Lifecycle Management for Docker UI

Gantry

Final Report (Semester 2)

Stephen Coady

20064122

Supervisor: Dr. Brenda Mullally

BSc (Hons) in Applied Computing

Gantry: Lifecycle Management For Docker UI

Stephen Coady

Declaration

Unless otherwise stated all the work contained in this report is my own. I also state that I have not submitted this work for any other course of study leading to an academic award.

ID: 20064122

Date: April 2017

Acknowledgements

I would like to sincerely thank my supervisor Dr. Brenda Mullally for her advice and support throughout this project. The guidance I received has been second to none and has made this project what it is. I also want to thank the wider computing staff - who have been instrumental in instilling a drive and hunger to learn which I did not know I had. My time at WIT will always be a fond memory due to this.

I would also like to thank my fellow students for their help and camaraderie throughout my four years. I have made some great friendships in my time at WIT which I know will last beyond my time here.

To Leigh Griffin, David Martin and indeed Red Hat in general I want to express my deepest gratitude. You have equipped me with tools seldom found in the lecture hall. To say I am a better developer as a result of your input and guidance is a gross understatement.

To my sister Emily, thank you for your support and encouragement over the last four years. To my parents John and Sylvia, thank you for everything. Your unwavering belief in me has been a deep source of motivation.

Finally, to Denise, you have been my source of inspiration over these past four years. I could not have asked for a better person to read all the boring papers I wrote and to listen intently to me drone on for hours at a time. For that I am eternally grateful. We can finally have our weekends back now, I promise.

Contents

Definition of Common Terms

API Application programming interface. A set of endpoints which lead to functions and application logic. Allows developers to expose application functionality without directly exposing the code within the application. 7, 9, 22

Backlog The remaining items which, when completed, will render the current release or version of a product 'done'. 10, 18, 19, 29, 38

Bootstrap A front end css framework designed to give developers more power with less work when creating html pages. Makes use of a standard 12-width grid system to provide an easy-to-use css styling method. 16

CI/CD The automated process of ensuring all integrated code is automatically deployed to a pre-defined location. Ensures that by releasing code it is automatically tested and then deployed depending on the results. 10

CLI Command Line Interface. A means of interacting with a computer program where a user can only enter commands in the form of successive lines of text. 7, 8

Code Coverage Refers to the percentage of code which has been executed by tests. Not an indication of test quality but instead can be used to identify pieces of code which has been missed by test cases. 22–24

Code smell A piece of code which may perform as expected in some or all cases but that introduce a possibility of an unexpected result. Should be avoided. They are normally decided upon by the community. 13, 23, 48

Continuous Deployment The act of continuously deploying software which has passed acceptance tests to a live system. 12, 23

Continuous Integration The act of continuously merging newly created software with the current working version, validated by automatic build and testing tools allowing for early detection of faults. 9, 12, 23, 26, 29, 31, 32

CRUD Create, Read, Update, Delete. CRUD defines the four cornerstones of API functionality. An API should provide these functions to any user using the API. 9

CSS Cascading Style Sheet. Used for applying styles to markup text primarily displayed on web pages. 16, 25

Docker An application which manages containers running on a host. 7, 11, 17

Docker container An isolated package which bundles everything required to run an application within a container. The application can then be run within this container without regard to the underlying architecture. 8, 11, 14, 15, 26, 31

Docker daemon The server-side component of the Docker Engine. 7

Docker host The computer, server or virtual machine on which the Docker application is installed. One user may manage several Docker host's, depending upon how many computers they have installed it on. 8, 11, 16

Docker image A template from which many containers can be started. Can be pushed an pulled to/from a remote Docker registry. 8, 11, 16, 24

Dockerfile A template from which a Docker image can be built. Defined in plaintext and interpreted by the Docker Engine. Can define which commands should be run etc. 34

DockerHub An online platform to store Docker images. Can be pulled from by anyone once the repository is public. Essentially a way to share any built Docker images. 24

Git A version control system which allows source code to be tightly controlled and stored in a central repository. 9, 20, 24, 25, 28

Github An online platform to store git repositories. Allows for multi-developer collaboration on projects. 9, 20, 24

HTML Hypertext Markup Language. A means of tagging text using "markup" to allow them to be positioned, styled and linked on web pages. 16

JWT JSON Web Tokens. A means to provide authentication over the internet using a simple string of characters. 17

REST Representational state transfer. A means of providing communication between computers across the internet. RESTful web services allow systems interact with each other using completely stateless operations. 8

Scrum An iterative and incremental agile framework where a software development team works as a unit to achieve a common goal. 18, 39

SonarQube A tool to scan source code. It can inform developers of coding best-practices and highlight "code smells", which are essentially when best practices are not adhered to. Also highlights technical debt. Code is deemed either 'passing' or 'failing', meaning the code has been deemed either sufficient or insufficient in the following categories: Code Test Coverage, Bugs, Vulnerabilities and Technical Debt Ratio. 9, 13, 24, 29

Sprint A short cycle of development - normally one or two weeks long. Allows for rapid change of development direction if project requirements change. 9, 18–20, 29, 36

Staging Server A computing unit exposed on the network which typically runs an incomplete or demo version of a piece of software. Generally used to show the in progress development to any product stakeholders. 24

Technical Debt The extra development work which arises as a result of implementing an easier but less robust solution. 13, 23, 29

Travis A website which allows for automatic builds of code and feedback of results. Can be used to automatically deploy code dependent on test results also. 9, 12, 24

UI User Interface. A graphical view towards an application which exposes functionality by using buttons or other components. Typically utilised using a computer mouse. 7, 8

Vagrantfile A template file which provides Vagrant with all the instructions required to create the desired environment. Written in the Ruby programming language. 13

Virtual Machine An abstracted virtual version of a computer. Normally runs on another host with a dedicated piece of software to manage it. Appears to the user as a real computer, however all or most components are defined by software. 11, 13

1 Introduction

This report will aim to guide the reader through the planning and development of the Lifecycle Management for Docker UI application, termed Gantry for the remainder of this report. After reading this report the reader should have a clear idea of why the application was built, what was used to build it and how the process was carried out.

This project is being undertaken with a local company, Red Hat, acting as product owners. Red Hat have a vested interest in solving the problem discussed in Section 1.1 as the Red Hat Mobile Application Platform uses containers extensively. This will ensure that the initial requirements will accurately shape the project and the delivered solution passes a product review by Red Hat Mobile.

This application will be built using open source principles and best practices, enabling it to be maintained and improved by any developer who wishes to contribute. For this reason many of the decisions made and processes employed were done so with an open source final product in mind.

1.1 Problem Statement

Currently the Docker application does not ship with any bundled UI. When installed, it is comprised of a client and a server side component (Docker, 2017*a*). The server side exposes itself through an API and is ultimately responsible for controlling all aspects of Docker on the host such as containers, images, networks and volumes etc. The API exposed by the server-side application of the Docker Engine is consumed by the Docker CLI which is the client side application. A graphical representation of the complete Docker engine can be seen in Figure 1.

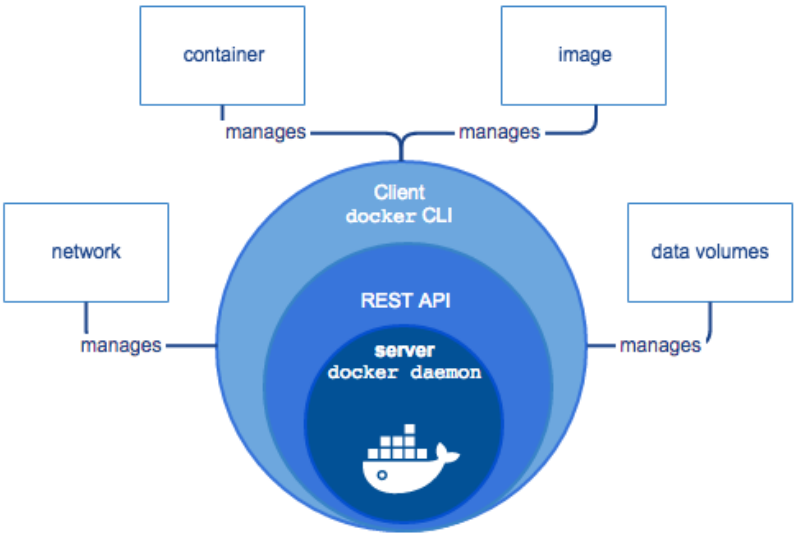

Figure 1: *Docker Engine Components. Credit: (Docker, 2017a)*

This model is extremely versatile as it allows the developer to control any Docker daemon (the server-side component of a Docker installation) once they have access to the command line of the

host Docker is running on. In fact, if the API exposed by the Docker daemon is exposed remotely then the developer does not need access to the host's command line, instead they can directly access the API remotely.

While the CLI gives developers full control over the server-side component of a Docker host it also has its drawbacks.

- Learning curve - the person using the command line must be familiar with typical commands used to achieve certain tasks. This precludes anybody without these skills from using Docker.
- Vast set of Docker commands - there are a vast number of commands available to use from the Docker CLI. This is also a learning curve as even a developer who is familiar with a CLI must first learn the Docker commands to be able to use the client-side application.
- User friendliness - The command line does not produce content that is easily readable and can often disrupt the data's format depending on things like screen size etc.

According to recent studies the usage of containers (and therefore of Docker) in production has increased at an exponential rate in recent years (Arijs, 2016). This is due to the many benefits provided by containers as a deployment mechanism. While the problems discussed above are a pain point for experienced developers they are the price one must pay to use a service. However, as new and possibly inexperienced developers become attracted to Docker the use of the command line will shift from being a pain point to a barrier to entry. This is where Gantry aims to provide an easy point of entry.

1.2 Aims and Objectives

The aim of this project is to address all of these problems whilst also trying to increase the functionality available to anybody who wishes to use Docker.

At a high level the primary objectives of this project are:

- Fully functional server-side application
- Expose this application through a RESTful API.
- A fully functional front-end application
- A Docker image built to allow easy distribution of the application

These objectives will then provide the following functionality:

- A graphical user interface
  - This user interface will remove the barrier to entry for new and inexperienced developers while also creating a fast and efficient overview to experienced developers who may not always require the command line for simple tasks. The overall aim of this UI is to provide an educational tool with the potential to be built upon.
- A Docker container which has no external dependencies. The benefit of this is that the user does not need to install the application in the traditional sense. This will be discussed further in Section 2.
- An independent API which can be consumed by any front end application

– This will provide flexibility if the front end framework needs to be changed further down the line

1.3 Semester One Summary

To gain a complete understanding of the process used to create this product and what was achieved previously the reader is encouraged to refer to the semester one report, which is available in Appendix F.

However, a brief summary of the prototype application will be provided. The prototype was completed in semester one and the development consisted of two sprints, the first lasting two weeks and the second taking place over three weeks.

1.3.1 Semester One Sprints

Sprint 1

Goal: Initial research and investigation of the core technologies which at this point were candidates for the project.

Achieved:

- The developer environment was set up (Vagrant + Docker installation).
- A basic Express API was created and run on test endpoints with no functionality.
- A basic Dockerfile was created to run the application within a Docker container.
- The Dockerode third party module was investigated and integrated into the project (Dias, 2017). It was used to provide CRUD functionality to the container and image endpoints.

Sprint 2

Goal: Completion of the continuous integration pipeline along with the addition of further API endpoints.

Achieved:

- The prototype application was improved to include further and more complicated CRUD calls.
- A Travis account was created and linked with the public Github repository of the application. Both of these can be seen in Appendices A and B.
- A travis.yml file was created which details the steps to be followed when building the application. The Travis repository in Appendix B will now build the application and run the unit tests every time a Git commit is made.
- A SonarQube account was associated with the Github repository and incorporated in the build pipeline. This can be viewed in Appendix C.

The final product of both sprints was a prototype application which provided a proof of concept. It also validated the technology decisions which were discussed in detail in report 1 and which will be discussed further in Section 2. While it was not a functionally complete application it did prove useful when reflecting on semester one.

1.3.2 Reflection

Once the prototype application had been built it offered valuable insight which could be used when moving forward into semester two.

Learning outcomes include:

- Using the prototype as an introduction to agile development significantly helped to stock the backlog and gain an understanding of exactly what would be needed to successfully create this product.
- Feedback from my support group in the form of my supervisor and Red Hat also helped prioritise the development and gave a much better sense of awareness with regard to what was required in the project.

Technical learning outcomes include:

- More time should be spent making the test suite robust. While complex tests cases require initial investment, this initial outlay can save time going forward.
- Creating a large section of the CI/CD pipeline was a good investment also as it meant an improved development process was in place for semester two.
- Running the application at all times within a container is good practice as it means testing the application in the environment it will ultimately run in.
- The decision to use Agile methodologies was positive for the product as it allowed for a flexible and fast development cycle. This is fitting when the project span is short at 12 weeks.

2 Technologies

In this section of the report the technologies used to create the application will be examined and explained in detail.

2.1 Docker

Docker is a platform which allows developers to package their applications into isolated containers which contain only the software dependencies required by the application. A container is different to a virtual machine in that a container does not contain a full operating system (Docker, 2016).

Since a primary objective of this application is to manage a Docker host it makes sense to leverage the capabilities of Docker and run this application within a Docker container. This provides several benefits over distributing source code, such as:

- Portability - If a Docker image can be built and uploaded to a public repository then it makes it easier for other developers to pull and run the application.
- Ease of use - As the application will be running in a container a developer does not need to install any third party components on their system to use the application. They do not need to worry about their environment as once their system has Docker installed it will run the application.
- Ephemeral - Docker containers are designed to be 'throw-away'. This means that if this application needs to be quickly stopped and restarted then containers are the perfect vehicle to do this.

2.2 Node JS

Node JS is a server-side JavaScript runtime, it is built on the same V8 engine that powers the popular Chrome browser. It uses an event-driven, non-blocking I/O model that makes it lightweight, efficient and very fast. Node JS' package ecosystem, NPM, is the largest ecosystem of open source libraries in the world (Nodejs.org, 2016). Node JS provides an excellent way to build highly scalable network application which are non-blocking and extremely performant (Griffin et al., 2011). This means that if the application was adapted in the future to deal with large numbers of servers then the technology choice will be able to deal with that.

Node JS was also deemed a good fit for this project as it has a large and extremely active online community. Since this is an open source project this will increase the likelihood of other developers taking part in the project and contributing. We can see in Figure 2 that there are currently (as of November 2016) more node modules available through the node package manager (NPM) than any other of the large package managers such as the ones used by Go, PHP, Python and Ruby.

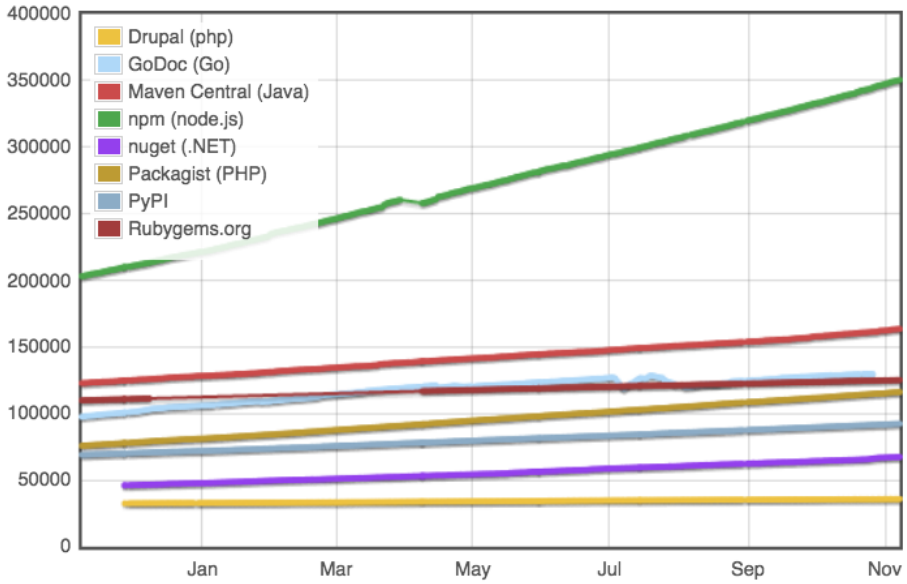

Figure 2: *Various Module Counts (Credit: modulecounts.com)*

The original idea for this project also came from the Red Hat Mobile Application which is a node-based project built using a large amount of Docker images (RedHat, 2016). Therefore it made sense to align this technology stack with this project.

2.3 Angular JS

Angular JS is a web framework for building dynamic web applications using Javascript as a controller. It uses HTML as its template language to display the information passed to it from the application controller (AngularJs, 2017).

For this application the front end requirements were relatively low. Once the framework provided a mechanism to show dynamic content then it was a candidate. For this reason several were evaluated and Angular was chosen as it had the lowest barrier to entry. The front-end application will be discussed further in Section 3.

One requirement when choosing the front end framework was that it was modular. If, at some point in the future, this project required a new front end framework for some unknown feature then it must be relatively easy to remove Angular and use the new feature. This is discussed further in Section 3.2.

2.4 Travis CI

Since this project was developed as an open source project it made sense to have a transparent and fully automated build process integrated into the project. For this reason continuous integration and continuous deployment in this project is handled by Travis CI. It is a feature-rich service free to use for open source projects and has a large user base. The set up used in this project for testing and continuous integration and deployment will be explored further in Section 4. The Travis repository for this application is available in Appendix B.

2.5 SonarQube

In general all projects tend to take shortcuts to solve problems. This is particularly true for a final year project where the student must balance developing a substantial code base alongside other modules. This can mean that the final product of many final year projects can suffer performance issues and still be working software. In industry technical debt is a huge problem as it accrues over time and must be paid off in the long run. It can become so ingrained that it can be almost impossible to refactor out of the code without huge cost, either time or financial. With this in mind all code in this project is held to a high standard. From the very beginning this project will aim to be a performant, reusable and readable system that other developers can use without confusion.

To achieve this an open source tool SonarQube will be used (SonarQube, 2016). This tool analyses the output of tests to give detailed information about test coverage and it also scans the code to find possible bugs and code smells. It can give useful information such as the current amount of technical debt. SonarQube will be discussed in further detail in Section 4.5. The SonarQube repository for this project can be seen in Appendix C.

2.6 Vagrant

Vagrant is a technology to create configurable, reproducible and portable environments by using a set of programmable steps to produce a virtual machine which the developer can use to develop in (Vagrant, 2017). While this is just one use-case of Vagrant it is the main reason it was used in this project.

Since this project is open source it is useful to have one standardised VM within which all development can take place. This virtual machine can then be shared (either as its own separate repository or included in the main application repository). This is useful as it enables any developer who wishes to contribute to the project to instantly have the required environment. For instance, a developer can contribute to this project by using the Vagrantfile supplied without requiring them to first install Docker or Node JS.

3 Design

The formal information modelling of the system remains unchanged from Report 1 and is available in Appendix G. However in this section an examination of the system overview and all aspects of design will be discussed.

3.1 System Architecture Overview

To gain a better understanding of the system it is important to look at it from a high-level architectural view. This will give the reader an idea of how all major components fit together.

A diagram showing each major component of the application can be seen in Figure 3 below.

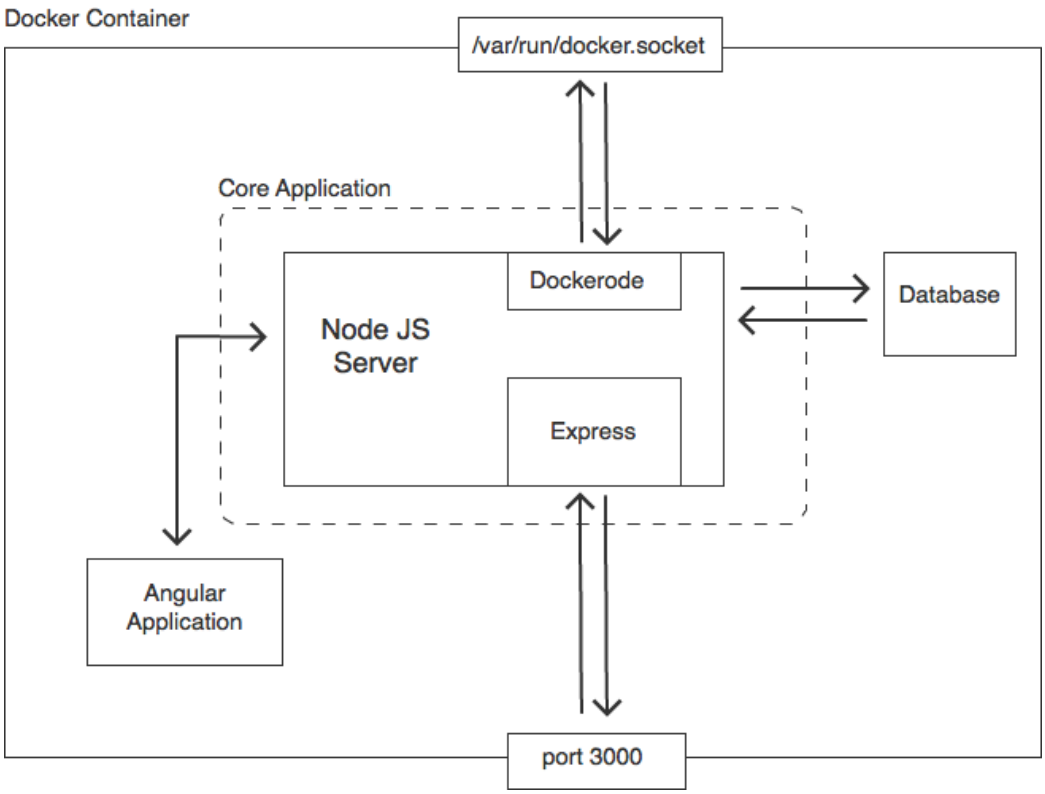

Figure 3: *System Architecture*

The whole application is bundled in a Docker container. This is represented in Figure 3 and means the architecture will never vary across different systems.

Most of the application's logic is in the Node JS server. It is the most important component as all others are built around it. It provides a mechanism to access the application through an Express API while also communicating with the Docker API using Dockerode as an external library. It also performs all communication with the database and serves the front end application.

It is important to note that the design of this application is deliberately modular. This means a low cohesion between each component of the application which in turn means that any one component can be swapped for another which provides the same functionality. For instance, if in the future it was decided that a different front end framework was required then Angular could easily be

replaced. Everything in Figure 3 outside of the dashed rectangle can be easily swapped for a different technology in this same fashion.

The Node JS application 'listens' on port 3000 within the Docker container which in turn maps to port 3000 on the host it is running on. This is completely configurable by the user and any available port can instead be chosen. It also maps the Docker unix socket '/var/run/docker.socket' as a volume on the Docker container.

3.2 Front End Design

As previously discussed the requirements for the front-end application are relatively low. They are:

- The application should allow for dynamic content.
- It should allow for re-use of code to keep the codebase small and manageable, i.e. 'templating'.
- As this project is open source it should be a relatively well known framework, meaning low barrier to entry for potential contributors.
- The application should have a clean, minimalistic interface using a sidebar to navigate all available pages.
- It should be intuitive to use, with minimal need for documentation.
- It would be responsive - meaning it would scale well on smaller devices such as mobile phones and tablets.

Therefore it was decided Angular was the best choice as it satisfied all of these conditions.

3.2.1 Wireframes

With these design decisions in mind an initial batch of wireframes were created which would be used when writing the front end code. These wireframes were used to elicit feedback from Red Hat and from Dr. Brenda Mullally. That feedback focused on usability as well as good user interface practices which helped focus the development of the front end ensuring it stayed simple and intuitive. Several iterations of wireframe development were conducted and the final product of these iterations is included in this report. One of these, the containers view, can be seen below in Figure 4 while the rest are available in Appendix H.

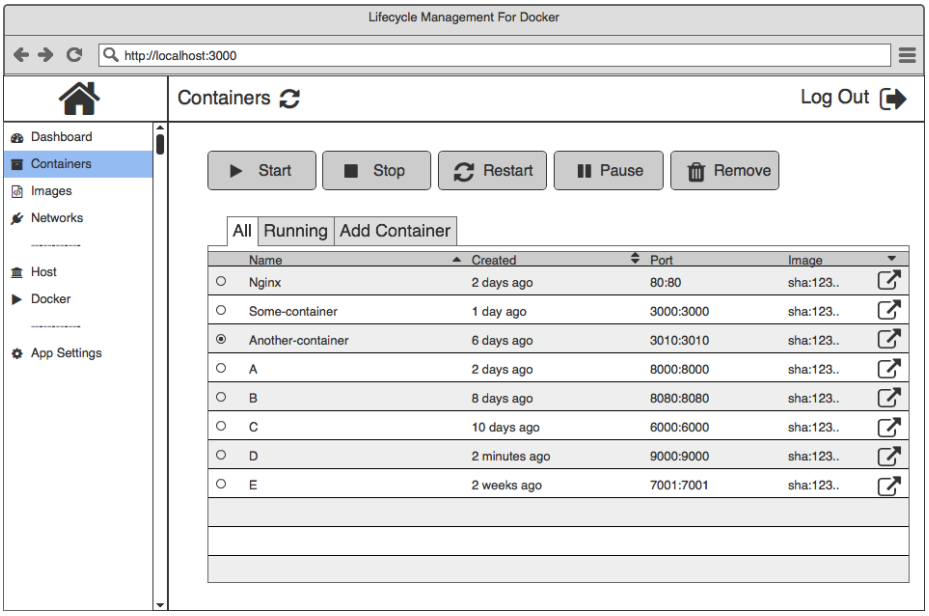

Figure 4: *Containers Mockup*

3.2.2 Mobile Web Application

The idea of creating a standalone dedicated mobile application was de-scoped early in semester one as it was decided that a web application would be a better fit for the following reasons:

- Since the application would also need to be installed it would not make sense to require the user to install another application just to manage the first one.
- Not restricting users to certain devices was a major factor in deciding to create a web application as any user with a web browser can manage their Docker host.

Bootstrap was therefore decided as the best method of delivering a fully functional mobile web application. Bootstrap gives the developer full flexibility in terms of choosing the arrangement of content on different sized devices. For example, on a large device the developer can decide to have a 2 column 2 row grid of items. If the size of the screen then shrinks to mobile-size it can rescale to a 1 column 4 row grid. To do this simply means adding CSS classes to the HTML code and then Bootstrap will reactively change the layout.

3.3 Enterprise Level Considerations

When developing any application that will potentially be used in a production environment it is important to consider the implications of this in the design phase. There are several headings under which careful consideration must be given to ensure the application performs as it should while also being usable.

Scalability On a system which will potentially be used by many different users at the same time, how it scales and adapts to an increase in usage is important. To ensure that any user has complete flexibility in this regard this application is distributed using a Docker image, and therfore it can

easily be used in a system which natively supports scaling such as Docker swarm or Kubernetes (Docker, 2017*b*; Kubernetes, 2017).

Performance A number of previously mentioned technology choices significantly increase the performance of the application.

- Node JS - Node JS has been shown to have significant performance benefits due to its asynchronous non-blocking event-driven model (Tilkov and Vinoski, 2010). It also has excellent scalability and so is a good fit for this project (Griffin et al., 2011).
- Docker - Since this application will run in a container it is designed to be lightweight. It will only contain the third party modules it absolutely needs and it should minimise the risk of memory issues on the host it is running on.

Security To secure an application the Express endpoints will be secured using JSON web tokens (JWT). The front end application will authenticate itself with the Node JS server and will then use the returned JWT token to access all Express endpoints.

It was also identified that a security method such as single sign on, where users gain access to multiple services using a single username/password combination, might be beneficial to this project. This was captured in the product backlog but it was decided that it was out of scope for this project.

Distribution To make distribution a simple process it has already been described how Docker will be the vehicle of choice. It makes downloading and running the application extremely easy for any developer who wishes to do so. Once the developer has Docker installed on their system all they need to do is run one command to download and begin using the application.

4 Methodology

This section aims to give the reader an insight into the process used to build the application. After reading this section the reader should have a clear understanding of all methodologies used and how they benefited the project.

4.1 Agile

The Agile movement helps teams develop in unpredictable circumstances by using incremental and iterative units of work. It provides a mechanism by which feedback is not only facilitated but actively encouraged, promoting greater transparency along the development cycle (Agile Methodology, 2016).

Agile was chosen as the development methodology for this project for the following reasons:

Quality Testing Testing is integrated into the development cycle, enabling the developer to continuously monitor the functionality and performance of the application. In Waterfall, testing is not carried out until the end when development is finished. This can lead to problems for a single developer as testing coverage and quality may not be as high.

Visibility Agile provides a great environment to see how expectations are managed effectively. It provides a clear view into the project scope and the current track it is on. With Waterfall all expectations and deliverables need to be forecast before any development begins. This can be difficult to do and can mean increased overhead of work.

Risk Management Incremental development cycles allow the developer to accurately assess any challenges in the early stages of developement and make it easier to respond and adapt. In Waterfall there is very little room for adapting to unforeseen challenges.

Flexibility Agile allows for change natively. Instead of setting a rigid time plan up front the timescale is set and each sprint allows for the requirements to change and for more to emerge as development continues.

Agile uses time-boxed units throughout the development cycle (Agile Methodology, 2016). These are short development cycles designed to give the developer achievable goals while also allowing for change at short intervals.

The Agile implementation used in this project will be discussed further in Section 5.

4.1.1 Scrum

Within Agile this project will also follow Scrum methodologies. This will include:

1. Backlog Grooming - Each sprint has a sprint planning session to initially scope the product backlog and to inform the total scope of the project. This typically involves the product owners who will help to shape the overall project direction by providing their priorities for the project.

2. Sprint Planning - Refining the backlog to break large development tasks into smaller tasks which will fit better in a sprint. This also helps to identify similar tasks which can be grouped together and reduce duplication.

3. Sprint - All development work is carried out here. There are daily stand ups which involve the developer taking into account the work achieved the previous day, what was going to be achieved today and also any difficulties encountered. While this project was a lone developer project it was still constructive as it forced consideration of the Jira board and also helped guide the day to day work.

4. Sprint Review - After a sprint is completed a sprint review is then performed which will evaluate goals achieved versus goals planned and the backlog is then re-prioritised accordingly.

5. Sprint Retrospective - A retrospective will then analyse sprint performance and help to highlight areas where improvement can be made. This will also enhance the accuracy of the next sprint.

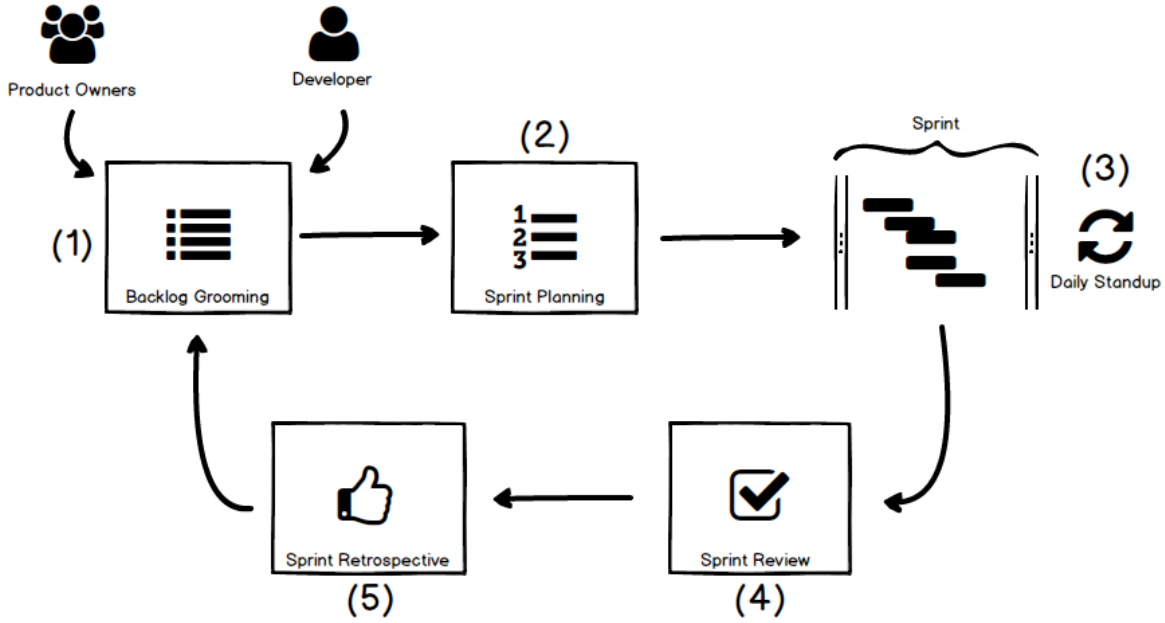

Figure 5: *Agile/Scrum Process*

To evaluate all tasks on the backlog a popular method of 'story points' was used. This involves the developer rating tasks based on estimated effort. It uses the Fibonacci numbers as a guide and all tasks should be relative to each other. It is not an exact science but does provide a way of judging the amount of work remaining. For example when it is well established that a backlog item with a story point rating of 2 may take a day then it is reasonable to assume an item with a rating of 3 may take a day and a half to complete.

The following roles were also identified within this project:

- Product Owner - Leigh Griffin (Red Hat)
- Scrum Master - Dr. Brenda Mullally
- Scrum Team/Developer - Stephen Coady

4.2 Jira

The project management tool Jira was used to aid with Agile implementation and to track all project activity (Atlassian, 2016). Jira is a professional grade project management software used by companies to execute a distributed version of agile. It provides functionality to graphically manage the product backlog and each sprint. The developer is presented with a dashboard which allows them to create and edit tasks by assigning descriptions, subtasks and estimation of work.

Granular control over backlog items is important in an Agile environment as change is expected and so being able to have fine-grain control over everything makes this process much easier.

4.3 Gitflow

As this project will be developed in an open source environment there will also be well defined procedures followed when using Git as a version control tool. This ensures that should any other developers wish to contribute to the project in the future then a set of guidelines will ensure all code is merged in a clean way which will in turn ensure complete transparency. This set of guidelines is referred to as a 'Gitflow'. The Gitflow used for this project can be seen below in Figure 6.

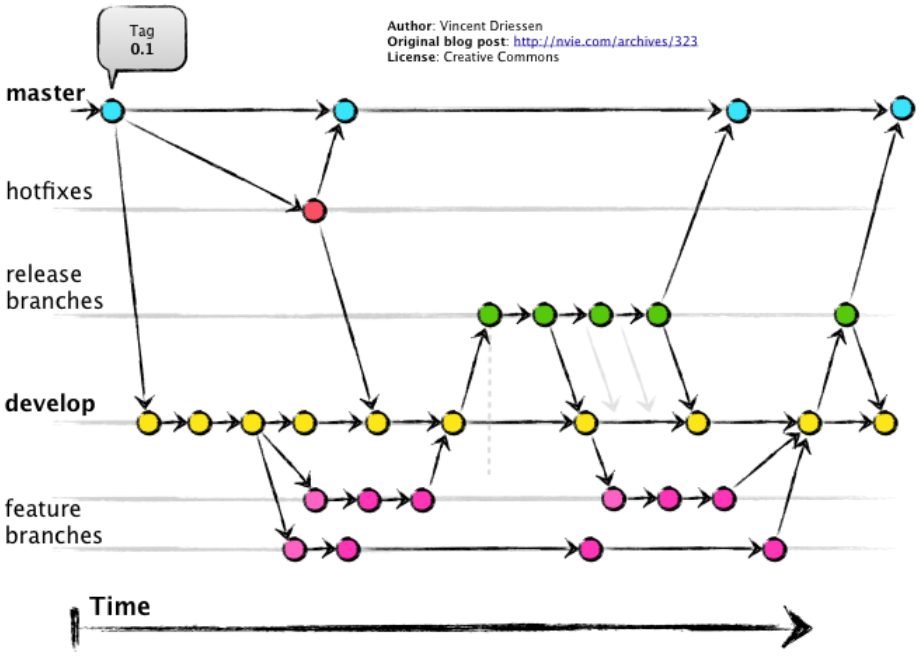

Figure 6: *Gitflow (Driessen, 2017)*

Using a well-defined Gitflow provides many benefits. Namely:

Tagged Versions

Tagging versions at release intervals means it is easy to see a desired version if the developer wishes to do so. For example if someone wishes to download a version 0.6.0 of software then they can navigate to the versions pane in Github and easily download that specific version. An example of this for this project can be seen in Figure 7. Here it shows how a developer can easily download a specific version.

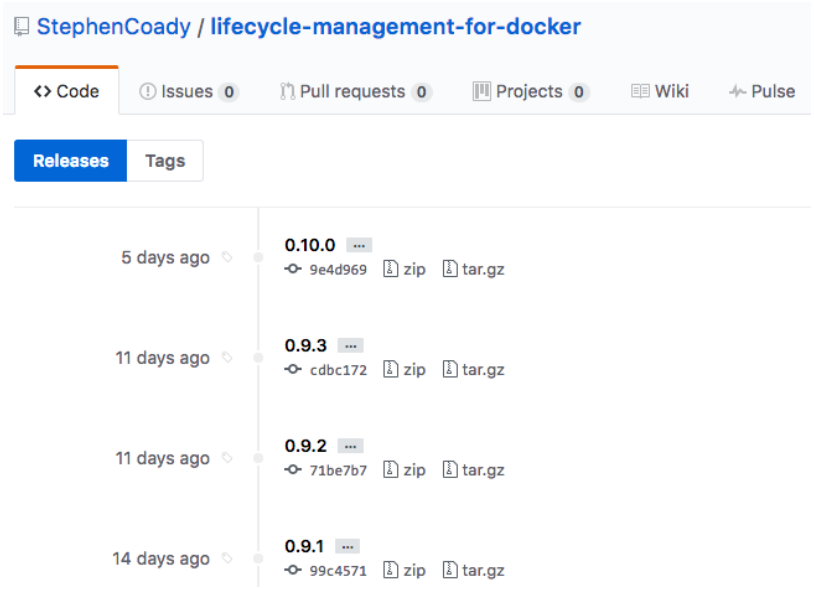

Figure 7: *Git Tagged Versions*

Safe Merges

When using multiple branches it can become difficult to safely merge code from two branches. Having a well-defined process to do this minimises this risk while also creating a clearer picture in the developer's mind about what code is being merged and where it is being merged.

Easier Troubleshooting

In the case where something has gone wrong if there is a reproducible set of steps which were followed then troubleshooting the Git environment becomes easier. It also makes it easier for novice developers who may be inexperienced with Git to contribute to the project if every step is documented.

4.4 Testing

Testing is a vital aspect of software development. It is for that reason that great consideration was given to the test plan for this application. Software testing in this project has been broken up into 4 distinct headings, namely:

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

Unit, integration and system testing in this project is all automated while acceptance testing is a manual process. Each of these will now be discussed whilst describing how they are implemented in this project in detail.

Unit Testing

Unit tests are the most basic form of automated testing and occur when one unit of a program is tested. This unit may be a basic variable or it can also be a function belonging to a module of

software. Unit tests on a function for example may call that function and compare the returned value with the expected output. It is a simple standalone test which does not rely on any other test within the test suite (STF.com, 2017*d*). An example of a unit test for this project can be seen in Figure 8.

```
1  it('should list specific container', (done) => {
2    request(app)
3      .get('/api/containers/' + testContainer)
4      .set('x-access-token', token)
5      .expect('Content-Type', /json/)
6      .end(function(err, res) {
7        expect(res.status).to.be.equal(200);
8        assert.equal(testContainer, res.body.container.Id);
9        done();
10     });
11 });
```

Figure 8: *A Simple Unit Test*

We can see on line 8 of Figure 8 that this unit test simply makes an API call and compares the result to the expected result. While this is powerful it is also too simple for modern day applications. Instead, we must test how several different functions acting in sequence behave, as this gives a more realistic simulation of an application being used by a real user.

Integration Testing

We therefore naturally arrive at integration tests as a means to increase testing potency. Integration tests are essentially a sequence of unit tests. This sequence is carried out with the express intent of exposing defects in software as a result of the interaction between multiple components (STF.com, 2017*b*).

We can see a trivial example of integration testing in this project in Figure 9. In this test case, we first stop the container and then restart it before checking that the container is actually started. This is to ensure that the act of stopping the container does not interfere with restarting.

System Testing

System testing is the act of testing a system in its entirety and in the same environment on which it will run (STF.com, 2017*c*). This will be discussed further in Section 4.6 but for the purposes of this section system testing refers to the process of all tests being in sequence and those tests should cover the complete application. This is referred to as code coverage and will again be discussed further in Section 4.5.

Acceptance Testing

Acceptance testing is the process of manually using an application to ensure it meets requirements and performance expectations (STF.com, 2017*a*). Since this project has a product owner in the shape of Red Hat Mobile they acted as the user. At the end of each sprint this involved the product owner using a staged version of the application to ensure it met all of their initial requirements for the newly developed feature(s).

```
it('container should be stopped', (done) => {
    request(app)
      .get('/api/containers/' + testContainer)
      .set('x-access-token', token)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        expect(res.status).to.be.equal(200);
        assert.equal("exited", res.body.container.State.Status);
        done();
      });
  });

  it('container should restart', (done) => {
    request(app)
      .post('/api/containers/' + testContainer + '/restart')
      .set('x-access-token', token)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        expect(res.status).to.be.equal(200);
        expect(res.body.message).to.equal("Container restarted successfully");
        done();
      });
  });

  it('container should be started', (done) => {
    request(app)
      .get('/api/containers/' + testContainer)
      .set('x-access-token', token)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        expect(res.status).to.be.equal(200);
        assert.equal("running", res.body.container.State.Status);
        done();
        });
  });
```

Figure 9: *A Simple Integration Test*

4.5 Code Quality/Coverage

It is vital in any project to keep code quality at a high standard. If the project wishes to attract other developers to contribute then a good codebase with current best practices implemented and few bugs will help. To do this the tool SonarQube will be used, as discussed in Section 2.5. SonarQube will scan the codebase an inform the developer of any code smells present and will also perform calculations such as technical debt and code coverage.

SonarQube is embedded in the continuous integration pipeline for this project. This will ensure that code is consistently monitored which in turn ensures code quality and test coverage is always at the forefront of the developers mind.

4.6 Continuous Integration/Deployment

In software development, continuous integration is the act of continuously ensuring software is of a suitable standard to be integrated into the current software package. This will ensure that any changes made to the code base will receive immediate test-feedback (Fowler, 2006). Even though this application was not built in a production environment, having a continuous build cycle ensured maximum quality code and also reduced the risk of a "bug bottleneck". Continuous deployment is the act of making an application available for use as soon as it has passed all tests contained in the integration section of the build pipeline.

For this project, a complete pipeline was built to facilitate a full continuous integration & deployment environment build and release process. This pipeline consists of the following:

- Git repository to house the code (stored remotely on Github)
- Travis CI build server to execute tests and carry out post-test tasks
- SonarQube Server to analyse and advise of improvement to code, based on best-practices
- DockerHub repository which the built image is pushed to
- A staging server which the built image is deployed to and then run on

This pipeline can be seen in Figure 10 and will work as follows:

1. Git commit is made
2. Travis CI build is triggered which runs all tests and code coverage reports
3. If successful a SonarQube push is triggered and the code is sent for evaluation
4. Once this is complete Travis then builds a Docker image from the newly passed code. This image is then pushed to the associated DockerHub repository to be made public
5. Once the push is complete Travis then deploys the application (by running a container) on the staging server.
6. Travis then pushes a notification to the developer to inform of success/failure status

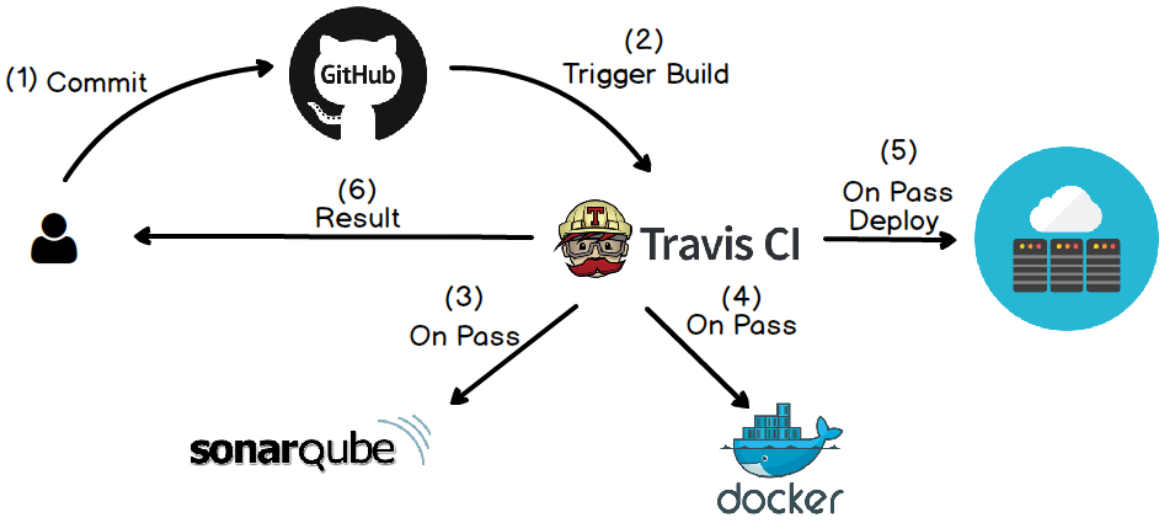

Figure 10: *Continuous Integration and Deployment Pipeline*

There are several advantages to this method of integration and deployment which are additional to those mentioned previously. These are as follows:

- The process is completely automated, meaning one git push is all that is required to fully build the application and make it ready for deployment.
- Using SonarQube as part of the integration pipeline means that code quality does not drop at any iteration of development
- Automatically releasing to DockerHub will also mean that the application is rapidly updated between software releases.

- Staging the application provides immediate feedback for anybody wishing to evaluate the current iteration of the project and whether or not it has met the requirements for the sprint.

4.7 Twelve Factor Application

A recent methodology has emerged regarding web applications and the development best practices associated with creating and deploying them. This methodology is known as 'The Twelve-Factor App' and dictates the guidelines to follow for an application to be easily setup, portable and suitable for deployment on various architectures (Wiggins, 2017). An examination of each of these twelve factors will now be completed and will include how they were applied during the development of Gantry.

1. Codebase
'One codebase tracked in revision control, many deploys'

Gantry is tracked in a single Git repository. It is not a deployable application in the traditional sense as each developer must deploy their own version of this application.

2. Dependencies
'Explicitly declare and isolate dependencies'

All external dependencies in this project such as NPM modules, CSS files and external libraries are bundled within the application. Gantry can be run without an internet connection and only external API calls will fail.

3. Config
'Store config in the environment'

While there is very little config in this application it is isolated to a config file and is not hard-coded within the application.

4. Backing Services
'Treat backing services as attached resources'

The only backing service within this application is the Nedb database and it is treated as a modular service which satisfies this factor.

5. Build, Release, Run
'Strictly separate build and run stages'

As this application uses Docker as both a build and deployment vehicle there is a clear separation by default here. The image is built, the container is deployed and they are both separate entities.

6. Processes
'Execute the app as one or more stateless processes'

Again Docker solves this problem as it completely isolates the environment and therefore the process.

7. Port Binding

'Export services via port binding'

This whole application is exposed via a single port listening on the host.

8. Concurrency

'Scale out via the process model'

This factor is not applicable to this application as the process to run this application is already a single unit and cannot be broken down any further. This factor deals with applications which may have multiple instances of themselves running across several hosts.

9. Disposability

'Maximize robustness with fast startup and graceful shutdown'

Due to the ephemeral nature of Docker containers this constraint is also satisfied. Containers can be stopped, removed and quickly started again without affecting the next instance's startup or shutdown procedures.

10. Dev/Prod Parity

'Keep development, staging, and production as similar as possible'

This factor is implemented by using Vagrant as discussed in Section 2.6 for the development environment to replicate the Docker engine environment. The continuous integration server Travis is then used with the same Docker engine. As any user running the application will also be using the Docker engine it ensures the application will always have the same environment available to it.

11. Logs

'Treat logs as event streams'

This application exports all application logs directly to the web application. This ensures they are always available to the user if required.

12. Admin Processes

'Run admin/management tasks as one-off processes'

This application does not provide the facility to run any administrative tasks.

We can see that by choosing to use Docker as a build and deployment vehicle many of these twelve factors have been automatically satisfied with minimal input from the developer.

4.8 Documentation

This section will discuss the tools used to document the entire process of building this application and also the document which accompanies it.

4.8.1 User Documentation

Since the project comes in two parts, the standalone API and the user web application there are technically two separate user guides. The first user guide is relatively short and can be seen in Appendix A. It is the form of the README file associated with the repository. In the open source community it is common practice to include everything in the README which the user requires to begin using the application.

The second set of user documentation uses a standalone application called Swagger UI to document the API (Swagger.io, 2017). Swagger allows any user to interact with the API through the documentation. For example, anyone with access to the documentation can make calls to the real API by filling in data and pushing buttons. It shows the user all current API endpoints and what each accepts as parameters and returns as a result. A screenshot of this can be seen below in Figure 11.

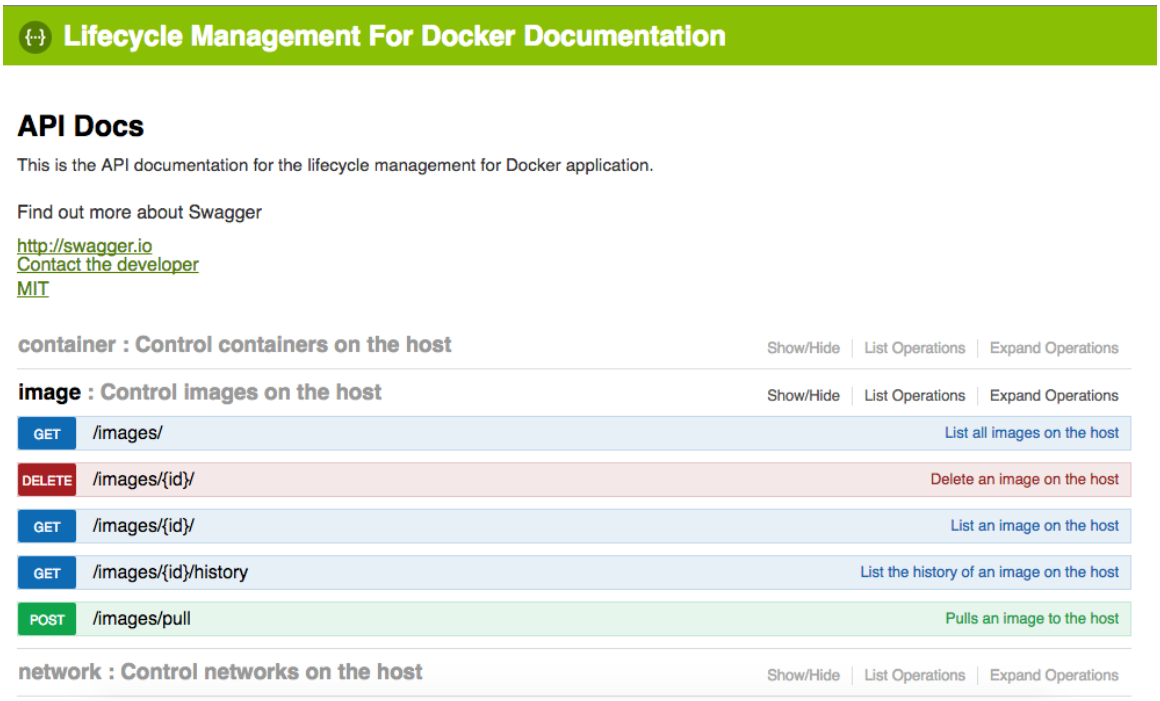

Figure 11: *Swagger Overview*

An example of how one individual API endpoint is shown in Swagger in Figure 12.

Figure 12: *A Single Swagger Endpoint*

Making it accessible

To make the documentation easily accessible to anybody who wishes to view it it has been embedded as part of the application. When the user runs the project the node application automatically serves the documentation up on the same host as the application. This can be seen by following the documentation link in Appendix E.

4.8.2 LaTeX

Latex is a software package which is used to write academic, high-quality papers. It is written using plain text with some markup which is then used to format. The plain text documents are compiled and Latex produces a pdf file (The Latex Project, 2017).

In this project Latex was used exclusively to write this document. This offered a number of clear benefits over traditional office-based software packages.

- Plain text - since Latex only uses plain text to create its documents the writer is free to concentrate on the important aspects of writing documentation.
- Source controlled - again as Latex is plain text it can be easily stored in a version control system such as Git.
- Powerful compilation engine - The compilation engine behind Latex offers huge benefits such as auto-referencing all sections and figures etc. It can also auto-create sections such as glossaries and tables of contents.

The source code used to create this report can be seen in Appendix F and will be made available as a template for future WIT students to use.

5 Implementation

This section will detail all technical work carried out in this project. It will use the project sprints to build a timeline for the reader.

The scrum development cycle can be seen in the previous Section 4.1.

5.1 Sprint 1

Sprint Planning

The planning session for this sprint identified that it would be best to have a functional API developed early in the project and so the tasks associated with this were prioritised and refined. It was also deemed important to have SonarQube integrated in the continuous integration pipeline early in the project. This ensured technical debt was minimised from the beginning. For the same reason it was decided that the continuous integration pipeline should be finalised in this sprint to maximise its effectiveness throughout the project. Without working tests in place the value of this pipeline is minimal therefore the testing tickets in the backlog were refined and placed in the sprint.

Sprint Review

Goals achieved:

- Large subset of necessary API endpoints created.
- SonarQube integration completed.
- Continuous integration pipeline finalised.
- Full test suite for all current API endpoints implemented.

This sprint delivered on all initial goals and with the experience gained the product backlog was re-prioritised accordingly.

Sprint Retrospective

Date: 01 Feb 2017
Participants: Leigh Griffin, Stephen Coady

What did we do well?

- Already had prototype in place to accelerate development.
- 3rd party module knowledge accelerated development.
- Guidance from Red Hat was crucial in keeping the sprint focused.
- Scope on tickets was well understood from Red Hats perspective.

What could we have done better?

- Story points were inflated because domain knowledge was higher than anticipated.
- Testing strategy needs to be revised, very time consuming.

Actions:

- Stephen Coady review backlog for story point accuracy based off of current domain knowledge.

- Stephen Coady add a ticket to review / spike testing strategies, feel free to consult David Martin and Leigh Griffin on specifics.
- Stephen Coady add a ticket for UI frameworks investigations and spikes, end result should be an Epic that we can triage and prioritise.

Sprint Burndown

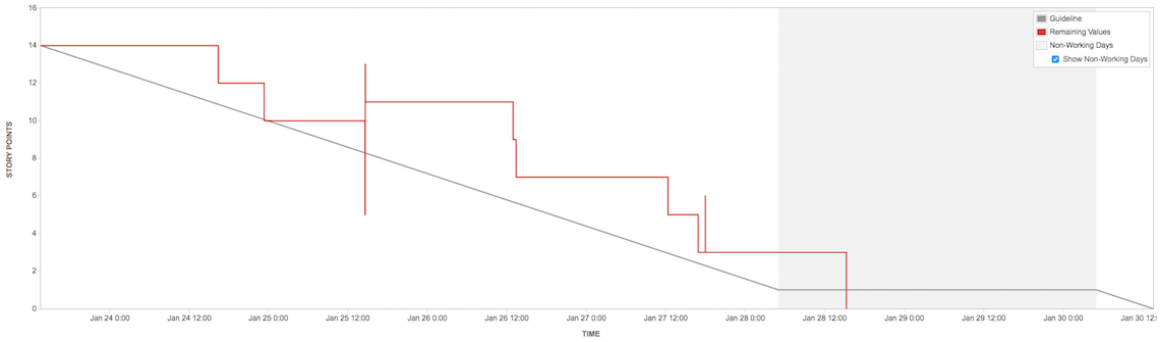

Figure 13: *Sprint 1 Burndown*

Personal Reflection

This sprint highlighted the need for code quality and a continuous integration pipeline. It allowed the developer to gain a deep understanding of testing a modern application and also highlighted some improvements which could be made in following sprints with regard to the test plan.

5.2 Sprint 2

Sprint Planning

While reviewing the backlog after Sprint 1 several API endpoints were identified as missing, therefore they were added to the backlog. It was decided that these would be developed in Sprint 2. Since the API should be a standalone component it is vital to have documentation available for any developer wishing to use it. For this reason a documentation tool investigation ticket was added to the sprint.

Sprint Review

Goals achieved:

- Express API made considerably larger by adding multiple end points.
- Documentation investigation resulted in Swagger being decided on as the best method of writing documentation (Swagger.io, 2017).
- Swagger implemented for existing API endpoints.

Issues encountered:

- Testing method was found to be laborious and was negatively impacting development speed.

Sprint Retrospective

Date: 15 Feb 2017

Participants: Leigh Griffin, Stephen Coady, David Martin

What did we do well?

- A backlog review was performed and this set the priority for the remaining 3 sprints.
- Progression of the sprint was excellent, good pace and good story pointing.
- Communication was good.
- Smooth sprint, story points and tickets were well scoped.
- Sprint Planning revisited the story points so there were few unknowns during the sprint.
- Team (Stephen) came to the Stakeholders (David & Leigh) with the plan for the next Sprint.

What could we have done better?

- Sprint started at a bad time in the college calendar, with other assignments due.
- Not keeping in touch with the sprint day to day (Dave & Leigh).

Actions:

- Stephen Coady to share wireframes as a mid sprint review asynchronously. Recommended emailing this to stakeholders David & Leigh.
- Stephen Coady to complete metrics spike insight (this sprint possibly).

Sprint Burndown

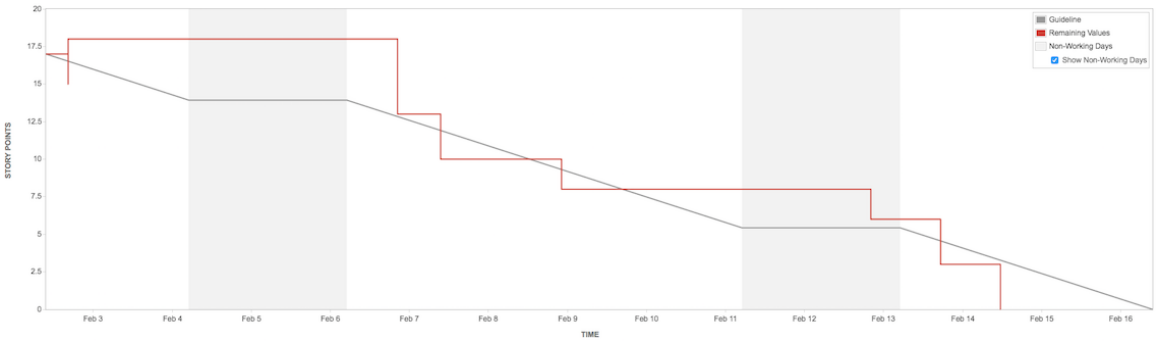

Figure 14: *Sprint 2 Burndown*

Personal Reflection

This sprint helped the developer to improve story pointing skills and it also improved the level of detail included in each development ticket. This meant less time at the beginning of each sprint deciding the desired outcomes as the ticket was clear and concise before work commenced.

5.3 Sprint 3

Since the application was now well-formed from a server-side perspective it was decided that the front-end application should be started. This would mean a more visual component for the project stakeholders to view which would provide more useful feedback moving forward. To start this process tickets were created relating to the system wireframes. Also, since it was intended that the application would have a front-end component by the end of this sprint it would be practical to build this into the continuous integration pipeline ensuring it was automatically deployed upon each build. As a result of the Sprint 2 retrospective it was decided that tests should be implemented on the staging server within a Docker container.

Sprint Review

Goals achieved:

- Continuous deployment implemented using bash scripts and Travis CI. Staging server can be seen in Appendix E.
- Wireframes created and passed to stakeholders for review. Following feedback they were revised and improved.
- Front end application was then built using these wireframes.

Issues encountered:

- Containerised testing was not implemented on the continuous integration server.

Sprint Retrospective

Date: 28 Feb 2017

Participants: Leigh Griffin, Stephen Coady

What did we do well?

- The UI is very usable, positive feedback and functionality visible now.
- Consistency in the velocity at 20 points.
- Got the wireframe relationship, it really helped with the developer's front end skills.
- Wireframe feedback was excellent, helped scope the work.
- Got to demo to Dr. Brenda Mullally which gave her insight to the product
- Overall work pace was judged well for the most part, consistent delivery.

What could we have done better?

- The story pointing on the skeleton was completely off, it could have derailed the entire sprint.
- Velocity in the last sprint was off.
- The developer should have de-scoped when it was realised how big the UI was.
- The developer should have re story pointed the UI mid sprint to allow a controlled de-scope.
- Tickets are not descriptive enough, need to add more metadata.
- Didn't de-scope the testing in a container ticket, this should have happened when it was realised how large the UI ticket was.

Actions:

- Stephen Coady to review the backlog with a view to WHAT and WHY being evolved in the tickets as well as story points.
- Stephen Coady to define the critical path through the project, 80 story points left with a 60 story point burn predicted.
- Stephen Coady to add some investigative tasks around KeyCloak SSO for future work i.e. out of scope of this project.

Sprint Burndown

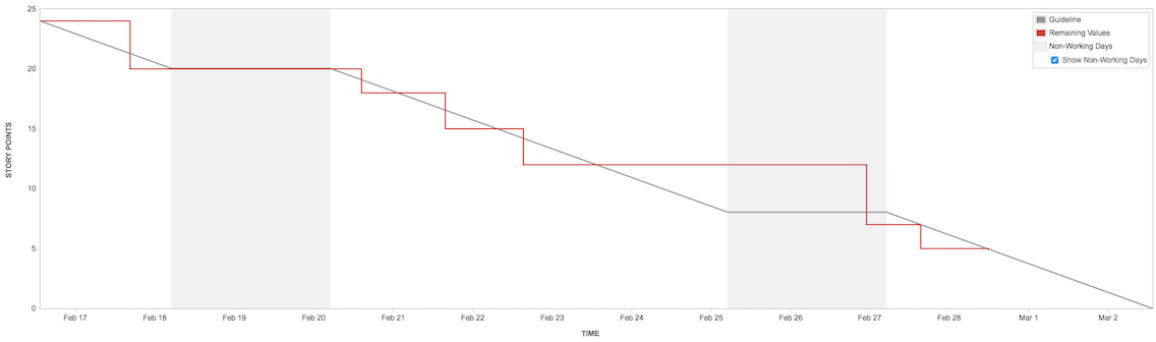

Figure 15: *Sprint 3 Burndown*

Personal Reflection

This sprint highlighted the fact that some tasks, such as front end related tasks, were underestimated and needed to be revised. It also proved valuable as it produced wireframes which improved my ability to create a front end application.

5.4 Sprint 4

Sprint Planning

For this sprint the general direction was aimed at creating a more complete front-end application. Therefore the backlog was groomed and re-prioritised accordingly. Metrics tools and a means of collecting them was also identified as a desirable outcome of this sprint. Further API endpoints were identified and added to the sprint.

Sprint Review

Goals achieved:

- Front-end application:
  - images, containers, networks and volumes views created
  - creation of volumes and networks
  - addition of volumes to containers
  - view a container's log
- Investigation of metrics tools
- Further API endpoints added

Sprint Retrospective

Date: 13 Mar 2017 Participants: Stephen Coady, Leigh Griffin

What did we do well?

- Better scoping of story points as a result of last retrospective.
- Re-scoped mid sprint which the developer hadn't done before - leads to more accurate metrics.
- Tickets were made more descriptive before this sprint. This meant less time spent before starting a task figuring out what was needed.

- The velocity of this sprint was much higher than previous sprints, this is a combination of the developer not having as many college commitments during this sprint and also as a result of better re-scoping mid-sprint.
- Agile methodology maturity is clear to see now. Tickets are more accurate, estimates are accurate and a mastery of the methodology is really clear from Red Hat's side (Leigh).

What could we have done better?

- Getting container logs had not been researched properly as the data was a stream but could also be requested as a JSON object, missed this flag which wasted a lot of time parsing streams to JSON.
- Should have read the Docker remote API documentation in detail sooner. Would have saved time instead of relying on third party module 'Dockerode' which has little to no documentation.

Actions

- Stephen Coady align defined critical path with remaining backlog so that 'nice-to-haves' are separated from 'must-haves'.
- Stephen Coady to decide on project naming conventions.

Sprint Burndown

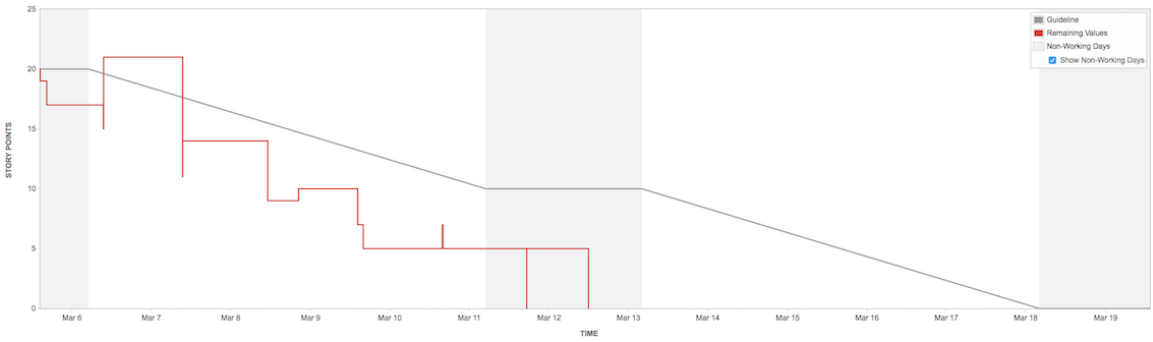

Figure 16: *Sprint 4 Burndown*

Personal Reflection

After this sprint the developer had a deep understanding of the Docker API as creating the multiple endpoints in this sprint required a complete understanding of the Docker documentation. It also proved valuable as I learned how to properly investigate tasks which may be required further into the project (metrics).

5.5 Sprint 5

Sprint Planning

Since this would be the last sprint of the project there were a number of core features remaining unimplemented. These were:

- Allowing the user to search images instead of requiring them know the image name beforehand
- Drag and drop image building using a Dockerfile

- Implementation of a database which should provide user details and therefore security
- Login features to make use of the database

Sprint Review

Goals achieved:

- Search feature implemented
- Database integrated into application along with startup script
- Security in the form of JSON web tokens put in place. This feature secures the front end application and also all API routes
- Dockerfile upload and build implemented behind a drag and drop user interface
- Created a page to allow the user to change their password

Sprint Retrospective

Date: 01 Apr 2017 Participants: Stephen Coady, David Martin, Jason Madigan

What did we do well?

- Majority of story pointing was very accurate. Lead to a very predictable sprint
- Used previous knowledge of JWTs to decrease the time taken to implement them
- Design choice of not using breadcrumbs was a good one, as validated by user acceptance testing
- Managed to close out the last of the college project deliverables

What could we have done better?

- JWT implementation impeded by forgetting to include re-writing of tests in the ticket
- Sprint retrospective not carried out immediately after sprint ended, made it a more laborious task

Sprint Burndown

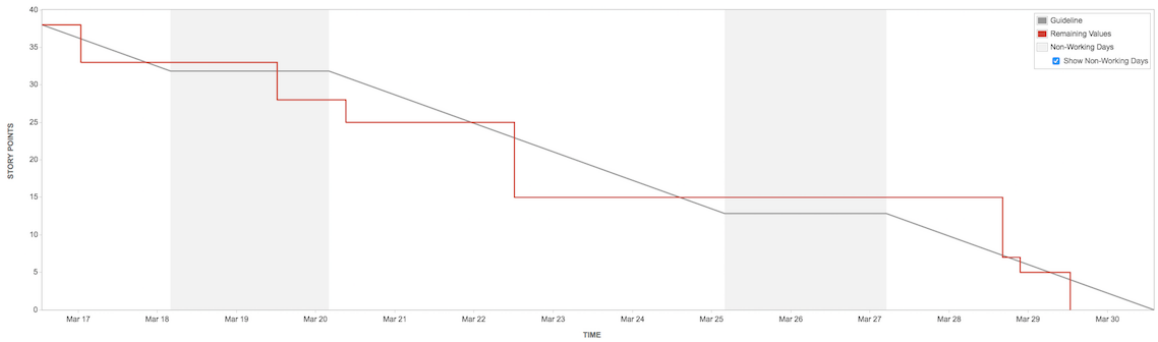

Figure 17: *Sprint 5 Burndown*

Personal Reflection

This sprint was valuable as I learned about security in a node application. It also allowed me to use drag and drop technologies which I had not previously been exposed to. I was also introduced to in-memory databases which I had not previously used.

5.6 Sprint Metrics

5.6.1 Sprint Velocity

Using Jira the velocity for each sprint can be easily viewed, which shows the story points completed in each sprint compared to the initial commitment for that sprint. This gives the developer an indication as to their performance but also maturity from an Agile perspective. The velocity for this project can be seen below in Figure 18.

With regard to the sprints in this project it is evident that initially the developer was completing far more than expected. In sprint 1 the developer completed almost 40% more story points than anticipated. In sprint 2 this was improved and the expected story points were closer to the completed. Problems arose in sprint 3 and not everything was completed. However, this set back was quickly remedied in sprint 4 as a break in college pressures allowed for accelerated development. By sprint 5 the developer had become much more accurate at assigning story points and so this last sprint was by far the most productive and also accurate from a metrics point of view.

5.6.2 Sprint Burndowns

Using burndown graphs for each sprint was a valuable resource as it enabled the developer to maintain a steady working pace while also keeping the deadline in sight. This meant that the volume of tasks remaining was never overwhelming and remaining effort could be accurately judged. All burndown charts for each sprint can be seen above in their respective sprint sections.

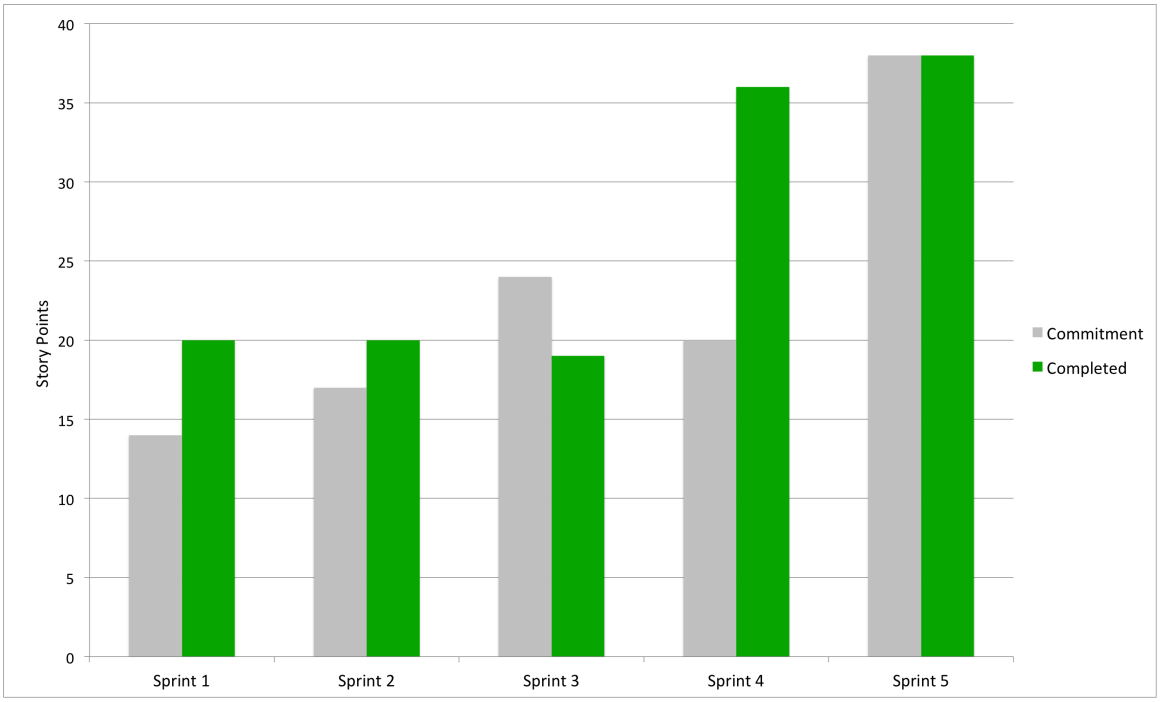

Figure 18: *Sprints Velocity*

6 Summary

In this Section the aim is to dissect the project, examine its proposed deliverables and compare them to the components it actually delivered. This will determine the overall success of the project. The direction of the project going forward will also be discussed in terms of the product backlog and the work which is still outstanding.

6.1 Review

At the beginning of the project all stakeholders decided on the core and stretch goals of the system - these would then form the initial requirements. This list was then broken down further and was the basis for the initial product backlog. This high-level list can be seen below in Figure 19.

Type	Initial Requirement	Status
Core	Fully functional UI	Complete
	Control Images	Complete
	Control Containers	Complete
	Server Side Application	Complete
	Containerise the whole application	Complete
	Full RESTful API	Complete
Stretch	Implement security	Complete
	Connection to multiple hosts	Incomplete
	Drag and Drop Dockerfiles	Complete
	Business Intelligence	Complete

Figure 19: *Initial Project Requirements*

It is clear that all core requirements were delivered and only one stretch goal was not delivered. However as this is now an open source project the tasks associated with this feature have been created and added to the product backlog which can be seen in Section 6.3.1.

The main three components this project has delivered will now be focused on. As the system is loosely coupled each component can be completely recycled in any other project or indeed be replaced within this project by a comparable component.

Server Side Application
The server side of this application is a node module which communicates with the Docker API by using the third party module, Dockerode, previously mentioned in Section 3.

Independent API
The Express JS API has been designed to be independent of any server side or front end code. While it is run by the node module it is not constrained by it. An API written in any other language could easily be placed in front of the node module also.

Front End Application
The front end application written using Angular is probably the most weakly coupled component delivered. It is a web application which consumes API endpoints and could just as easily be written using another front end framework.

6.2 Learning Outcomes

The learning outcomes of this project can be categorised under technical and non-technical.

6.2.1 Technical

As a result of developing this application many technical skills were gained. These include:

- Gained a deep understanding of container technologies, both on the command line and by using the APIs exposed by services such as Docker.
- Learned how to use open source software effectively. i.e while using a package it may be necessary to git clone the software and run it locally. Once software has been run locally it makes it much easier to interact with the community by asking questions.
- Developing an application which has some elements of all modern applications - i.e a server side component, a database, an API and a front end - has resulted in valuable experience of developing the 'full-stack'.
- Using current best practices by creating a fully-functional continuous integration and deployment pipeline has enforced strict coding practices which is invaluable when working as part of a multi-developer team.
- Using code quality tools such as SonarQube has meant code quality is no longer an afterthought when developing applications. This ultimately means a better and more reliable software developer.

6.2.2 Non-Technical

- Working as part of an agile team, even as a lone developer, has meant increased awareness of the overall goal within a project and therefore a more focused method of development.
- Working with a Scrum master has also enforced the ideology of performance and code reviews, again contributing to a better overall development experience and therefore a higher quality end product.
- Receiving input and advice throughout the project from product owners Red Hat mobile has been instrumental to professional development. It has lead to a more focused and organised process. Therefore the quality of the end product has also increased exponentially as a result.
- Communication skills have been greatly improved as a result of constant contact with both the product owner and Scrum master.

6.3 Project Direction

As this product was developed from the very beginning under the open source software ethos it is currently still being developed. The current direction can be categorised under the following headings.

6.3.1 Continued Development

Currently the product backlog can be found here and currently is as follows:

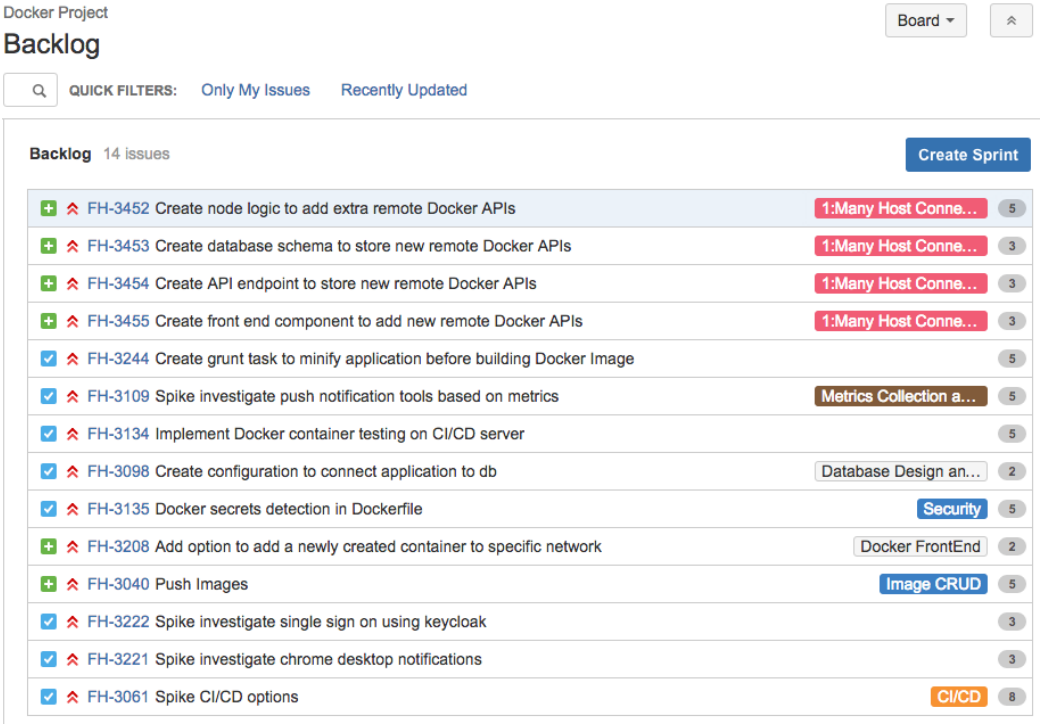

Figure 20: *Current Backlog*

At the current project velocity this backlog contains approximately two sprints. It will implement several features which will improve the overall usability of the application.

By creating a cumulative flow diagram it is clear how much progress has been achieved to date and also how much is remaining. This gives a good indication of time remaining before all items on the backlog are complete if current velocity is maintained. This is evident in Figure 21.

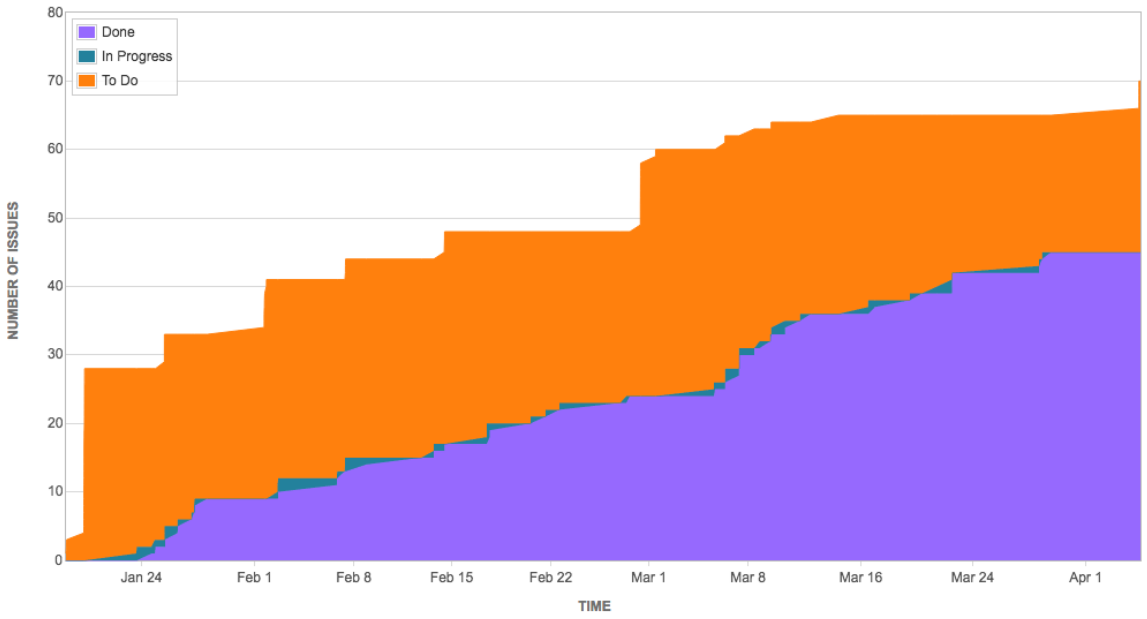

Figure 21: *Cumulative Flow Diagram*

This shows that while much work has been completed development is not finished.

6.3.2 Encourage Developer Contributions

As Docker grows in popularity so too will the interest in peripheral tools and so this project may gain interest due to this. However, to raise its profile further steps like posting on developer community message boards or using social media to attract developers may be viable.

6.3.3 Increase the Feature Set

While the current feature set is large it could be increased further to provide users with more power from the UI. The Docker remote API is extremely large, therefore the amount of possible actions is only limited by the functionality provided by this remote API.

6.4 Reflection

From a personal perspective this project has been a huge milestone. It has shown me how the building blocks I have been learning about for the past four years can be put together to build one large, cohesive project. While it is a technical process it is also ultimately about the journey of creating this application. Starting with nothing and self-managing to the point where the end result is a fully functional application has taught me a lot about project and time management. This is an extremely valuable skill and one which I could not have honed without this project as a platform.

Developing the early prototype model in semester one also taught me about investing an initial amount of time into planning and feasibility investigation. In this project this was invaluable as it highlighted some strengths of the technology choice and also unearthed some which could have been

costly if not caught so soon. To the untrained developer an initial technical feasibility may seem like a waste of time but this project has certainly shown me otherwise.

While developing this project the act of constant self-evaluation paired with always assessing the priority of tasks has shown me how to be the most productive developer possible. This is an incredibly useful skill as it is vital in industry to retrospectively evaluate oneself. Without this skill it is difficult to learn from one's own mistakes, no matter how small or seemingly trivial those mistakes were.

The experience of having technical experts (in the shape of Red Hat) to consult with and seek advice from was also invaluable. I would encourage any future student who is given this opportunity to take it with both hands. I feel I made the most of this relationship by being proactive in my interactions with Red Hat. My experience has shown that companies are not afraid to engage with students - ultimately on the students own terms - and this can, in the end, be beneficial to both the student and the company involved.

I found that implementing to a strict timeline in the form of sprints was a major factor in the success of this project. By initially setting the amount of time (2 weeks) and exactly how many sprints there would be I created a finite timeline which needed to be worked through. The danger of a project like this in a college setting is that the student may devote too much or too little to the project dependent on other subjects. If this happens either the project or the other subjects are bound to suffer and by creating a definite timeline of 10 weeks in which all development will take place this risk was minimised for me. At the beginning of this project Dr. Brenda Mullally and I travelled to the Red Hat offices where all three parties worked through and agreed on the initial high level tasks of this project. In my opinion this, paired with the previously mentioned definite timeline, was the reason I was able to finish the project with 2 weeks of the semester remaining. This enabled me to focus on this document without the distraction of technical tasks - ultimately producing a more detailed and complete document.

Overall I personally would deem this project a complete success. Without a doubt I have grown as a developer during the past 8 months as a result of the work I have completed but ultimately this project was the final learning outcome of the past four years.

Bibliography

Agile Methodology (2016), 'The Agile Movement'. [online] Accessed: 27/11/2016.
URL: *http://agilemethodology.org/*

AngularJs (2017), 'What is angularjs?'. [online] Accessed: 10-03-2017.
URL: *https://docs.angularjs.org/guide/introduction*

Arijs, P. (2016), 'Docker usage statistics: Increased adoption by enterprises and for production use', http://www.coscale.com/blog/docker-usage-statistics-increased-adoption-by-enterprises-and-for-production-use. Accessed: 23-03-2017.

Atlassian (2016), 'Jira Software'. [online] Accessed: 27/11/2016.
URL: *https://www.atlassian.com/software/jira*

Dias, P. (2017), 'Dockerode'. [online] Accessed: 13-04-2017.
URL: *https://github.com/apocas/dockerode*

Docker (2016), 'What is Docker'. [online] Accessed: 27/11/2016.
URL: *https://www.docker.com/what-docker*

Docker (2017*a*), 'Docker overview'. [online Accessed: 08/03/2017].
URL: *https://docs.docker.com/engine/understanding-docker/*

Docker (2017*b*), 'Swarm mode key concepts'. [online] Accessed: 19-03-2017.
URL: *https://docs.docker.com/engine/swarm/key-concepts/*

Driessen, V. (2017), 'A successful git branching model'. [online] Accessed: 27-03-2017.
URL: *http://nvie.com/posts/a-successful-git-branching-model/*

Fowler, M. (2006), 'Continuous Integration'. [online] Accessed: 27/11/2016.
URL: *http://martinfowler.com/articles/continuousIntegration.html*

Griffin, L., Ryan, K., de Leastar, E. and Botvich, D. (2011), 'Scaling instant messaging communication services: A comparison of blocking and non-blocking techniques'. [online] Accessed: 10-03-2017.
URL: *http://repository.wit.ie/1636/*

Kubernetes (2017), 'What is kubernetes?'. [online] Accessed: 19-03-2017.
URL: *https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/*

Nodejs.org (2016), 'Node.js'. [online] Accessed: 27/11/2016.
URL: *https://nodejs.org/en/*

RedHat (2016), 'Red Hat Mobile Application Platform'. [online] Accessed: 27/11/2016.
URL: *https://www.redhat.com/en/technologies/mobile/application-platform*

SonarQube (2016), 'SonarQube'. [online] Accessed: 27/11/2016.
URL: *http://www.sonarqube.org/*

STF.com (2017*a*), 'Acceptance testing fundamentals'. [online] Accessed: 27-03-2017.
URL: *http://softwaretestingfundamentals.com/acceptance-testing/*

STF.com (2017*b*), 'Integration testing fundamentals'. [online] Accessed: 27-03-2017.
URL: *http://softwaretestingfundamentals.com/integration-testing/*

STF.com (2017*c*), 'System testing fundamentals'. [online] Accessed: 27-03-2017.
URL: *http://softwaretestingfundamentals.com/system-testing/*

STF.com (2017*d*), 'Unit testing fundamentals'. [online] Accessed: 27-03-2017.
URL: *http://softwaretestingfundamentals.com/unit-testing/*

Swagger.io (2017), 'Swagger ui'. [online] Accessed: 28-03-2017.
URL: *http://swagger.io/swagger-ui/*

The Latex Project (2017), 'An introduction to latex'. [online] Accessed: 28-03-2017.
URL: *https://www.latex-project.org/about/*

Tilkov, S. and Vinoski, S. (2010), 'Node.js: Using javascript to build high-performance network programs', *IEEE Internet Computing* **14**(6), 80–83.

Vagrant (2017), 'Why vagrant?'. [online] Accessed: 10-03-2017.
URL: *https://www.vagrantup.com/docs/why-vagrant/*

Wiggins, A. (2017), 'The twelve-factor app'. [online] Accessed: 28-03-2017.
URL: *https://12factor.net/*

Appendices

A Application Repository

A.1 Location

The Git repository for this project is available here:
https://github.com/StephenCoady/lifecycle-management-for-docker.

A.2 Contributing

Since this project is open source anybody is welcome to contribute. To do so:

1. Clone the repository
2. Make a branch
3. Make the changes you want
4. Submit a pull request on your code

Once the pull request has then been reviewed it will be merged into the main code base.

A.3 Raising an issue

If you find a bug, please raise an issue on Github and it will then be resolved as soon as possible.

A.4 Running the application

If you wish to run the application please first clone the repository and then run 'node index.js' from within the directory.

Alternatively, you can use Docker. To do this please follow the instructions found in the README.md file in the repository.

B Travis Repository

B.1 Location

The repository can be found at: https://travis-ci.org/StephenCoady/lifecycle-management-for-docker

B.2 Build Statistics

We can see some statistics for the builds of this project below in Figure 22.

Number of Builds	84
Failures	12
Nightly Deploys	42
Successful Pull Requests	18
Releases	12

Figure 22: *Project Build Statistics*

We can also see the builds per day and time taken per day to complete a build below in Figures 23 and 24.

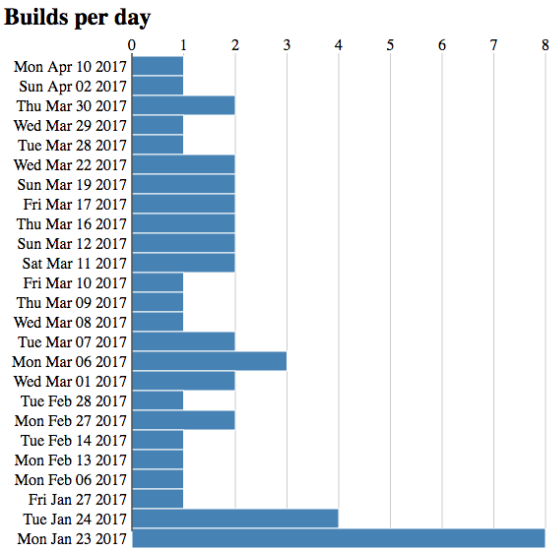

Figure 23: *Project Builds per Day*

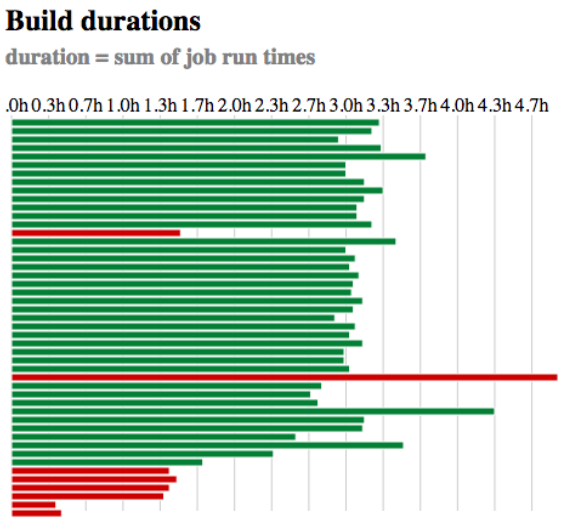

Figure 24: *Project Average Build Time*

C SonarQube Repository

C.1 Location

The SonarQube quality gate for this application can be found at: https://sonarqube.com/dashboard?id=lifecycle-management-for-docker

C.2 Quality Statistics

We can see the overall SonarQube analysis statistics below in Figure 25.

Code Coverage	0.9
Bugs	0
Code Smells	10
Technical Debt	34 mins
Complexity Score	109

Figure 25: *SonarQube Statistics*

C.3 Code Coverage

In Figure 26 below we can see that the code coverage for this project was consistently maintained at 90%.

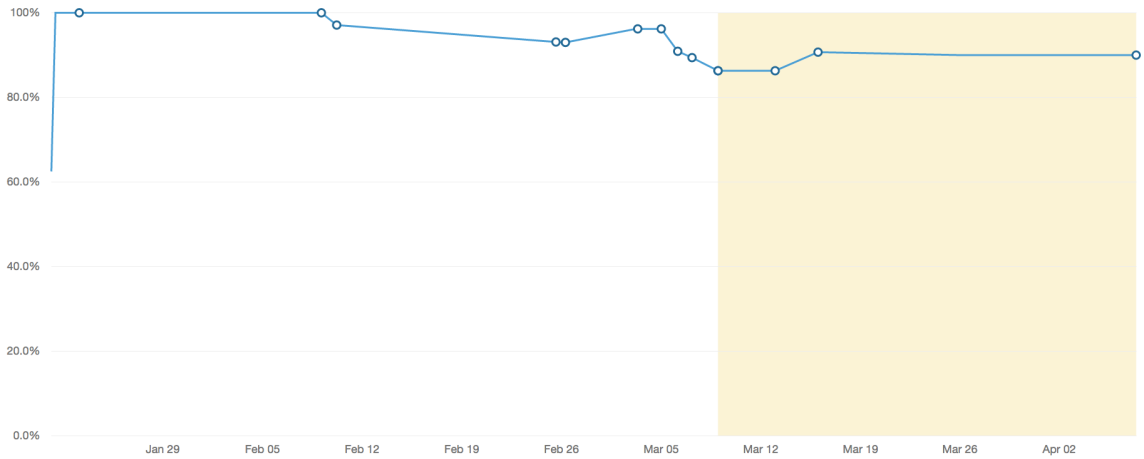

Figure 26: *Code Coverage*

Technical debt was also focused on and as a result was maintained at an average 30 minutes for the duration of the project. This can be seen in Figure 27.

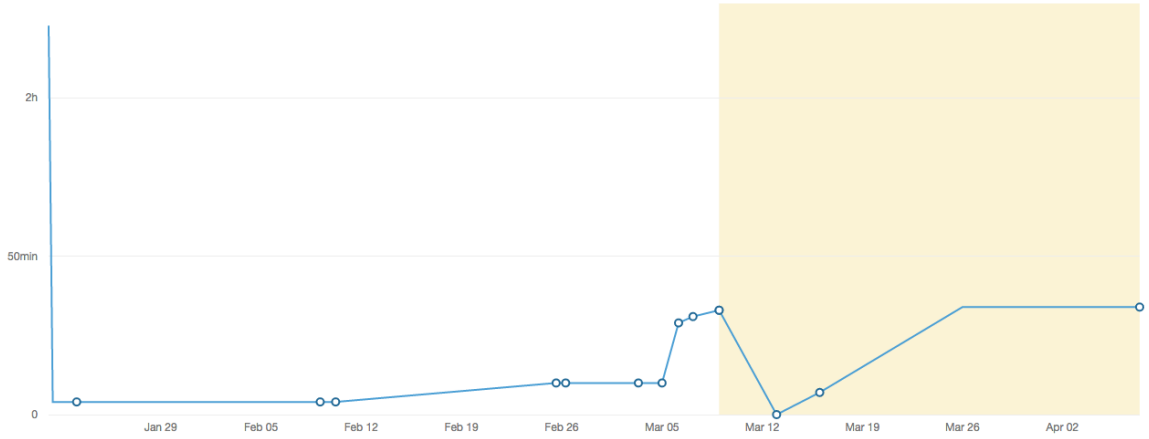

Figure 27: *Technical Debt*

The number of code smells was also maintained as low as possible, with the number averaging at 10. This can be seen below in Figure 28.

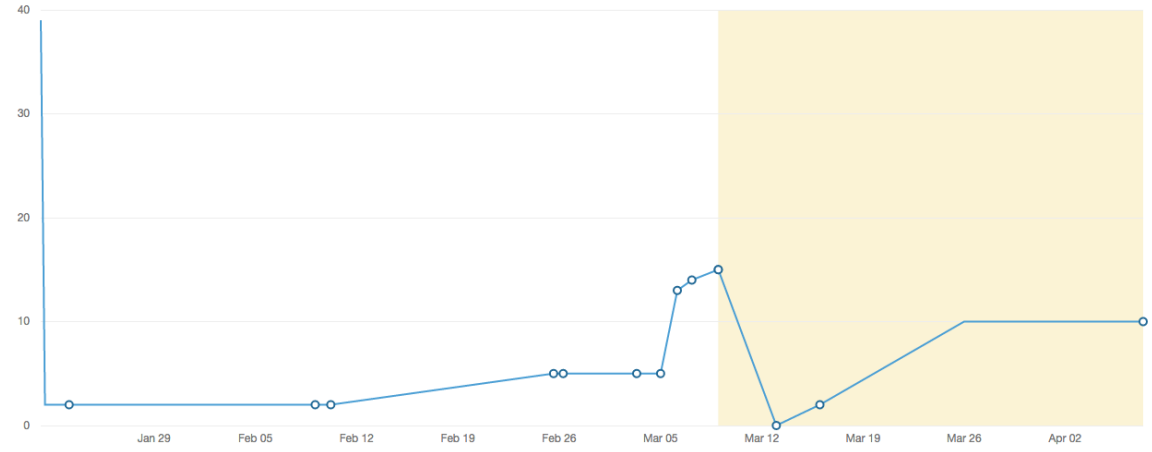

Figure 28: *Code Smell*

D DockerHub Repository

D.1 Location

The repository for the Docker image of this project can be found at: https://hub.docker.com/r/scoady2/lifecycle-management-for-docker/

D.2 Running The Container

To run the container Docker must first be installed on the desired system. Once Docker is installed the command:

```
1 docker run -d -p 3000:3000 -v /var/run/docker.sock:/var/run/docker.sock scoady2/
      lifecycle-management-for-docker
```

will start the container.

E Staging Server

This application is staged on a public instance which can be accessed at:
application: http://87.44.18.55:3000/
documentation: http://87.44.18.55:3000/docs

To use the application one must first log in, this can be done using the username and password provided below.
username: admin
password: admin

F Report Source Code

All LaTeX files for this paper are available here: https://github.com/StephenCoady/college-papers

G Formal System Models

G.1 Class Diagram

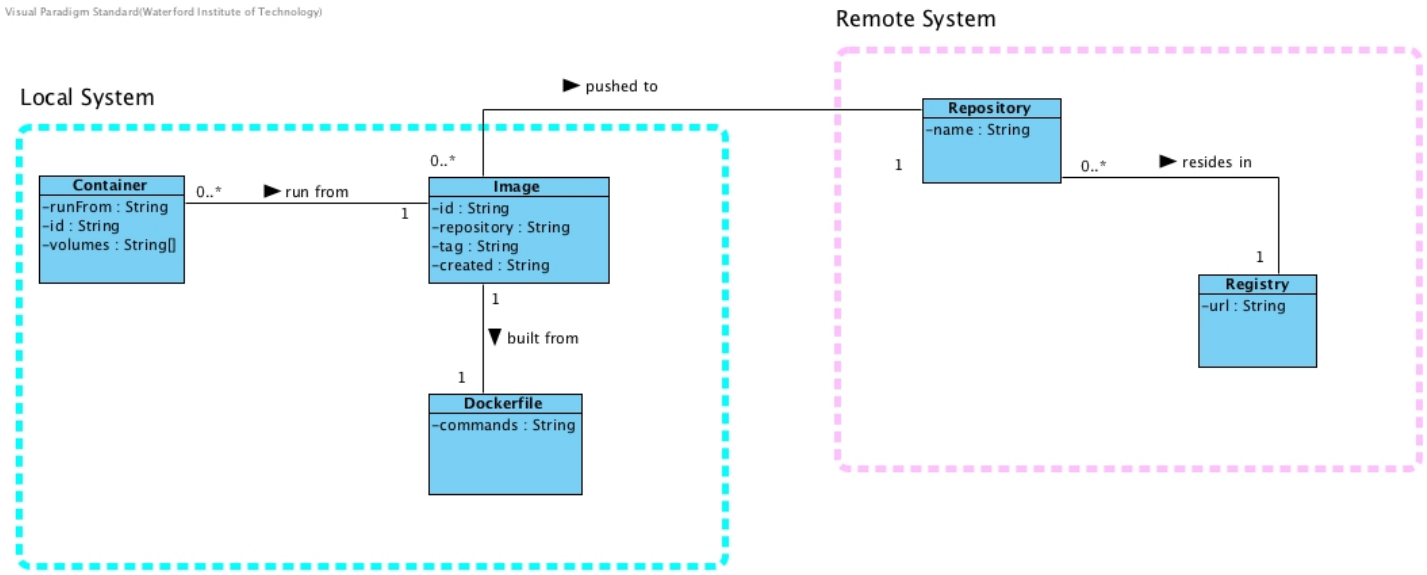

Figure 29: *Class Diagram*

We can see the class diagram for this system in Figure 29. Each class is explained below in list form.

- Container - the container on the system, state is either running or not running. It is run from an existing image on the system.
- Image - the template from which a container is created. May be local or remote.
- Dockerfile - the template from which an image is built. Each line in the Dockerfile leads to a unique Docker image.
- Repository - a remote location where images can be pushed. For an example of a Docker repository please see Appendix D.
- Registry - a remote location which houses one or more repositories.

G.2 Sequence Diagrams

An example of how the sequence diagram shown in Figures 30 and 31 should be read is:

1. The web server calls the list container API endpoint
   1.1. the endpoint passes the container ID to the Docker API
      1.1.1. if the container exists the docker daemon acknowledges this
      1.1.2. the container is then returned to the Docker API
   1.2. the Docker API returns an OK status message and also the container information

1.3. the API returns the container information in JSON form

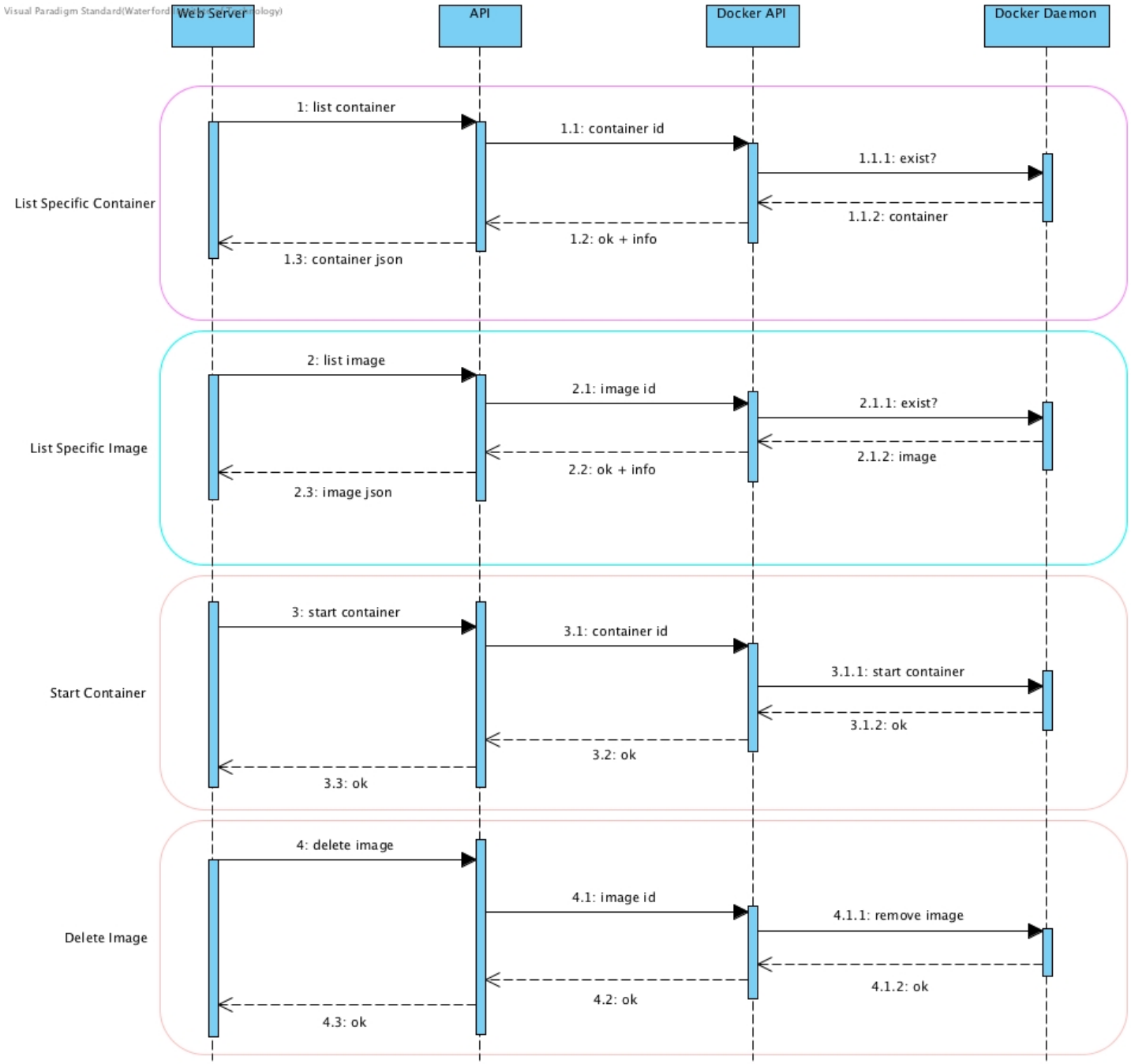

Figure 30: *Sequence Diagram*

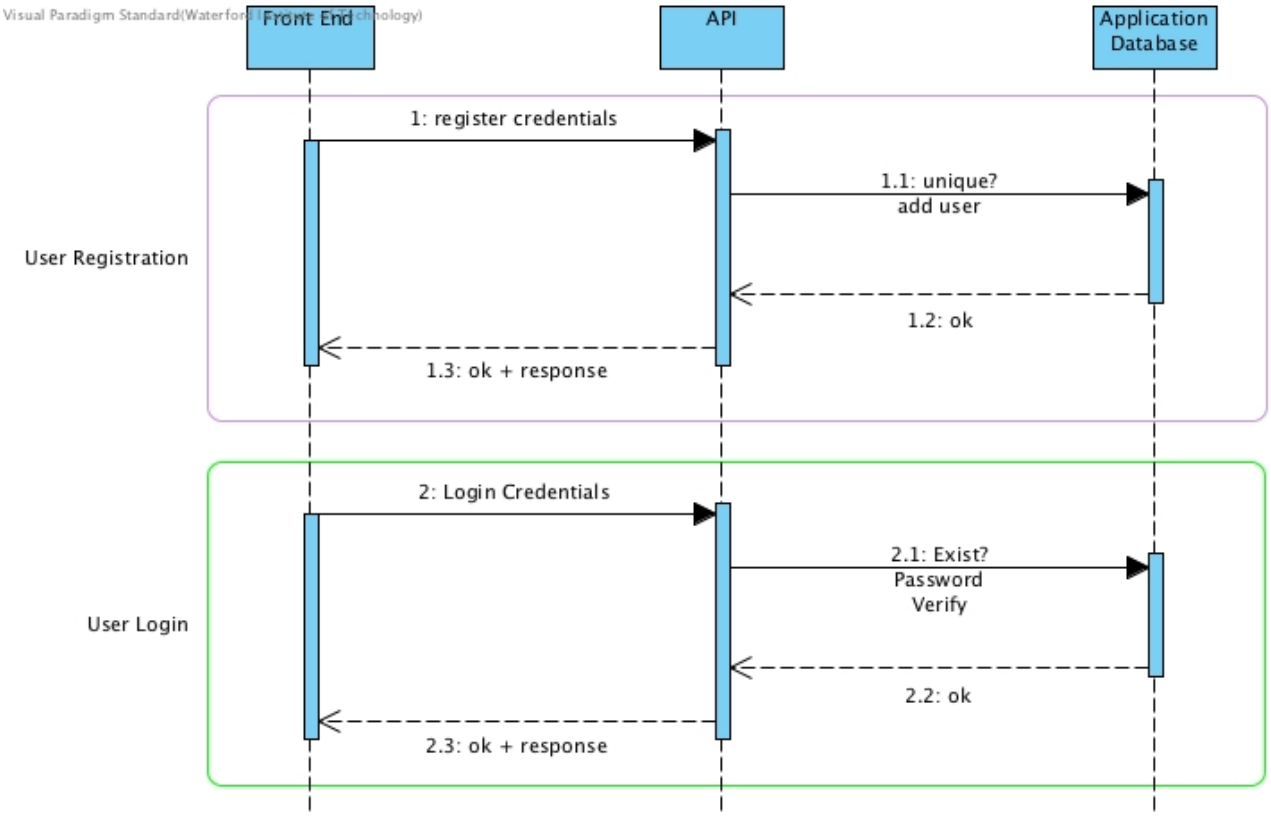

Figure 31: *User System Sequence Diagram*

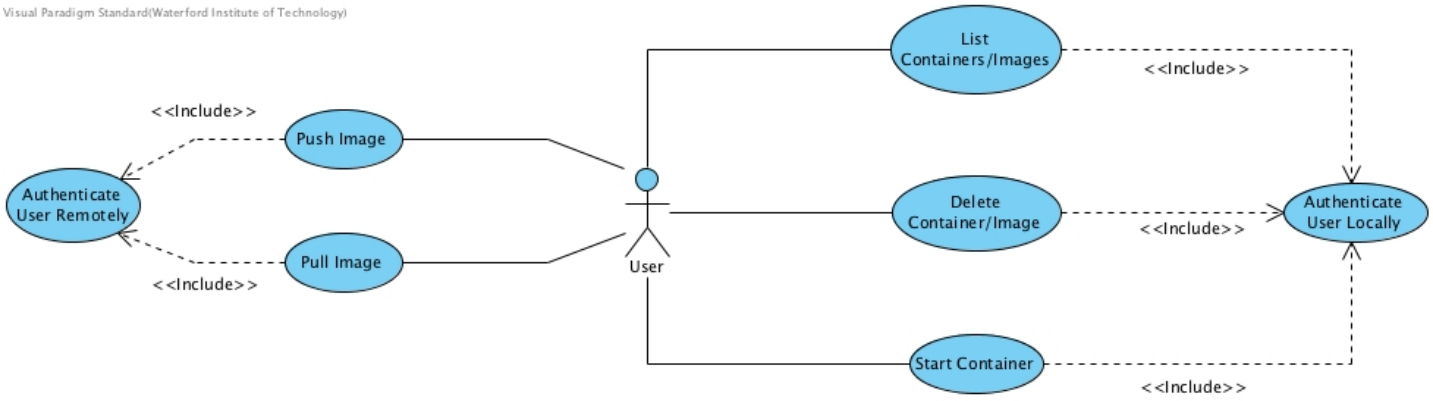

Figure 32: *Use Case Diagram*

Authenticate User Remotely - if the user wishes to push or pull to/from a remote repository then they must login
Push Image - the act of pushing an image from a local machine to a remote one
Pull Image - finding an image not currently on the system and making it available by 'pulling' it from a repository
List - getting a list of all containers/images on a host
Delete - remove a container/image from a host
Start - start a container from an image on a host
Authenticate Locally - if a user wishes to perform any action on the Docker engine they must be authenticated. This is always true unless the Docker remote API has been restricted

H Wireframes

This is the landing page when the application is first visited. It will show a few high level details about the host.

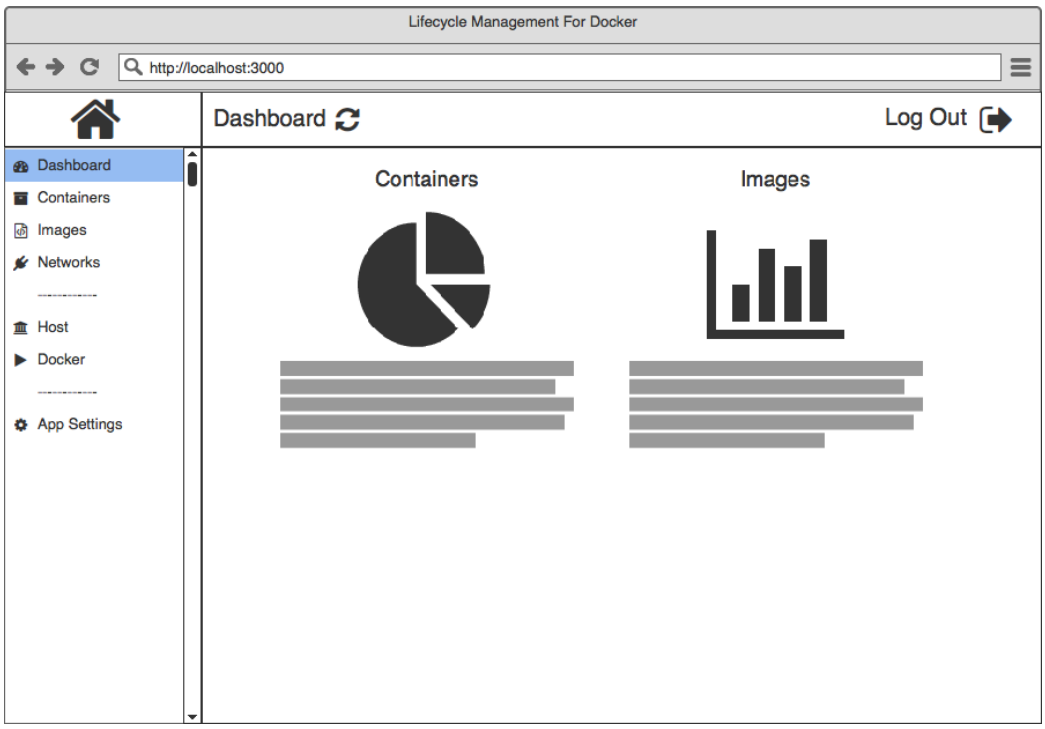

Figure 33: *Dashboard Mockup*

This is the container control view. It shows the main view which can be used to perform most actions on containers.

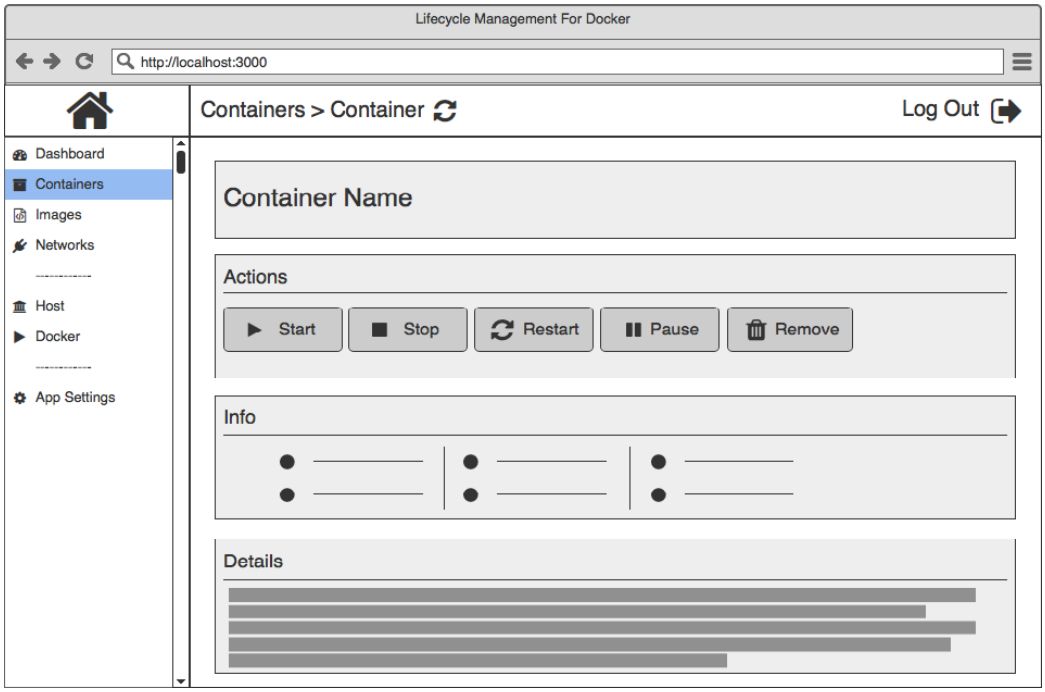

Figure 34: *Container Mockup*

This is the image control view. It shows the main view which can be used to perform most actions on images.

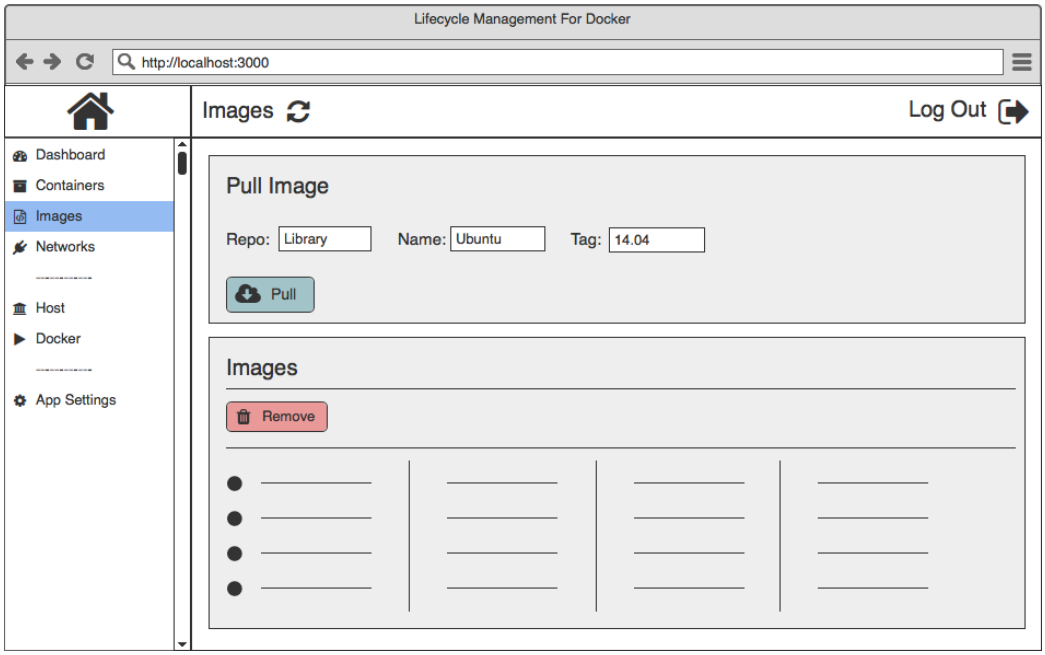

Figure 35: *Images Mockup*

The individual view for an image. It will give a high level overview of that image.

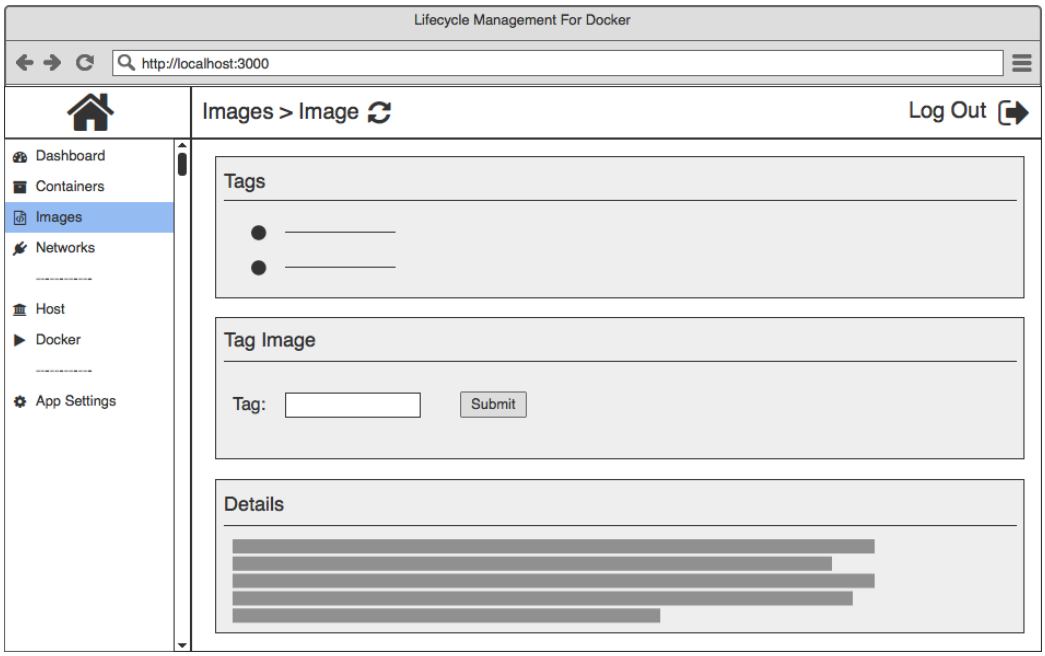

Figure 36: *Image Mockup*

This is the network control view. It shows the main view which can be used to perform most actions on networks.

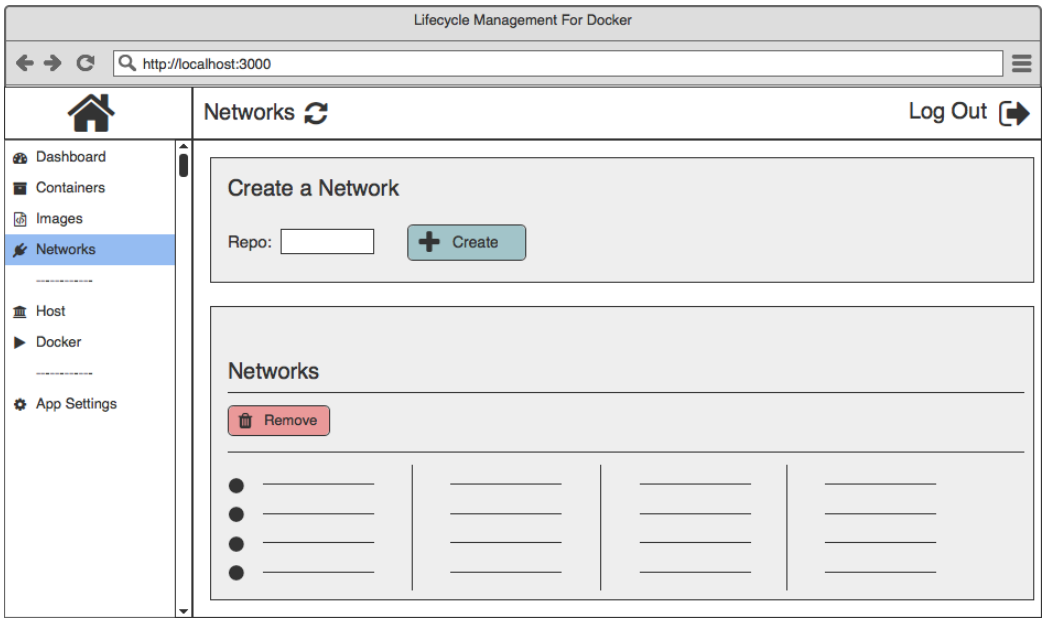

Figure 37: *Networks Mockup*

The individual view for a network. It will give a high level overview of that network.

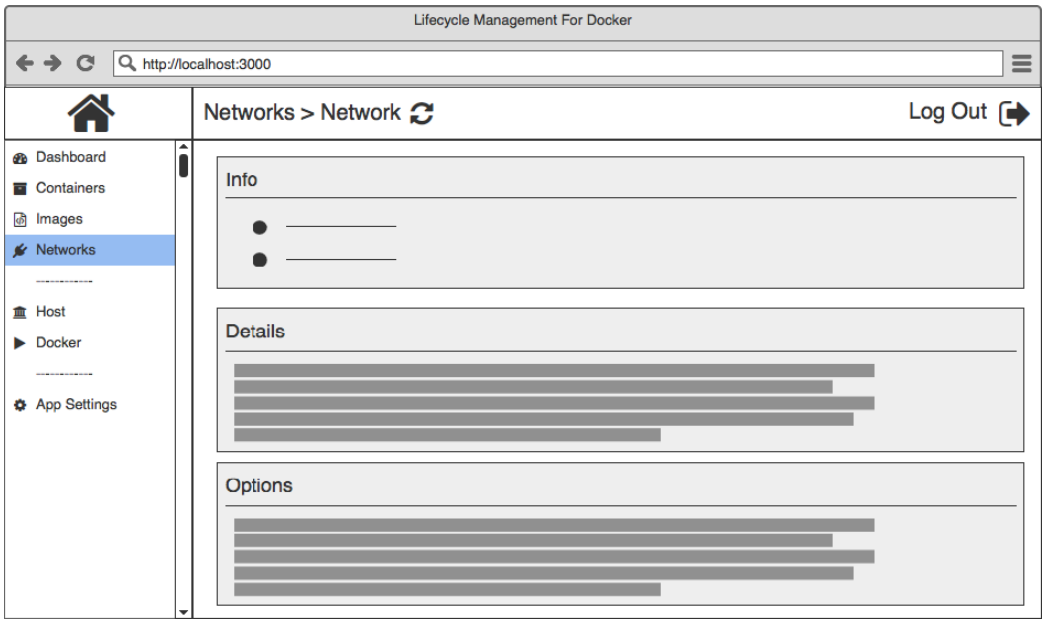

Figure 38: *Network Mockup*

This page will give all relevant information about the Docker installation. This page shows the host

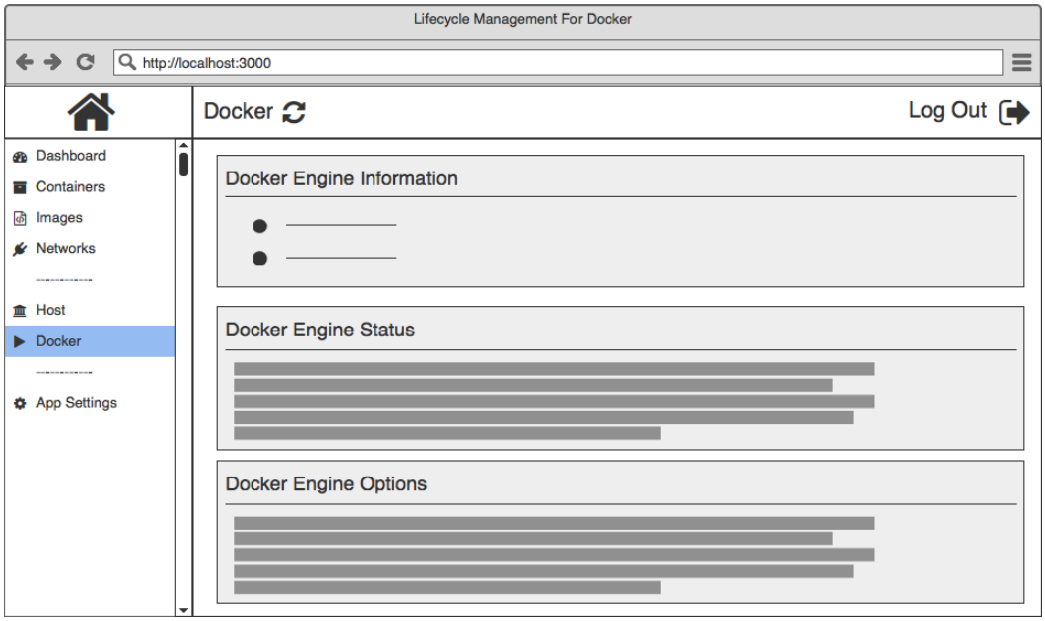

Figure 39: *Docker Mockup*

information page. Any relevant information which shows details of the host Docker is installed on will be displayed here.

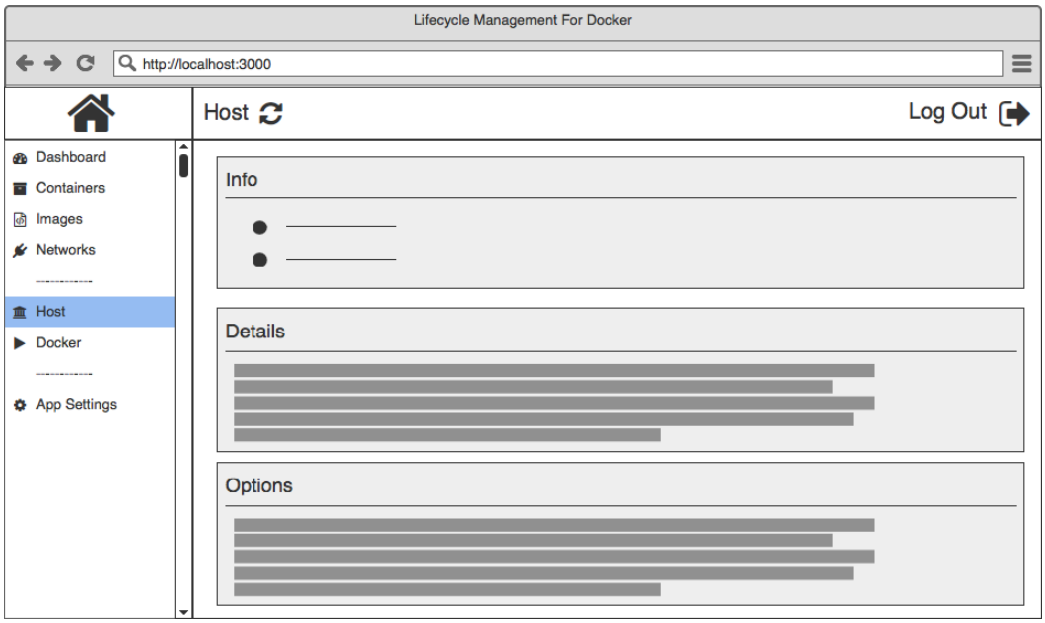

Figure 40: *Host Mockup*