Contents

1 Project Overview

1.1 Problem

Currently, container lifecycle management is predominantly achieved through 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.

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.

1.2 Industry Example

As a real-life example, Red Hat Mobile's Application platform is made up of between 20-25 Docker images. So in a development environment that means that running all of those images gives around the same number of containers to manage. While this is no small feat, it can be mitigated by things like scripts, Docker Compose etc. However when we start running these containers in production that number 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 Continuous Integration/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.

1.3 Solution

With the aim of solving the problem described in Sections 1.1 and 1.2, 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.

1.4 Goals

In terms of goals of the project, the core or primary goals can be broken up into functional and non-function goals as follows:

1.4.1 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 1.1. 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. We will look at the reason for this in Section 2.1.

1.4.2 Non-Functional

Testing
This project will aim to deliver a product with extensive unit tests. It will follow the behavioral driven development (BDD) method which will ensure code is hardened and minimize 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 3.3.

Security
The end product will also be secure. The Docker daemon needs to run on a host with root privileges, which can have disastrous affects 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.

1.4.3 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
If this project is successful, I intend to release the source code as open source, where it can be built upon and improved by the community.

Containerized 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.

2 Investigation

2.1 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 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 tools called sprints throughout the development cycle (Agile Methodology, 2016). These are short incremental development cycles which are planned at the beginning of each sprint itself and analyzed at the end.

Each sprint in this project will have a well-defined goal, which will allow for accurate judging of the project scope and whether or not it is on track. As part of the Agile process there will be a planning session to initially scope the product backlog and to inform the total scope of the project. At the

end of the initial sprint a sprint review will then be performed which will evaluate goals achieved versus goals planned and the backlog will then be re-prioritized accordingly. A sprint retrospective will then analyze sprint performance and help to highlight areas where improvement can be made. This will also enhance the accuracy of the following sprint.

2.2 Continuous Integration

Continuous Integration is the act of continuously ensuring software is of a suitable standard to be integrated into the current software package. This will ensure that any changes made to the code base will receive immediate test-feedback. 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.

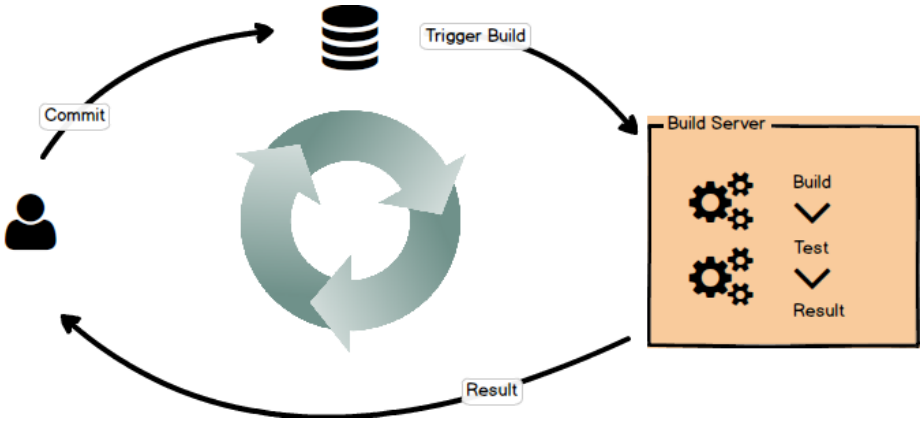

Figure 1: *Continuous Integration*

2.3 Existing Solutions

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

Lack of Features

One of these projects "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 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 I believe that 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 1.1.

Cost

Docker themselves provide a lifecycle management solution, however it comes as a monthly subscription (Docker, 2016). 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.

3 Technologies

3.1 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).

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 2.

One benefit of Node using a single event loop is that it is less CPU intensive since it does not need a new thread for every new request. Node is also non-blocking, which when combined with using a single event-loop mean it is very fast.

Some advantages of using Node for this project are:

- Large online community - vital for troubleshooting
- One of the largest online ecosystems providing a vast amount of third party modules, meaning it will not be necessary to "re-invent the wheel"
- It is Javascript so the code base will not be verbose

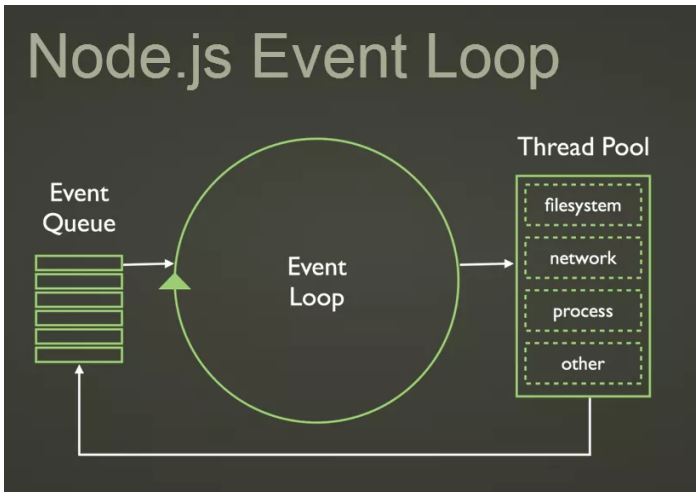

Figure 2: *Node Event Loop*

3.2 Mongo

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.

3.3 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 standardize 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 behavioral-driven, meaning the components will be tested individually with the end-goal of the passing test being validating a certain behavior of the application, as opposed to validating the functionality of each individual component.

In this way, the project will follow the behavioral 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.

4 Technical Feasibility

To evaluate the technological stack of the project a prototype application was built to communicate with the Docker API. This application is available in Appendix A. This application consisted of the following components:

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

4.1 Node.js Application

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 3 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.

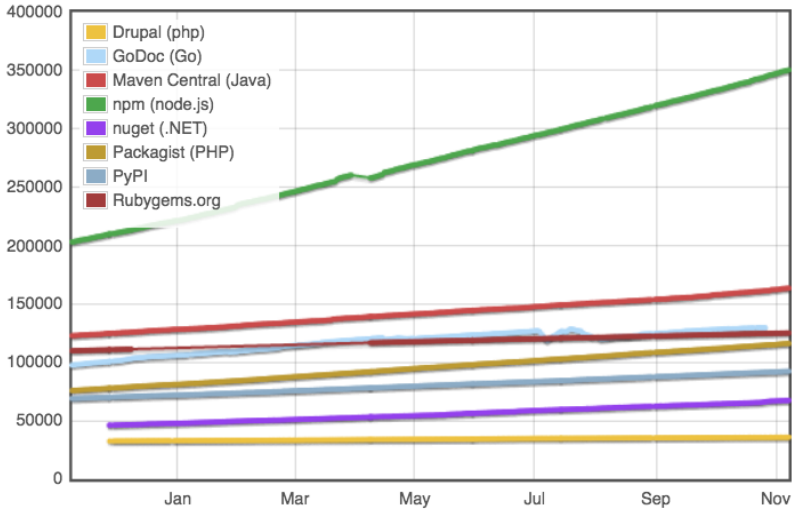

Figure 3: *Various Module Counts (Credit: modulecounts.com)*

4.2 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 Listing 1, which generates the JSON response shown in Listing 2.

```
1  const Docker = require('dockerode');
2  const docker = new Docker({
3    socketPath: '/var/run/docker.sock'
4  });
5
6  exports.listContainers = (req, res, next) => {
7    docker.listContainers({'all': 1}, (err, data) => {
8      if (data === null) {
9        res.status(404).json({
10         response: "No containers found",
11         error: err
12       })
13     } else {
14       res.status(200).json({
15         response: data
16       })
17     }
18   });
19 }
```

Listing 1: Listing All Containers on a Host

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

Listing 2: Response From Server

4.3 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 Listing 3. 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;
```

Listing 3: Creating an Express.js Router

4.4 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 Listing 4.

```
1  FROM node:6.9.1-slim
2
3  ADD . /app
4  WORKDIR /app
5  RUN npm install
6  EXPOSE 3000
7
8  CMD ["node", "app.js"]
```

Listing 4: Creating a Docker Image

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

Appendices

A Application Repository

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

B Dockerode

https://github.com/apocas/dockerode

Bibliography

Agile Methodology (2016), 'The Agile Movement'.
URL: *http://agilemethodology.org/*

Ahlquist, K. (2016), 'UI For Docker'.
URL: *https://goo.gl/Y8MjxK*

Bowes, J. (2016), 'Agile vs Waterfall: Comparing project management methods'.
URL: *https://goo.gl/nI9dZ0*

Docker (2016), 'Docker Cloud'.
URL: *https://cloud.docker.com/features/deploy/*

MongoDB (2016), 'Mongo - NoSQL Explained'.
URL: *https://www.mongodb.com/nosql-explained*

Nodejs.org (2016), 'Node.js'.
URL: *https://nodejs.org/en/*