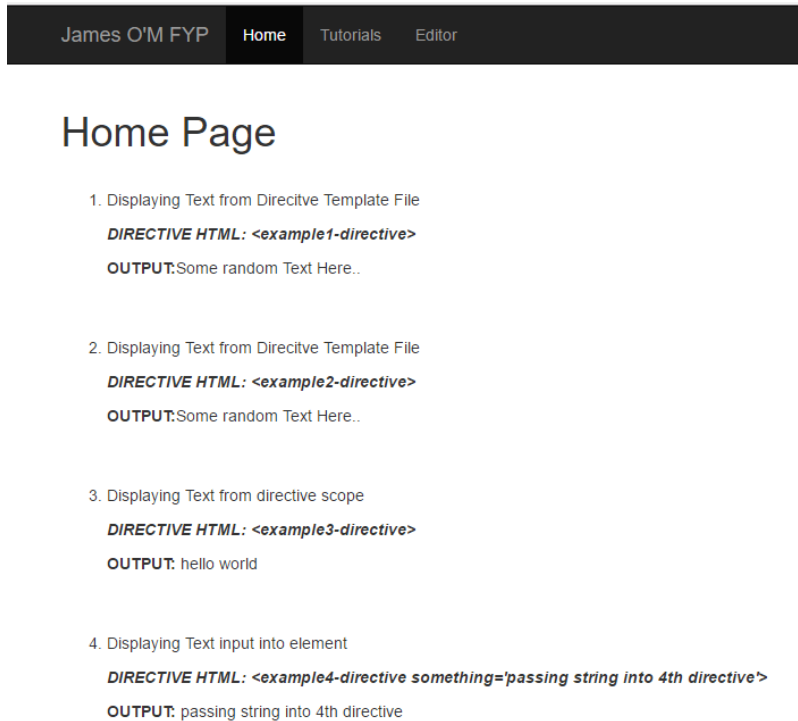


Figure 59 Example Directives home page

6.15.3 Embedding plunker and jsfiddle

I added support for plunker and jsFiddle in my app, since my app takes on a tutorial wise approach, it needed some functionality for the user to create & edit code while they are looking at tutorials. Plunker and jsfiddle have a lot of work put into them, and are good at allowing the developer to test something and prototype certain things.

For example, you have a complex build structure, an app that must be fully rebuilt each time, and wait for the app to be refreshed. These two methods allow for rapid prototyping, say you want to test your button out, layout of something, or a basic prototype that does not need any data or access to your server or other component sin your app. Plunker and jsfiddele are a good way to go.

With this addition to my app, users will have quick access to a development suite in their browser, if they haven't got it set up locally, if they did not know about it before hand, then it will broaden the knowledge of the technologies available online. As plunker and jsfiddle are great code sharing applications, which I believe ties in nicely with my project.

Plunker and jsfiddle provide html to embed their product if your app/site. So, I put this into an angular directive, one for jsfiddle and one for plunker. With angular directives, this means I now have 2 custom elements, and can put plunker and jsfiddle anywhere in my app.

```
module.exports = function(app) {  
  app.directive('embeddedJsfiddleDirective', [function () {  
    return {  
      // controller: 'mainController' ,  
      restrict: 'E',  
      replace: true,  
      scope: {  
      },  
      templateUrl: './app/client/src/directives/templates/embeddedJsfiddle.ejs',  
      link: function(scope) {  
      }  
    }  
  }]);  
};
```

Figure 60 Angular Directive for jsFiddle

```
<div>  
  <iframe width="100%" height="600" src="//jsfiddle.net/IgorMinar/ADukg/embedded/" allowfullscreen="allowfullscreen" frameborder="1"></iframe>  
</div>
```

Figure 61 Angular directive html for jsFiddle

7 Issues

Jenkins Integration testing

My problem with Jenkins Integration testing, End to End click through testing was causing me issues, I was trying for Jenkins to run its integration tests in chrome, or a headless browser. I was having issues trying to install drivers for chrome and configuring my Jenkins instance to find the drivers. This will work perfectly fine on my local machine, but Issues were had attempting to run it on Jenkins.

Jenkins memory

An issue that I came across was with the vm's I had available, since I was using a free tier of amazon EC2, the virtual machines that I was using were a micro instance with small memory, this worked perfectly fine up until a point in the project where it got a bit more complicated, Jenkins was running multiple jobs and intensive ones, and all my jobs were failing. I found the problem, and it was due to the fact that memory was fully utilized and the server would freeze, I fixed this by adding a swap file to my virtual machine instance.

Styling

Styling was a thing I have not really touched on before this project, and for me personally I think this required some level of attention to make the app look good, but its priority would fall lower down than much other components. If I had more time towards the end I would Have spent longer ensure the app could look its best.

Level of detail on some tutorials

As the heading says: I found that when first looking at the new technologies I got side tracked sometimes and found it hard to keep up with the documentation and documentation how to install and configure it. The level of detail on these tutorials are supposed to be for beginners, and beginners moving on to intermediate, A lot more time would have been required to do full in-depth tutorials on good practices and digging down to the complex levels of each component.

Getting correct code coverage

This was a problem with the coverage for unit tests, As I was building/bundling my app, I did this into one file. This caused a huge problem for acquiring correct coverage of code. Say for example I had 1000 lines of code x number of branches etc. With the way, I was bundling files I had everything included (source code templates and dependencies) this meant that even though if my application was fully tested, the code coverage would only display about 15-20%. To fix this issue I could configure my bundling task to bundle the app into separate files, one for the source code, one for dependencies and another for template files. This proved hugely successful as now the code coverage is correct.

8 Conclusion

Original Aims:

- **Build a single page application using NodeJS/AngularJS to display tutorials on the relevant technologies for new developers/students.**
A Node & Angular App was made, in which the user can navigate through different views to see tutorials, topics and overview of the section.
- **Configure and setup a build system for NodeJS/AngularJS application.**
A standalone Linux server from Amazon Web Services was set up to run Jenkins as a build server for the app.
- **Deploy and configure a continuous integration platform capable of automating the build process.**
The Jenkins build server has the second server (VM) set up as a slave machine, so that it can SSH to it and use it as a workspace to pull latest code, install and then restart the node server.
- **Host Build server and app on separate servers (amazon web services, Microsoft azure etc.).**
Both Jenkins the build server and the hosting server, are on separate instances of Linux machine's hosted by Amazon web Services. This meant that both machines had to be configured separately and setup for its need.
- **Evaluate several test frameworks and rigorously test the application, from unit tests, through to integration tests etc.**
Evaluation was done on different testing frameworks for unit testing and integration testing, there was a choice for unit tests between karma, mocha, jasmine and chai, in the end karma & jasmine was picked. For integration testing, there was an all-rounder Framework called protractor, this was chosen and installed, configured and demonstrated.
- **Allow the user to test out their code using the application whilst following a particular tutorial (they won't have to clone a project etc.).**
Embedded jsfiddle and plunker was added into the app, this means that users can click on the specific tab and begin editing code in the Brower using my app.
- **Generate coverage reports/metrics and ensure that the code meets high test coverage.**
The unit test suites were configured to provide highly accurate test coverage. It can create coverage reports with metrics it in different file formats if needed, e.g. html, .json etc.
- **Configure tasks to run test automation locally and as build jobs on build server.**
There are 2 options for running tests automatically and continuously, one is an all-rounder, which will run unit tests, server and client side as well as integration tests if files have changed. While the other is a unit test specific automation task. This will do the same as the previous, but is faster and more efficient in showing the output as this is a standalone task for running unit tests continuously as code or tests is altered.

- **Configure automation tasks to run locally for development purposes, should detect file changes and run appropriate tasks such as building app and appropriate testing.**

Grunt a task manager has been set up that includes tasks to deal with test automation and building of the app. It also has tasks that can be run while the developer is developing new source code or tests.

- **Create a visualization of the main topics and components covered by the tutorials in the app.**

D3.js was chosen for creating a visualization of the main topics and components as this gives a unique, interactive fun way to look at and learn what is connected to what. Other options included making a flow chart to demonstrate flows between components, but this would have grown considerably and may potentially have been more confusing as there would be multiple charts. But with D3.js, it can lay out all the key words, phrases as separate nodes, and allow the user to see them all in one go, hover over each one and see what it is related to, since it will make its neighbors stand out, while dimming out the other nodes of no interest.

Based on the goals and results above, All the goals set out to me by IBM, and supervisor have been met. The project was shown to fellow students and got a great reaction, especially for the D3 graphs,

The application was fully functional, and has quite several tutorials to go along with it. This app is for entry level, and intermediate level developers, who have some knowledge of MVC architecture and JavaScript. The project also has shown full support for how a continuous integration system can be set up. It also has a complete unit test and integration test automation framework.

9 Evaluation

Looking at the results achieved during the time of the project, all the goals that have been set out, have been met.

This Idea was given by a supervisor from my placement in IBM, they were more than accommodating and understanding about the level of involvement and took a back foot on the project and where happy for it to be taken in any direction deemed necessary for it to a final year project.

When I approached my supervisor, he was happy to supervise this idea, and happy with the level of engagement that IBM were providing. Disclaimer: I am not doing this project as a next “new” product for IBM, this was “my” final year project, something I am very interested in. If I required any help or direction, then IBM were more than willing to help. This project has an aspect of industry led (since the topic was given by a company leading in this space), but ultimately academic on the front that this is a project to investigate and document the full end to end process of an AngularJs App from development to CI, testing and deployment.

I am happy that I got all the goals achieved, but if I had more time I could have spent more time on each component to improve it and especially keep up to date with the documentation. I would have liked to get integration testing fully completed on the build server, but was having difficulty getting drivers to work, so I had to park it and move on to other important things.

From this project I am happy at what I have achieved, I have learnt so much more about AngularJS,NodeJS, Test Automation frameworks, test frameworks, continuous integration/continuous deployment and working with Linux virtual machines.

10 Future Work

There are a few things that could be improved over time on this app, of which are:

- Updating the D3 force graphs to expand and close nodes up, this may only be possible for nodes that act as a tree structure, as some nodes are connected by many other nodes.
- Install drivers for, IE11, Chrome, Firefox on Jenkins build server to allow for integration testing.
- Adding screenshot reporting for integration testing, meaning each integration test will take a screenshot when the test passes or fails.
- Create an output in Jenkins to show all metrics for client side and server side unit tests,
- Update tutorials with more in depth information. Tweak them If necessary to make them easier to read.

11 Bibliography

- Angularjs.org. (2017). AngularJS — Superheroic JavaScript MVW Framework. [online] Available at: <https://angularjs.org/> [Accessed 30 Mar. 2017].
- Foundation, N. (2017). Node.js. [online] Nodejs.org. Available at: <https://nodejs.org/en/> [Accessed 29 Mar. 2017].
- npm. (2017). *grunt*. [online] Available at: <https://www.npmjs.com/package/grunt> [Accessed 30 Mar. 2017].
- Npmjs.com. (2017). *npm*. [online] Available at: <https://www.npmjs.com/> [Accessed 30 Mar. 2017].
- npm. (2017). *karma*. [online] Available at: <https://www.npmjs.com/package/karma> [Accessed 30 Mar. 2017].
- npm. (2017). *watch*. [online] Available at: <https://www.npmjs.com/package/watch> [Accessed 29 Feb. 2017].
- W3schools.com. (2017). *AngularJS Tutorial*. [online] Available at: <https://www.w3schools.com/angular/> [Accessed 30 Mar. 2017].
- npm. (2017). *protractor*. [online] Available at: <https://www.npmjs.com/package/protractor> [Accessed 16 Feb. 2017].
- Docs.angularjs.org. (2017). *AngularJS*. [online] Available at: <https://docs.angularjs.org/api> [Accessed 30 Mar. 2017].
- npm. (2017). *mongodb*. [online] Available at: <https://www.npmjs.com/package/mongodb> [Accessed 27 Nov. 2016].
- npm. (2017). *browserify*. [online] Available at: <https://www.npmjs.com/package/browserify> [Accessed 17 Nov. 2016].
- npm. (2017). *grunt-protractor-runner*. [online] Available at: <https://www.npmjs.com/package/grunt-protractor-runner> [Accessed 30 Mar. 2017].
- Jenkins.io. (2017). *Jenkins*. [online] Available at: <https://jenkins.io/> [Accessed 30 Mar. 2017].
- Amazon Web Services, Inc. (2017). *AWS Free Tier*. [online] Available at: https://aws.amazon.com/free/?sc_channel=PS&sc_campaign=acquisition_UK&sc_publisher=google&sc_medium=cloud_computing_hv_b&sc_content=aws_core_e&sc_detail=amazon%20web%20services&sc_category=cloud_computing&sc_segment=188877744876&sc_matchtype=e&sc_country=UK&s_kwcid=AL!4422!3!188877744876!e!!g!!amazon%20web%20services&ef_id=VsZA@AAAAb1f7Smd:20170330223109:s [Accessed 30 Mar. 2017].
- GitHub. (2017). *karma-runner/grunt-karma*. [online] Available at: <https://github.com/karma-runner/grunt-karma> [Accessed 30 Mar. 2017].
- npm. (2017). *jasmine*. [online] Available at: <https://www.npmjs.com/package/jasmine> [Accessed 30 Mar. 2017].
- Chaijs.com. (2017). *Chai*. [online] Available at: <http://chaijs.com/> [Accessed 30 Mar. 2017].
- Mochajs.org. (2017). *Mocha - the fun, simple, flexible JavaScript test framework*. [online] Available at: <https://mochajs.org/> [Accessed 30 Mar. 2017].
- KeyCDN Blog. (2017). *Gulp vs Grunt - Comparing Both Automation Tools*. [online] Available at: <https://www.keycdn.com/blog/gulp-vs-grunt/> [Accessed 30 Mar. 2017].

- En.wikipedia.org. (2017). *Comparison of synchronous and asynchronous signalling*. [online] Available at: https://en.wikipedia.org/wiki/Comparison_of_synchronous_and_asynchronous_signalling [Accessed 30 Mar. 2017].
- Npmjs.com. (2017). *npm*. [online] Available at: <https://www.npmjs.com/> [Accessed 30 Mar. 2017].
- Modulecounts.com. (2017). *Modulecounts*. [online] Available at: <http://www.modulecounts.com/> [Accessed 30 Mar. 2017].
- npm. (2017). *bower*. [online] Available at: <https://www.npmjs.com/package/bower> [Accessed 30 Mar. 2017].
- Mocha, D. (2017). *Difference between Karma and Mocha*. [online] Stackoverflow.com. Available at: <http://stackoverflow.com/questions/23272521/difference-between-karma-and-mocha> [Accessed 31 Mar. 2017].
- Thejsguy.com. (2017). *Jasmine vs. Mocha, Chai, and Sinon - The JS Guy*. [online] Available at: <http://thejsguy.com/2015/01/12/jasmine-vs-mocha-chai-and-sinon.html> [Accessed 31 Mar. 2017].
- Helm, C. (2017). *Using Jasmine to Test Node.js Applications*. [online] via @codeship. Available at: <https://blog.codeship.com/jasmine-node-js-application-testing-tutorial/> [Accessed 31 Mar. 2017].