Lifecycle Management For Docker UI

Final Report (Semester 2)

Stephen Coady

20064122

Supervisor: Dr. Brenda Mullally

BSc (Hons) in Applied Computing

Plagiarism 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.

Contents

Glossary

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, 24

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

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

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. 25, 26

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, 26

Continuos 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. 12, 19, 26

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, 18

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. 9, 11, 14, 15, 19

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. 7, 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, 17, 27

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. 26, 27

Git A version control system which allows source code to be tightly controlled and stored in a central repository. 9, 18, 22, 26, 29

Github An online platform to store git repositories. Allows for multi-developer collaboration on projects. 9, 23, 26

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

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, 26

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. 26, 27

Technical Debt The extra development work which arises as a result of implementing an easier but less robust solution. 13, 26

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, 26

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

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. 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 Mobile, acting as product owners. Red Hat have a vested interest in solving the problem discussed in Section 1.1 and so will act as product owners in the development of this product. 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.

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

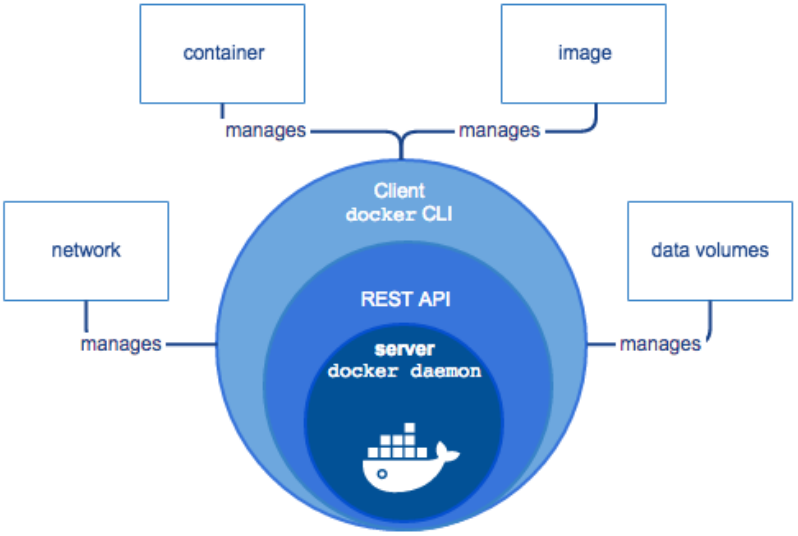

Figure 1: *Docker Engine Components. Credit: (Docker, 2017a)*

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 format the data it is trying to present in an odd fashion depending on things like screen size etc.

1.2 Aims and Objectives

The aim of this project is to address all of these problems while 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 UI which will

  - allow users to manipulate images, containers etc on the host with the same capabilities as the command line
  - allow them to do this remotely
- 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

As this is the second of 2 reports detailing the progress of this project it is important to provide a summary of report one. The work completed in semester one was achieved in the form of 2 sprints, the first lasting 2 weeks and the second taking place over 3 weeks.

1.3.1 Sprints

Sprint 1:

- 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 added to the application to provide CRUD functionality to the container and image endpoints

Sprint 2:

- 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.

To gain a complete understanding of the process and what was achieved semester one report is available in Appendix G.

The final product of these 2 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 we will discuss 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.

- More time should be spent making the test suite robust. While complex tests cases require initial investment that outlay can save time in the long run.
- Creating a large section of the CI/CD pipeline was a good investment also as it meant a better 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 a good one as it allowed for a flexible and fast development cycle. This is ideal 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 at all, 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.

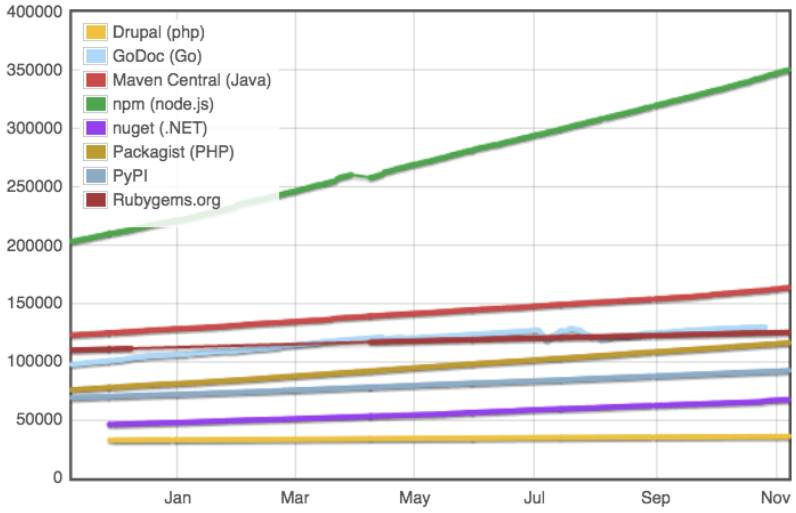

Figure 2: *Various Module Counts (Credit: modulecounts.com)*

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.

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 continuos integration 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 will be explored further in Section 4.

2.5 SonarQube

All code in this project should be held to a high standard. Since this project is open source it is important to ensure all code is bug free and follows current best practices. While this is important it also ensures more educational value for the writer.

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 also scans the code to find possible bugs and code smells. It can also 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 I. However in this section we will examine the system overview and all aspects of design will be discussed.

3.1 System Architecture Overview

To gain a better understanding of the system we will now 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.

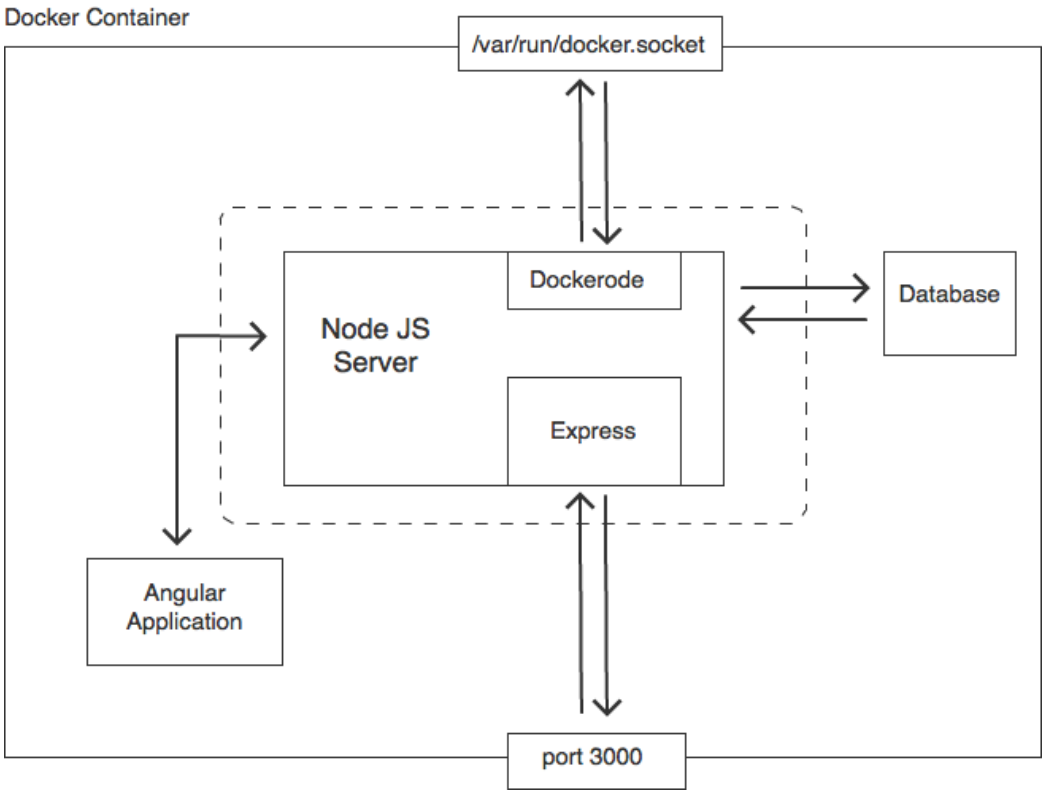

Figure 3: *System Architecture*

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 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 any 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. 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 would have a clean, minimalistic interface using a sidebar to navigate all available pages.
- 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. One of these, the containers view, can be seen below in Figure 4 while the rest are available in Appendix J.

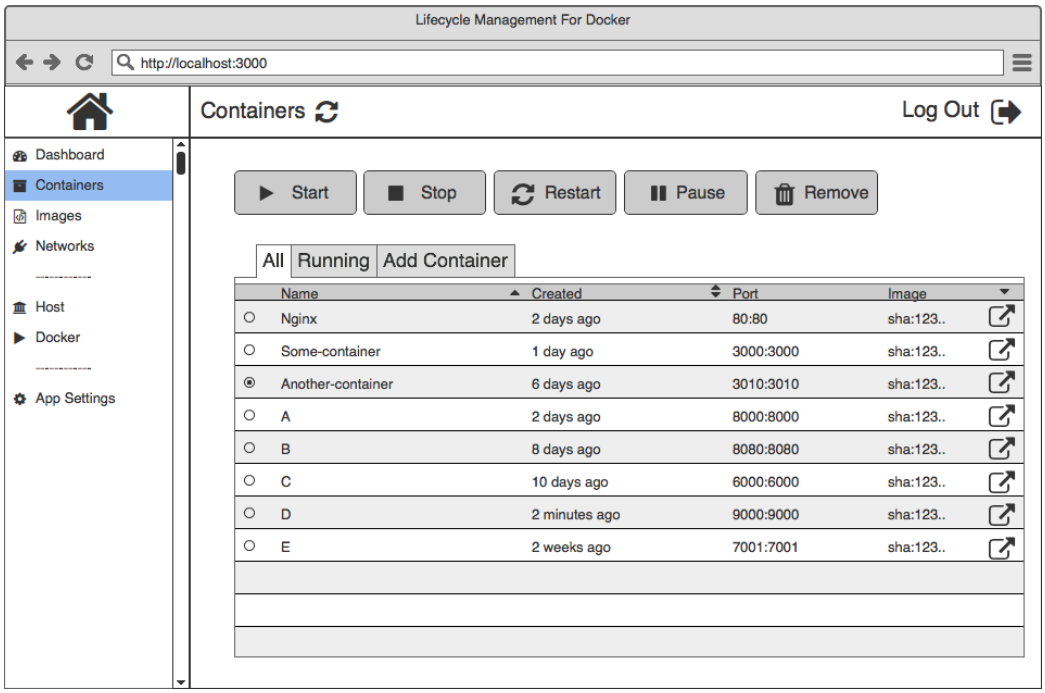

Figure 4: *Containers Mockup*

3.2.2 Mobile Web Application

Instead of creating a dedicated mobile application on a proprietary platform it was decided that a web application would be better. This is mainly because:

- 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 handle 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 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 my 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 (Kubernetes, 2017; Docker, 2017*b*).

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).
- 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.

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.

3.3.1 12 Factor Application

A recent methodology has emerged regarding to 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). We will now examine how each of these twelve factors influenced the design of this application.

1. Codebase

'One codebase tracked in revision control, many deploys'

This application 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. The application can even 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 and 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 and removed and quickly started again without affecting the next instance's startup and 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 continuos 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 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, 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 very 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 has begun. 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 early on 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 even for more to emerge as development continues.

Agile uses time-boxed units called sprints 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.

While following the Agile methodology, this project will employ some Scrum techniques to better manage development. The complete process can be seen in Figure 5.

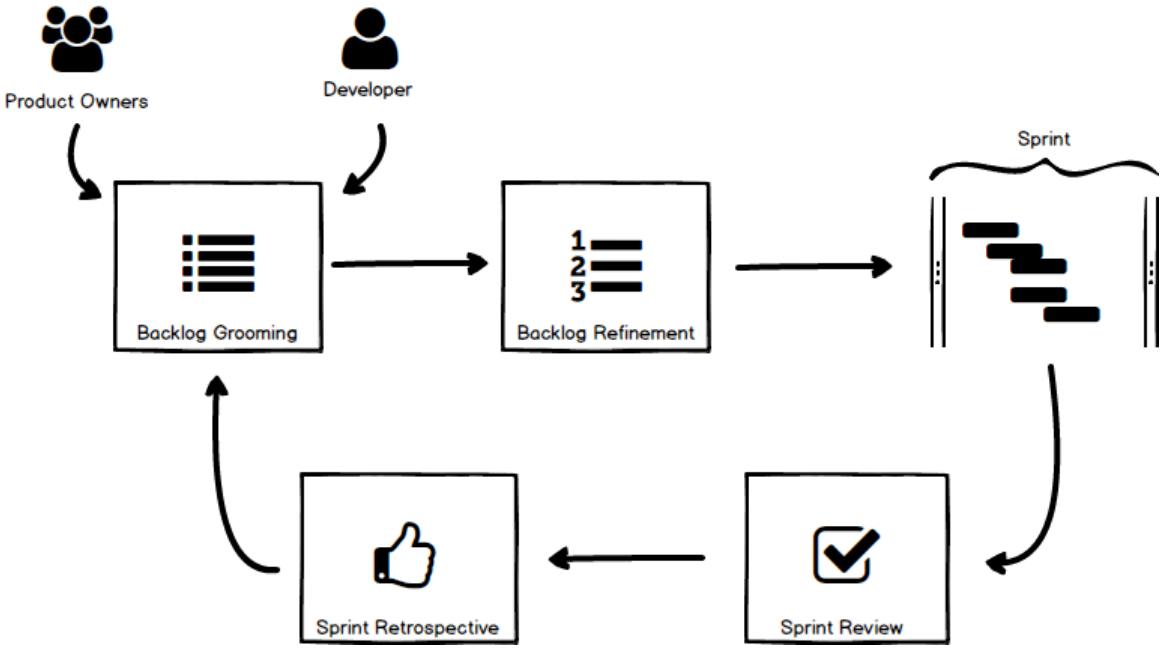

Figure 5: *Agile/Scrum Process*

- 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.
- Backlog Refinement - 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.
- 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.
- 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 following sprint.

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 which

is currently used by Red Hat Mobile. 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, estimation of work and subtasks.

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.

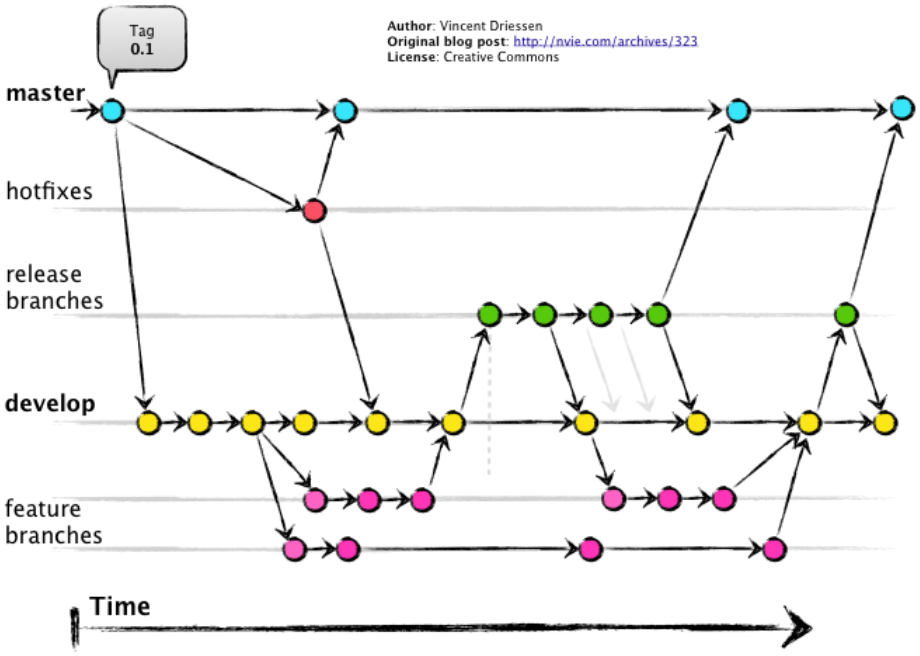

Figure 6: *Gitflow (Driessen, 2017)*

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 we can see how a developer can easily download a specific version.

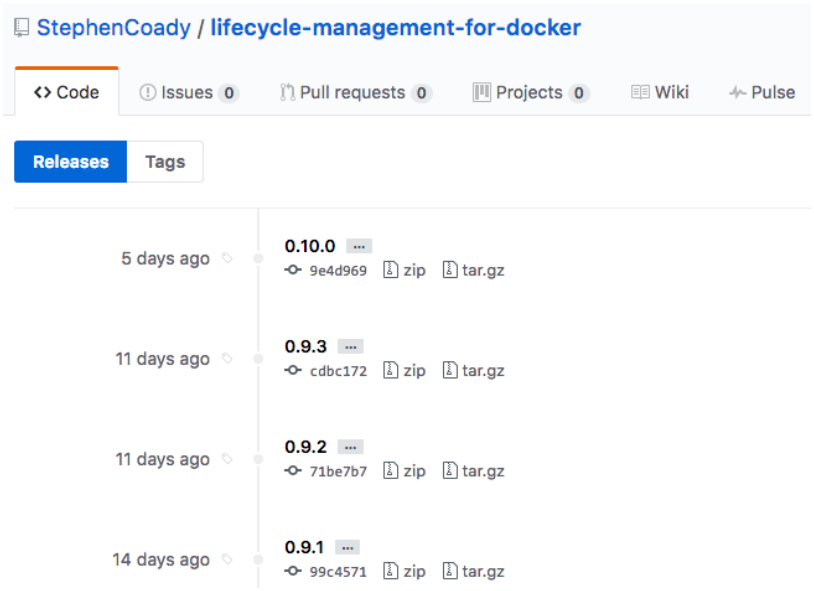

Figure 7: *Git Tagged Versions*

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 to 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 and it is for that reason great consideration was given to the test plan for this application. Software testing in this project can be 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. We will now look at each of these and 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 some 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

```
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*

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*). We will discuss this further in Section 4.6 but for the purposes of this 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).

4.5 Code Quality/Coverage

For an open source project it is vital 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.

As we will see in the next section SonarQube is embedded in the continuos 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

continuos 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".

To achieve this, a complete integration pipeline was built. This pipeline consisted 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 in the form of a running container

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

- Git commit is made
- Travis CI build is triggered which runs all tests and code coverage reports

- If successful a SonarQube push is triggered and the code is sent for evaluation
- Once this is complete a 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
- Once the push is complete Travis then deploys the application (by running a container) on the staging server.

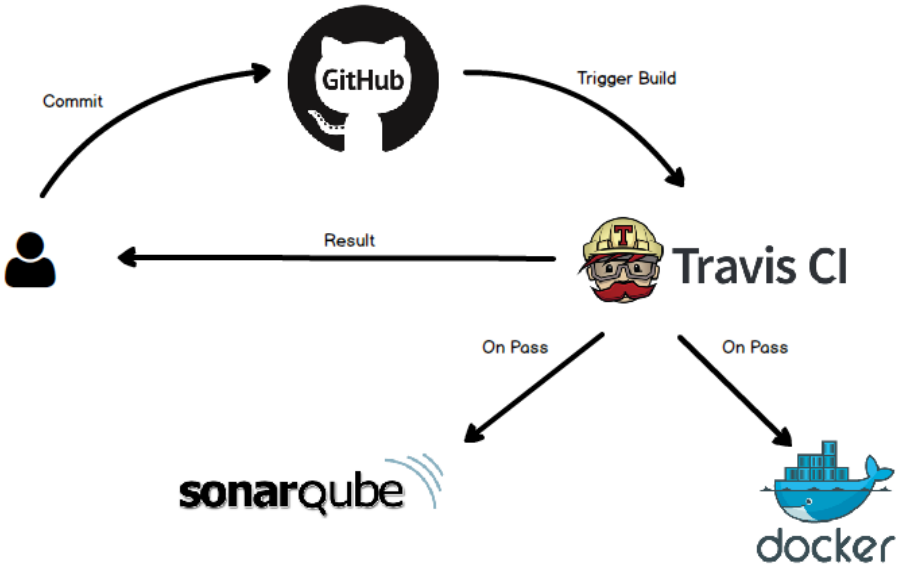

Figure 10: *Continuous Integration*

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

- The process is completely automated, meaning one git push is all that is required to completely build the application and make it ready for deployment.
- Using SonarQube will mean 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 Documentation

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

4.7.1 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 being 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 in Figure 11. We can also see how one individual API endpoint is shown in swagger in Figure 12.

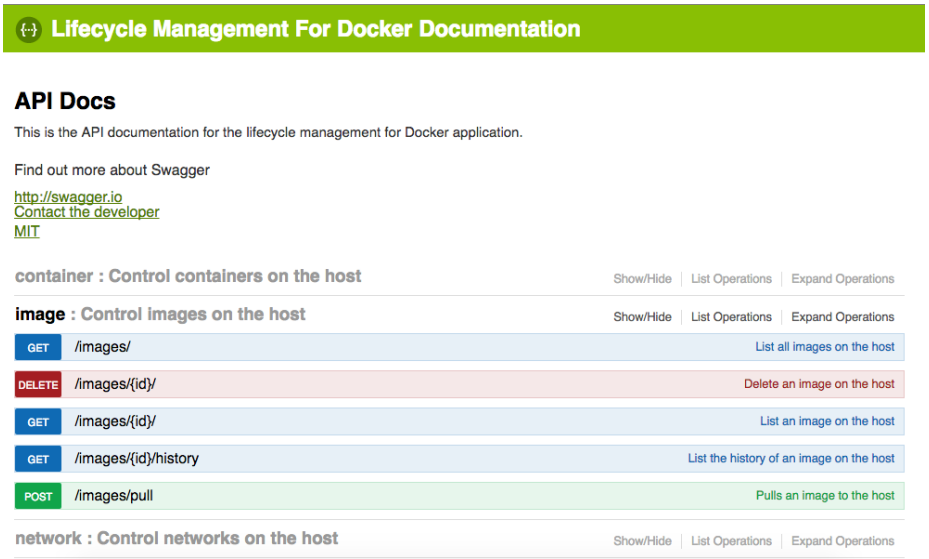

Figure 11: *Swagger Overview*

Making it accessible

To make the documentation easily accessible to anybody who wishes to view it it has been embedded

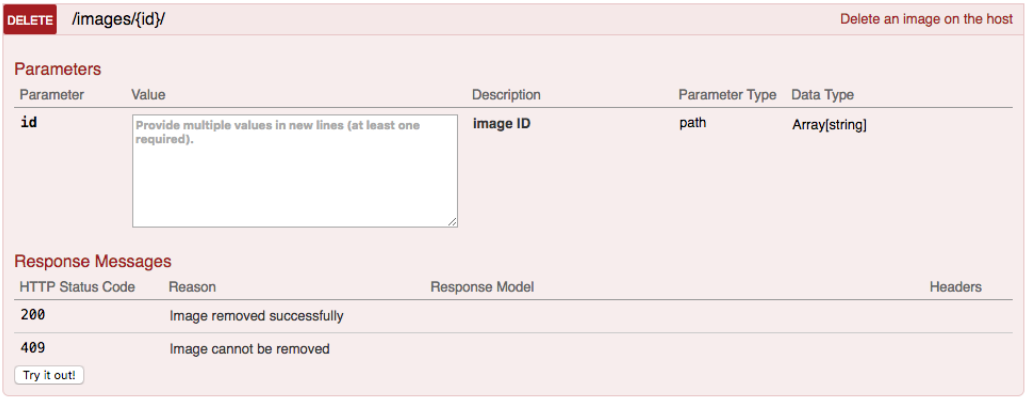

Figure 12: *A Single Swagger Endpoint*

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.

4.7.2 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 also can auto-create sections such as glossaries and tables of contents.

The source code used to create this report can be seen in Appendix G.

5 Implementation

6 Summary

6.1 Reflection

6.2 Project Direction

6.3 Review

Appendices

A Application Repository

https://github.com/StephenCoady/lifecycle-management-for-docker

B Travis Repository

https://travis-ci.org/StephenCoady/lifecycle-management-for-docker

C SonarQube Repository

https://sonarqube.com/dashboard/index?id=lifecyle-management-for-docker

D DockerHub Repository

https://hub.docker.com/r/scoady2/lifecycle-management-for-docker/

E Staging Server

application: http://87.44.18.55:3000/ documentation: http://87.44.18.55/docs

default username: admin
default password: admin

F Dockerode

https://github.com/apocas/dockerode

G Report Source Code

https://github.com/StephenCoady/fyp-documentation

H Sprint Retrospectives

2017-02-01 Docker Project Sprint 1 Retrospective

Date 01 Feb 2017
Participants Leigh Griffin Stephen Coady

Retrospective

What did we do well?

- Already had prototype in place to accelerate development
- 3rd party module knowledge accelerated development
- Guidance from Red Hat really helped focus the sprint
- Scope on tickets well understood from Red Hats perspective

What should 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

2017-02-15 Docker Sprint 2 Retrospective

Date 15 Feb 2017
Participants Leigh Griffin Stephen Coady David Martin

Retrospective

What did we do well?

- We performed a backlog review and set the priority for the remaining 3 sprints
- Progression of the sprint was excellent, good pace to it and good story pointing
- Comms was pretty good
- Smooth sprint, story points and tickets were well scoped
- Sprint Planning revisited the story points so very little surprises
- Team (Stephen) came to the Stakeholders (Dave & Leigh) with the plan for the next Sprint

What should we have done better?

- Sprint started at a bad time college wise, 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. Would recommend emailing that to us both.
- Stephen Coady metrics spike insight when you get to it (this sprint possibly)

2017-02-28 Docker Sprint 3Retrospective

Date 28 Feb 2017
Participants Leigh Griffin Stephen Coady

Retrospective

What did we do well?

- The UI is very usable, lots of nice feedback and functionality visible now
- Consistency in the velocity at 20 points
- Got the wireframe relationship, it really helped with my front end skills which I was not confident in
- Wireframe feedback was excellent, really helped scope the work
- Got to demo to my supervisor which gave her a lot of insight
- Overall work pace was judged well for the most part, consistent delivery

What should we have done better?

- The story pointing on the skeleton was completely off, it could have derailed the entire sprint
- Velocity last sprint was off
    - you should have descoped when you realised how big the UI was
    - you should have re story pointed the UI mid sprint to allow a controlled descope
- Tickets are not descriptive enough, need to add more metadata
- Didn't descope the testing in a container ticket, should have done that when the UI became so big

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

I Formal System Models

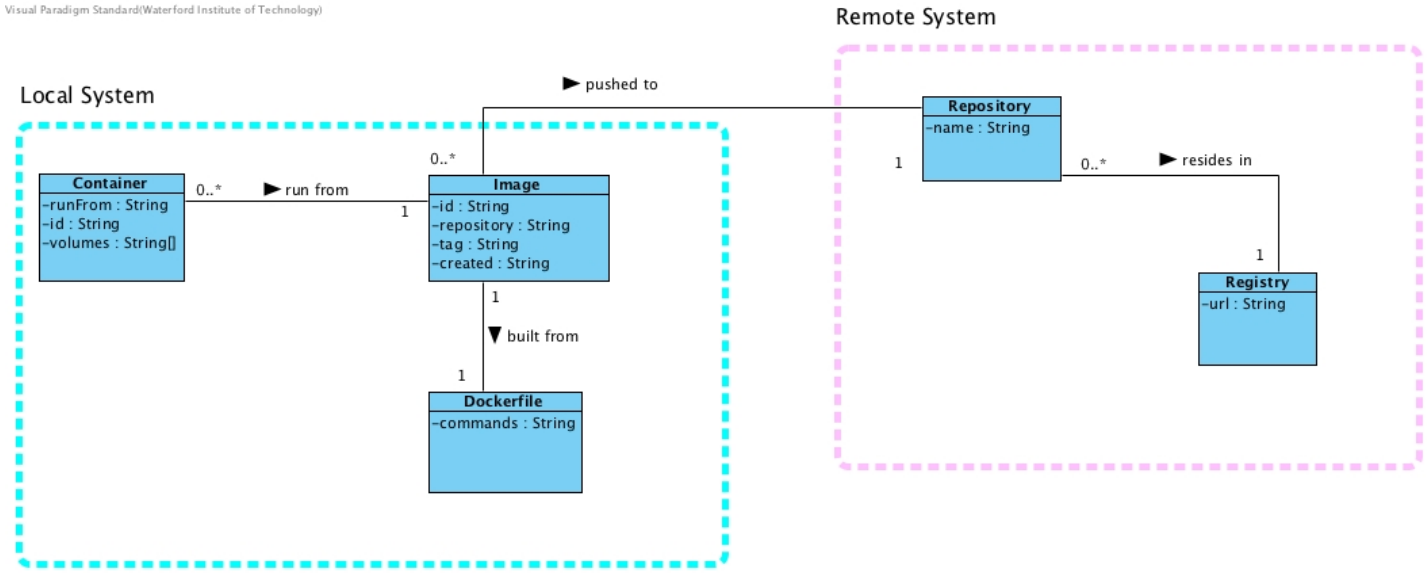

Figure 13: *Class Diagram*

J Wireframes

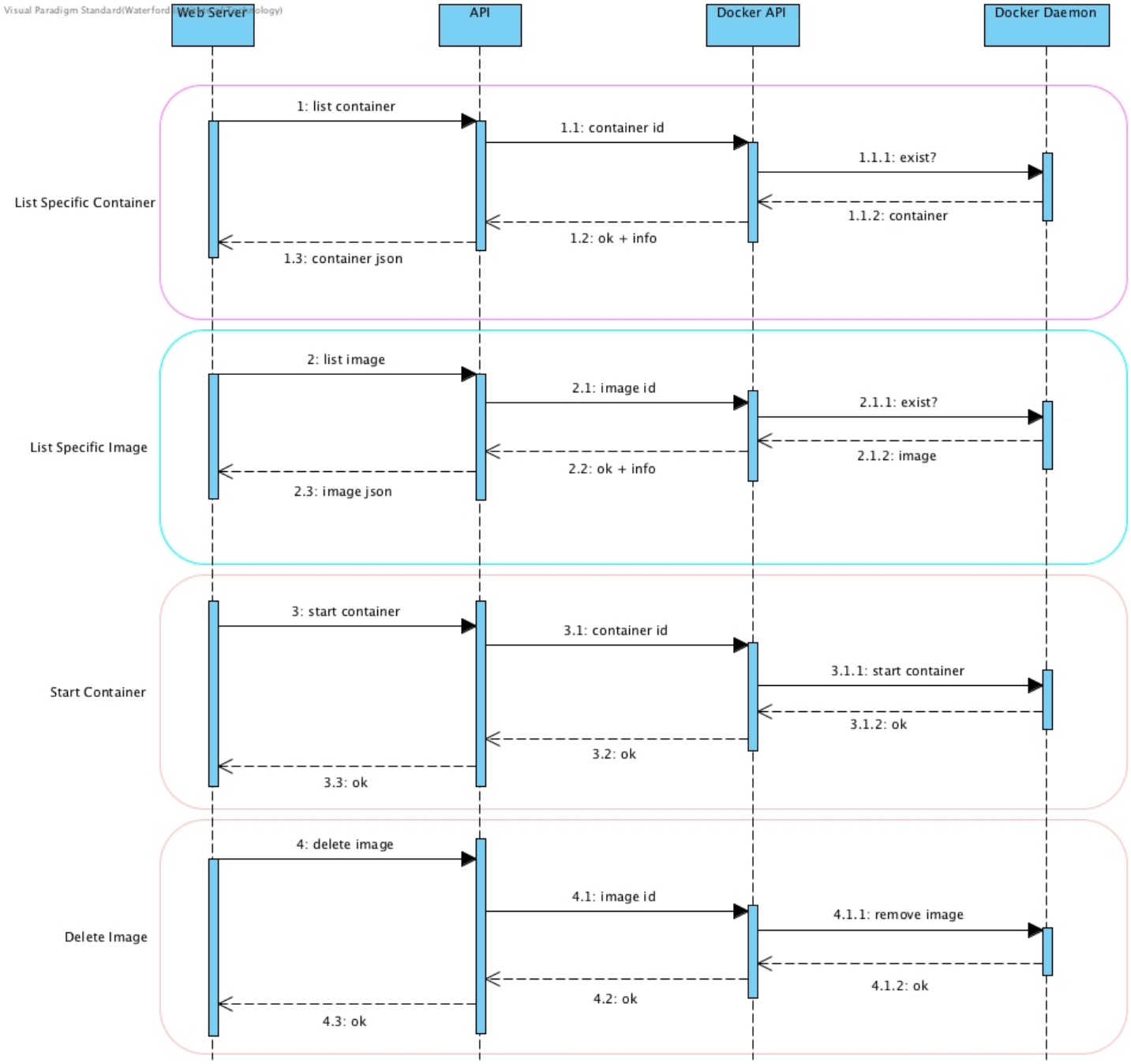

Figure 14: *Sequence Diagram*

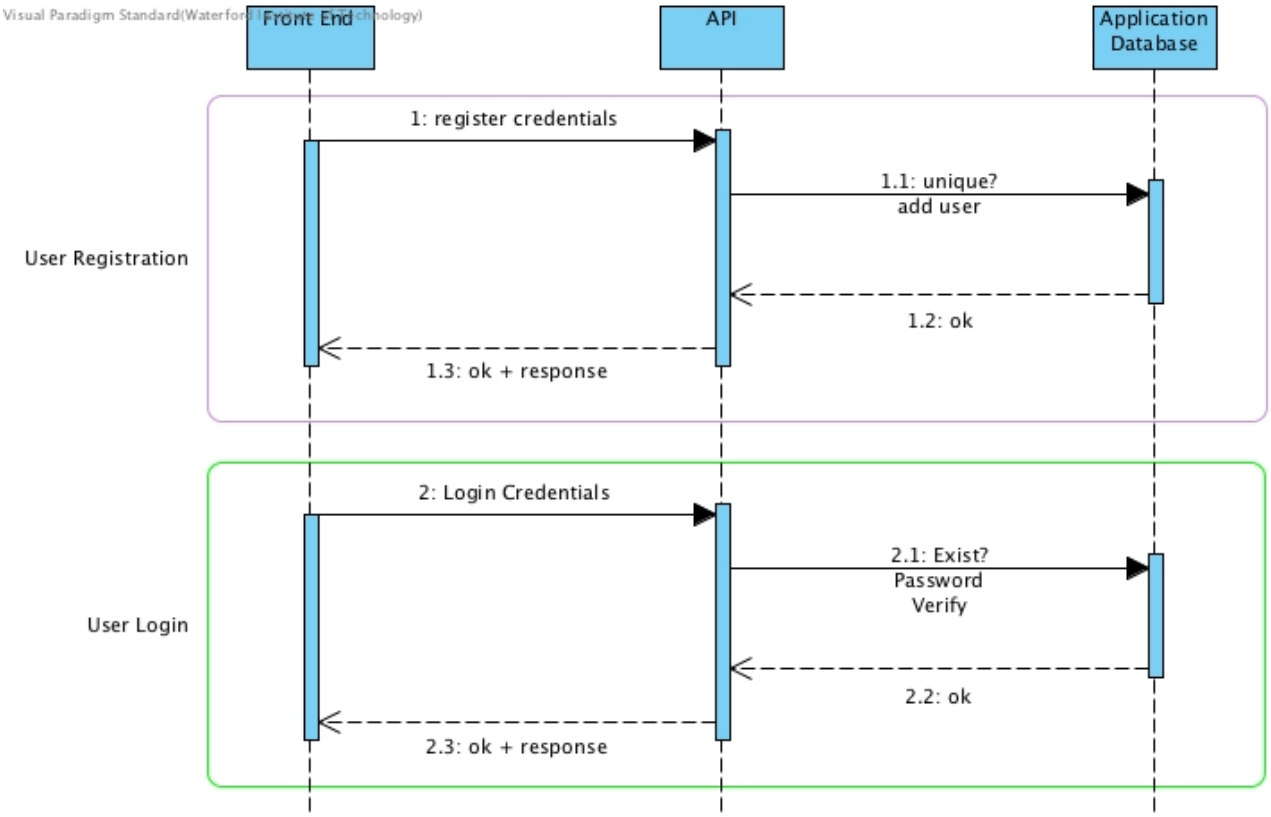

Figure 15: *User System Sequence Diagram*

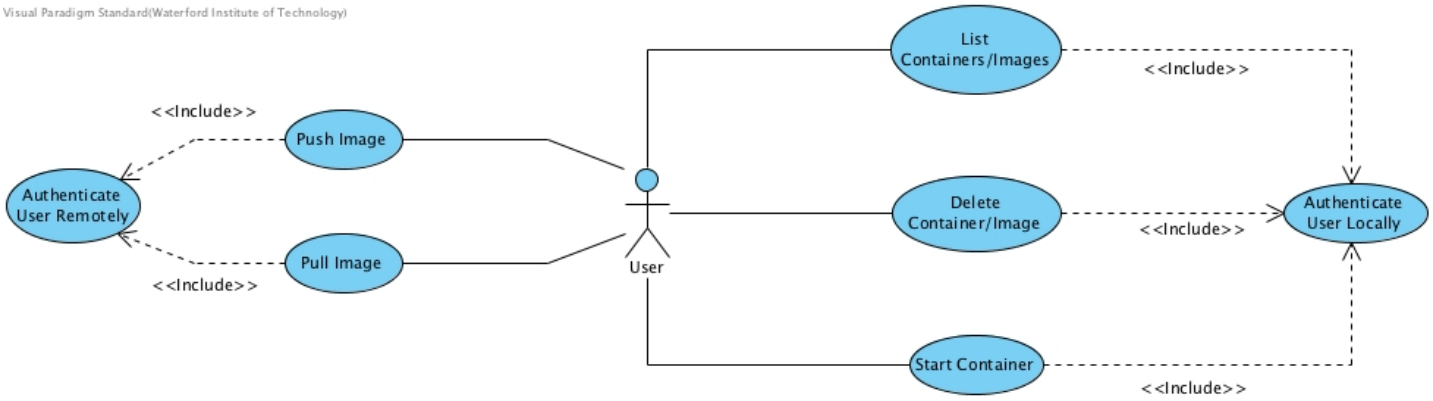

Figure 16: *Use Case Diagram*

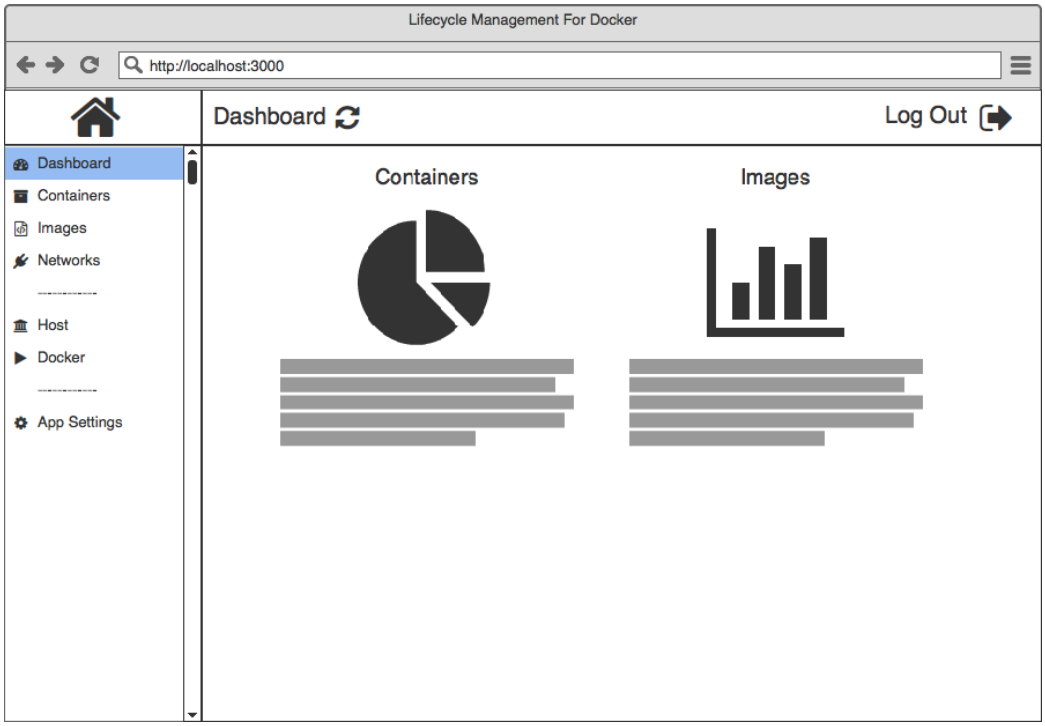

Figure 17: *Dashboard Mockup*

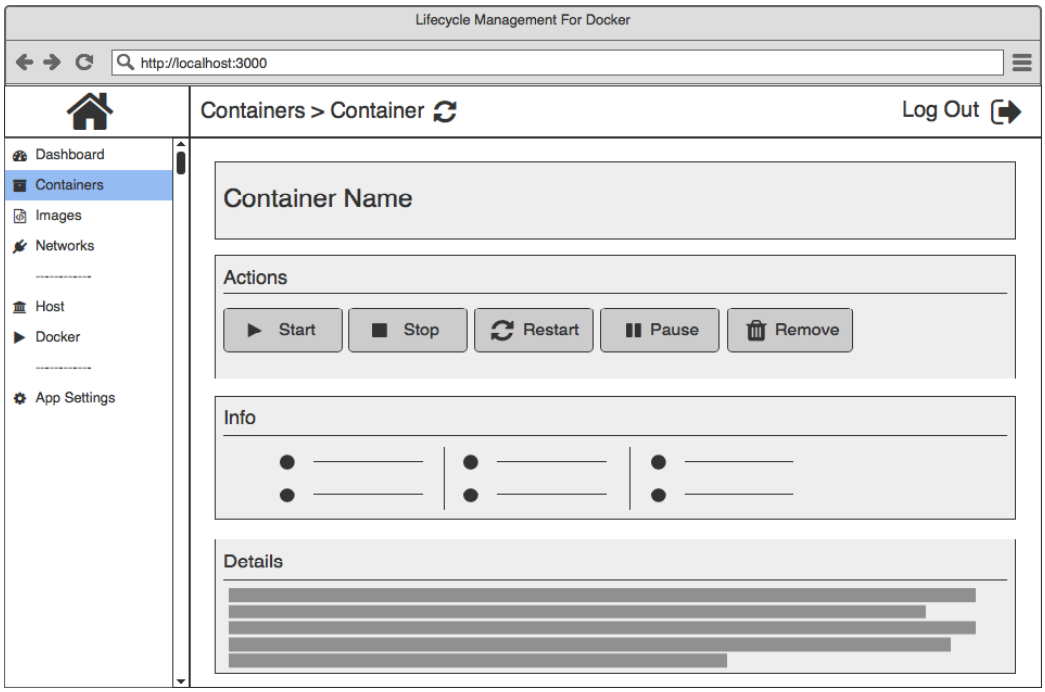

Figure 18: *Container Mockup*

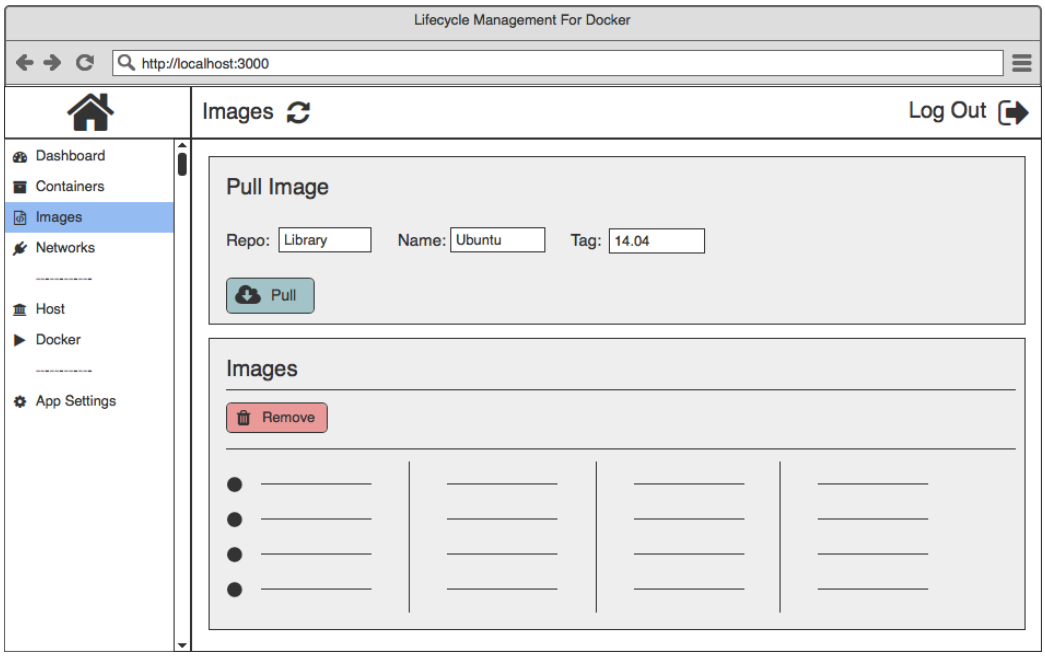

Figure 19: *Images Mockup*

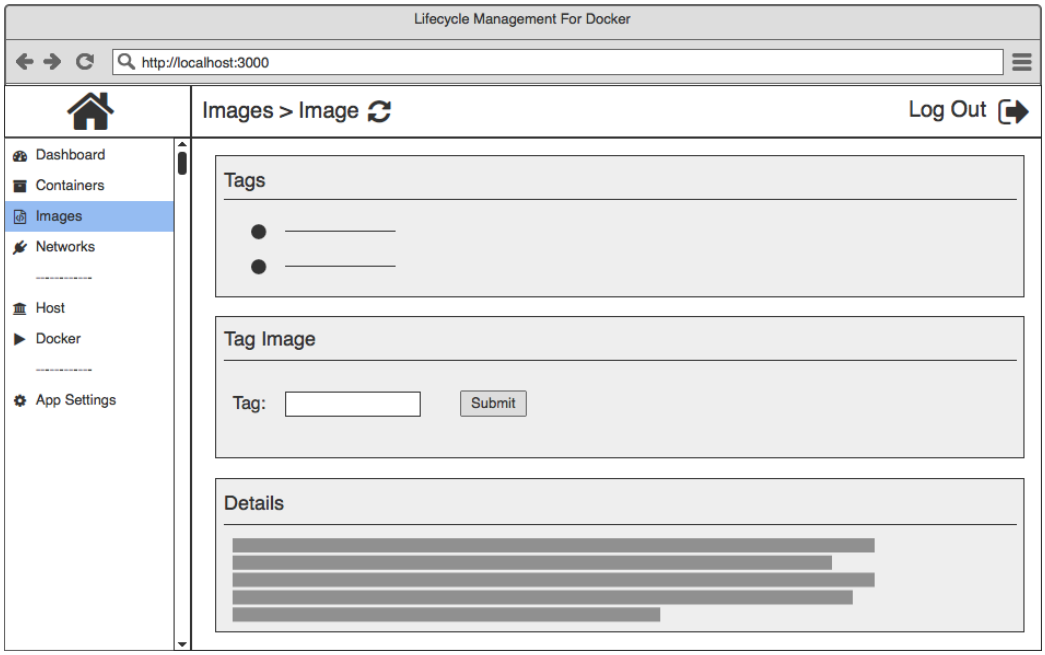

Figure 20: *Image Mockup*

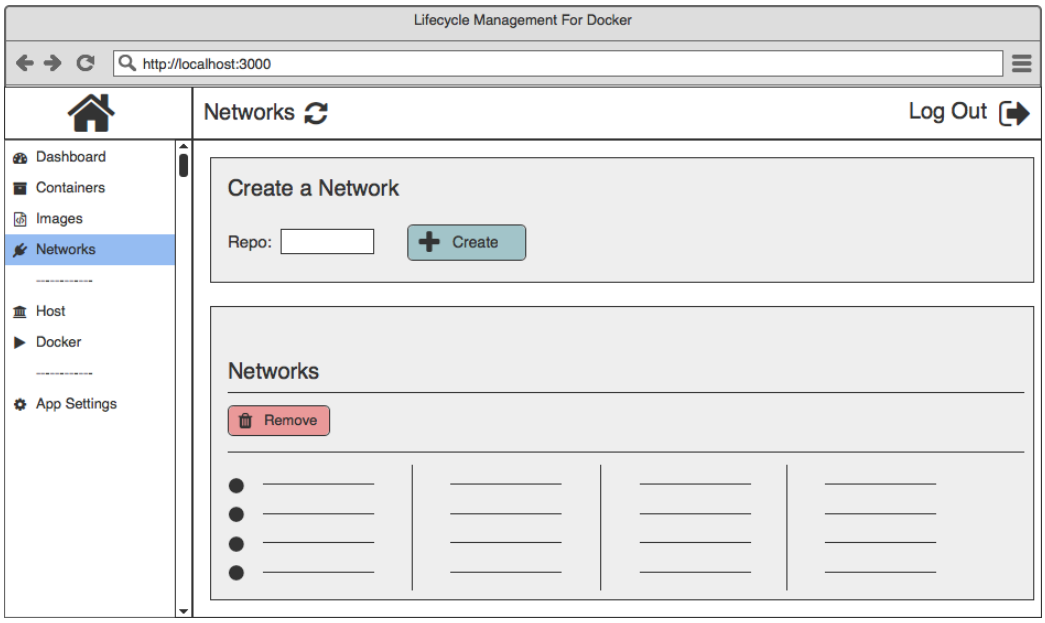

Figure 21: *Networks Mockup*

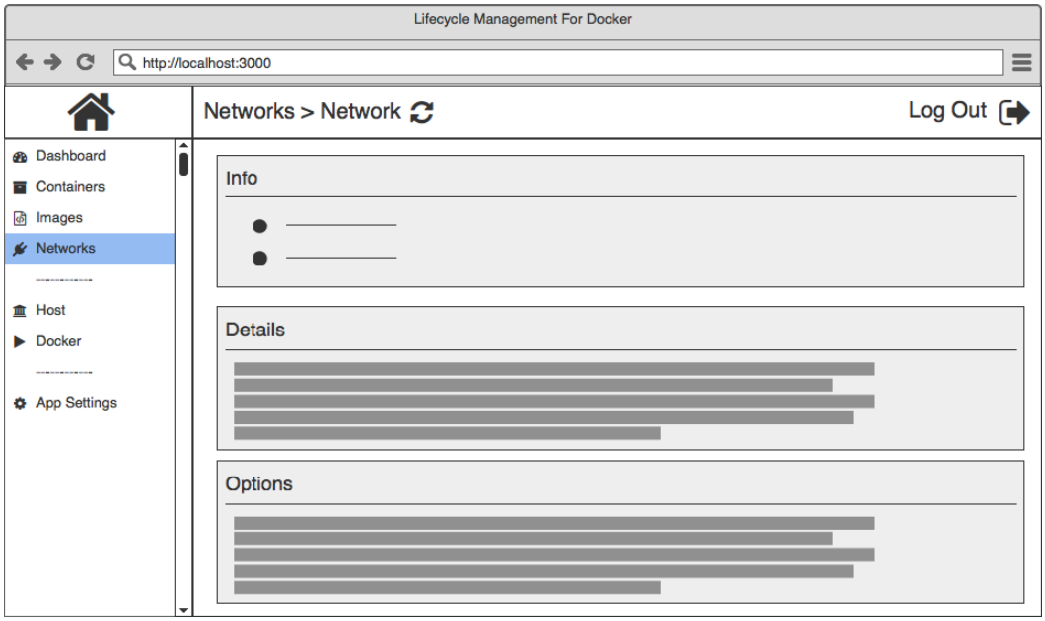

Figure 22: *Network Mockup*

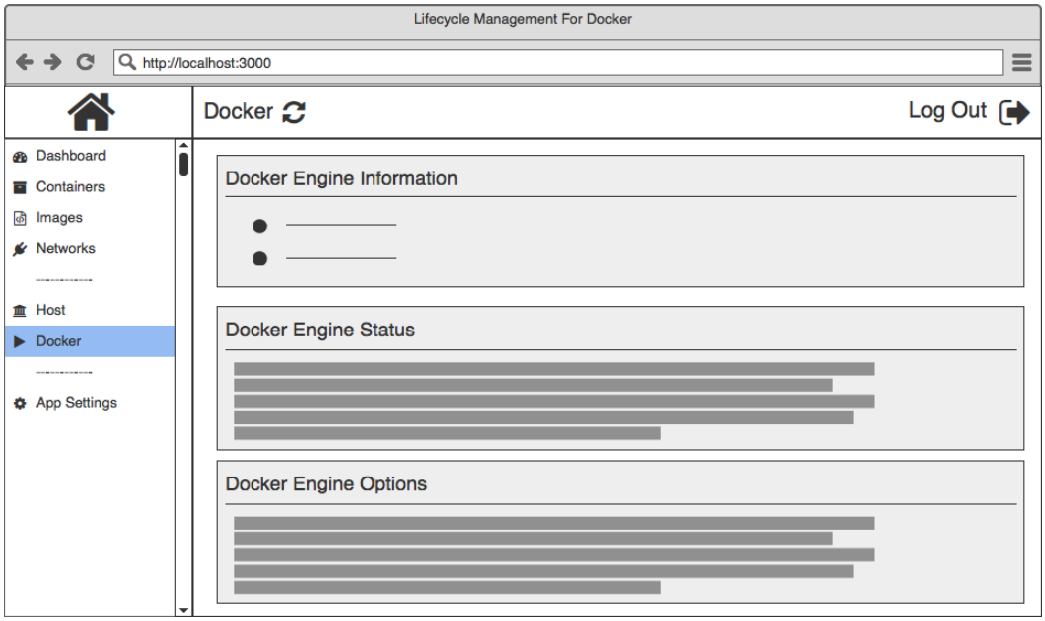

Figure 23: *Docker Mockup*

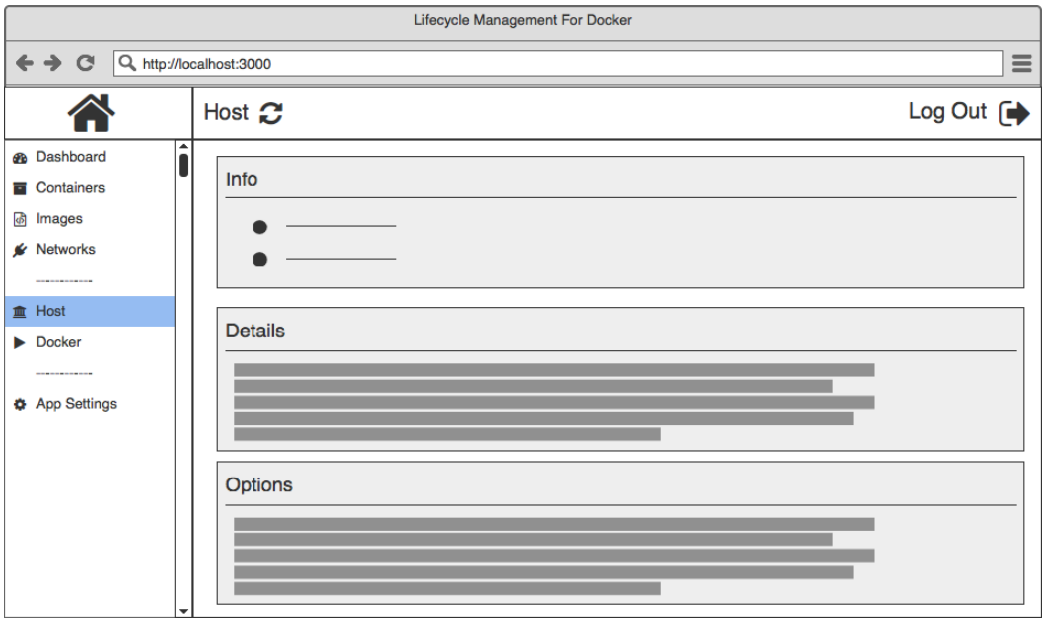

Figure 24: *Host Mockup*

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*

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

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/*

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/*