Advanced Web Engineering

MSc Emerging Technology

Assignment 5

Andrew Hall

Tutor: David Walker

Student: Andrew Hall

Registration Number: 14041593

Submitted: October 19th 2015

Word Count: 5738

Table of Contents

[Introduction 3](#_Toc432969642)

[1. Getting Mean 4](#_Toc432969643)

[1.1 High Level Architecture 4](#_Toc432969644)

[1.2 Getting Mean with Attire 5](#_Toc432969645)

[1.3 Development Schedule 6](#_Toc432969646)

[1.4 Dependency Management 6](#_Toc432969647)

[2. Versioning and Release 7](#_Toc432969648)

[2.1 Versioning Strategy 7](#_Toc432969649)

[2.2 Release Strategy 9](#_Toc432969650)

[2.3 Configuration Management 9](#_Toc432969651)

[3. Attire Server 10](#_Toc432969652)

[3.1 Data Management 10](#_Toc432969653)

[3.2 Express Middleware 11](#_Toc432969654)

[3.3 Views and Templates 13](#_Toc432969655)

[3.4 Application Programming Interface 13](#_Toc432969656)

[3.5 Mongoose 14](#_Toc432969657)

[4. Attire Client 14](#_Toc432969658)

[5. Summary 16](#_Toc432969659)

[Credits 18](#_Toc432969660)

[Bibliography 20](#_Toc432969661)

[Appendix A – Project Planning 25](#_Toc432969662)

[A1. Planning Board 25](#_Toc432969663)

[A2. Progress Dashboard 25](#_Toc432969664)

[A3. Velocity and Productivity 25](#_Toc432969665)

[A4. Defect Tracking and Trends 25](#_Toc432969666)

[Appendix B – Dependency Management 26](#_Toc432969667)

[B1. Future Packages 26](#_Toc432969668)

[Appendix C - Version Control 28](#_Toc432969669)

[C.1 Commit History 28](#_Toc432969670)

[C2. Release Tagging in Git 28](#_Toc432969671)

[Appendix D – Post Install Build 29](#_Toc432969672)

[Appendix E – Data Model 30](#_Toc432969673)

[Appendix F – Synthesizing Test Data 31](#_Toc432969674)

[Appendix G – Attire Folder Structure 32](#_Toc432969675)

# Introduction

The purpose of this report is to plan, design and implement a single page web application using open source software components across the whole application technology stack. The report critically evaluates the benefits of such an approach from a number of diverse perspectives using a fictitious online clothing retailer called Attire as a case study. This study builds on the user experience design presented in assignment three by adding full stack functionality to serve content and manage data.

The structure of this report is aligned around the different technologies used to build the application. The first section serves as an introductory overview, describing the goals and scope of the implementation, the overall delivery plan and the high level architectural design for the Attire site. The architecture introduces the key technologies of the MEAN stack; MongoDB, Express, AngularJS and NodeJS and describes how they work together. The role and importance of dependency management is introduced.

The second section presents an integrated version control and release management strategy, explaining how these support the overall objectives of the delivery plan. This section also explores the use of build automation through task runners to customise the deployment of our application to a cloud hosting provider called Heroku.

The third section examines the server side components of the MEAN stack in more detail discussing the concept of middleware and reflecting on how best to organise code to better support an automated test strategy. The benefits of using a template engine to improve developer productivity and encourage code reuse are considered in the context of the Attire navigation design. Data storage, schema and migration strategies are considered, comparing design patterns for NoSQL databases to more traditional relational data models.

The fourth section shifts attention to the client and describes the benefits of the single page application paradigm. The client side architecture of the application is presented with reference to design patterns such as dependency injection and inversion of control frameworks that promote code modularity and loose coupling of components.

Finally the report raises the discussion to a more strategic level describing the importance and benefits of open source but also focussing on why many organisations opt to be tied into expensive and restrictive vendor contracts.

# Getting Mean

According to Dickey, J. (2015) the MEAN stack enables a very different style of web application, one where the client does more of the work and is responsible for creating the user experience and interaction. In this model servers provide data, typically in a JSON format, rather than pages in a HTML format. The server is therefore an API to be consumed by the client and the client chooses how to render the data. The distinction is important when we consider servers that support both web and mobile clients. An API approach allows the same server side code to support all clients.

The MEAN stack is a collection of four separate technologies (Mongo, Express, Angular and Node) that work seamlessly together to deliver a full software stack for building modern, dynamic, performant and engaging web applications, from browser to database. In this stack Node and Express are used to create server side APIs, Mongo for data storage and Angular for building the client side web application.

## High Level Architecture

The diagram below shows how the components of the MEAN stack fit together.

|  |  |
| --- | --- |
| C:\Users\145987\Desktop\Masters\Assignment 5\High level Architecture.emf | On the browser, AngularJS provides a sophisticated framework for building single page web applications. In this style of application there is only ever one web page served from the server. This page knows how to fetch the data it needs from the server and is capable of dynamically changing the view the user sees by manipulating the document object model in response to user interactions.  This contrasts with traditional web applications where each user interaction triggers a new request to the server for a fresh page.  The single page model provides a much cleaner separation of concerns where the server is responsible for providing raw data and the client responsible for presentation and handling user interaction. As we have noted this allows us to reuse code to support mobile clients. Offloading work to the client is likely to yield performance gains. |

On the server side, NodeJS is a javascript runtime liberated from the confines of the browser. Node enables us to build generic servers using javascript. The asynchronous, non-blocking and event driven architecture of NodeJS make it an effective technology choice for simple input output based request handling typical in web and chat servers. (Dahl, R. 2009).

Express is a web framework for NodeJS used to build the server API our single page application will interrogate. It somewhat simplifies working with NodeJS but more importantly introduces powerful request and response middleware to route client requests to an endpoint via a configurable processing pipeline. This processing pipeline could perform any number of useful tasks such as logging, authentication or validation ahead of processing the actual request which will usually involve fetching some data.

The final component of the stack is MongoDB which is a NoSQL or document database technology. Document databases work quite differently to relational databases because they are able to store and query complex hierarchical JSON documents. The data does not have to be deconstructed to fit a predefined table schema and it does not have to be reassembled at query time through expensive join operations.

The MEAN stack offers several advantages. Firstly because each component of the stack is coded using javascript developers can quickly become productive and more easily understand each other’s work. They can move seamlessly from building client side to server side or even database access without friction. Code can become isomorphic, running on both client and server and this might be advantageous when validating user input for example.

Secondly because each part of the stack understands JSON as a data structure there is no need for data transformation between client and server or between server and database. We can simply use the same data structures throughout and this is another productivity gain for the developer and a performance gain for the user. We avoid the complexity of object relational mapping.

However there are some downsides. The technology stack is still relatively immature and best practices are continuing to evolve. MongoDB in particular has fantastic potential but is not suited to all problems (Mei, S. 2013). Denormalised data storage offers fast retrieval speed and horizontal scalability beyond that of relational database technology but limited support for atomic transactions and deferred write to disk make it an unlikely choice for applications handling financial transactions.

## Getting Mean with Attire

To investigate the MEAN stack further we shall develop a web application for an online clothes retailer called Attire. The goal of the web application is to be the primary sales channel so showcasing the stock, helping customers find the products they need and using reviews to create reassurance are important sales conversion features of the site.

The key initial features for the Attire site allow users to browse the product catalogue, read reviews and recommendations for products and add items from the catalogue to their shopping basket. A minimum viable product would have a wider scope including authentication, checkout and payment features.

## Development Schedule

The diagram below depicts a milestone plan showing the development schedule for the site. To organise the work the development process follows an agile methodology, using weekly iterations to create software releases that incrementally build the required features. This approach establishes a sustainable pace for development effort enabling frequent course corrections based on measured progress rather than guesswork. The iterations each encapsulate a theme or functional area of the site and the plan allows time for research to understand how best to use the technology and identify other complimentary tools.

C:\Users\145987\Desktop\Masters\Assignment 5\Milestone Plan.emf

The milestone plan is supported by a product backlog comprising of user stories and fine grained tasks organised around the sprint and release cycle. The backlog was created using an agile planning and tracking tool called [Yodiz](http://www.yodiz.com/) which is depicted in Appendix A.

## Dependency Management

In the previous section we described NodeJS as the server environment for our applications. In practice Node is more generic and is increasingly used to build many different types of application package. Technically a Node package is a javascript file that conforms to the CommonJS specification for importing dependencies and exporting a public interface (Ertz, B. 2014). They are easy to create and publish and Node has large eco system of packages to suit most needs.

One of the most powerful parts of Node is the Node Package Manager (NPM) which is both a registry of published node modules and a command line tool for querying and installing code from the registry.

When NPM is used to install a package for an application, we can record the dependency in a package.json file. This file is important when we share or deploy our code. Because the file declares what packages the application needs to run, NPM can recreate the application dependencies on another environment by downloading from registry. Our code becomes more portable as a result.

Node packages are installed in either a global or a local configuration. The choice depends on how the package will be used. Global packages are installed under the node folder and are, as the name implies, globally accessible from the command line. Global configurations are typically used for development tools such as task runners and generators.

In most cases local configurations are preferred. Local package installation creates the node module under the current working directory. This means that different applications can use different versions of the same package and those installations can coexist on one machine without interfering with each other. Once installed our application code can use of the package through the require syntax and the Attire source code provides numerous examples of this concept.

When developing code we use packages such as build tools and test runners to help support the development workflow. These dependencies are known as development dependencies; they are not required by our application to run in live operation but they are required to build and maintain the application.

C:\Users\145987\Desktop\Masters\Assignment 5\Node Modules.emf

The package.json file shows a clear distinction between application and development dependencies as shown in the example from the Attire site.

The packages required by Attire to run in production are shown in the dependencies section alongside their semantic version numbers. In this scheme the caret character denotes the minimum version within the same major version family and the tilde character denotes the minimum version within the same minor version family.

An explanation of each of these modules is provided in the Appendix B.

NPM will install or update packages listed as dependencies and this forms part of our release and deployment strategy.

# Versioning and Release

## 2.1 Versioning Strategy

To manage the source code for the Attire site we need a code versioning and release strategy. This is particularly important in a team environment where we need a safe way to integrate code from multiple authors into the main codebase. The versioning and release strategy is the foundation of a sound development workflow. It directly supports the development plan and schedule.

In this study Git was used for code versioning and release. Git is a distributed, disconnected version control system. Unlike traditional source code repositories there is no centralised place where developers manage the code base although in practice it is a good idea to designate a master branch to hold stable released code. Instead developers can pull code from each other to integrate their work locally before committing or issuing a pull request for integration with the main codebase. Branches are cheap to create in Git as no code is actually cloned and this enables a workflow of branch, fix and commit in small increments, reducing merge issues (Vogel, L. 2015).

The diagram below illustrates how Git was used to support development of the case study. There is no single right way to do this and both Driessen, V. (2010) and Whitney, D (2013) offer variations on this model.

C:\Users\145987\Desktop\Masters\Assignment 5\Versioning Strategy.emf

The basic principle of the strategy is that there is a master branch of our source code that holds only stable, tested and released code. In the diagram the master branch is shown in red and is hosted on GitHub [here](https://github.com/ah903/assignment05). Commit access to the master branch is normally restricted and developers issue a pull request to the owner of the master branch when their work is ready for integration.

To manage work in progress developers clone the master branch, creating a local copy of the code. This is the integration branch in the diagram. They may work directly on the integration branch or create further branches to organise their own work. In the case study each weekly release was developed on a separate local branch shown in green. In hindsight a better strategy would have been to branch the code for each user story in the release creating smaller contained units of code.

As work is completed it is merged into the integration branch. This is a two way point of synchronisation. Firstly we may wish to merge code from different developers and test together before promoting. Secondly, the code on the master branch may have changed for example if an emergency hotfix were needed. In this case the hotfix can be integrated locally with our work in progress. Occasionally merging branches may result in conflicts, usually if two people have changed the same line of code in the same file. Merge conflicts require manual resolution.

With only one developer the model is simple; build on a development branch integrate locally, tag the release and push to the master branch. For larger teams and products there may be multiple layers of staging and integration, perhaps to integrate the work of many teams. Integration branches are natural hook points for continuous integration servers configured to watch for new commits and run a series of tasks such as code linting and test suite execution.

Each time a release is created the code is tagged with a release number and pushed to the master branch. The tag provides a quick way of identifying significant commit points such as releases. Appendix C illustrates the commit history, tags and release schedule of the Attire site.

## 2.2 Release Strategy

The production version of the Attire site is hosted at Heroku and may be accessed [here](https://attire.herokuapp.com/). There are a number of ways of releasing code onto the Heroku platform but the simplest is to push to a git repository monitored by Heroku.

When code is pushed, Heroku runs an appropriate build pack that fetches the source code, installs the right language runtime and any dependencies our application requires. For NodeJS applications the build pack uses NPM and the package.json file to fetch the right dependencies. The whole assembled bundle is deployed onto a virtual machine known as a dyno and Heroku starts the server by executing a user definable script. It is also possible to configure post deployment tasks to execute additional scripts (Ravulavaru, A. 2014)

This deployment process is depicted in the diagram below and is completely automated, triggered by a push to the Heroku release repository.

C:\Users\145987\Desktop\Masters\Assignment 5\Release Strategy.emf

For the most part the process looks identical to that depicted in the previous section. The key difference lies in the post deployment script that uses Gulp to optimise our source code for a production environment, copying, concatenating and minifying files to reduce download times. Appendix D describes the process in more detail.

## 2.3 Configuration Management

Versioning and release strategies need to support different configurations for different target environments. We have already seen one example with a post installation script that optimises code for performance in the production environment. More generally we would expect application data such as port numbers and connection strings to vary between environments.

A common approach to managing configuration data is for the application to load settings from a file and effectively configure itself. Command line arguments passed to the application identify the right configuration file. Whilst this approach is simple it is not ideal from a security perspective unless the data is encrypted. Even so it is wise to direct git to ignore configuration files.

NodeJS servers will accept command line arguments as a configuration strategy and Heroku extends this concept further through configuration variables that are mapped to the Node process environment variable at runtime.

The Attire case study uses a Node module called nconf to manage configuration. This works with command line arguments, environment variables or configuration files in any combination. The module builds a configuration based on the data it collects from each of these sources using an order of precedence to manage conflicts. In production the primary source is Heroku configuration variables which are secured on the platform. In development the configuration is loaded from a JSON file.

In this section we have described version and release management strategies to manage different environments. Continuous integration processes constructed using tools such as gulp provide a means to execute a test suite whenever code changes to check that no defects have been inadvertently introduced.

Continuous deployment takes the automation concept further and pushes successfully tested code to staging or production environments again using purpose built scripts. This helps to ensure repeatability of the process and de-risk releases.

# Attire Server

Attire is a web application that creates a shopping experience and brand that customers will enjoy. It relies on high quality content such as photographs of the product range coupled with concise descriptions and reviews to inform the buying choice.

## Data Management

To store this content Attire uses a Mongo database hosted at MongoLabs. Mongo is a document database, one of new generation of NoSQL databases that are optimised for performance, horizontal scalability and distributed storage. Mongo is able to achieve read and write scalability by constraining the scope of database transactions and by storing data in a denormalised form (Dayley, B. 2014).

One of the big advantages of Mongo is that it is very simple for the web developer to work with. The toolset is instantly familiar as the data is managed in JSON format and stored as a lightweight binary version of JSON known as BSON. As a result NoSQL databases and Mongo in particular are popular choices when time to market is an important factor. The biggest challenge in working with Mongo is to design the right data structures to balance the needs of the application, performance and data usage patterns.

Mongo is designed to handle rich hierarchical data without a strictly defined schema. We can therefore store and retrieve data as an object graph without having to disassemble and reassemble into flat tabular formats. This naturally leads to choices about how best to represent different data structures, as embedded objects within a larger document or as references to objects in different collections (Data Modelling Introduction 2015).

In the case of Attire we could store all the reviews for a product as part of the product document. Similarly we could store orders as part of the customer document. Alternatively we could choose to store products, reviews, customers and orders as separate document collections using ids as joining references and normalising the data model. The best design is not always obvious without deeper consideration of three factors.

* Write operations are only consistent and atomic at the document level. Transactions across documents and collections are not supported so transaction handling requirements often define a data boundary.
* Different collections will grow at different rates and potentially require different scale out strategies.
* Repeating multiple copies of the same data item as an embedded object in different documents is costly from a storage and data consistency perspective although better for query performance. Understanding what data is updateable and when is important.

Appendix E shows the data model for the Attire application based on these considerations and Appendix F describes how test data meeting the design was synthesised at scale using a JSON data generator.

## Express Middleware

The remainder of our server environment is based on Express, a sophisticated middleware framework that runs on NodeJS and provides the API for our application.

Like all node modules Express is installed using the Node Package Manager. Helpfully there are also a number of different generators to scaffold Express servers. For the Attire project a basic node package called express-generator was used to setup the folder structure and incorporate boilerplate error handling and default routing. On reflection I found the generator folder structure for the application too simplistic, especially when adding client side Angular code. The Yeoman generator developed by Larsen, L (2013) might have proven a more complete scaffold given out of the box support for Angular and Heroku. As a result the final project structure for Attire had been modified to better separate server and client side code and Appendix G documents the final structure.

Middleware is a series of modular functions that execute sequentially to create processing pipelines. Requests for data from a client are directed via the middleware to an end point that will generate the response as shown in the diagram overleaf.

When a HTTP request for a resource arrives at the server it is handled by a series of middleware functions configured to act on different types of request. Each middleware performs a discrete task such as logging and processing moves to the next middleware in line. Middleware might alter the request, making more data available to downstream processes such as authentication tokens, query parameters or form input.

Once the middleware has run a route handler for the resource request is invoked. Route handlers are responsible for building and returning a response to the client. This could be as simple as serving static content such as an image file or as complex as querying a database to generate dynamic content.

C:\Users\145987\Desktop\Masters\Assignment 5\Middleware.emf

Just as middleware creates a request processing pipeline, responses can also be constructed in stages for example to assemble results from multiple data sources or in the case of Mongo from different collections.

Express middleware is very powerful. It allows a highly modular, pluggable design permitting different processing pipelines to be constructed for different types of HTTP request, potentially varying the pipeline for different endpoints. Furthermore middleware can alter the processing pipeline at runtime by invoking other middleware based on some event for example when no documents match a query or when an exception occurs.

Attire takes this concept of modularity further and separates middleware for different parts of the application into modules. From a unit testing perspective this is a necessary but insufficient step to make the server testable. Middleware is often built using anonymous functions which means we can only really test the endpoints of the API. To improve on this Singer, M (2014) argues that we need to restructure our code and implement middleware functions as methods on an object that is required by both our test suite and our endpoint, further using mocking libraries to fake request and response objects without the need to pass HTTP.

## Views and Templates

One of the packages included by the express generator is a template engine called Jade which is used to build the containing page in our single page application. Template engines such as Jade can render templates on the server, returning HTML to the client or they can render javascript into a page to interact with a server API for example (Mardan, A. 2015).

The big advantage of templates is reusability. Not only can consistent page layouts be created from a template, snippets of reusable HTML can be incorporated into different pages. Templates allow changes to be managed centrally without having to change many individual pages. The Attire application uses Jade to layout the main page in our single page application, making extensive use of mixins to create the menu structure for the application. The approach is an extension of that suggested by Bracey, K. (2014).

The biggest question for a full MEAN stack application is whether Jade and Angular should be used together because Angular also includes a template engine. It is possible to create Angular partials from Jade templates and Bahmutov, G. (2013) suggests an approach. However this adds complexity to the design and is perhaps better suited to large applications.

## Application Programming Interface

The API for Attire implemented with Express is designed to be client agnostic. Its job is to receive requests, take appropriate actions and return data in JSON format along with a HTTP response code that indicates the success or failure of the operation.

The API follows RESTful design principles using HTTP verbs to describe the required action and nouns to describe the entity upon which the take the action (Sahni, V. no date). It is documented in the file “Attire API End Points” on the GitHub repository.

In many cases the API design is straightforward because the relationships between our data entities are also straightforward and intuitive. For example, each product in the catalogue has reviews so we should be able to obtain these from and endpoint like:

api/products/[productId]/reviews.

In other cases we may to look at the data from the opposite perspective for example find all the products reviewed by a user:

api/reviews/[userId]/products

Cross cutting perspectives such as data filters or sort order specifications do not easily fit into this model (Sahni, V. no date) but can be implemented using query string parameters as shown in the example below:

api/products/?category=Trousers&group=Men&promotion=New

The first version of the Attire API could be improved in many ways. The most obvious gap is the lack of authentication on some endpoints which should be restricted to identified users. Middleware libraries such as PassportJS offer proven authentication and session management features and this work forms the next phase of the development plan.

The design could also be improved with regard to paging. The API supports parameters allowing the user to restrict the number of documents returned from a query and to request a specific page of results. A more mature design would return the total number of pages in a dataset and provide links to endpoints to retrieve next and previous pages (Beale, M. 2013).

Lastly a versioning strategy for the API is lacking. The simplest approach is to embed the version number into the API endpoint as shown in the example below (Harrigan, D. 2012)

api/v1/products/[productId]/reviews.

## Mongoose

To interact with the database from API endpoints Attire uses Mongoose as a data mapping layer, encapsulating and extending the capabilities of the Mongo database driver (Dayley, B. 2013). Mongoose allows developers to create schema for their documents so incoming data can be validated. This is important because Mongo is schemaless and will accept a document of any format into a collection. Although this is a powerful feature that supports emergent design it is typically not what we want and most applications require data to conform to some basic rules. Mongoose data models, based on a schema, define and articulate the rules data must adhere to before it will be accepted.

Attire uses Mongoose on the server in a very basic way and validation is one area where we might benefit from full stack isomorphic approach, validating data on the client and server using the same schema, rules (Karpov, V. 2015).

# Attire Client

The Attire client is a single page web application implemented with AngularJS and visually styled using Bootstrap. Single page applications offer a snappier experience to the user as much of the interaction processing is done locally. The application responds to user interaction by updating the user’s view without having to fetch new pages from the server. If the new view requires new data this is fetched using the Attire API, asynchronously in the background (Sevilleja, C. 2014).

AngularJS is an opinionated framework for building web applications. It is based on a model view controller design pattern that decouples the data (model) from the presentation (view) from the business logic and data binding (controller). Although the learning curve is steep following the Angular way leads to less code and more maintainable and testable applications. The client side application architecture is shown overleaf.

**C:\Users\145987\Desktop\Masters\Assignment 5\Application Architecture.emf**

Angular applications are modular, built from one or more modules, each of which has an associated configuration. This improves reusability. Modules are associated with a page through the ng-App directive. This effectively tells Angular how to bootstrap the application.

An optional Angular module called ng-route delivers single page capabilities. User navigation is handled through the routing configuration which loads different views into the containers on the page (Annunziato, J. 2015).

Attire uses this mechanism to display a common navigation section, swapping different views into the main canvas of the page in response to user actions. Views in Angular are associated with controllers which create a scoped two way data binding with the view. The view is automatically updated when data in the controller changes and data in the controller is automatically updated when the user interacts with the view for example by completing a form.

To manage data binding Angular uses a template system to define the data mappings and set of directives to execute those mappings. Directives provide clear statements of intent and effectively extend the capabilities of HTML (Whalin, D. 2013). A good example is the ng-click directive that invokes a function on the controller when a page element is clicked.

Interaction with the outside world is managed through modules known as factories or services. These components are responsible for interacting with our server API. They translate user interactions into the HTTP calls needed to retrieve data, save an order or authenticate a user. Factories are singleton objects; they do not need to be constructed and are injected into controllers at runtime following an inversion of control paradigm.

The Attire application showcases the Angular architecture to deliver an organised, demarcated codebase that is easy to maintain and extend. The code further demonstrates some more advanced Angular techniques such as the use of promises to handle asynchronous server calls, sharing scope between controllers via root scope and conditional routing and presentation based on authentication status.

# Summary

In this paper we have planned and built a web application using the MEAN stack to investigate the advantages and disadvantages of a full stack application utilising open source technology. We have Git and GitHub for distributed version control and integrated source code management into development and deployment workflows.

Using a single technology across the whole stack has proven advantageous from a developer productivity perspective and we might expect this to translate into gains for the enterprise as a whole. Firstly less people with individual specialisms are required, reducing communication and coordination overhead. Secondly the opportunities for code reuse and the incentives to create reusable code are greater when the technology stack is uniform across client and server.

However, as with any technology, time and effort must be devoted to achieve mastery of the MEAN stack. Best practices continue to evolve and understanding when to use the technology is as important as how. NoSQL databases change established design paradigms necessitating a mature reflection on their applicability to different problems. Yared, P. (2014) argues that full stack is something of a misnomer as cloud hosting, configuration management, mobile applications and data analytics require diverse skills. Given the technical diversity of most enterprise IT estates full stack developments are likely to yield local project level benefits in the first instance.

The story is similar when considering adoption of open source tools and technologies. Despite the dominance of technologies such as Apache, Linux and WordPress on the web (W3Techs, 2015) widespread adoption of open source in the enterprise is progressing more slowly.

Historically commercial enterprises have been sceptical of using open source software to run business critical operations, often citing security and support as concerns. Open source license and contractual models do not readily fit into existing business models and Rubens P. (2014) argues corporate procurement frameworks often presume terms that tend to exclude open source options.

Yet there are compelling advantages. Open source software is free to license offering large savings compared to proprietary alternatives, especially if we account for the time and effort expended by license analysts and contract lawyers. The legal risk of under-licensing also evaporates. From a support perspective, it is possible to purchase service and support agreements for popular open source packages addressing many corporate concerns. Open source tends to follow and in some cases drive open standards which reduces vendor lock-in.

Many commercial organisations are concerned about security flaws in open source code and high profile incidents such as HeartBleed reinforce these prejudices. All software has defects but there is a school of thought that open source projects achieve better security because code is open to scrutiny and review. Heath, N (2014) argues that just because code can be examined does not mean that it will be or even that reviewers are qualified to judge. What is more important is an understanding of the risks, a plan to mitigate and a means to distribute trusted patches quickly and effectively. Many organisations have solved these challenges with proprietary software and although the solutions might differ for open source, sensible risk management should deliver the security assurances enterprises require. In short the barriers to using open source look increasingly surmountable.

As part of this assignment we have seen how distributed source control can be used to improve developer workflows and create strongly automated integration and deployment processes. Technically the ability to work locally and in disconnected mode is powerful. The ability to directly integrate code at a peer level also establishes a frictionless agile working environment (Bruno, E. 2013).

Despite these advantages, organisations understand the value and intellectual property that resides in source code. They generally will not want their employees posting code onto public sites such as GitHub and take active steps to prevent data exfiltration. Whilst it is possible to host Git privately behind the corporate firewall but this does not fully alleviate security concerns. The fact that history is not immutable makes for a difficult conversation with corporate compliance and IT managers may feel Git offers too much flexibility to establish a good corporate standard.

In many cases a compromise could be possible where developers use Git for all the efficiencies of peer to peer code sharing and disconnected working but promote to the officially sanctioned centralised version control system at key points in the workflow.

# Credits

|  |  |  |  |
| --- | --- | --- | --- |
| **Frameworks** | | | |
| NodeJS Package Manager | Package management for server components and development tools | | [NodeJS](https://nodejs.org/) |
| NodeJS | Non-Blocking Event Loop Server | | [NodeJS](https://nodejs.org/) |
| NodeMon | Utility to restart NodeJS server when watched files change | | [Sharp, R.](https://github.com/remy/nodemon) |
| Node Inspector | Debugger for NodeJS servers | | [StrongLoop](https://github.com/node-inspector/node-inspector) |
| Express Generator | Generator application for scaffolding Express applications | | [StrongLoop](http://expressjs.com/starter/generator.html) |
| Express | Web server and middleware running on top of NodeJS | | [StrongLoop](http://expressjs.com/) |
| MongoDB | NoSQL database | | [MongoDB](https://www.mongodb.org/) |
| Mongoose | Database Object Data Manager for schema management | | [LearnBoost](http://mongoosejs.com/) |
| Jade | Template engine deployed Server Side | | [ForbesLyndsey](http://jade-lang.com/) |
| AngularJS | Client side MVC Framework for building single page applications | | [Google](https://angularjs.org/) |
| AngularJS Routing | Angular add-in module to handle client side routing | | [Google](https://docs.angularjs.org/api/ngRoute) |
| Gulp.JS | In memory task pipeline for build automation | | [Gulp](http://gulpjs.com/) |
| Browser Live Reload | gulp-browser-sync | [Browser-Sync](http://www.browsersync.io/) |
| File and folder management | Del | [Sorhus, S.](https://www.npmjs.com/package/del) |
| File Concatenation | Gulp-concat | [Fractel](https://www.npmjs.com/package/gulp-concat) |
| CSS minification | Gulp-cssmin | [Lijung, C.](https://www.npmjs.com/package/gulp-cssmin) |
| Script dependency injection | Gulp-inject | [Carlstein, J.](https://www.npmjs.com/package/gulp-inject) |
| Javascript Linter | Gulp-jshint | [Alger, S.](https://github.com/spalger/gulp-jshint) |
| Javascript minification | Gulp-uglify | [Stock, T.](https://www.npmjs.com/package/gulp-uglify) |
| JQuery | Javascript library for adding interactivity to sites and managing legacy browsers | | [JQuery](https://jquery.com/) |
| Bootstrap | CSS Grid Framework | | [Bootstrap](http://getbootstrap.com/) |
| **Hosting** | | | |
| Heroku | Site hosting  Published version https://attire.herokuapp.com/ | | [Heroku](http://www.heroku.com/) |
| MongoLabs | Hosting for MongoDB database  ds051873.mongolab.com:51873/heroku\_hl48jg51 | | [MongoLab](https://mongolab.com/) |
| **Development Tools** | | | |
| JSON generator | Used to generate product and review test data for the application | | [Omanashvilli, Vazha](http://beta.json-generator.com/) |
| Bash | Mac shell for creating executable utility scripts, used to load test data | | [Bash](https://www.gnu.org/software/bash/) |
| Sublime with Emmet | Text editor and productivity tools | | [Sublime](http://www.sublimetext.com/) |
| Node Inspector | Server side process debugger for Node.JS applications | | [Bajtos, M.](https://github.com/node-inspector/node-inspector) |
| Postman | Used to build and replay HTTP GET, POST, PUT and DELETE payloads to API endpoints | | [Postman](https://www.getpostman.com/) |
| GitHub | Source Code Repository  Code built locally and pushed to assignment 5 repo  <https://github.com/ah903/assignment05.git>  Code deployed to Heroku by pushing to a separate repository  [git@heroku.com:attire.git](mailto:git@heroku.com:attire.git) | | [Assignment 5](https://github.com/ah903/assignment05.git) |
| Visio | Visio for creating architectural diagrams | | [Microsoft](https://products.office.com/en-gb/visio/flowchart-software) |
| **Testing Tools** | | | |
| Jasmine | Researched Only | | [PivotalLabs](http://jasmine.github.io/) |
| Karma | Researched Only | | [Karma](http://karma-runner.github.io/0.13/index.html) |
| Mocha | Researched Only | | [Mocha](https://mochajs.org/) |
| **Project Management Tools** | | | |
| YoDiz | Agile Planning tool for backlog, story, sprint and release planning and management | | [YoDiz](http://www.yodiz.com/) |

# Bibliography

Dayley, B. (2014) *Node.js, MongoDB and AngularJS Web Development* 2nd Edition. New Jersey USA: Addison Wesley

Dickey J (2015) *Writing Modern Web Apps with the MEAN Stack* 1st Edition USA: Peachpit Press

Chacon, S. & Straub, B. (2014) *Pro Git* 2nd Edition. New York USA: Apress

Masse, M. (2012) *REST API Design Rulebook* 1st Edition CA USA: o’Reilly Media Inc

**Git**

Vogel, L. (2015) *Git Tutorial* <http://www.vogella.com/tutorials/Git/article.html#firstgit_repocreation> (Accessed 22 Aug 2015)

Orsini, L. (2013*) GitHub for Beginners, Commit Push and Go* Available at: <http://readwrite.com/2013/10/02/github-for-beginners-part-2> (Accessed 22 Aug 2015)

Atlassian (no date) *Setting up a repository* Available at: <https://www.atlassian.com/git/tutorials/setting-up-a-repository/> (Accessed 22 Aug 2015)

Atlassian (no date) *Git Workflows and Tutorial* Available at: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow> (Accessed 22 Aug 2015)

Atlassian (no date) *Making a Pull Request* Available at: https://www.atlassian.com/git/tutorials/making-a-pull-request/ (Accessed 22 Aug 2015)

Driessen, V. (2010) *A Successful Git Branching Model* Available at http://nvie.com/posts/a-successful-git-branching-model (Accessed 23 Aug 2015)

Whitney, D. (2013) A Source Control Strategy that Just Works Available at: <http://www.davidwhitney.co.uk/Blog/2013/09/11/a-source-control-strategy-for-git-that-just-works> (Accessed 23 Aug 2015)

Bruno, E. (2013) *Git in the Enterprise* Available at: <http://www.ericbruno.com/papers/Collabnet/Enterprise_Git_Survey.pdf> (Accessed 30 Sep 2015)

**Node.js**

Dhal, R. (2011) Introduction to NodeJS Available at: <https://www.youtube.com/watch?v=jo_B4LTHi3I> (Accessed 30 Aug 2015)

Tutorials point (2015) *Node.js Tutorial* Available at: <http://www.tutorialspoint.com/nodejs/> (Accessed 25 Aug 2015)

Geisendorfer, F. (no date) *Felix’s Node.js Guide* Available at: <http://nodeguide.com/beginner.html#debugging-node.js-apps> (Accessed 25 Aug 2015)

Ford, A. (2014) *Debugging node.js w/node-inspector* Available at: <https://www.youtube.com/watch?v=03qGA-GJXjI> (Accessed 25 Aug 2015)

Gilani, H. (2015*) Bootstrapping a NodeJS app for dev and prod environment* Available at: <http://himanshu.gilani.info/blog/2012/09/26/bootstraping-a-node-dot-js-app-for-dev-slash-prod-environment/> (Accessed 22 Sep 2015)

Ertz, B. (2014) *Creating and publishing a NodeJS module* Available at: <https://quickleft.com/blog/creating-and-publishing-a-node-js-module/> (Accessed 23 Sep 2015)

Voss, L (2014*) NPMs year in numbers* Available at: <http://blog.npmjs.org/post/106746762635/npms-year-in-numbers-2014> (Accessed 30 Sep 2015)

Read, B. (2014) NodeJS Apps *What’s the difference between installing locally and globally* Available at: <http://www.bartread.com/2014/02/17/whats-difference-locally-globally-installing-npm-packages> (Accessed 30 Sep 2015)

**Hosting**

MongoLab (2015) *Fully managed MongoDb as a service* Available at: <https://mongolab.com/> (Accessed 22 Sep 2015)

Heroku (2015) *Heroku Platform* Available at: <https://www.heroku.com/> (Accessed 22 Sep 2015)

Ravulavaru, A. (2014) Deploying Node Applications to Heroku Available at: <http://www.thejackalofjavascript.com/deploying-node-application> (Accessed 22 Sep 2015)

**Mongo**

MongoDb (2015) Data Modelling Introduction Available at: <http://docs.mongodb.org/manual/core/data-modeling-introduction> (Accessed 03 Sep 2015)

Leite, M (2015) *Creating* *a RESTful API with the MEAN Stack* Available at: <https://www.mongodb.com/blog/post/building-your-first-application-mongodb-creating-rest-api-using-mean-stack-part-1> (Accessed 01 Sep 2015)

MongoDB (2015) *Managing Social Content with MongoDB* Available at: <https://www.mongodb.com/presentations/webinar-managing-social-content-mongodb> (Accessed 31 Aug 2015)

Annunziato, J. (2015) *MongoDB and Mongoose* Available at: https://www.youtube.com/watch?feature=youtu.be&v=YH5sFprHgl0&app=desktop (Accessed 03 Sep 2015)

Ward, A. (2015) [*Importing json from file into mongodb using mongoimport*](http://stackoverflow.com/questions/30380751/importing-json-from-file-into-mongodb-using-mongoimport) Available at: http://stackoverflow.com/questions/30380751/importing-json-from-file-into-mongodb-using-mongoimport (Accessed 01 Sep 2015)

Mei, S. (2013) Why you should never use MongoDB Available at <http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/> (Accessed 01 Sep 2013)

**Mongoose**

Mongoose (no date) *Mongoose Quick Start* Available at*:*  <http://mongoosejs.com/docs/> (Accessed 01 Sep 2015)

Hernandez, M. (2013*) NodeJS and MongoDB Getting started with Mongoose* Available at: <http://blog.modulus.io/getting-started-with-mongoose> (Accessed 01 Sep 2015)

LearnCode.Academy (2014) *Node.js MongoDB Tutorial using Mongoose* Available at: <https://www.youtube.com/watch?v=5e1NEdfs4is> (Accessed 01 Sep 2015)

Karpov, V. (2015) *Better AngularJS Form Validation with Mongoose* Available at: <http://thecodebarbarian.com/2015/02/20/better-angularjs-form-validation-with-mongoose> (Accessed 15 Sep 2015)

**Express**

Larsen, L. (2013) *Generator - Node Express* Available at: <https://www.npmjs.com/package/generator-node-express> (Accessed 10 Sep 2015)

Shtylman, R. (2015) *ExpressJS Middleware Demystified* Available at: <https://www.safaribooksonline.com/blog/2014/03/10/express-js-middleware-demystified/> (Accessed 03 Sep 2015)

Shtylman, R. (2015) *Paramerterised Routes in Express.JS* Available at: https://www.safaribooksonline.com/blog/2014/03/13/parameterized-routes-express-js (Accessed 03 Sep 2015)

Express (no date) *Using Middleware* Available at: <http://expressjs.com/guide/using-middleware.html> (Accessed 03 Sep 2015)

**RESTful API Design**

Learn Coding Tutorials (2015*) MEAN Stack RESTful API Tutorial* Available at: <https://www.youtube.com/watch?v=kHV7gOHvNdk> Accessed (Accessed 01 Sep 2015)

Sahni, V. (no date*) Best Practices for Designing a Pragmatic RESTful API* <http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api> (Accessed 01 Sep 2015)

Wheelan, H. (2012) *HATEOAS 101 An Introduction to a REST API Style* Available at: <http://apigee.com/about/blog/technology/hateoas-101-introduction-rest-api-style-video-slides> (Accessed 01 Sep 2015)

Harrigan, D. (2012) *Versioning your API* Available at: <http://daneharrigan.com/2012/08/versioning-your-api> (Accessed 01 Sep 2015)

Beale, M. (2013) *Suggested REST API Practices* Available at: <http://madhatted.com/2013/3/19/suggested-rest-api-practices> (Accessed 01 Sep 2015)

Smith, S. (2014*) Beer Locker – Building a RESTful API with Node* Available at: <http://scottksmith.com/blog/2014/05/05/beer-locker-building-a-restful-api-with-node-crud/> (Accessed 02 Sep 2015)

Sevilleja, C. (2014) *Build a RESTful API with Node and Express 4* Available at: <https://scotch.io/tutorials/build-a-restful-api-using-node-and-express-4> (Accessed 03 Sep 2015)

Kernan, B. (2014) *Create a REST API with NodeJS* Available at: [http://bigspaceship.github.io/blog/2014/05/14/how-to-create-a-rest-api-with-node-dot-js](http://bigspaceship.github.io/blog/2014/05/14/how-to-create-a-rest-api-with-node-dot-js/) (Accessed 03 Sep 2015)

**AngularJS**

Whalin, D (2013) *AngularJS In Sixty Minutes* Available at: http://fastandfluid.com/publicdownloads/AngularJSIn60MinutesIsh\_DanWahlin\_May2013.pdf (Accessed 04 Sep 2015)

Annunziato, J. (2015) *Introduction to AngularJS* Available at: <https://www.youtube.com/watch?v=iT5rwGcEAYs> (Accessed 05 Sep 2015)

Annunziato, J. (2015) *AngularJS Routing* Available at: <https://www.youtube.com/watch?v=jelXe-t2MPQ> (Accessed 05 Sep 2015)

Deitz, F. (no date) *Recipes with AngularJS Using Route Location to Implement a Navigation menu* Available at: <http://fdietz.github.io/recipes-with-angular-js//urls-routing-and-partials/using-route-location-to-implement-a-navigation-menu.html> (Accessed 20 Sep 2015)

Ben-Yosv, A. (2015) *Notifying about changes from Services to Controllers* Available at: <http://www.codelord.net/2015/05/04/angularjs-notifying-about-changes-from-services-to-controllers/> (Accessed 17 Sep 2015)

McKeachiein, C. (2014) *UI-Router: Why many developers don’t use AngularJS’s built-in router* Available at: http://www.funnyant.com/angularjs-ui-router (Accessed 15 Sep 2015)

Charemza, M. (no date) *Using and Chaining Promises in AngularJS* Available at: <https://www.airpair.com/angularjs/posts/angularjs-promises> (Accessed 16 Sep 2015)

Sevilleja, C. (2014) *Setting up a MEAN Stack Single Page Application* Available at: https://scotch.io/tutorials/setting-up-a-mean-stack-single-page-application (Accessed 03 Sep 2015)

**Jade**

Mardan, A. (2015) *To Engineers who tried to use Jade Template Engine and can’t get Started* Available at: <http://webapplog.com/jade/> (Accessed 09 Sep 2015)

Lindesay, F. (no date) *Jade Node Template Engine* Available at: <http://jade-lang.com/reference/mixins/> (Accessed 09 Sep 2015)

Bracey, K. (2014) *Baking Bootstrap Snippets with Jade* Available at: <http://webdesign.tutsplus.com/tutorials/baking-bootstrap-snippets-with-jade--cms-22798> (Accessed 06 Sep 2015)

Bahmutov, G. (2013) *AngularJS inside Jade Templates* Available at: <http://bahmutov.calepin.co/angularjs-inside-jade-templates.html> (Accessed 06 Sep 2015)

**Gulp**

Imhof, S. (2015) *Introduction to GulpJS* Available at: <http://stefanimhoff.de/2014/gulp-tutorial-1-intro-setup/> (Accessed 20 Sep 2015)

McCrae, C. (2014) *Building with Gulp* Available at: http://www.smashingmagazine.com/2014/06/building-with-gulp (Accessed 20 Sep 2015)

Ramires, M (2015*) Express Generator + Gulp* Available at: <http://marcoramires.com/article/Express-Generator-Gulp> (Accessed 20 Sep 2015)

Sotelo, C. (2015) *A Healthy Gulp Setup for AngularJS projects* Available at: <http://paislee.io/a-healthy-gulp-setup-for-angularjs-projects> (Accessed 21 Sep 2015)

Schmitt, M. (2014) Gulp.js Tutorial: SASS, Browserify, jade and workflow optimization Available at: <http://maximilianschmitt.me/posts/gulp-js-tutorial-sass-browserify-jade> (Accessed 25 Sep 2015)

**Testing**

Singer, M. (2014) *Unit Testing Express Middleware with Express and Mocha* Available at: https://www.youtube.com/watch?v=BwNMUVzo3vs (Accessed 20 Sep 2015)

Gil, Ignacio (2015) *Automated testing on the MEAN Stack* Available at: <http://www.techinsight.io/review/devops-and-automation/automating-unit-and-integration-testing-with-the-mean-stack> (Accessed 15 Sep 2015)

Drucker, B. (2014) Testing AngularJS Apps Using Karma Available at: <https://www.airpair.com/angularjs/posts/testing-angular-with-karma> (Accessed 15 Sep 2015)

Sarcevic, I. () Available at: <https://semaphoreci.com/community/tutorials/getting-started-with-node-js-and-mocha> (Accessed 15 Sep 2015)

Cruz, R (2015) Getting Started with Mocha Testing Available at: http://ricostacruz.com/til/get-started-with-mocha.html (Accessed 15 Sep 2015)

**Passport**

Annunziato, J. (2015) *Introduction PassportJS* Available at: https://www.youtube.com/watch?v=dXkqNWLyJ7E (Accessed 07 Sep 2015)

Annunziato, J. (2015) *Use PassportJS* to authenticate and manage user access to MEAN Available at: https://www.youtube.com/watch?v=sgVL2kw0hSA (Accessed 07 Sep 2015)

**Full Stack**

Yared, P. (2014) *The rise and fall of the full stack developer* Available at: http://techcrunch.com/2014/11/08/the-rise-and-fall-of-the-full-stack-developer (Accessed 22 Sep 2015)

**Open Source**

Bridge, R. (2013) *Open Source Software, the Advantages and Disadvantages* Available at: <http://entrepreneurhandbook.co.uk/open-source-software> (Accessed 30 Sep 2015)

Rubens, P (2014) *How to overcome hidden barriers to open source adoption* Available at: <http://www.cio.com/article/2464160/open-source-development/how-to-overcome-hidden-barriers-to-open-source-adoption.html> (Accessed 30 Sep 2015)

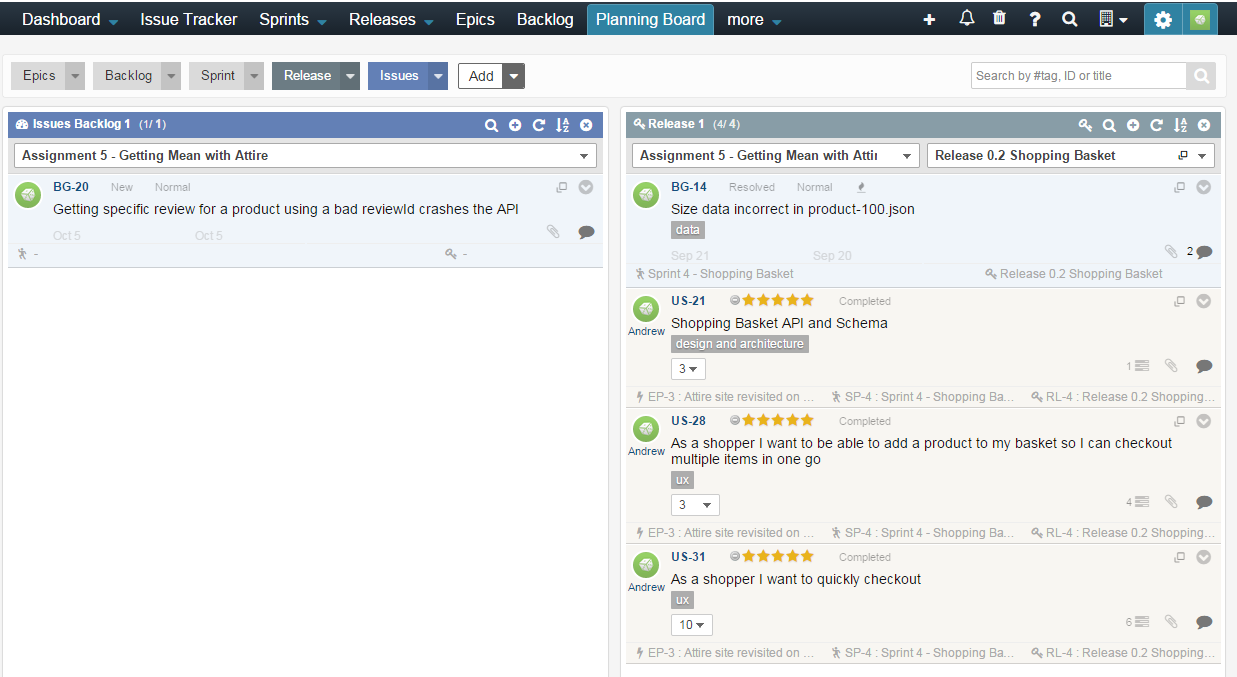
W3Techs (2015) *Usage statistics and market share of Apache for Websites* Available at: <http://w3techs.com/technologies/details/ws-apache/all/all> (Accessed 30 Sep 2015)

Heath, N (2014) *Six open source security myths debunked and eight real challenges to consider* Available at: <http://www.zdnet.com/article/six-open-source-security-myths-debunked-and-eight-real-challenges-to-consider> (Accessed 30 Sep 2015)

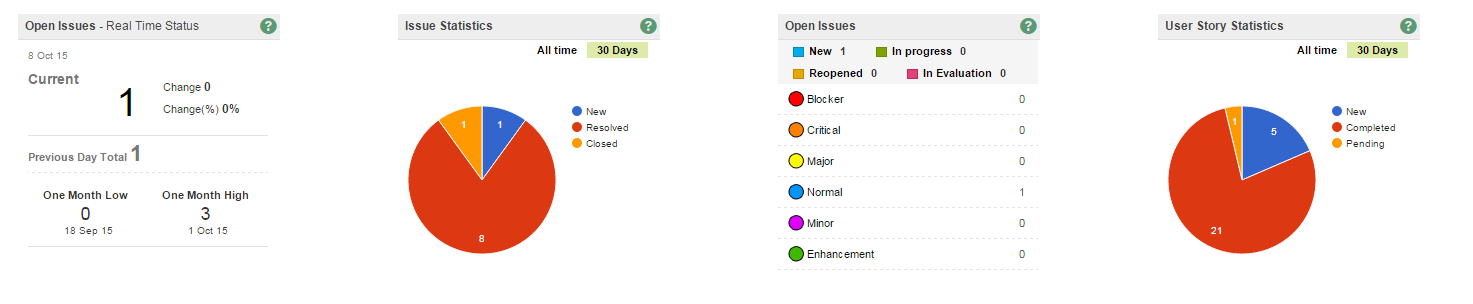
# Appendix A – Project Planning

The charts below were extracted from Yodiz to illustrate planning and progress tracking thorughout the Attire development. Access to the project is available using the email invitation sent separately or by request.

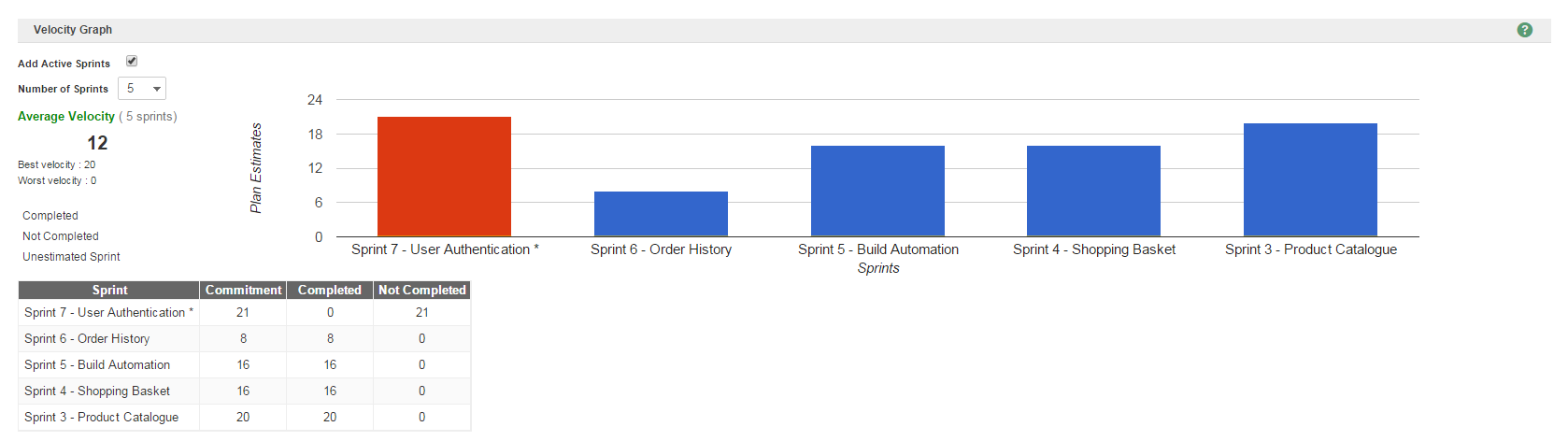
## A1. Planning Board



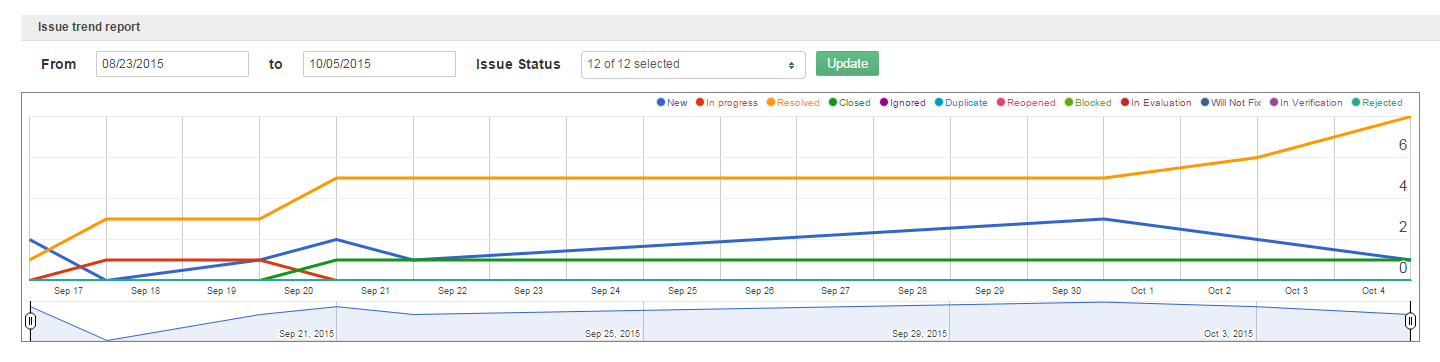
## A2. Progress Dashboard



## A3. Velocity and Productivity



## A4. Defect Tracking and Trends



# Appendix B – Dependency Management

The table below describes the dependencies for the Attire web application, detailing the NPM packages used for development and those installed locally.

Some packages such as Express and Gulp are installed both locally and globally. The Attire application uses the local version but it is also useful to be able to run the application from the command line to scaffold new applications and run build tasks.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Package | Version | Description | Scope | Dependency Type |
| bower | 1.5.2 | Client side package manager | Global |  |
| express | 4.13.3 | Express server | Global |  |
| express-generator | 4.13.1 | Express generator | Global |  |
| gulp | 3.9.0 | Build Engine | Global |  |
| node-inspector | 0.12.2 | Debugger for Node applications | Global |  |
| nodemon | 1.7.1 | Node server watcher | Global |  |
| npm | 2.14.2 | Node package manager | Global |  |
| yo | 1.4.8 | Yeoman Scaffolding | Global |  |
| body-parser | 1.13.3 | Express middleware used to extract data from request body | Local | Dependency |
| browser-sync | 2.9.3 | Live reload utility | Local | Dev Dependency |
| cookie-parser | 1.3.5 | Express middleware used to extract cookie data from the request | Local | Dependency |
| debug | 2.2.0 | Utility for debug logging | Local | Dependency |
| del | 2.0.2 | Utility package to manage files and folders | Local | Dev Dependency |
| express | 4.13.3 | Express server | Local | Dependency |
| gulp | 3.9.0 | Gulp build pipeline | Local | Dev Dependency |
| gulp-concat | 2.6.0 | Gulp file concatenation task | Local | Dev Dependency |
| gulp-cssmin | 0.1.7 | Gulp css minification task | Local | Dev Dependency |
| gulp-inject | 3.0.0 | Gulp file dependency task | Local | Dev Dependency |
| gulp-jshint | 1.11.2 | Gulp javascript lint task | Local | Dev Dependency |
| gulp-uglify | 1.4.1 | Gulp javascript minification task | Local | Dev Dependency |
| jade | 1.11.0 | Jade template engine | Local | Dependency |
| mongoose | 4.1.5 | Server side Mongo object data model | Local | Dependency |
| morgan | 1.6.1 | Server side logging | Local | Dependency |
| nconf | 0.8.0 | Server side configuration management | Local | Dependency |
| serve-favicon | 2.3.0 | Express middleware to serve site favicon | Local | Dependency |

## B1. Future Packages

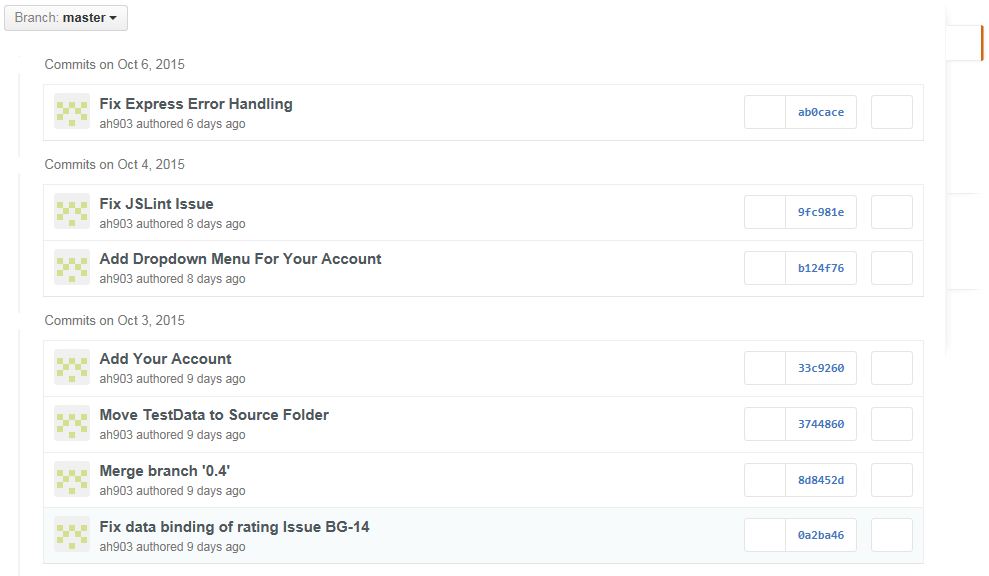
The table below shows packages that we could be used in the ongoing development of the Attire web application. The two key areas of focus are authentication and test automation and this is reflected in the package choice

|  |  |  |  |
| --- | --- | --- | --- |
| Package | Description | Scope | Dependency Type |
| gulp-if | Could be used to merge production and development build scripts conditionally testing environment variables to build the right outputs | Local | Dev Dependency |
| PassportJS | Authentication middleware for NodeJS applications used in conjunction with express sessions to establish encrypted session cookies upon login. The users identity may then be parsed from the cookie when accessing server resources. Could also be used to allow authentication strategies based on OAuth for example. | Local | Dependency |
| gulp-mocha | Gulp test runner for mocha test suites used to run server side test suite  var gulp= require(“gulp”);  var mocha=require(“gulp-mocha”);  gulp.task(“Test”, function(){  return gulp  .src(“test/\*.js”)  .pipe(mocha());  }); | Local | Dev Dependency |
| chai | Assertion framework that works with different test frameworks but commonly paired with Mocha to allow tests assertions such as  assert(x).equal(10)  assert(x).not.equal(“Test”)  expect(x).to.equal(10)  x.should.equal(y) | Local | Dev Dependency |
| sinon | Used in conjunction with mocha and chai to test the parameter and return values inside callback and other functions | Local | Dev Dependency |

# Appendix C - Version Control

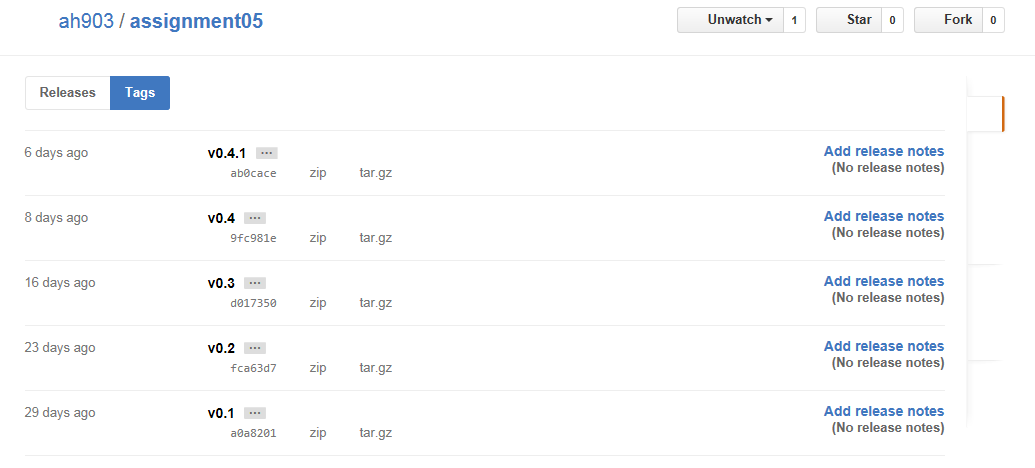
## C.1 Commit History

The screen shot below shows a subset of the commit history for the project detailing the SHA1 hash associated with each action.



## C2. Release Tagging in Git

The screen shot below illustrates the regular weekly release cycle of completed code. Releases are tagged with a version number where all numbers are less than one indicating the code is not yet production ready. Note the last release v0.4.1 is a patch release as identified by the additional minor digit in the tag.



# Appendix D – Post Install Build

The diagram below shows the gulp build tasks that are executed post deployment to Heroku. In production the Attire application is served from a folder named public. This folder does not exist in the source code and is ignored by git if created. The job of the production build gulp script is to copy source code assets in the src folder to the public folder, optimising for performance as along the way.

C:\Users\145987\Desktop\Masters\Assignment 5\Production Build.emf

By default gulp executes tasks in parallel although this behaviour can be modified by making one task depend on the output or completion of another (Schmitt, M. 2014). The diagram illustrates this with arrows for example all tasks at the second level of the hierarchy will execute in asynchronously parallel but only after the Clean Public Folder task has finished

The attire project also includes two additional gulp scripts designed to support the development workflow. The first script called synchronise.js monitors source files and reloads the browser when any changes are detected, saving the developer some valuable time.

The second gulp script, called devbuild.js currently carries out similar tasks to the production build. Ideally this script would evolve in a different direction and execute an automated test suite developed using Jasmine or Mocha. Test automation provides a safety net checking that code changes do not introduce unforeseen side effects. Of the two test frameworks Jasmine is more mature and comprehensive but overall the choice of tool is less important than the principle of test automation.

# Appendix E – Data Model

The diagram below shows the data model for the Attire web application. The data domain is made up of four collections, customers, orders, products and reviews. With the exception of review documents all documents are include other data structures embedded within them so a customer document includes an address and a card object. Similarly the products document includes lists of available colours and sizes.

**C:\Users\145987\Desktop\Masters\Assignment 5\Data Model.emf**

The design separates the collections based on an understanding of how the data is used and how it might grow over time. For example although reviews relate to products, review data is created by the user and product data by Attire. There is no need for products and reviews to be updated together. Similarly the growth of reviews is likely to exceed the growth of products, perhaps by quite a large multiple so separating the collections gives is scalability options.

Similarly the relationship between customer and orders is a reference rather than as an embedded document. This is a less clear cut design choice especially as customers might update their details whjen placing an order. Here the decision to separate orders from customers and products is predicated by a desire to maintain a history so we always have a point in time record of where an order shipped to without having to rely on the customer record.

Document references are implemented much as in traditional relational databases through foreign keys although Mongo offers the possibility of implementing many to many relationships with intermediating permutation tables. It is worth noting that querying referenced data may require a multiple queries so for example if we wanted to get a product and all of its reviews we would need to execute two queries and join the resulting outputs.

# Appendix F – Synthesizing Test Data

This assignment required a volume of test data for products and reviews beyond what is feasible to create by hand. Testing with representative data as soon as possible in the development process is a good way to identify performance issues and validate the data design and API meets the needs of the application.

One of the advantages of using Mongo as a database technology is that its use of JSON and schemaless nature makes it easy to create data files and incrementally refine as development progresses.

To create the data files a tool called JSON Generator by Omanashvilli, V. (no date) was used. The tool generates output based on a Mustache like template (see for example product-data-script.json and review-data-script.json files in the git repo). Unfortunately the tool is limited to generating one hundred items at a time which is a little annoying but still a big improvement on manual construction and a big win for repeatable processes.

With the output files in place the bash script dataload.sh was used to create a data build and load process using the mongoimport command line utility to connect to an instance of Mongo and upload the data (Ward, A. 2015).

The bash script is required to perform a little gymnastics to fix up the Mongo ids generated by our JSON generator. Whilst the generator creates perfectly serviceable id values it outputs them as strings where Mongo by default expects to find the string id packed within an object. This is corrected using a search and replace regular expression.

With more time it would be possible to rework this script to run through gulp and optionally incorporate database teardown and build as part of the automated deployment process.

# Appendix G – Attire Folder Structure

The diagram below shows the folder structure for Attire. This is based on the express-generator scaffold but reorganised separate server side code, client side code and utility code. In production client side Attire code is served from the public folder and build scripts copy assets from src to public.

In development it is possible through command line switches or configuration files to serve directly from src which is useful for live reload or from a local copy of public.

C:\Users\145987\Desktop\Masters\Assignment 5\Folder Structure.emf