Lifecycle Management For Docker UI

Status Report (Semester 1)

Stephen Coady

20064122

Supervisor: Dr. Brenda Mullally

Second Reader: Dr. Peter Carew

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. 8, 9, 19, 20, 31

Command Line The program through which the operating system can be interacted with. The user enters a command they wish to execute on a text line. 7

Container An isolated piece of software bundled with everything it needs to run on the host. 7, 8

Continuos Deployment The act of continuously deploying software which has passed acceptance tests to a live system. 7

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. 7, 12

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. 31

Docker A piece of software which enables orchestration and management of containers on a host. 6, 7

Docker Compose A tool developed by Docker to allow developers to easily orchestrate multi-container systems without requiring them to manually manage containers. 7, 31

Docker Daemon The server-side component of the Docker Engine. 9

Docker Image A template from which many containers can be started. Can be pushed an pulled to/from a remote Docker registry. 7

Git A version control system which allows source code to be tightly controlled and stored in a central repository. 13, 31

Github An online platform to store git repositories. Allows for multi-developer collaboration on projects. 13, 31

Lifecycle Management The management of a container over its complete lifecycle, from its beginnings as an image to starting, stopping and restarting the container and managing the interactions it has with other containers on the system. 6, 7, 14

Open Source Software which encourages input and feedback from other developers and allows for any and all modifications to the current software package with the intent to release it under a new package. 7, 10

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. 13, 31

Technical Debt The extra development work which arises as a result of implementing an easier but less robust solution. 9

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. 13, 31

1 Introduction

The purpose of this report is to detail the lifecycle management for Docker application. It will aim to explain all elements of the project by examining the proposed methodologies and technologies which will be used and evaluating the risks associated with a project such as this one. It will also look at the design and implementation by using formal tools such as the unified modelling language (UML).

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 2.1 and so will act as product owners in the development of this product. This will ensure that the initial requirements will accurately aim to solve the problem.

The project supervisor, Dr. Brenda Mullally, will also act as Scrum master. This will ensure that the product backlog accurately reflects the state of the project at any moment in time.

2 Project Overview

2.1 Problem

Docker is an open source software solution to package any piece of software in a completely isolated runtime which contains everything the packaged software needs to run (Docker, 2016*b*). This isolated environment is called a container. This project will deal with the management of these containers from their inception as images, to starting and stopping the containers, and managing the interactions they have with each other. This whole process is known as the lifecycle management of containers.

Currently, container lifecycle management is predominantly achieved through the command line. This is due to the fact that the people currently working with Docker are developers who are comfortable using the command line. This means that managing images, running containers and managing their interactions with each other can all be time-consuming tasks. One way in which this is aided is by the use of scripts. While this is useful, it does also mean that the user must be comfortable with the command line to manage the containers. However as the industry moves towards using containers (Datadog, 2016) it will mean the need to cater for users who are not comfortable with the command line.

Managing containers can quickly become a tedious and error-prone task when the number of containers starts to grow. In a production system it can mean provisioning and managing dozens of containers and their dependencies. This, coupled with the fact that the container must be managed not just at start up but over its complete lifecycle, means that the advantage of using containers can quickly be negated by their lifecycle management.

2.2 Industry Example

As a real-life example, the Red Hat Mobile Application platform (RedHat, 2016) is made up of between 20-25 Docker Images. In a development environment this translates to the same number of running containers to manage. While this is no small feat, it can be mitigated by things like scripts, Docker Compose etc. However in a production environment the number of containers can quickly grow, depending on resiliency and load requirements. At this point it is now a much larger task to manage all of these containers.

Red Hat Mobile also employ a Continuos Integration/Continuos Deployment (CI/CD) model when it comes to their software. Whenever a pull request is opened for a component of their application, it triggers a CI build, which in turn (depending on whether it passes or not) can trigger a build of a Docker Image related to this component. So essentially for each Docker image there may be many

tagged versions of that image built and ready to be used. This process could be aided by a visual component to view and inspect these images.

2.3 Solution

With the aim of solving the problem described in Sections 2.1 and 2.2, the overall goal is to provide a tool to aid with the management of a container's lifecycle. In other words, the end product of this project will aim to tie together the basic container and image controls to build one cohesive management unit. This unit will allow for a graphical information panel and console to carry out any tasks needed.

It will aim to provide a means to remotely manage a server and orchestrate the containers and container images on that instance, therefore greatly reducing the overhead in management of that server.

2.4 Goals

The core or primary goals can be broken up into functional and non-functional as follows:

2.4.1 Functional

- Image Manipulation/Control
- Container Manipulation/Control
- Remote Control of the application through an API
- View Running statistics of containers
- Search Docker Hub/Private Registry

These are the minimum goals which will solve the problem discussed in Section 2.1. Some stretch goals, which may prove too large to fit into the project are:

- Mobile Application
- Having the final application itself run within a container
- Connecting to multiple servers running the application at once
- CI/CD Integration

Although these goals are not deemed immediately important and potentially undeliverable it is still important to keep them on the backlog. This is due to the methodology the project will follow, which we will look at further in Section 3.1.

2.4.2 Non-Functional

Testing

As the time span of this project is relatively short it is important to keep the technical debt at a minimum. To achieve this a third party tool called SonarQube will be used (SonarQube, 2016). This tool will be used to constantly monitor the quality of the code in the project to ensure that it is of the highest possible quality. This is important for a number of reasons, mainly to reduce the previously mentioned technical debt but also essentially meaning the codebase can be easily understood and maintained by the community after the project has finished.

This project will also aim to deliver a product with extensive unit tests. It will follow the behavioural driven development (BDD) method which will ensure code is hardened and minimise the amount of bugs. This will also mean greater development speed as code will be re-usable with greater efficiency. This is discussed further in Section 5.6.

Security

The end product will also be secure. The Docker Daemon needs to run on a host with root privileges, which can have disastrous effects for the host if not managed correctly. Since this application will effectively allow remote control of this daemon it is important that only a trusted user be allowed to use the application.

Scalability

When the application can run on a server and allow the containers on that server be controlled it is a very useful application. However, in industry it is rarely just one server housing the application. For this reason the end product of this project will need to scale *with* the server it is managing. To achieve this the application will need to be able to communicate with several servers at once.

2.4.3 Deliverables

The deliverables of this project are as follows:

Server-side Application

This is the main part of the application and will act as the controller of the Docker API.

Front-end Application

This will be the interface through which the user will communicate with the server application. It will be exposed through an API to allow for remote calls.

Open Source Node Module

This project will produce a fully open source node module, available for the wider-developer community to maintain and build upon if they so wish.

Containerised Application

Although a stretch goal, a possible deliverable from this project would be to have the whole application running within a container. This would allow for rapid deployment of the application and would also be a huge learning outcome.

3 Investigation

3.1 Methodology

The methodology which this project will be carried out under is an important decision. It will have a significant impact on timelines, deliverables and overall product quality. Therefore which methodology to use should be given careful consideration. The guide by (Bowes, 2016) was used as reference when making this decision.

The two to be considered are Agile and a more traditional approach named Waterfall. While both have strengths and weaknesses Agile was chosen for this project as it was deemed a better fit 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 lone-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 1.

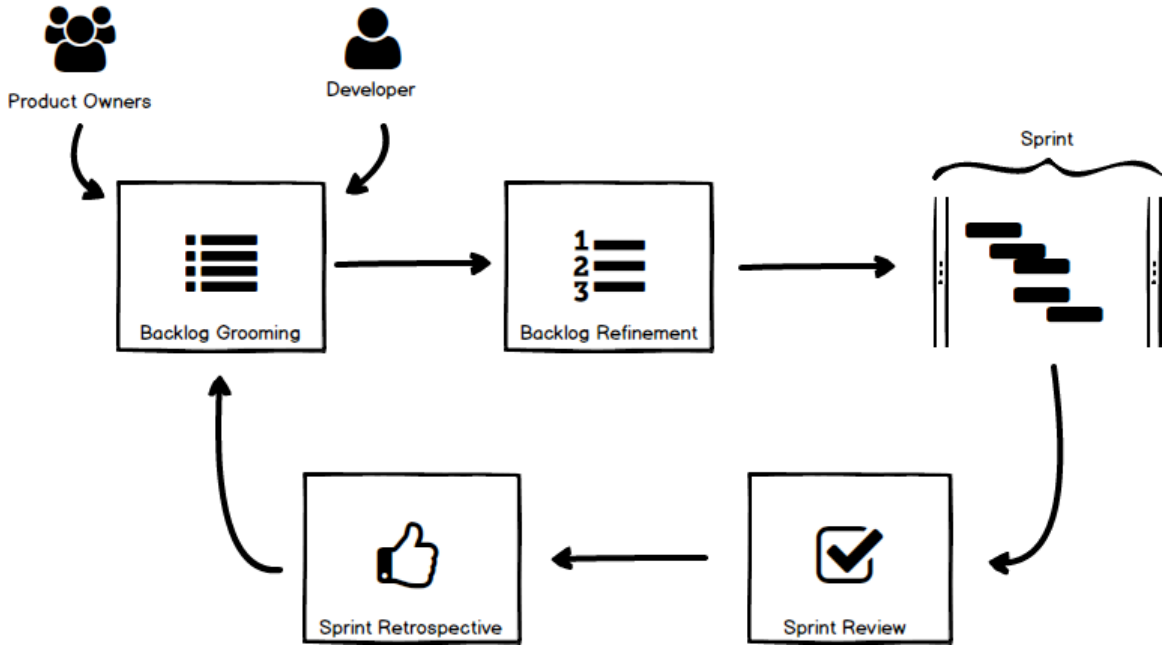

Figure 1: *Agile/Scrum Process*

- Backlog Grooming - There will be a planning session to initially scope the product backlog and to inform the total scope of the project. This will involve the product owner who will help shape the overall project direction by giving their priorities.
- Backlog Refinement - Refining the backlog to break large development tasks into smaller tasks which will fit better in a sprint. This will also help to identify similar tasks which can be grouped together and reduce duplication.
- Sprint Review - After a sprint is completed a sprint review will then be performed which will evaluate goals achieved versus goals planned and the backlog will then be 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.

3.2 Continuous Integration

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 will not be

built in a production environment having a continuous build cycle will ensure maximum quality code and also reduce the risk of a "bug bottleneck" further down the line.

To achieve this, a complete integration pipeline will be built. This pipeline will consist of the following:

- Git repository to house the code (stored remotely on Github)
- Travis CI build server to execute unit tests and carry out post-test tasks
- SonarQube Server to analyse and advise of improvement to code, based on best-practices
- Docker Repository to build a container with the latest application bundled

This pipeline can be seen in Figure 2 and will work as follows. When a git commit is made to the remote git repository a build will be automatically triggered on the associated Travis CI server which will then carry out all unit tests in an isolated environment. Travis will then inform of the results, and if all unit tests have passed will carry out a push to a remote SonarQube server which will analyse code. Once this is complete Travis will then build a Docker image of the application and push it to a public repository.

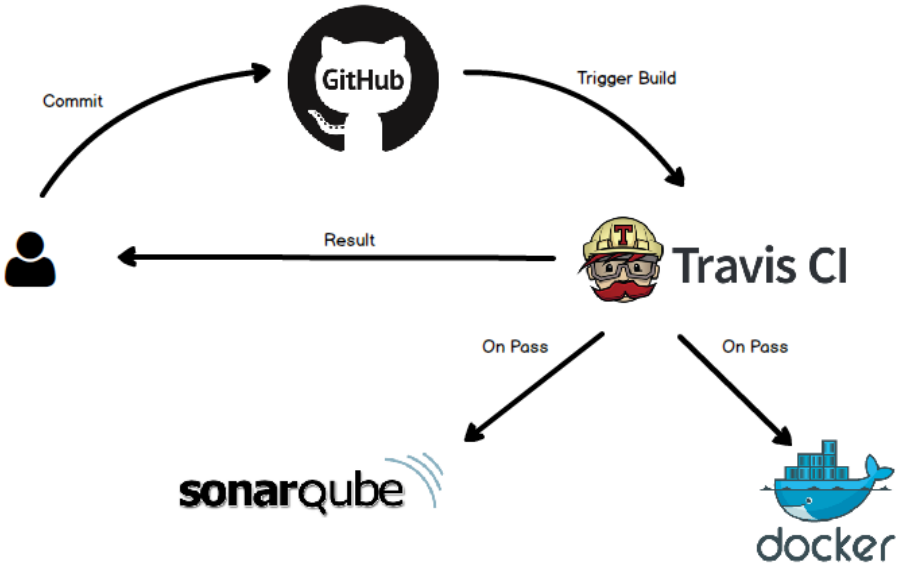

Figure 2: *Continuous Integration*

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.

3.3 Existing Solutions

Currently several other projects exist which aim to provide the same solution as this one does. However there are subtle differences in either the feature set or implementation of these solutions which validates this project as a worthwhile undertaking.

Lack of Features

"UI For Docker", is an open source project written in Golang (Ahlquist, 2016). It shares the same basic feature set as this project however it does not allow for advanced controls such as starting a container from a Dockerfile or searching for new images to start containers from. Also, upon experimentation with UI For Docker the programs error handling and user feedback was not found to be satisfactory. As an example if renaming a container was attempted while it was still running the container would simply crash instead of telling the user it must be stopped first. While it is an excellent project this combined with the lack of the previously mentioned features means it is not powerful enough to solve the domain problem discussed in Section 2.1.

Cost

Docker themselves provide a lifecycle management solution, however it comes as a monthly subscription (Docker, 2016*a*). It is not currently available to run privately which makes the product rather restricted. While the subscription fee is relatively small it is still a barrier for smaller teams or single developers. Since this project aims to produce an open-source application it is catering for an opening in the market.

4 Risk Analysis

This section will aim to look at the various types of risks involved in the development of this product. It will also show how these risks will be minimised and in some cases completely mitigated.

4.1 Requirements Risks

When building the requirements for the project it is important to accurately and definitively scope all necessary requirements. This ensures the project is not over-scoped effectively meaning an undeliverable end product.

However some risks here are:

- System requirements initially missing key components
- Over/under estimating the complexity of requirements
- Changes to requirements

To mitigate these risks, a full Agile/Scrum process will be followed as discussed in Section 3.1. This will ensure that:

- All requirements are recorded and accurately rated based on complexity and priority. This will ensure that the most important requirements are dealt with first. To achieve this a backlog grooming and refinement session was carried out with the input of both product owner and scrum master.
- Agile also facilitates changes to product requirements. If, for instance, during a sprint new requirements emerge - they can be added to the product backlog and in the next sprint planning session they can be taken in as new tasks in that sprint.

4.2 Skills Risks/Technological Risks

In software development there is always the risk that the technologies used will not allow for a feasible product to be built. To test this and avoid any risk once development has started, a prototype application was built to show complete technical feasibility. This application is available in Appendix A and is discussed in detail in Section 5.

The result of this prototype build was successful, and showed that all proposed components can be used together to create the complete application.

4.3 Project Management Risks

Project management is a vital aspect of any project, having a significant impact productivity and therefore the overall level of "completeness" of the project.

To ensure that the project is managed correctly, a formal process will be followed. Using Agile and also using the issue tracking system Jira will ensure that the project is managed correctly (Atlassian, 2016).

Also, having a formal definition of when a piece of software is done is key to project management, as it ensures any development undertaken will be complete by a common standard. This ensures consistency within the project. In this project there will be two definitions of done, the definition for each feature and also the definition for each release.

The definition of done for each feature is:

- The new code passes unit tests.
- All code is commented.
- When the new code is integrated with the current software package and all previous unit tests pass.
- Documentation comments are added to code.

The definition of done for each release is:

- The new release is passing all unit tests.
- The release branch is pushed and built successfully by the integration server as discussed in Section 3.2.
- SonarQube analysis carried out and code quality deemed "passing".
- Documentation comments added to each feature is built.
- A complete Docker Image is built and pushed to DockerHub.
- A working release is pushed to a staging server which can be used for demonstration purposes.

5 Technical Feasibility

To evaluate the technological stack to be used in the project it was felt that a prototype application should be built. This would give the best possible view as to how well the chosen technologies would go together. In this section we will look at each aspect of the application and the technologies used to create it. The complete prototype application was completed in two sprints (we will look at this further in Section 7.1). The prototype application consists of the following elements:

- Node.js Server Application
- Express.js API
- Mongo Database
- Docker Image + Container
- Docker Engine installed on Mac OS X & Linux 14.04

5.1 Node.js Application

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).

It is node's asynchronous nature which makes it so powerful. Being asynchronous essentially means that it does not create a new thread every time a request needs to be dealt with. Instead, it relies on a single event loop to handle requests which never blocks I/O. This can be seen in Figure 3.

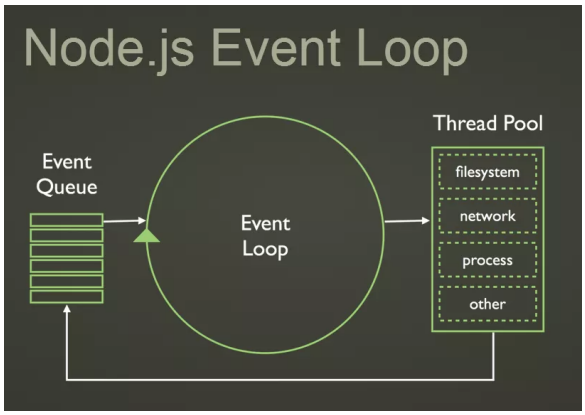

Figure 3: *Node Event Loop*

Node.js was deemed a good fit for this project as it has an excellent community and ecosystem which will increase development speed and allow for more complex features with less overhead. We can see in Figure 4 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.

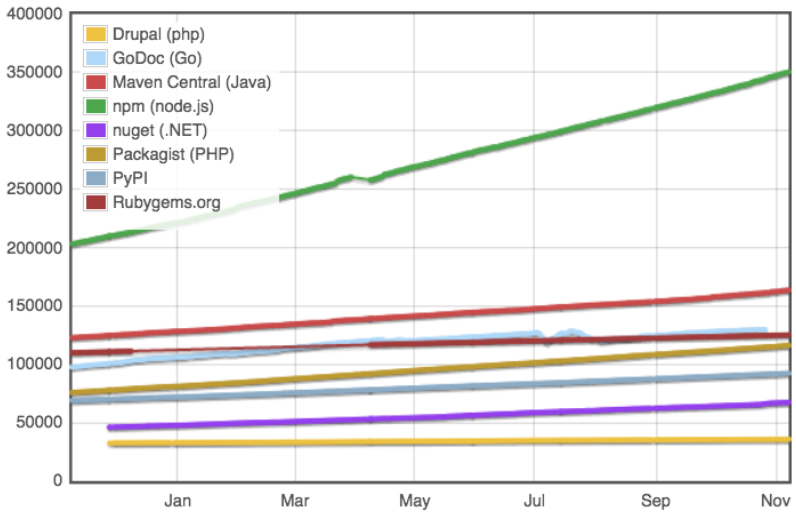

Figure 4: *Various Module Counts (Credit: modulecounts.com)*

5.2 MongoDB

MongoDB is an open source document-oriented database. It is a NoSQL database which means Mongo does not support relationships between tables (MongoDB, 2016). Instead, MongoDB uses JSON-like "documents" to store information in an ordered way. This is advantageous for a number of reasons, namely:

- Flexibility - Since Mongo does not have relations it is easy to change the database as development continues, without the need for migrations of the current schema.
- Javascript - Since the application itself will be written in Node it will be extremely beneficial to have a database which stores its information in JSON.
- Simplicity - When a database is schema-less it removes the need for complex actions such as joins, while still providing the power of complex queries.

Since we are using a javascript runtime on the server we can see that it is trivial to use connect to Mongo using Mongoose ODM in Figure 5.

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');

var Container = mongoose.model('Container', { name: String, id: String });

var built_container = new Container({ name: 'NGNIX', id: '4f5t5e3'});
built_container.save(function (err) {
  if (err) {
    console.log(err);
  } else {
    console.log('Saved!');
  }
});
```

Figure 5: *Communicating with Mongo*

5.3 Dockerode

The Node JS app is the main component of the application, and serves a way to communicate with the Docker Engine. This is done using the Docker API, which is exposed by default on any system Docker is installed on. As it is bound to a non-networked Unix socket it was necessary to communicate with this socket somehow. To communicate with this API a third party module was used, Dockerode. This module is available in Appendix B. Using this module allows the application to retrieve any information needed from the Docker API quickly and without needing to make complicated API calls to it.

Dockerode was chosen as the primary method to communicate with the Docker API using node for the following reasons:

- Active developers - new major release on average once a month
- Full test suite
- Feature rich - aims to keep all Docker API features implemented and tested
- It has a large set of contributors (49 as of November 2016) with an active issues board, meaning a greater chance of any issues encountered being resolved

An example call to the dockerode module can be seen in Figure 6, which generates the JSON response shown in Fig 7.

```
const Docker = require('dockerode');
const docker = new Docker({
  socketPath: '/var/run/docker.sock'
});

exports.listContainers = (req, res, next) => {
  docker.listContainers({'all': 1}, (err, data) => {
    if (data === null) {
      res.status(404).json({
        response: "No containers found",
        error: err
      })
    } else {
      res.status(200).json({
        response: data
      })
    }
  });
}
```

Figure 6: *Listing All Containers on a Host*

```
  {
    "response": [
      {
        "Id": "f8ea3a2c0...",
        "Names": [
          "/my_nginx_container"
        ],
        "Image": "nginx",
        "ImageID": "sha256:e43d838...",
                .
                .
                .
      }
    ]
  }
```

Figure 7: *Response From Server*

5.4 Express.js

To expose the Docker API externally a web API is needed. Upon investigation several strong choices were available and any of these would have made a suitable API. However Express.js was chosen for its simplicity and as with node, its large online community. To create a router which our API will use is a relatively straight forward process, a small sample of which can be seen in Figure 8. Express also supports many different types of authentication, which is a non-functional goal of this project.

```
let express = require('express');
let containers = controllers.containers;
let router = express.Router();

router.get('/', function(req, res, next) {
    res.send('Welcome to the Docker Lifecycle Management v1 API');
});

/* Container Routes */
router.get('/containers', containers.listRunningContainers);
router.get('/containers/all', containers.listContainers);
router.get('/containers/:id/', containers.listSpecificContainer);

module.exports = router;
```

Figure 8: *Creating an Express.js Router*

5.5 Docker

To evaluate whether or not it was feasible to put the final application in a container a sample Docker image was built locally and a container was run from this image. The file this container was built from can be seen in Figure 9.

```
FROM node:6.9.1-slim

ADD . /app
WORKDIR /app
RUN npm install
EXPOSE 3000

CMD ["node", "app.js"]
```

Figure 9: *Creating a Docker Image*

Once this container was run the complete application could be run locally from within a container.

5.6 Supporting Technologies/Processes

Vagrant

Vagrant is a development tool to provision and 'sandbox' the complete development environment from the host machine it is running on. This is very beneficial, as it means all necessary dependencies can be bundled into one virtual machine and updated or changed when needed.

Vagrant also allows the developer to standardised all of the different environments the application will run on. This means that if the code will be deployed to a specific version of a specific operating

system then the developer can easily replicate this on the local development environment. It allows the developer to build the application on one (local) environment and deploy to the *exact same* remote environment. This is extremely beneficial in terms of code reliability and stability - as the runtime can be reproduced easily.

Ansible

Ansible is a tool to provision virtual machines running locally or remotely. A formal process of all deployment and provisioning will be employed in this project. Meaning that the same route from running code locally to it running on a remote server will always be followed.

Ansible paired with Vagrant in this project will mean any discrepancies or bugs in the code will not be introduced by the environment or the deployment/provisioning process. This is very important as it can reduce the development workload.

Testing

This project will use third party node modules to perform its unit tests. These unit tests will be behavioural-driven, meaning the components will be tested individually with the end-goal of the passing test being validating a certain behaviour of the application, as opposed to validating the functionality of each individual component.

In this way, the project will follow the behavioural driven development (BDD) model. The advantage of this model over pure unit testing in test-driven development (TDD) is that BDD keeps the end result in mind at all times, instead of just the conditions for an individual unit test to pass. This allows the developer to constantly keep in mind the goal of the module being tested.

6 Design and Implementation

6.1 Front End Application

To conceptually show how the application may look a wireframe of the user-facing application was built. It will give the reader some indication of what the final application will look like. While it is only at conceptual level it does encapsulate the primary goals and how they will be implemented. It shows how images and containers can be listed and interacted with, allowing advanced actions to be performed on them. This wireframe can be seen in Figure 10.

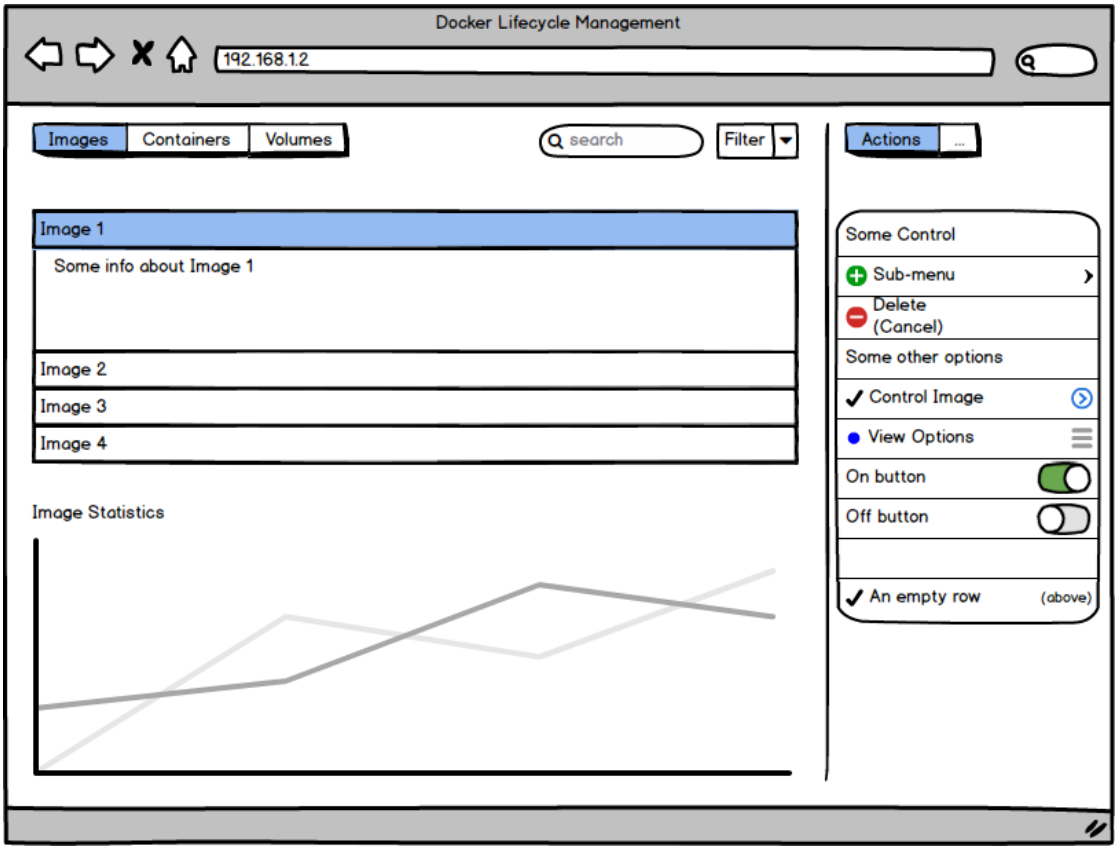

Figure 10: *Wireframe*

6.2 Class Diagram

The conceptual class diagram for the system can be seen in Figure 11. It should be noted that while all aspects of the Docker ecosystem are modelled in Figure 11 they are not all local objects. This

is highlighted by distinguishing between the local and remote systems. It is still important to show the remote system's relationship with the local system as many local actions will be affected by the remote system. In Docker, and indeed for this project, the local system can be thought of as the host on which the Docker engine is running, i.e. the host which will run containers and house images to be used to run containers. A remote system is any non-local Docker engine which the application will communicate with. For example a remote Docker registry which stores various types of Docker images is still a Docker engine and communicates with a local Docker installation as such.

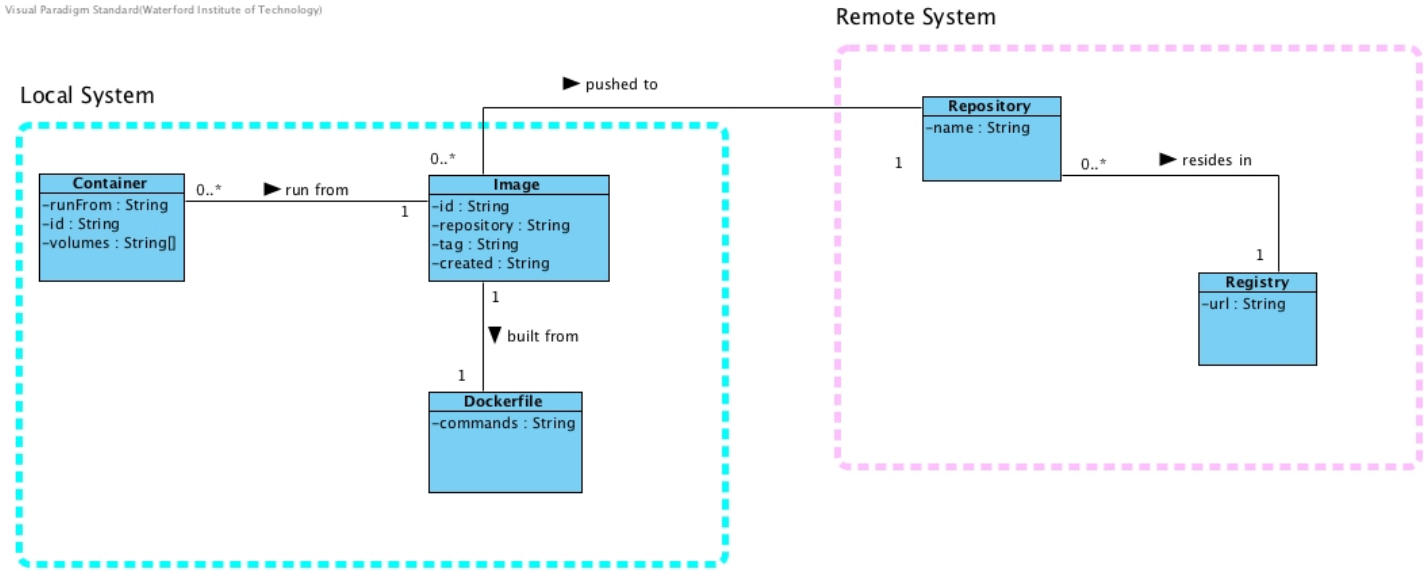

Figure 11: *Class Diagram*

Figure 11 shows that from a data-modelling perspective the system is quite small, this is due to the nature of Docker and how most data will be stored and managed by Docker itself. Since the database being used will be NoSQL the concept of relationships does not strictly apply, however they are modelled for clarity.

6.3 Sequence Diagram

To obtain a clearer understanding of how basic create, read, update and delete functions will operate throughout the various levels of the application sequence diagrams can be used. Figure 12 aims to show the sequence of events contained in these basic functions. Since there are a multitude of functions not all will be modelled using sequence diagrams, however the sequence of events for each are quite similar to that shown in Figure 12.

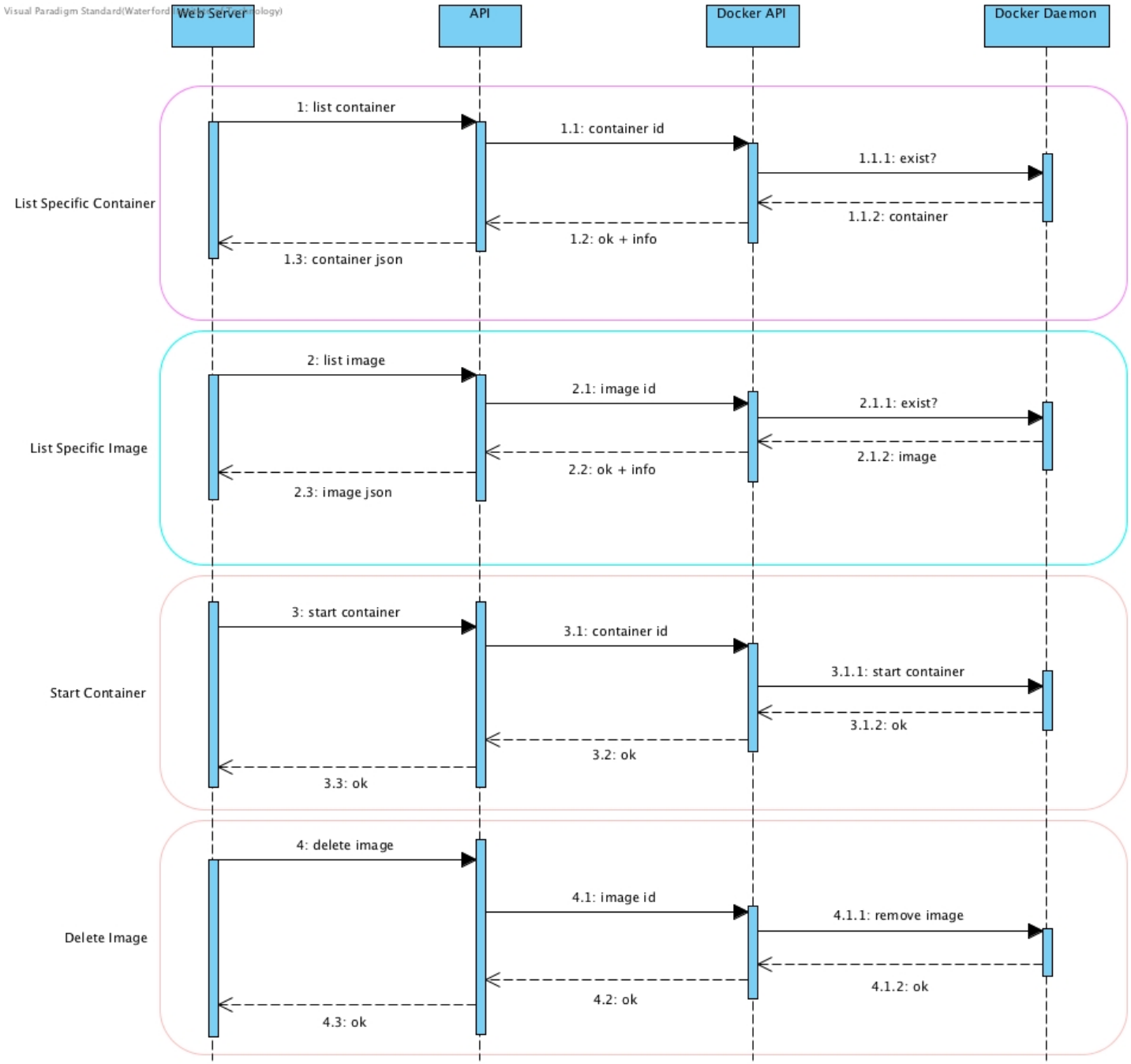

Figure 12: *Sequence Diagram*

Sequence diagrams can also be used to show the flow of the user login and registration aspect of the application since this is quite different to the Docker side. This aims to show the sequence of events

when a user tries to register with the application and then when they try to log in. This can be seen in Figure 13.

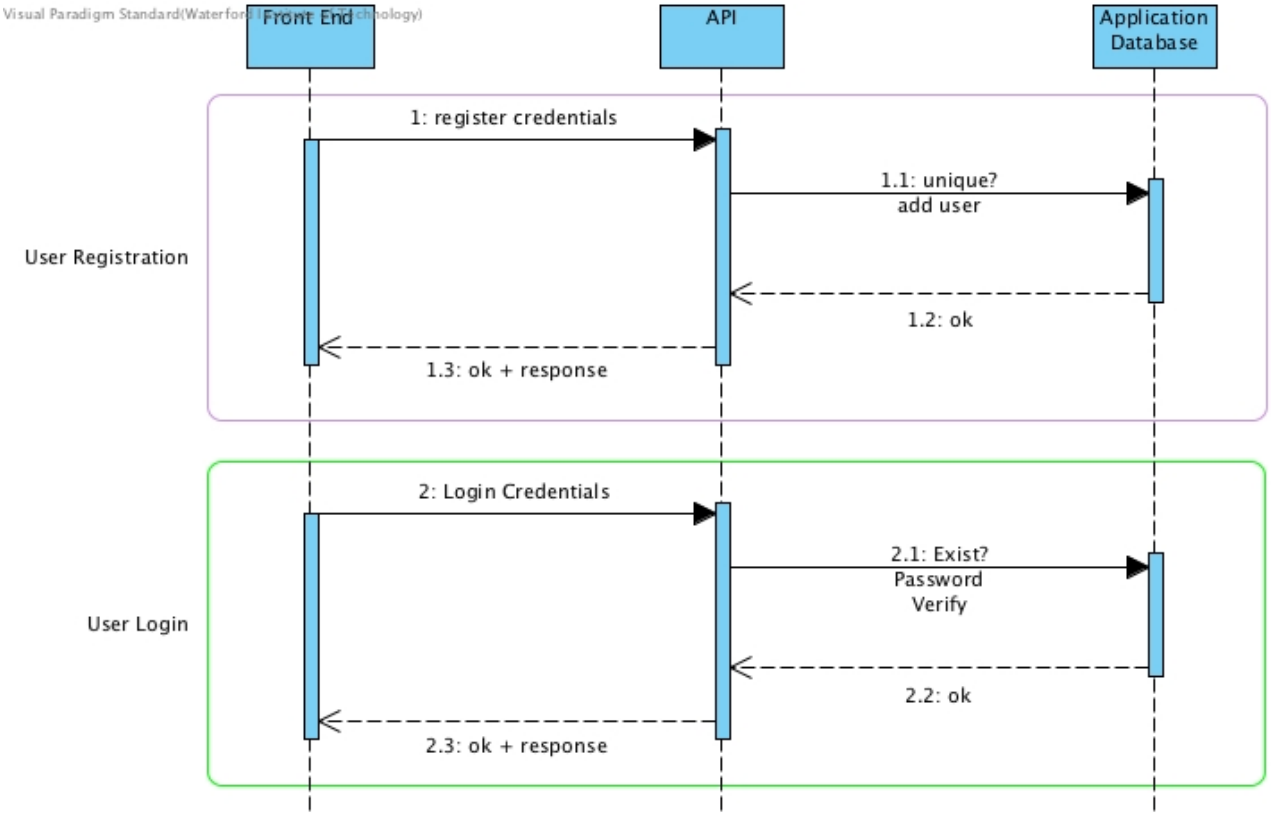

Figure 13: *User System Sequence Diagram*

6.4 Use Case Diagram

Now that we have an understanding of the components of the system and how they interact with each other to perform specific functions we can now look at a use case diagram which aims to tie together the basic use cases of the system. This can be seen in Figure 14. As before, a subset of use cases which accurately encompass other uses of the system have been chosen here, as a larger use case diagram would not be beneficial to understanding the system better.

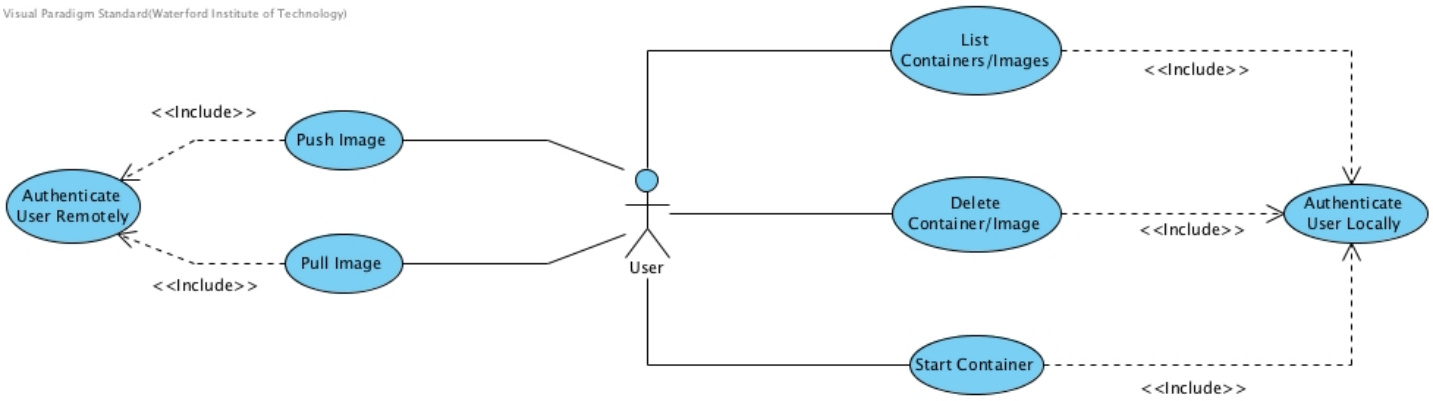

Figure 14: *Use Case Diagram*

6.5 Implementation

As previously mentioned, the project management tool which will be used in this project is provided by the product owners, Red Hat Mobile. Jira is a tool to organise projects and their associated tasks (Atlassian, 2016). It allows the developer to view all current tasks on the project and also gives useful reporting options such as burndown and gantt charts. It comes bundled with a sprint management tool, an example of which can be seen in Figure 15.

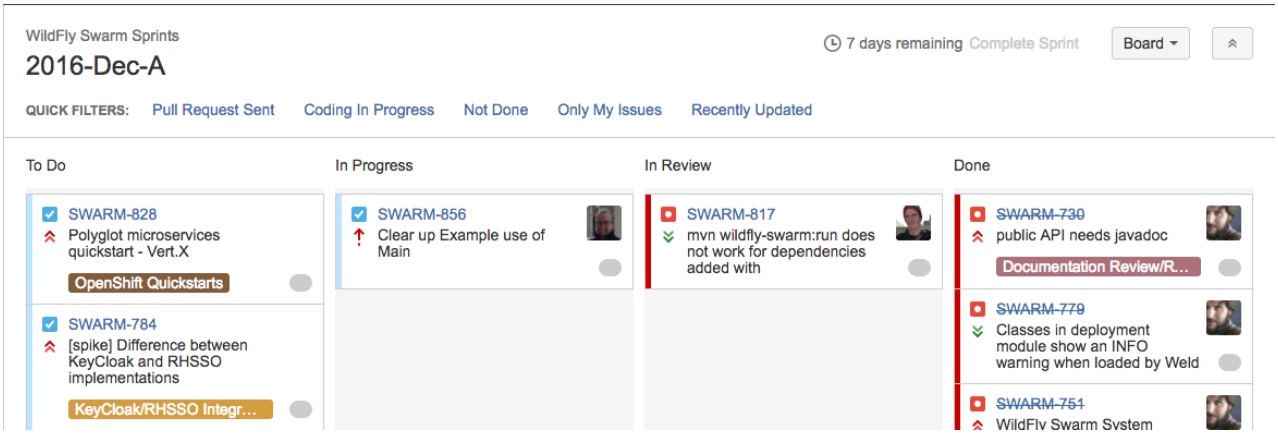

Figure 15: *Jira Sprint*

It also stores all current tasks in the project backlog. These tasks can then be pulled into each sprint as required. An example backlog can be seen in Figure 16.

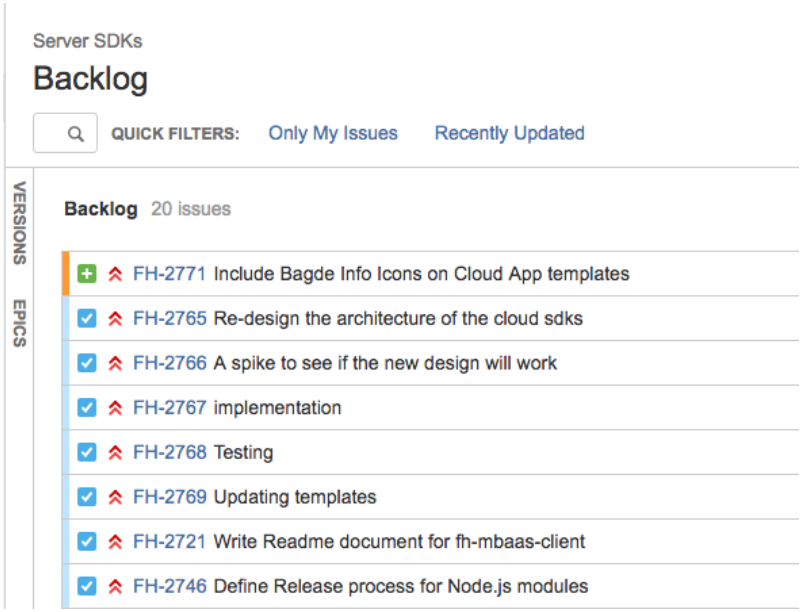

Figure 16: *Jira Backlog*

To implement the system outlined in this section, some initial tickets were detailed. These were grouped under the Agile headings of:

- Epics - which describe a large feature set of the system.
- Tickets - the sub group under which the tasks to get an epic completed can be placed
- Sub Tasks - which are the smallest units of work. When all sub tasks for a ticket are completed then that ticket is said to be completed.

To prioritise the tasks within this project each ticket was given two scores. The first was a priority score which gives an indication as to how vital this ticket is to the overall project. An indication of this can be seen in Section 2.4. The second score is an estimate of the complexity of completing said task. Together, these two scores are combined to give an overall weighting to each ticket, thereby making it easier to decide which tasks should be taken first. Tasks with a higher overall score are more likely to be worked on sooner. We will now look at the first draft of project epics and their respective tickets.

Docker Image CRUD

This section will look at the basic Docker Image tasks of the system. Completion of this epic will allow the user to have complete control over all Docker images on the system.

Epic	Ticket	Sub Task	Priority	Effort	Rank
Image CRUD	List Images	Write endpoint to return a list of all images on the host	5	1	500
	Delete Images	Write endpoint to delete an image on the host	5	1	500
	Pull Images	Write endpoint to pull an image to the host	5	2	250
	Filter Images	Write endpoint to filter images based on certain criteria	4	2	200
		Write function to manipulate images based on criteria			
	Push Images	Write endpoint to push an image to a registry	4	5	80
		Write call to docker hub endpoint allowing images to be pushed here			
		write function to authenticate with docker hub api			
	Search Docker Hub	Write endpoint to search for an image on a registry	4	3	133
		Write function to make call to docker hub api			
		Write function to download chosen image			

Docker Container Management

This section will deal with container management. When completed, a user should have total control over all containers, both running and otherwise on the system.

Epic	Ticket	Sub Task	Priority	Effort	Rank
Container Life Cycle	List all Containers	Write endpoint to return a list of all containers on the host	5	1	500
	Start Containers	Write endpoint to start a specific container on a host	5	1	500
	Stop Containers	Write endpoint to stop a specific container on a host	5	1	500
	Restart Containers	Write call to Docker API endpoint	5	1	500
		Write App endpoint			
	View a Container's log	Write App endpoint	4	2	200
		Write call to Docker API endpoint			
	Remove Containers	Write App endpoint	5	1	500
		Write call to Docker API endpoint			
	Filter Containers	Write endpoint to filter images based on certain criteria	4	2	200
		Write function to manipulate containers based on criteria			

Security Epic

Security of the system is also an epic. This will mean how the system is accessed by a user, as certain users may be accessing the system remotely. It will also include how the access occurs, i.e. will the application use SSL certificates to encrypt data flow to and from the application.

Epic	Ticket	Sub Task	Priority	Effort	Rank
Security	User Login	Create an endpoint so that a user can log in	3	5	60
		Create function so that a user can be logged in and returned a token			
		Add verification to functions requiring a user be logged in to do anything			
	Remotely Exposing Docker	Investigate method to expose docker daemon over TCP	2	3	67

Core Application Epic

This section will look at some of the broader details of the project.

Epic	Ticket	Sub Task	Priority	Effort	Rank
Core Application	Database	Create initial document database set up	4	3	133
		Create Dockerfile to run database in a container			
		Create configuration to connect node application to database			
		Create intial seed data			
	Containerise the Application	Create Dockerfile	2	3	67
		Create Compose file for all components to be started together			
	Front End	Investigate which front end framework will be used	3	8	38
		Create wireframes for the initial look of the application			
		Create login page so that a user in the database will be able to log in to the application			
		Create base dashboard page			

7 Summary

This section will try to give the reader a complete picture of the current state of the project. We will look at the total work completed so far and then what the project will entail going forward.

7.1 Work Completed

The work completed so far has been achieved in the form of 2 sprints, the first lasting 2 weeks and the second taking place over 3 weeks. A high level overview of the tasks completed in each Sprint can be seen below with the finished prototype application available in Appendix A.

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 C.
- A travis.yml file was created which details the steps to be followed when building the application. The Travis repository in Appendix C 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 D.
- A docker-compose.yml file was created to allow for quick startup of the application using Docker Compose, including any other possible containers which may be needed further into development.

After these initial 2 sprints a sprint review was carried out. This sprint review highlighted the fact that some smaller tasks such as creating the developer environment and a skeleton Express API were underestimated in the initial backlog grooming session.

A sprint retrospective was then carried out which highlighted some key performance issues. One such issue was time management, it was felt that too much time was spent on smaller tasks like incorporating SonarQube into the build pipeline.

7.2 Project Direction

At the end of the two sprints mentioned above a further backlog grooming session was carried out between the author, the product owners (Red Hat Mobile) and the scrum master (Dr. Brenda Mullally). The topic of this grooming session was to re-prioritise the backlog with the previous two sprints in mind.

After this grooming session a new and better defined backlog was available. This was broken down and the next sprint was then constructed. It was decided that this would include:

- A basic application front end to be created for graphical interaction with the application.
- Documentation tools for the application are to be investigated for suitability.
- Multiple databases are to be created to allow for persistence of data across all environments i.e. development, CI and staging.

Appendices

A Application Repository

https://github.com/StephenCoady/docker-test-app

B Dockerode

https://github.com/apocas/dockerode

C Travis Repository

https://travis-ci.org/StephenCoady/docker-test-app

D SonarQube Repository

https://sonarqube.com/dashboard/index?id=na

Bibliography

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

Ahlquist, K. (2016), 'UI For Docker'. [online] Accessed: 27/11/2016.
URL: *https://goo.gl/Y8MjxK*

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

Bowes, J. (2016), 'Agile vs Waterfall: Comparing project management methods'. [online] Accessed: 27/11/2016.
URL: *https://goo.gl/nI9dZ0*

Datadog (2016), '8 Surprising Facts About Real Docker Adoption'. [online] Accessed: 27/11/2016.
URL: *https://www.datadoghq.com/docker-adoption/*

Docker (2016*a*), 'Docker Cloud'. [online] Accessed: 27/11/2016.
URL: *https://cloud.docker.com/features/deploy/*

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

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

MongoDB (2016), 'Mongo - NoSQL Explained'. [online] Accessed: 27/11/2016.
URL: *https://www.mongodb.com/nosql-explained*

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/*