TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science

ITI40LT

Siim Kaspar Uustalu 134295IAPB

Implementing automation in the software development lifecycle using Docker

Bachelor's thesis

Supervisor: Marko Kääramees
PhD
Assistant Professor

Tallinn 2016

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut

ITI40LT

Siim Kaspar Uustalu 134295IAPB

AUTOMAATIKA RAKENDAMINE TARKVARAARENDUSE ELUTSÜKLIS DOCKERI ABIL

Bakalaureusetöö

Juhendaja: Marko Kääramees
PhD
Dotsent

Tallinn 2016

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Siim Kaspar Uustalu

23.05.2016

Abstract

The thesis provides an implementations of automation within the software development lifecycle utilizing Docker, Vagrant and Travis CI. Automatic provisioning processes for development environment setup are provided for three common types of web software projects. In addition the process of creating release artifacts is automated utilizing the Travis continuous integration platform with a Docker based build process. As part of the build process, a method to automatically run functional tests on the release artifact within a mock infrastructure is implemented.

Processes implemented and described within the thesis follow best practices outlined by the principles of continuous delivery and have the aim of providing consistency between different deployment environments.

This thesis is written in English and is 40 pages long, including 4 chapters, 25 figures and 0 tables.

Abstract

Automaatika rakendamine tarkvaraarenduse elutsüklis Dockeri abil

Käesolev bakalaureusetöö keskendub automaatika rakendamisele tarkvaraarenduse elutsüklis *Docker*i, *Vagrant*i ning *Travis CI* tööriistade abil. Töös on toodud lahendused kolme laialt kasutatava veebirakenduse tüübi arenduskeskkonna ülesseadmise protsessi automatiseerimiseks. Vaadeldavate rakenduste hulgas on Javascript ning HTML rakendus, *RESTful* veebiteenus kirjutatud PHP keeles ning *RESTful* veebiteenus kirjutatud Go keeles. *Vagranti* abil üles seatud protsess töötab Windows, Linux ning OSX operatsioonisüsteemidel ning kasutajapoolne sekkumine ei ole vajalik. Arenduskeskkonna infrastruktuur koosneb Dockeri konteinerites sisalduvatest teenustest ning sel põhjusel analüüsitakse töös ka rakenduse konteineriseerimise protsessi.

Lisaks on autor välja pakkunud lahenduse *Docker*i abil *continuous delivery* protsessi rakendamiseks. *Travis CI* platvormil toimiv automatiseeritud protsess on modulaarne ning väheste muudatustega ka muudel *continuous integration* platvormidel rakendatav. Töös on näitlikustatud, kuidas on võimalik luua automatiseeritud *build*-protsessi käigus ajutine infrastruktuur, kuhu paigutatuna testitakse rakendust realistlikes olukordades.

Töö tulemusena leiti, et *Docker*i baasil on võimalik rakendada keeruka infrastruktuuri automaatset seadistamist. Käesolevas bakalaurusetöös autori poolt analüüsitud protsesse saab kõigi kolme töös käsitletud rakenduse tüübi puhul ilma muudatusteta kasutusele võtta ning need on laiendatavad ka teistele sarnastele projektidele.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 40 leheküljel, 4 peatükki, 25 joonist ja 0 tabelit.

List of abbreviations and terms

Bash	Bourne Again Shell — a sh-compatible shell developed by the GNU project conforming to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard.
Chakram	A REST API testing framework written in Node.js offering a behavior-driven testing style and fully exploiting promises.
Container	An isolated Linux system running on a shared kernel. In the context of this thesis the term container is used to note Docker containers.
Containerization	The process of encapsulating an application in a container to provide a lightweight alternative to virtualization.
Continuous integration	A development practice putting the focus on integrating developer changes as soon as possible.
Docker	A software project that allows developers to package an application with all of its dependencies into a standardized unit for software development. Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in.
Dockerfile	A text document containing all the commands a user could call on the command line to assemble an image.
DSL	A domain specific language. Programming language, or a subset of a programming language focused on a single problem domain.
Go	Go is a open source statically typed programming language originally developed at Google with a focus on building simple, reliable and efficient software.
HHVM	The HipHop Virtual Machine is an open-source virtual machine, using the JIT compilation approach, designed for executing programs written in Hack and PHP
HTTP	*Hypertext Transfer Protocol.* An application protocol for delivery of hypermedia resources.
Memcached	Free & open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database

	load.
MySQL	An open-source relational database management system.
Node.js	A Javascript runtime built on the Chrome V8 Javascript engine used to execute Javascript outside the browser.
Npm	A package management solution for node.js applications.
React	A JavaScript library for creating user interfaces originally built by Facebook and Instagram
Redux	A predictable state container for JavaScript apps, that helps developers write applications which behave consistently in multiple environments and are easy to test.
Vagrant	An open source command line utility for managing the lifecycle of virtual machines supporting multiple provisioning methods
Vagrantfile	A ruby syntax file declaring the configuration of a virtual machine for the Vagrant virtual machine management utility.
VM	Virtual machine
YAML	YAML Ain't Markup Language — a human friendly data serialization standard for all programming languages.

Table of Contents

List of Figures

1 Introduction

A rising number of organizations [4] are building their infrastructure on isolated Linux user-space instances, called containers, running on a shared kernel. These deployments are large scale enough to warrant attention, with Google reportedly running over 2 billion containers in 2014 [1] . The main advantage of a container based infrastructure is the lowered overhead cost of isolation compared to traditional virtualization methods [2] . Containerized applications have lower provisioning times, measured in seconds instead of minutes [2] and as such ease the process of horizontal scaling to handle higher than usual transaction volumes [2] .

The focus of this thesis is on the process of containerizing applications with Docker in order to implement a process of continuous delivery and to create consistency between development and deployment environments. Docker was chosen as the containerization engine due to its open source status, widespread industry adoption [4] and relative maturity as compared to similar open source offerings, such as CoreOS rkt.

Criterions of success must be separately established for both the containerization of applications and provisioning of development environments. The implemented development environment provisioning process must be fully automated, requiring no human interaction to create a ready instance of the application. Another key criterion is the requirement of architectural consistency between development and production environments. Extendibility must be ensured by keeping the configuration modular and with low complexity.

The criterions of automation and modularity are also applicable to the containerization of applications. Results of the containerization process must follow best practices established for continuous delivery of release artifacts. The process itself must be as portable as possible to avoid possible vendor lock-in.

Containerization solutions will be provided for three common types of web applications. Among these are a traditional static Javascript and HTML web application,

an RESTful API written in PHP and a RESTful API written in Go. Both of the web services utilize MySQL as their main data store and Memcached as an in-memory cache. The applications are varied enough to showcase different issues faced with containerization.

In addition, a solution to run functional tests on release artifacts in a mock infrastructure during continuous integration is provided. This will ensure the correct functioning of the applications before being shipped to deployment environments. Contrasted with unit testing the functional tests help ensure that components are able to operate as part of a system.

The thesis consists of 3 chapters focusing on containerization within different stages of the traditional software development lifecycle and the fourth providing a summary of the results. The first chapter is focused on automating the provisioning process for a containerized development environment using Docker and Vagrant. In the following chapter a continuous integration process to create containerized release artifacts is provided. Additionally a solution to test these release artifacts in a mock infrastructure before distribution is provided.

2 Provisioning the development environment

Each tool introduced to the development process adds additional complexity and lengthens the process of onboarding new developers. A way to mitigate the initial confusion of a new developer introduced to a project is to make the set up process unobtrusive. Automation of configuration and installation of the supporting components behind a software project (such as a database engine or a message queue) achieves this.

Within the scope of this thesis the requirements of a development environment provisioning process are defined as following:

1. The provisioning process must be fully automated and repeatable.
2. No non-application specific configuration is needed within the scope of a single project.
3. A working instance of the application must be available after provisioning.
4. The development environment must be as close as possible to the deployment environment.

Due to Docker depending on the Linux kernel, a virtual machine must be used to avoid imposing a platform restriction on developers. To ensure consistency the virtual machines must be kept identical between developers. Distributing these pre-configured virtual images could be implemented by using the built in solutions provided by virtualization applications. Doing this would mean that all projects must have a manually created virtual machine image containing the tools and configuration. This was found to be unsuitable due to the loss of change auditability in a version control system and distribution issues introduced by the rather large size of the images.

A process of automatic provisioning and configuration on top of a common virtual machine image was chosen. Vagrant was chosen as the provisioning tool due to it enabling declarative definition of a virtual machine using a Ruby based DSL (*Domain*

specific language) within a single file called a Vagrantfile. Multiple provisioning methods are supported by Vagrant such as Puppet, Chef and Ansible, but the shell script based method was chosen to reduce complexity. Virtualbox is used as the virtualization backend under Vagrant due to its cross platform availability.

Four files pertaining to the configuration of the development virtual machine will need to be added to the application codebase. Firstly, a generic Vagrantfile, contents of which are presented in Figure 1, responsible for declaring the configuration properties of the virtual machine and invoking the provisioning shell scripts. The other three are shell scripts listed in the Vagrantfile and invoked by Vagrant. Installation of the Docker daemon and container orchestration tools is performed within docker-setup.sh and requires no application specific modification. Contents of the final two, app-setup.sh and app-start.sh, depend on the application type.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
Vagrant.configure(2) do |config|
 # Based on a install of Debian Jessie x64
 config.vm.box = "debian/jessie64"
 # Private network avail. from host
 config.vm.network "private_network", ip: "192.168.33.10"
 # Sync folder from host to guest, preserving notifications
 config.vm.synced_folder "./", "/synced/app", type: "rsync"
 # Virtualization provider specific configuration
 config.vm.provider "virtualbox" do |vb|
      vb.memory = "4096" # Define available memory (in MB)
 end
 # Set up the docker engine & orchestration tool
 config.vm.provision "shell", path: "docker-setup.sh"
 # App specific first-run provisioning
 config.vm.provision "shell", path: "app-setup.sh"
 # Start up script run each time
 config.vm.provision "shell", path: "app-start.sh", run: "always"
end
```

Figure 1. Contents of the generic Vagrantfile defining the development virtual machine

Editing the code takes place outside of the virtual machine, under the host operating system. This is enabled by the multiple host-to-guest filesharing methods provided by both Virtualbox and Vagrant. In this case the rsync method is utilized to preserve

filesystem events. Requirement of rsync availability is thereby introduced and must be taken care of on non-unix platforms.

Docker container and network provisioning will be handled with the Docker compose container orchestration tool. The advantage of using a container orchestration tool over manual provisioning via shell commands is the ability to declaratively define an infrastructure consisting of linked containers in a YAML (*YAML Ain't Markup Language*) syntax file. This enables developers to have a clear overview of the infrastructure of their application.

2.1 Containerizing the Javascript application

Containerization of the client-side Javascript application written using the React library is performed with Docker. In case of the development environment, the build process, run with the Webpack build tool running on Node.js must be containerized. In this environment Webpack's embedded server is used to serve the application due to its “hot reload” capability. Figure 2 presents a HTTP request performed in the implemented architecture.

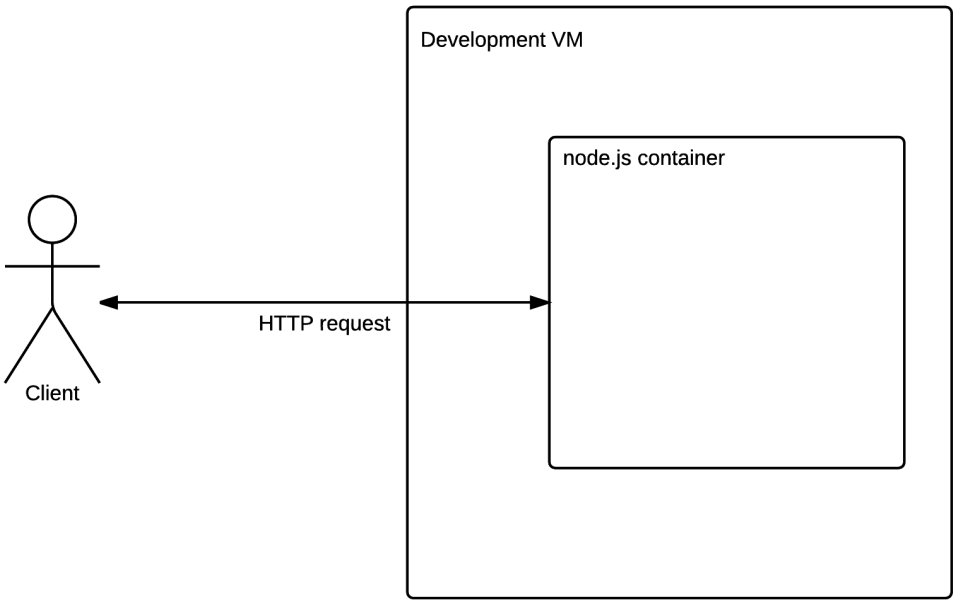

Figure 2. HTTP request against the embedded Javascript application container

Application and its static assets will be served to clients on port 8080 of the virtual machine, serviced by a Node.js container running Webpack's embedded webserver. Figure 3 presents the Dockerfile, Docker's image definition file, defining the custom image used for the Javascript application. The need for a custom image stems from the requirement for the Webpack build tool.

```
# Basing it on the official open source image
FROM node:4.4.3
MAINTAINER Siim Kaspar Uustalu <siim@mooncascade.com>

RUN npm install -g webpack

WORKDIR /app
```

Figure 3. Contents of the Dockerfile for the Javascript application development image

During application specific setup process, presented in Figure 4, the image is built from its definition. Once the image is registered in the local registry, the development dependencies required by the application are installed using the npm dependency management tool running within a temporary container created from the new image.

```
#!/bin/bash
cd /synced/app
# Build the development container image
docker-compose build
# Install dependencies required by the application
docker-compose run js-container npm install
```

Figure 4. Contents of the Javascript application specific setup script (app-setup.sh)

Application setup is run on the first vagrant up invocation or when provisioning is requested. Only the application specific start-up script in Figure 5 is run on subsequent invocations.

```
#!/bin/bash
docker-compose -f /synced/app/docker-compose.yml up -d
```

Figure 5. Contents of the application start up script (app-start.sh)

Figure 6 presents the infrastructure definition for the Javascript application. The infrastructure consists of a single container created from an image based on the Dockerfile presented in Figure 3. A volume containing the synced application source code is mounted inside the container. The container will also have its port 8080 bound to the virtual machine's external interface to enable outside access.

```
version: '2'
services:
 js-container:
  build:
   context: ./
   dockerfile: Dockerfile.development
  ports:
   - "8080:8080"
  volumes:
   - /synced/app:/app
  command: npm start
```

Figure 6. Contents of the infrastructure definition file for the Javascript application

With these files in place the automatic provisioning process for a Javascript application has been established. Invoking vagrant up within the application directory on the host machine will start the provisioning process. After provisioning and startup has completed, the Javascript application will be reachable from port 8080 of the virtual machine. Any changes made to the application source code on the host operating system will trigger a hot-reload of the application.

2.2 Containerizing the RESTful web service written in PHP

The RESTful web service communicates with clients using HTTP. Request and response bodies are JSON encoded. The service accesses MySQL and Memcached. Long term data storage is handled by the RDBMS, while results of time expensive queries are cached in the Memcached instance to provide cache expensive queries.

HHVM (*HipHop Virtual Machine*) is used to execute PHP with nginx configured as a reverse proxy (Appendix 1 – Containerized nginx in reverse proxy configuration) to handle incoming HTTP requests.

In a traditional development environment all parts would be run on the host operating system as services without any additional isolation. In case of a dockerized application each container will only contain a single process. These containerized services are then able to communicate over a network managed by Docker. The single process limitation may be worked around by utilizing a process control system, such as supervisord, but doing so is not recommended [6] .

Service discovery between these isolated services will be handled by the Docker embedded DNS server. Implementing isolated communication between the services requires creation of two networks: one for communication between nginx and HHVM containers and a second one to enable communication between the HHVM container and the data layer. Figure 7 illustrates the architecture of the service and the additional isolation provided by separated networks.

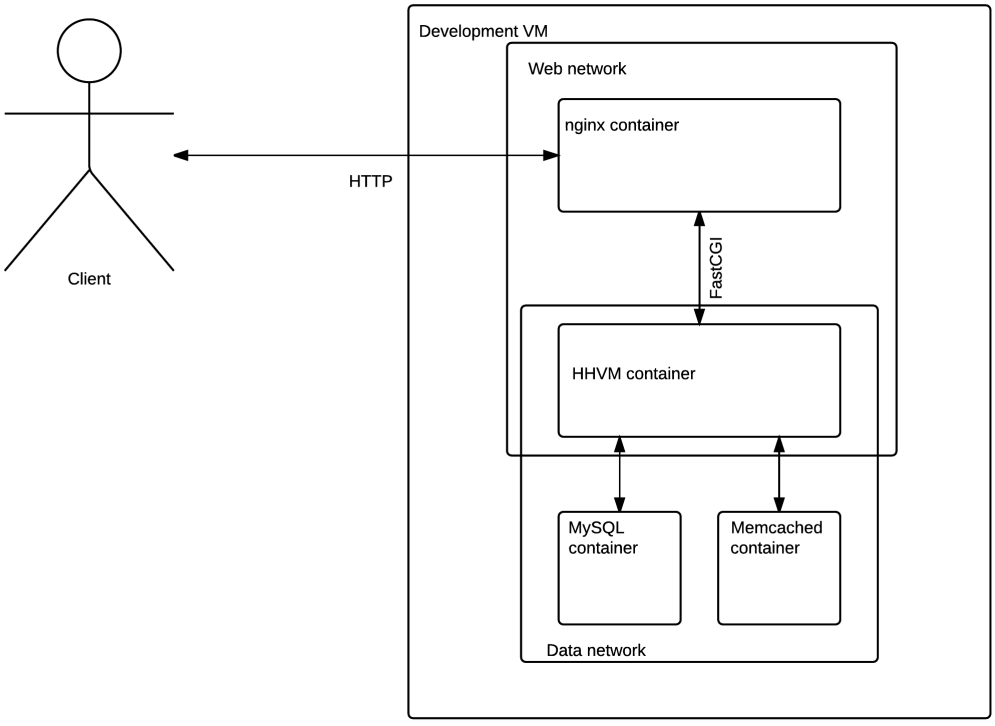

Figure 7. Client communication with the PHP API development environment

The nginx container's port 8080 is bound to the development VM's external interface. Only HTTP protocol is handled in all deployment environments as HTTPS (*HTTP over SSL*) termination will be handled by an external load balancer.

Application specific changes only need to be introduced in app-setup.sh and in the infrastructure definition. The full contents of the application setup script are displayed in Figure 8. During the application setup process the Docker images required are either built or pulled from the central registry. After that the infrastructure is started and the database schema created using the database migration tool provided in the application framework.

```
#!/bin/bash
cd /synced/app

# Build or pull the required images
docker-compose build && docker-compose pull

# Start the containers
docker-compose up -d
sleep 40
# Build the database from incremental migrations
docker-compose run hhvm hhvm --php -d
hhvm.hack.lang.look_for_typechecker=0 artisan migrate
```

Figure 8. Contents of the application specific setup script (app-setup.sh) for the PHP project

Figure 9 presents the implementation of the proposed architecture as an infrastructure definition. Additional runtime configuration, possibly containing sensitive credentials, is applied by using environment variables. In the infrastructure definition file these are passed to the containers by using the env_file attribute. Contents of these files are in the form of NAME_OF_VALUE=value on isolated lines. These are then set as environment variables inside the container. Associating configuration with the environment in such a way is considered a best practice [7] .

```
version: '2'
services:
 nginx:
  build: ./infra/nginx
  ports:
   - "8080:8888"
  networks:
   web:
    aliases:
     - nginx
 hhvm:
  build: ./infra/hhvm-dev
  env_file:
   - ./env/application.env
  # Make project source code available as a volume
  volumes:
   - ./app:/var/www/app
  networks:
   web:
    aliases:
     - hhvm
   data:
 mysql:
  image: mysql:5.5.44
  volumes:
   - /volumes/mysql:/var/lib/mysql
  env_file:
   - ./env/mysql.env
  networks:
   data:
    aliases:
     - mysql
 memcached:
  image: memcached:latest
  networks:
   data:
    aliases:
            - memcached
networks:
      web:
      data:
```

Figure 9. Contents of the infrastructure definition file for the web service written in PHP

With the modifications in place the PHP web service is automatically set up and providing information on port 8080 of the virtual machine without any developer intervention. The created infrastructure consists of multiple containers each providing a single service. Communication between them happens over Docker controlled networks, providing isolation between the data and web layers.

2.3 Containerizing the RESTful web service written in Go

Go is a statically typed systems programming language from Google introduced in 2009. Applications written in this language can be compiled into a single, statically linked binary ready for redistribution. Static compilation makes it possible to produce comparatively small Docker images by removing the requirement for a more fully featured Linux environment inside the container.

The web service communicates with clients over HTTP. Request and response bodies are JSON encoded. In addition to service resources, an embedded web server will serve HTML documentation covering the service. MySQL will be used as the primary data store with resource expensive queries cached in Memcached. Figure 10 presents the development environment architecture.

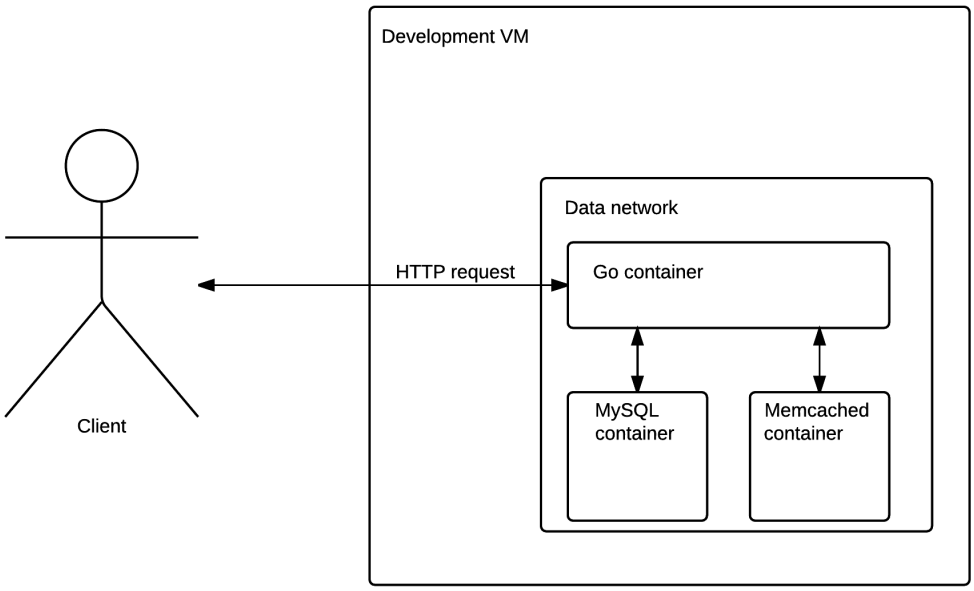

Figure 10. HTTP request against the Go web service development environment

The service setup process is presented in Figure 11. During setup, all of the images are either built or pulled from the central registry. Project dependencies are then installed by running the ephemeral container. Database schema is again managed with a migration tool. Compared to the PHP web service, the migration tool is not related to the application framework and as such containerized separately. In this case a shell script wrapper has been created to create ephemeral containers to run the migrations process.

```
#!/bin/bash
cd /synced/app
# Pull & build the images
docker-compose build && docker-compose pull
# Load the dependencies
docker-compose run go-container make dependencies
# Bring things temporarily up
docker-compose up -d
sleep 40
# Run incremental database migrations with an external tool after
building image for it
make migrator-image
./migrate.sh up
```

Figure 11. Contents of the application specific setup file for the web service written in Go

During development a Docker container containing a Go development environment is used. Project source code is made available and program compilation performed during container startup. Updating the binary then requires restarting the container.

The web service is capable of servicing HTTP requests by itself and as such the nginx container has been removed along with the web network. Only supporting insecure HTTP is accepted as production environments will have an external load balancer responsible for SSL termination. A Docker network is created to facilitate communication between the service and data layer. Figure 12 presents the Go service container definition within the infrastructure definition file. Definitions for the accompanying containers have been omitted for brevity.

```
version: '2'
services:
 go-container:
  build:
   context: ./
   dockerfile: Dockerfile.development
  env_file:
   - ./application.env
  volumes:
   - ./:/go/src/github.com/organization/app
   - /gopath:/go
   - ./doc:/doc
  ports:
   - "8080:8080"
  working_dir: /go/src/github.com/organization/app
  command: /bin/bash -c "go build -v . && ./app server"
  networks:
   data:
# Omitted for brevity
networks:
 data:
```

Figure 12. Definition of the Go service's container

The infrastructure for the Go application created by the provisioning is rather similar to the one required by the PHP web service. The process requires no manual user interaction and is applicable to other projects of similar nature with no modification.

3 Continuous delivery of release artifacts

Deployment may often be seen as something tedious and to be done rarely. This fear stems from a lack of automation bringing a greater chance of human error and waste of time [4] . Reliance on a manual process puts the burden of deployment on a single person or a team [4] due to the need for systems access.

Before automation can be implemented, the distribution method and the release artifact must be defined. In case of Docker the release artifact is an image containing the packaged application, which is distributed using the mechanisms built into Docker itself. The specifics of setting up a production environment based on Docker falls out of the scope of this thesis, but an implementation of continuous delivery for release artifacts along with configuration is provided.

In order to outline a process for continuous delivery of these release artifacts a few guiding principles must be defined.

- **Accountability –** Developers and operators must be able to identify and audit the exact version and configuration of software deployed to any instance.
- **Automation** – Creation of the release artifact should require no explicit actions from a dedicated build engineer.
- **Configurability –** Environment specific configuration should either be included within the release artifact or made available to the deployment target by other means.
- **Uniformity –** Each environment must be consistent with its peers.
- **Variant management** – Multiple versions of the same software with variable amounts of stability should be available.

3.1 Defining the build process

Defining a build process which is able to abide all the established guiding principles is possible with Docker. Uniformity is achieved by utilizing Docker images. Automating the production of these images in an accountable fashion requires the development of organizational tooling and processes.

Accountability and variant management require the ability to individually identify every release artifact deployed to any environment. A tagging system utilizing the image tagging feature of the Docker registry is used. Each tag consists of two parts — a short identifier of the deployment environment and a commit identifier within the VCS (*Version Control System)* used. The tags will be applied to Docker images before making them available from the remote registry. This enables project members to verify issues with the exact version of the deployed artifact by examining it specifically.

In the scope of this thesis the tag format used is **prod/dev-#######**. The prod identifier identifying releases meant to be deployed to the more stable production environment and dev those deployed to staging. Specific implementations of source code management flows focusing on variant management do not fall under the scope of this thesis. Assumption has been made that the projects discussed within the thesis have implemented a workflow based on the Github Flow (Appendix 2 – Github Flow based workflow) to manage different variants of their codebase.

Avoiding the human element requires introduction of a build automation tool. A wide variety of tools is available on the market, both proprietary and open source. In the scope of this thesis the Travis CI (*Continuous Integration*) platform will be used to perform builds. Travis was chosen over Jenkins due to its integration with Github, ease of configuration and support for more deployment targets than CircleCI. Although due to the nature of Docker the build process outlined may be implemented on any of the alternative CI platforms supporting Docker with minimal modification.

3.2 Producing the release artifact

The basis for each of the release artifacts will be a Dockerfile defining the contents of the resulting image. The Dockerfile for each of the application types is different, but re-

usable within similar types of projects. A key metric to minimize is the size of the resulting Docker images.

The Travis build process is configured with a YAML syntax .travis.yml file available in the project's source code repository. Commands listed will be run at defined points in the build lifecycle [3] . Re-usability of the CI configuration file is achieved by containing the build logic in external BASH scripts. Figure 13 presents the contents of the configuration file utilized.

```
sudo: required
language: node_js
branches:
 only:
  - master
  - develop
services:
- docker
# Steps to be run in order
install: . build/before_build.sh
script: . build/build.sh
before_deploy: . build/before_deploy.sh
deploy:
 # Deploy step instructions removed for brevity
```

Figure 13. Contents of the Travis CI configuration file (travis-ci.yml)

Any configuration settings must be specified in environment variables. The name of the variable must be suffixed with an environment identifier to avoid confusion. Any potentially confidential information stored in these variables (such as service credentials) can be protected by using features offered by Travis.

The configuration consists of a generic .travis.yml file accompanied by 5 BASH scripts and the deployment Dockerfile in a directory called build. The first, before_build.sh script is responsible for installing any dependencies and preparing the build environment. Second, build.sh is a wrapper script for the actual application specific build and functional testing scripts, build_artifact.sh and test_artifact.sh. Functional testing is discussed further in the chapter on Testing the release artifact in a mock infrastructure. Final script, before_deploy.sh is rather generic and only includes steps

made before the artifact can be deployed. In the scope of application types discussed in the thesis this would mean pushing the image to the appropriate remote Docker registry.

3.2.1 The Javascript application

In a production context the webpack internal development server is unsuitable. Instead a distributable application bundle is produced from the source code. The bundle will contain a minimized Javascript file and static assets required by the application. Any HTTP capable web server can then be used to serve these assets. Nginx (Appendix 3 – Containerized nginx serving static assets) was chosen in this case due to both its performance characteristics and developer familiarity. To minimize the size of the distributed release image the minimal Alpine Linux base image was chosen. Figure 14 presents the Dockerfile for the production image. Architecturally this does not introduce a large change as only the underlying Javascript delivery mechanism was changed.

```
FROM alpine:3.2
MAINTAINER Siim Kaspar Uustalu <siim@mooncascade.com>

# Install nginx stable
RUN apk add --update nginx && rm -rf /var/cache/apk/*
# Bake in nginx configuration
ADD build/config/nginx.conf /etc/nginx/nginx.conf
# & the single page app itself
ADD index.html /var/www/app/index.html
ADD favicon.ico /var/www/app/favicon.ico
ADD static /var/www/app/static
#Nginx will be exposed on 8080
EXPOSE 8080
WORKDIR /tmp # Working directory
CMD nginx # Command run on container start
```

Figure 14. Production ready Dockerfile for the Javascript application

The generic Travis configuration in Figure 13. Contents of the Travis CI configuration file (travis-ci.yml) is used without any modification. During the before_build phase application dependencies will be installed using npm and the rest of the environment

will be prepared for build. The appropriate target environment is decided from the code branch within the VCS. Figure 15 presents the environment determination and dependency installation.

```
# Determine target environment from code branch
case $TRAVIS_BRANCH in
      "master")
            export ENV_LONG="production"
            export ENV="prod" ;;
      "develop")
            export ENV_LONG="development"
            export ENV="dev" ;;
      *)
            export ENV_LONG="development"
            export ENV="dev" ;;
esac

export ENV_UPPER=$(echo "$ENV" | tr '[:lower:]' '[:upper:]')

# APP TYPE SPECIFIC:
# Install the build tool & and application dependencies
npm install -g webpack && npm install
```

Figure 15. Contents of the before_build.sh file for the Javascript application

During the build process both the applications Javascript and asset bundle with the containing release artifact will be created. The resulting Docker image will be tagged with an identifying tag in the local Docker registry. The resulting tag will also be placed in the deployment definition file Dockerrun.aws.json. Configuring the webpack build tool to properly produce the Javascript application bundle falls outside of the scope of this thesis and as such the configuration files used will be not be included. The release artifact creation is presented in Figure 16.

```
## APP SPECIFIC:
## Building the bundle
# Replace placeholder with real URL to handle forwarding within nginx
if [ $ENV = "prod" ]
then
      sed -i -e "s/replace_url_in_build/${HOST_PROD}/g"
build/config/nginx.conf
else
      sed -i -e "s/replace_url_in_build/${HOST_DEV}/g"
build/config/nginx.conf
fi

# Build the correct bundle
CONFIG_FILE="webpack.${ENV_LONG}.config.js"
NODE_ENV="production" webpack -p --config "${CONFIG_FILE}"

##
## Generic & re-usable
##
# Create tag from VCS and environment information
COMMIT_SHA1=`git rev-parse --verify --short HEAD`
export IMAGE_TAG="${ENV}-${COMMIT_SHA1}"
export IMAGE_FULL_TAG="{$ORG}/${PROJECT}:${IMAGE_TAG}"

# Build the docker image
docker build -t "${IMAGE_FULL_TAG}" .

# Place correct tag in deployment declaration
sed -i -e "s/latest/${IMAGE_TAG}/g" Dockerrun.aws.json
```

Figure 16. Contents of build_artifact.sh script to create the release artifact of the Javascript application

As the resulting release artifact contains only static HTML and assets no further testing will performed on it during the build. No changes will need to be introduced to the before_deploy.sh script either and as such after the container has been pushed to the remote registry the deployment process may be started.

3.2.2 The RESTful web service written in PHP

The process for release artifact creation process aside from the Dockerfile remains mostly the same. No major changes will be introduced to the deployment infrastructure either compared to the one used within the local development environment.

The release image is based on the HHVM Dockerfile used in the development environment with minimal changes. Differences between the Dockerfiles are listed in Figure 17. Rather than mounting the source code from the container host, the image now contains the application source code. Dependencies will only be installed within the container during the artifact build and as such the dependecy installation step in the pre-build step is removed. Doing everything in the image being built means that no application specific changes need to be applied to build_artifact.sh.

```
13a14
> COPY ./app /var/www/app
15d15
< VOLUME /var/www/app
16a17
> RUN composer install --prefer-source –optimize-autoloader
```

Figure 17. Diff of the changes made to the HHVM Dockerfile used in the dev. environment

Environment specific configuration will be applied by using environment variables. Using Docker to manage these is rather trivial as demonstrated in the previous chapter. During the pre-deployment step configuration values will be injected into the deployment description file called Dockerrun.aws.json replacing placeholders. Figure 18 presents the contents of the variable replacement script. Adding additional configuration parameters requires them to be listed in the presented script.

```
#!/bin/bash
# Environment specific variables
VARIABLES="APP_DEBUG APP_ENV APP_KEY DB_USER DB_PASS MEMCACHED_HOST"
VARIABLES="$VARIABLES DB_HOST"

# Set environment specific variables under generic name
for variable in $VARIABLES
do
      TEMP_VARIABLE="${variable}_${ENV_UPPER}"
      sed -i -e "s/{$variable}/${!TEMP_VARIABLE}/g" Dockerrun.aws.json
done

docker login -e $DOCKER_EMAIL -u $DOCKER_USERNAME -p $DOCKER_PASSWORD
docker push $IMAGE_FULL_TAG
```

Figure 18. Contents of the before_deploy.sh BASH script for the PHP application

Producing the release artifact for the PHP web service required more complex modifications to be applied to the deployment description file due to configuration requirements. Overall, the Dockerfile describing the release artifact required minimal modification from the one used within local development environments.

3.2.3 The RESTful web service written in Go

Size of the resulting release artifact must be taken into consideration, when developing the build process. The image containing the Go toolset is measured in the hundreds of megabytes and as such it is necessary to produce the service binary beforehand. Image size can be further lowered by removing the need for OS libraries by producing a statically linked binary (Appendix 4 – Producing statically linked binaries with Go). Figure 19 presents the minimal Dockerfile to be used with the statically linked binary.

```
FROM scratch
MAINTAINER Siim Kaspar Uustalu <siim@mooncascade.com>
ADD app /
ADD doc /doc
ENTRYPOINT ["/app"]
```

Figure 19. Contents of the Dockerfile for the image containing a static Go binary

Overall, rest of the build process remains the same in regards to the injection of configuration settings in environment variables and making the image available for deployment.

3.3 Testing the release artifact in a mock infrastructure

One of the requirements of establishing a dependable continuous delivery process is the existence of a comprehensive test suite [5] . While unit testing helps verify that individual pieces of an application work as intended, issues introduced by a combination of components may be missed. More realistic functional tests performed against an actual running instance may catch errors missed by other methods of testing.

Performing these tests requires that a realistic mock of the production infrastructure to be created. For each test run, all services are provisioned similarly to a production deployment and in the observed case a test suite utilizing Chakram is run against the release artifact. Encountering any errors during the testing process cancels the in-progress build.

The mock infrastructure used within the tests is provisioned with the Docker compose orchestration tool in the same way as in development environments. Any third party services utilized during these tests should be emulated if possible to ensure repeatability of the tests. Services can be emulated by containerizing already existing emulation software such as s3rver for S3 or a generic one such as WireMock. Additionally, emulation enables control over service responses to produce and verify handling of edge case scenarios. Although a problem may arise when communication with these services occurs over secured protocols. For example, to test the web service written in Go, an additional service which emulates Amazon S3 is provisioned. Communicating with this emulated service requires that the protocol be changed to insecure HTTP or certificate validation errors be ignored.

Services may not be ready to answer requests after starting the container. In order to determine whether or not different infrastructure services have started accepting connections and are in a state where the test can be started, netcat running in port

scanning mode is used in an ephemeral docker container connected to the appropriate network.

Once the infrastructure is in a state to begin testing the system, the database schema is created. Some of the tests may depend on specific data existing within the database which can not be inserted by utilizing the application under test. Overcoming this limitation introduces a requirement to access the database. Database schema creation is handled in an incremental way with migrations, this means that temporary additional migrations may be created to perform data insertion or other database manipulation required to achieve test conditions. A new migration script is created from the test specific SQL scripts and then made available to migration software.

A container is created from a image containing the Mocha test framework and Chakram (Appendix 5 – Containerized Mocha test runner) and connected to the network containing the service under test. Code for the tests is mounted to a volume on the container. Figure 20 presents the command used to invoke the test runner container during the testing process. Running the test suite produces a testing report detailing test status. In addition any test failures will be indicated with the proper exit code.

```
docker run \
            --rm \
            -v `pwd`/testing/suite:/tmp/testsuite \
            --net test
            mocha-test-runner \
            /tmp/testsuite \
            --ui bdd \
            --recursive \
      && RETURN_VALUE=0
```

Figure 20. Invoking the Mocha framework test runner

Put together the implementation allows automation of functional tests on Docker images before the distribution step. The established implementation can also be run in the development environment to check for regressions introduced by locally made changes. The actual testing method used can be swapped out with analogous methods. Performing these automated tests provides the development team with a degree of confidence that the application works as intended.

4 Summary

Aim of the thesis was to investigate using containerization to create an automated process for continuous delivery of release artifacts and provisioning of local environments. Functional testing of release artifacts was also investigated as part of the continuous delivery process. Solutions were provided for three common types of web applications.

Provisioning the development environment was implemented using a combination of Vagrant, Docker compose and provisioning shell scripts. Resulting provisioning process is generic enough to apply to projects using the same technology stack. Following the examples implemented within this thesis the work can be extended to other types of applications.

Further, the creation of release artifacts was automated using a container based build process. Although the implementation was done using Travis CI, the non vendor specific configuration is applicable to all environments where Docker is available. Build steps were implemented using containers to ensure reproducibility. Environment specific configuration for the containers was implemented using environment variables set by Docker.

Additionally a way to automatically test the release artifacts resulting from the build process was implemented. The testing portion was implemented by using the Mocha testing framework with Chakram. Replacing external services for testing purposes using Docker was investigated and demonstrated with the example of Amazon S3.

Docker and its orchestration tools were found to be suitable for the purpose of managing service infrastructure in small scale (development) deployments. Building the automation implementation on top of Docker ensured compatability with production deployments in every stage of the software development lifecycle.

References

[1] Beda, J., Containers At Scale — 2014 [WWW] https://speakerdeck.com/jbeda/containers-at-scale (01.03.2016)

[2] Coggin, M., Craven, K., Fernandes J., Herrmann, L., Juengst, D., Melia, I., Puri, S., Owens, K., Thirumalai, K., Yellumahanti, S., Linux Containers: Why They're in Your Future and What Has to Happen First — 2014 [WWW] http://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/openstack-at-cisco/linux-containers-white-paper-cisco-red-hat.pdf (28.02.2016)

[3] Customizing the Build — *Travis CI documentation* [WWW] https://docs.travis-ci.com/user/customizing-the-build/ (09.05.2016)

[4] Devops.com, ClusterHQ, The current state of container usage: identifying and eliminating barriers to adoption — 2015 [WWW] https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2015.pdf (28.02.2016)

[5] Farley, D., Humble, J., Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation : Pearson Education, 2010.

[6] Turnbull, J., The Docker Book: Containerization is the new virtualization : James Turnbull, 2014.

[7] Wiggins, A. (2012), Config — *The Twelve-Factor App* [WWW] http://12factor.net/config (15.05.2016)

Appendix 1 – Containerized nginx in reverse proxy configuration

Presented in Figure 21 is the nginx configuration file used within the containerized nginx instance in the development VM.

```
daemon off;

http {
  log_format bodylog '$remote_addr - $remote_user [$time_local] '
     '"$request" $status $body_bytes_sent '
     '"$http_referer" "$http_user_agent" $request_time '
     '<"$request_body" >"';
  sendfile on;

  server {
        listen 8080;
        access_log /dev/stdout;
        error_log /dev/stdout;
        root /var/www/app/public;
        location / {
            try_files $uri $uri/ /index.php?$query_string;
        }
        location ~ \.(hh|php)$ {
          access_log /dev/stdout bodylog;
          fastcgi_keep_conn on;
          fastcgi_pass  hhvm:9000;
          fastcgi_index index.php;
          fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
          include    fastcgi_params;
        }
      }
}
```

Figure 21. nginx configured as a reverse proxy for PHP application

Appendix 2 – Github Flow based workflow

The workflow used within the projects discussed in the thesis are using the git version control system. Due to multiple developers working on the project the Github Flow was chosen as a base for the workflow due to its lightweight nature and focus on peer review.

Similarly to the Github Flow, the master branch is used to represent the state of the production system. Additional branches are added which represent different deployment targets. In the projects discussed in the thesis, the master and develop branch were used to represent the code deployed to production and staging environments. This flow makes heavy use of pull requests and peer reviews, ensuring that everyone working on the project has a view of the larger picture.

After receiving a new issue to work on within the internal issue tracking system, each developer creates a new feature branch which will contain the work. After completing the issue, a pull request is opened against the develop branch and reviewed by a peer. This code review focuses on both ensuring the correct implementation of the issue at hand and any code standards agreed upon within the team.

Once the code has passed review and is merged, a new release artifact is produced by the CI platform and deployed to the staging environment for verifaction by QA and project management. Once it has been tested within the staging environment and no problems found the changes are deployed to production by opening up a pull request against the master branch.

Appendix 3 – Containerized nginx serving static assets

Although the container only provides its assets over HTTP, a redirect has been added to forward any user requesting the application over insecure HTTP to HTTPS according to the X-Forwarded-Proto header appended by the user facing load balancer. Figure 22 presents the nginx configuration used.

```
daemon off;
worker_processes auto;
events {
      worker_connections 4096;
}
user nginx;
http {
      sendfile on;
      server {
            listen 8080;
            if ($http_x_forwarded_proto = 'http') {
               return 301 https://replace_url_in_build$request_uri;
            }
            include mime.types;
            access_log  /dev/stdout;
            error_log   /dev/stdout;
            root /var/www/app;
            index index.html;
            location / {
                  try_files $uri /index.html;
            }
      }
}
```

Figure 22. nginx configuration to serve a static Single Page Javascript application

Appendix 4 – Producing statically linked binaries with Go

Creating truly statically linked binaries with Go may be achieved by either disabling cgo or performing static linking against glibc. The former is unsuitable in this case as disabling cgo loses the ability to interface with C code. Performing a static link against against glibc may introduce issues with licensing.

Due to these issues, the statically linked binaries are compiled under Alpine Linux using an alternative implementation of the libc called musl. An Alpine Linux based Go Docker image distribution exists but requires customization due to our requirement to perform a static link against musl libc. A custom Docker image is created for the purpose based on the Dockerfile presented in Figure 23.

```
FROM golang:1.5.3-alpine
MAINTAINER Siim Kaspar Uustalu <siim@mooncascade.com>

RUN apk add --update gcc git musl-dev
# Fetch dependencies with Go get and build binary passing linkerflags
CMD go get && go build -v -ldflags '-extldflags "-static"'
```

Figure 23. Contents of Dockerfile for Go language image

Mounting the source code to the image and setting the proper working directory will result in a static binary being built on running the container. Figure 24 presents the compiler invocation.

```
docker run --rm \
      -v gopath/on/host:/gopath/in/container \
      -w /gopath/in/container/application \
      golang-builder
```

Figure 24. Running the container to produce a statically linked binary

Appendix 5 – Containerized Mocha test runner

Chakram, written in Node.js, is used to perform automated test queries and analyse the results in order to perform functional testing of RESTful web services. In order to make the build process more portable containerization is required. A minimal Node.js image will be created containing both the Mocha test framework and Chakram. The test suite will be made available to the runner by mounting to a Docker volume on the created container. Figure 25 presents the contents of the Dockerfile for the Mocha testrunner image.

```
FROM mhart/alpine-node:4.2.6
MAINTAINER Siim Kaspar Uustalu <siim@mooncascade.com>

RUN npm set progress=false \
      && npm install -g mocha chakram \
      && npm cache clean

ENV NODE_PATH /usr/lib/node_modules
VOLUME /tmp/testsuite

WORKDIR /tmp/testsuite

ENTRYPOINT ["mocha"]

CMD ["/tmp/testsuite"]
```

Figure 25. Contents of Dockerfile for containerized Mocha test runner