

# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ  
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INTELLIGENT SYSTEMS

## SOFTWARE VERSIONS MANAGEMENT IN CONTAINERS DEPLOYMENT

BAKALÁŘSKÁ PRÁCE

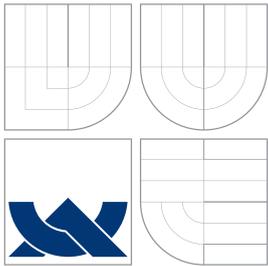
BACHELOR'S THESIS

AUTOR PRÁCE

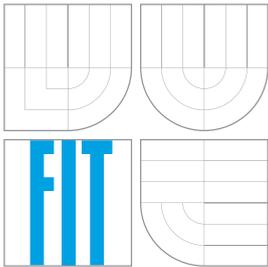
AUTHOR

ADAM RŮŽIČKA

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ  
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ  
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INTELLIGENT SYSTEMS

# VYUŽITÍ SPRÁVY VERZE SOFTWARE V LINUXOVÝCH KONTEJNERECH

SOFTWARE VERSIONS MANAGEMENT

IN CONTAINERS DEPLOYMENT

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

ADAM RŮŽIČKA

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2015

## Abstrakt

Cílem této práce je analyzovat požadavky a navrhnout řešení pro využití projektu s otevřeným zdrojovým kódem Katello pro správu softwaru nainstalovaného na obrazech aplikace Docker. Nejprve je diskutováno srovnání virtualizačních technik a následně také různých implementací kontejnerové virtualizace na rozdílných platformách. Následně jsou popsány možnosti využití projektu Katello pro správu verzí softwarových balíčků RPM a návrh a implementace samotného systému pro správu obrazů aplikace Docker. Funkcionalita aplikace byla experimentálně testována a ověřena a byly navrženy možnosti pro další rozšíření.

## Abstract

The aim of this thesis is to analyse requirements and design a solution for using the open-source project Katello to manage software present on Docker images. In this thesis there are introduced virtualization technologies with emphasis on container virtualization and also projects Foreman and Katello and their usage for content management of RPM software packages. Furthermore, it deals with design and implementation of extension for project Katello allowing managing Docker images. The functionality of the application was experimentally tested and verified and further possibilities for extension were outlined.

## Klíčová slova

Foreman, Katello, Ruby, Docker, správa verzí softwaru

## Keywords

Foreman, Katello, Ruby, Docker, content management

## Citace

Adam Růžička: Software Versions Management  
in Containers Deployment, bakalářská práce, Brno, FIT VUT v Brně, 2015

# Software Versions Management in Containers Deployment

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Uvedl jsem všechny publikace a prameny, ze kterých bylo při psaní čerpáno.

.....  
Adam Růžička  
May 20, 2015

## Poděkování

Rád bych poděkoval vedoucímu mé práce Ing. Aleši Smrčkovi, Ph.D. a Ing. Ivanu Nečasovi z firmy Red Hat za vedení práce a cenné rady při jejím řešení. Dále bych rád poděkoval celé firmě Red Hat za vypsání této práce a také svým blízkým za podporu po celý čas studia.

© Adam Růžička, 2015.

*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Container Virtualization Technology</b>	<b>4</b>
2.1	Types of Virtualization . . . . .	4
2.1.1	Emulation . . . . .	4
2.1.2	Paravirtualization . . . . .	4
2.1.3	Container Virtualization . . . . .	5
2.2	History of Container Virtualization . . . . .	5
2.2.1	Change File-system Root - chroot . . . . .	5
2.2.2	FreeBSD Jails . . . . .	5
2.2.3	Solaris Zones . . . . .	6
2.2.4	OpenVZ Linux Containers . . . . .	6
2.2.5	LXC Project . . . . .	6
2.2.6	Docker Project . . . . .	7
2.2.7	Comparison of Presented Container Virtualization Technologies . . . . .	8
<b>3</b>	<b>Content Management with Foreman and Katello</b>	<b>10</b>
3.1	Introduction to Foreman and Katello . . . . .	10
3.2	Building Blocks for Content Management . . . . .	11
3.3	Principles of Content Management . . . . .	12
3.3.1	Recommended Workflow . . . . .	12
3.3.2	Keeping Content Hosts Updated . . . . .	13
<b>4</b>	<b>Design of System for Keeping Docker Images Updated</b>	<b>14</b>
4.1	Design of System for Managing Content on Docker Images Using Katello . . . . .	14
4.1.1	Use-case Analysis . . . . .	14
4.1.2	Requirements Analysis . . . . .	16
4.2	Build Service . . . . .	17
4.2.1	OpenShift Origin . . . . .	18
4.2.2	ImageFactory . . . . .	18
4.2.3	Dock . . . . .	18
<b>5</b>	<b>Implementation of System for Managing Content on Docker Images</b>	<b>20</b>
5.1	Building Docker Images . . . . .	20
5.2	Bulk Builds of Docker Images . . . . .	21
5.3	Updating Docker Images . . . . .	21
5.4	Build Orchestration . . . . .	22

5.4.1	Container Actions	22
5.4.2	Docker Image Build Config Actions	22
5.4.3	Image Actions	23
5.4.4	System Actions	23
5.4.5	Dock Plugins	24
5.5	Web User Interface	24
5.5.1	Docker Images	25
5.5.2	Docker Image Build Configs	27
5.5.3	Build Resource	29
5.5.4	Permissions and Security	29
5.6	Metrics	30
<b>6</b>	<b>Conclusion</b>	<b>31</b>
6.1	Ideas for Further Extension	31

# Chapter 1

## Introduction

In the early days of computing, when computer programs were defined by the states of computer mainframe's patch cables and switches, the computers were able to compute a single task at a time. As technology advanced, computers gained the ability to perform several tasks at once and reached the point, where deploying single service per machine was highly inefficient. To address this issue without sacrificing security, people started to deploy virtual machines on servers and run services in those virtual machines, which lead to better hardware utilization. To further increase service density, container virtualization, which reduces the amount of things needed to be run, was invented.

Container virtualization has been around for more than 15 years now, but working with containers was cumbersome, required the user to know his way around and required time to deploy. This changed with the arrival of Docker. It allows the user to quickly deploy containers distributed in the form of Docker images from the Docker hub and share his own images with other users.

With rising popularity of container virtualization and especially Docker the users of projects Foreman and Katello started to ask for support of Docker. An answer to these requests was a ruby gem `foreman-docker` and series of patches for Katello, which allowed to manipulate with Docker containers the same way as with regular virtual or baremetal machines and to work with Docker images in a fashion similar to working with RPM packages or Puppet modules.

But important part of the Docker experience is building Docker images. This thesis should focus on enhancing the build process and joining it with content management features of Foreman and Katello projects. Its purpose is to design and implement a solution for managing content on Docker images using projects Foreman and Katello.

The thesis is divided into six chapters. Chapter 2 introduces different approaches to virtualization with focus on container virtualization and briefly explain why is Docker an appealing platform and what caused its rather swift rise. Chapter 3 provides an introduction to projects Foreman and Katello and describes the basics of content management using these projects. Chapter 4 analyses the possible use-cases and describes a design of how previously mentioned projects could be used for managing content on Docker images. Chapter 5 talks about implementation of the design from previous chapter and the last chapter summarizes the whole thesis, talks about achieved goals and suggestions for further development.

## Chapter 2

# Container Virtualization Technology

This chapter provides introduction to container virtualization in general and describe the evolution from single system call to full fledged container virtualization solutions.

### 2.1 Types of Virtualization

There are three major approaches to virtualization: emulation, paravirtualization and container virtualization. Each of these will be described in this section.

#### 2.1.1 Emulation

This kind of virtualization is performed by a process emulating the whole hardware for the guest operating system. This provides good isolation of the guest, but comes with the cost of high overhead and thus low performance. There is no need to modify the guest operating system, which can be run as-is. For better understanding see Figure 2.1. Examples of this kind of virtualization are Oracle Virtual Box, VMware player or BHyVe.

#### 2.1.2 Paravirtualization

Paravirtualization is partial virtualization in a way that both the host and the guest share the same hardware. A thin layer called hypervisor controls the hardware and manages the guest operating systems. This approach has lower overhead than emulation, but has one significant drawback - the guest os has to be modified to be able to run in such environment. Interesting thing about paravirtualization is that the operating system controlling the hypervisor is in fact a guest with higher privileges than the other guests. For better understanding see Figure 2.2. Examples of this kind of virtualization are Xen, Microsoft Hyper-V and VMware ESXi.

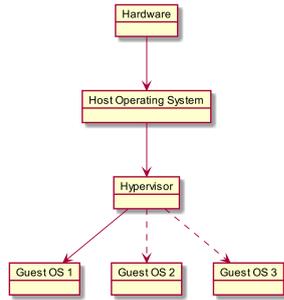


Figure 2.1: Type 2 hypervisor

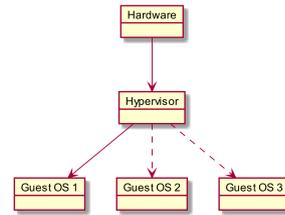


Figure 2.2: Type 1 hypervisor

### 2.1.3 Container Virtualization

This form of virtualization uses kernel features of the given operating system to run applications in isolated execution environments. This provides very low overhead and allows to dynamically distribute resources among guests and the host. However since the guests and the host run on the same kernel, container virtualization cannot be used to virtualize another platforms and all the solutions are platform specific.

## 2.2 History of Container Virtualization

This section describes several container virtualization technologies used on various platforms with focus on Docker, since it is the most important one for this thesis.

### 2.2.1 Change File-system Root - chroot

The first step towards what we now call *container virtualization* was introduction of the *chroot* system call in version 7 AT&T UNIX in 1979. It allowed the programmer to limit program's access to a subtree of the filesystem hierarchy for his or her program. Later, in 1986, a standalone chroot utility was presented in BSD4.3-Reno, which allowed to change root for any program. However, there were ways, which could be used to break out from chroot environment. Isolating access to a certain part of the directory tree laid the foundation on which other, more advanced tools could be built. [11]

### 2.2.2 FreeBSD Jails

The so-called jails first appeared in FreeBSD 4.0 in 2000. Jails are basically the first true container virtualization technique. They provide filesystem, network and privilege isolation, as well as disk, memory and CPU quotas. When creating a jail, a fresh copy of the base system is installed into the destination directory thus creating a clean environment, which can be used. Jails can be transferred to another machine and run there, as long as the host cpu architectures match, however portability is not their number one feature.

Jails are mainly designed as an additional layer of protection. If an attacker breaks into your application running directly on the host, he can compromise the whole system. If an attacker breaks into your application running in jail and even if he manages to elevate his privileges to superuser, he will not be able to get outside the jail. [10]

### 2.2.3 Solaris Zones

The Solaris Zones are Oracle's take on container virtualization. Previously known as Solaris containers are a part of Oracle Solaris and provide isolation layer for applications and services. There is one special zone called global zone, which is the zone of the physical machine. Boot of global zone is equal to boot of the system's hardware. The non-global zones or just zones are completely isolated environments, which cannot detect presence of other zones. Each zone has its own identity, which contains its name, numeric ID and path to its root, which is relative to the root of the global zone. Zones can be created and configured only from the global zone, which means zones cannot be nested.

Zones can be classified as zones with sparse root or whole root. Whole root means that the filesystem access for the zone is not limited and is equal to the global zone's access. Sparse root means the zone can only access a specific subtree of the global zone's root.

Currently only 8191 non-global zones can be created within a single instance of Solaris. Although Solaris Zones are aimed towards security and better utilization of hardware than portability they can be migrated among machines. [1]

### 2.2.4 OpenVZ Linux Containers

The OpenVZ is container virtualization solution for Linux. It allows launching of secure and isolated containers, which perform exactly like physical machines. Each container has its own set of user IDs, IP addresses, memory and processes. It consists of optional custom kernel and command-line tools. The major drawback of OpenVZ is that although it can be run on regular kernel, use of custom kernel is recommended for better security. [8]

### 2.2.5 LXC Project

LXC is a user space interface for containment features in Linux kernel released mostly under Lesser GPL2 license. It consist of API and necessary tooling needed for proper manipulation. The main goal of the LXC project is to create an environment as close as possible to standard installation without separate kernel. It uses kernel namespaces for interprocess communication, hostname, mountpoints, process IDs, network access and mapping of user IDs. Furthermore, it utilizes kernel CGroups for setting quotas on hardware usage and has AppArmor and SELinux profiles and seccomp<sup>1</sup> policies. [6]

**Cgroups** Linux Control Groups are a kernel feature, which allows to aggregate and partition sets of tasks into hierarchical groups. A Cgroup associates a set of tasks with set of parameters for one subsystem. Subsystem is usually a resource controller, which can schedule, limit and account access to the resource. [13] There can be an arbitrary amount of Cgroup hierarchies present and active on a system. Each of these hierarchies has an instance of Cgroup virtual file system associated. User-level code may create and destroy Cgroups in their instance of the virtual file system, specify and query to which Cgroup a task is assigned and list PIDs assigned to a Cgroup. [9]

**Namespaces** Linux kernel namespaces are wrappers around global system resources. To processes within a namespace it appears as if they had their own isolated instance of the global resource. This is essential for container virtualization.

---

<sup>1</sup>secure computing mode

## 2.2.6 Docker Project

Docker is a recently released platform for rapid development, shipping and deployment of PaaS<sup>2</sup> applications. Docker as a platform consists of two major parts. The Docker engine, which is the container virtualization platform, and Docker hub, which is a SaaS<sup>3</sup> platform for sharing and managing Docker images. [15]

It uses client-server architecture, with Docker daemon running in the background and doing all the lifting and the client ordering the daemon through sockets or RESTful API. The three essential parts of Docker are images, containers and registry.

*Docker images* are read-only templates used for creating containers. Docker makes it easy to build new images, update them or to simply download images other people have created. They consist of layers being mounted atop of each other using union mount.

*Docker container* is basically a launched image with read-write layer added on top. It contains everything the application needs to run and it can be started, stopped, moved and deleted.

*Docker registry* is a service provided by Docker, which provides public hosting of Docker images.

**Building Docker images** There are two main approaches to building Docker images. The first is starting an image, doing changes in the running container and committing the changes. This creates new image with user's changes added as a layer on top of the original image.

Second approach is to use a *Dockerfile*. Dockerfile is Docker's recipe for creating images. It can be sent to the Docker daemon, which basically does what is described in the first approach, but with one significant difference. Docker daemon creates a new container for each and every one command in the Dockerfile.

An example of Dockerfile can be seen in Figure 2.3. This Dockerfile uses an image from repository *centos* with tag *centos7* as its base image, runs commands to install a HTTP server and PHP interpreter, generates *index.php* to show php info page, sets default command to run the HTTP server without forking to background and exposes the port, on which the HTTP server is listening.

Figure 2.3: Example of Dockerfile

```
FROM centos:centos7
RUN yum install -y httpd php
RUN echo '<?php phpinfo(); ?>' > /var/www/html/index.php
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
EXPOSE 80
```

**Namespaces and CGroups** Docker uses Linux namespaces to provide additional security for containers. A set of namespaces is created for each launched container. Furthermore, it utilizes Cgroups to limit containers' access to host's resources. This allows the user to set limits and quotas for containers.

---

<sup>2</sup>Platform as a Service

<sup>3</sup>Software as a Service

**Union File Systems** Union file systems operate by mounting directories atop of each other. This is heavily used in Docker as this allows to reuse common parts of containers. At the time of writing this document Docker supports AUFS<sup>4</sup>, btrfs<sup>5</sup>, vfs and DeviceMapper as unionfs drivers.

**Container format** Docker combines these components into a wrapper called *container format*. Default format is libcontainer, but LXC is supported too.

Docker creates an excellent platform aimed towards fast delivery, ease of deployment, scaling and management and higher density of service. Since its first release it has been getting more and more popular, be it for its ease of usage or for being the only one providing this set of features in a bundle.

**Comparison of virtual machine and Docker based deployments** Figures 2.4 and 2.5 show the difference between virtual machine and Docker based deployments. To achieve application isolation in virtual machine based deployment, one has to have whole guest operating system with its own kernel, libraries and binaries to run the application.

The Docker deployment lowers the overhead by removing the need for guest operating system since the containers are basically run directly on the host using its kernel features for isolation.

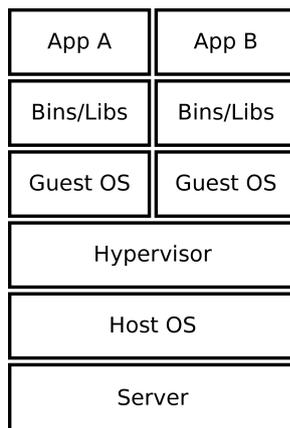


Figure 2.4: Typical deployment in virtual machine based environment.

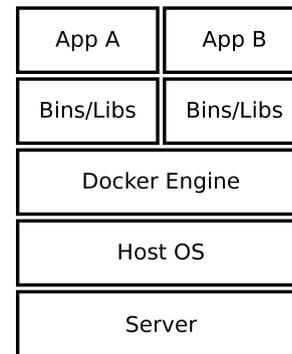


Figure 2.5: Dockerized deployment

### 2.2.7 Comparison of Presented Container Virtualization Technologies

The Table 2.1 shows comparison of container virtualization technologies described in Section 2.2 sorted chronologically. It shows the most important things about each of the techniques, namely platform they are available at, original release year, isolation level, their primary purpose and whether they can be nested.

<sup>4</sup>Another Union File System

<sup>5</sup>B-Tree File System, UnionFS like behavior is achieved via Copy-on-Write

Table 2.1: Comparison of Presented Container Virtualization Technologies

<b>Name</b>	<b>Platform</b>	<b>Available since</b>	<b>Isolation level</b>	<b>Aim</b>	<b>Nesting</b>
Chroot	Unix	1986	Filesystem only	Security	No
Jails	FreeBSD	2000	Complete	Security	Yes
Zones	Solaris	2005	Complete	Security	No
OpenVZ	Linux	2005	Complete	Security	No
LXC	Linux	2008	Complete	Security	Yes
Docker	Linux	2013	Complete	Portability	Yes

## Chapter 3

# Content Management with Foreman and Katello

This chapter introduces projects Foreman and Katello and describes what is their purpose, as well as how they can be used for content management.

### 3.1 Introduction to Foreman and Katello

Foreman is an open-source project providing system administrators with the possibility to provision, orchestrate, configure and monitor their systems. It can be controlled via web UI or command-line interface and has RESTful API, which enables developers to build higher level logic. There is an extensive library of Foreman plugins, adding the possibility to provision hosts on various providers or use another config management tool.

It allows the system administrator to discover, provision and upgrade his entire infrastructure, be it local or cloud, group hosts and manage them in bulk and audit changes done to the infrastructure from one place.

Foreman is currently deployed in many organizations, managing from tens to thousands of servers. [2]

Katello is an open-source project aimed at making the task of managing multiple machines in predefined configuration and with defined versions of packages installed. It is a Ruby on Rails engine for Foreman. It used to be a standalone application, but having some functionality overlap with Foreman, it was decided to convert Katello to a plugin for Foreman to make development and usage easier. [3]

It groups several open-source projects to provide a solution for provisioning and content and config management. The used projects are:

- Foreman - provisioning
- Pulp - content management
- Candlepin - subscription management
- Puppet - configuration management
- Elasticsearch - database indexing
- AMQP - communication between components

Figure 3.1 shows typical deployment of Foreman and Katello. Arrows mean communication direction, cylinders mean persistent storage and rounded rectangles mean services. Katello is a plugin for Foreman so its enclosed by Foreman. User works with it using the web UI or API. Foreman and Katello then communicate with Candlepin, Pulp and Puppet using their API. Foreman, Katello and Candlepin store their data in PostgreSQL database, Pulp uses MongoDB. ElasticSearch is used for indexing and searching in PostgreSQL database. AMQP is used for communication between components.

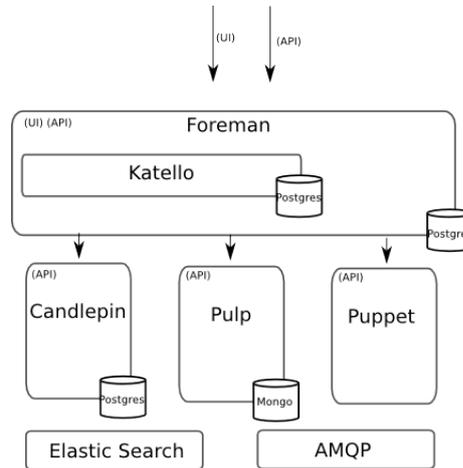


Figure 3.1: Architecture of typical deployment

## 3.2 Building Blocks for Content Management

This section contains definitions of several terms, which will be used throughout the rest of the document and not having solid definitions of them could cause confusion and misunderstanding.

**Content** in Katello is basically anything that can be stored in its repositories and promoted through environments. At the time of writing this paper it includes RPM software packages, package groups, errata, Puppet modules and Docker images.

**Product** is a collection of content repositories. It serves the purpose of logically grouping content with similar purpose or origin. One can have a product containing only Puppet modules called *configuration* and another product called *EPEL* containing RPM repositories for different releases and CPU architectures.

**Content View** is a logical grouping of products and thus RPM packages, erratas, Puppet modules and Docker images. Its content can be filtered using inclusive or exclusive filters based on package group or package name. They can also be snapshotted.

**Content View Version** is a snapshot of content view. When a new content view version is published, the repositories in the content view are cloned and their content filtered with associated filters. content view versions are immutable and can be promoted through environments.

**Lifecycle Environment** is a step in promotion path through software development lifecycle. Example of software development lifecycle can be seen in Figure 3.2. Moving through the promotion path is done by publishing content view versions and promoting them to environments.

	Dev	Test	QA	Prod
Content Views	3	1	0	0
Content Hosts	0	0	0	0

	Unstable	Testing	Stable
Content Views	2	2	1
Content Hosts	0	0	0

Figure 3.2: Example of Lifecycle Environments in Katello

**Host** is a concept representing a provisioned machine, holding various information about it. This includes facts reported by Puppet, such as operating system or IP addresses on network interfaces, or which Puppet classes should be assigned to it.

**Content Host** is a part of the host managing content and subscription related tasks. This means it provides reports about installed packages and state of the host's configuration.

### 3.3 Principles of Content Management

After defining content management building blocks this section can now describe mechanics of content management and how Katello keeps its content hosts updated.

#### 3.3.1 Recommended Workflow

Content management workflow in Katello is very complex, but simplified version for the scope of this thesis could be summarized as:

1. Create lifecycle environments
2. Create content view
3. Register content hosts to consume content from content view
4. Create product
5. Create repository
6. Add product to content view
7. Synchronize repository
8. Publish new version of content view
9. Promote version to correct lifecycle environment

10. Repeat from point 7

[12]

### 3.3.2 Keeping Content Hosts Updated

When a host is registered to consume content from Katello, a content host object is created to represent it in Pulp. This object holds among other things information about RPMs installed on the host. Any package management operation on the host triggers update of the installed RPMs list and attached repositories in Pulp. Having correct information about packages installed on a host and content view, in which the host belongs, makes Pulp able to determine which packages can be updated. The relationship described in this paragraph is shown in the Figure 3.3.

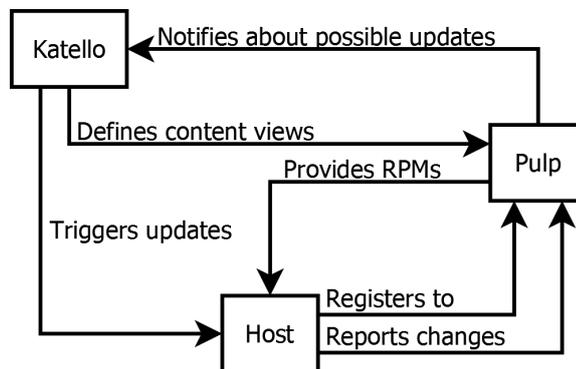


Figure 3.3: Katello - Pulp - Content host interaction diagram

## Chapter 4

# Design of System for Keeping Docker Images Updated

The main effort of this bachelor's thesis is to design a solution for keeping software on Docker images in sync with the content view it is in. The expected workflows are described in Subsection 4.1.1 and build service used for building Docker images in Section 4.2.

### 4.1 Design of System for Managing Content on Docker Images Using Katello

In order to design such system, use-cases and later requirements must be analyzed. Both analyses are described in this section.

#### 4.1.1 Use-case Analysis

There are four major use-cases, building image from already existing environments, bulk builds of images, automatic builds triggered by content view version publish and updates of existing images:

##### **Building Single Image from Existing Environments**

This use-case demonstrates desired workflow when user wants to build Docker image from his own Dockerfile and packages stored in Katello. This expects that the user has content views with RPM packages promoted into environments and wants to build Docker images from those environments.

The input needed from this user for this case is a git URL to a repository with the Dockerfile to build, compute resource, content view, environment and repository, where the built image should be stored. Optionally the user can provide git commit hash and select a base image to override the one provided in Dockerfile.

The user's input, should be transformed into a build configuration for the build service. Passing the build configuration to the build service should trigger a build of the Docker image. Build service should clone the git repository, pull the base image and force the build to use packages from provided content view and environment to build the image. The built image should be stored in the selected repository.

When the build is finished, the user should be able to add the repository to a content view, publish new version of it and promote it to lifecycle environments. The built image should be usable as a base image for building more images.

### **Bulk image builds**

This use-case demonstrates the desired workflow when the user wants to build image using the latest available packages, that means from the Library environment.

To be able to do bulk builds, the user has to create a Docker image build config object. It needs to have a URL to git repository with the Dockerfile, a content view and a repository, where the built images should be stored once built. Optionally the user can provide git commit hash and different base image.

When the user has several Docker image build configs created, he can select many of them and a compute resource and trigger their build. The build should be done in a way that failing to build one image should not affect the others, in other words the build should keep going. The built images are stored in their associated repositories.

### **Automatic builds**

The expected workflow is to provide the possibility to turn on automatic builds for some build configs. The general idea behind this is that the user should have the possibility to have images with latest content built automatically.

This should be done by enabling automatic builds for a Docker image build config. When there is a new version of the build config's content view published, a build should be started. As with regular builds, built images are stored in the associated repository.

Since this kind of build is triggered without any intervention from the user, there is a need to specify which compute resource should be automatically selected for the build. Solution to this problem is described in Section [4.1.2](#).

### **Updating Docker Images**

This use-case demonstrates desired workflow when the user wants to upgrade his Docker images to have the latest available versions of packages installed. This expects the user to have several Docker images stored in Katello, as well as RPM packages promoted into environments.

When the user synchronizes newer RPM packages and promotes them to environments, he should be able to find out three important things.

He should be able to determine, which images are able to be updated, i.e. which were built using a Docker image build config. Images imported from external sources or images built using single builds described in Subsection [4.1.1](#) cannot be updated since we don't know what packages are installed on the image or how to rebuild it.

The user should also be able to find out which packages present on an image are not in sync with the ones present in the associated content view and which erratas are applicable to this image. Having the layered nature of Docker images in mind, it is also important to determine which updates are applicable to the image itself and which it inherits from its base image.

Another thing which is important to know is whether the image is based on a current image. For example the user can have a Docker image build config set in a way that it uses a base image from *Production* environment. If the user builds an image on top of it

and later promotes another image to that environment, thus replacing the original one, he should be able to find out that the newly built image is not based on the latest base image.

When the user knows all the things described in previous paragraphs, he can select which images should be updated and start the updates.

#### 4.1.2 Requirements Analysis

To make the workflows described in Subsection 4.1.1 doable several things will have to be implemented. This contains user interface for providing input and notifying the user about possible updates, build service, interface to transform user's input to build service configuration, a way to represent Docker images as content hosts and a way to store information about Docker images in Katello.

The schematic of a system theoretically capable of doing so is shown in the Figure 4.1. Katello has RPM and Docker image repositories with content synchronized from external repositories. It also has number of content hosts representing machines consuming content from Katello's repositories.

My contribution is making build service use content from Katello to build Docker images and save them in Katello's repositories, adding associations for representing Docker images as content hosts and adding user interface for controlling the build service. It is marked in the schematic with dotted lines.

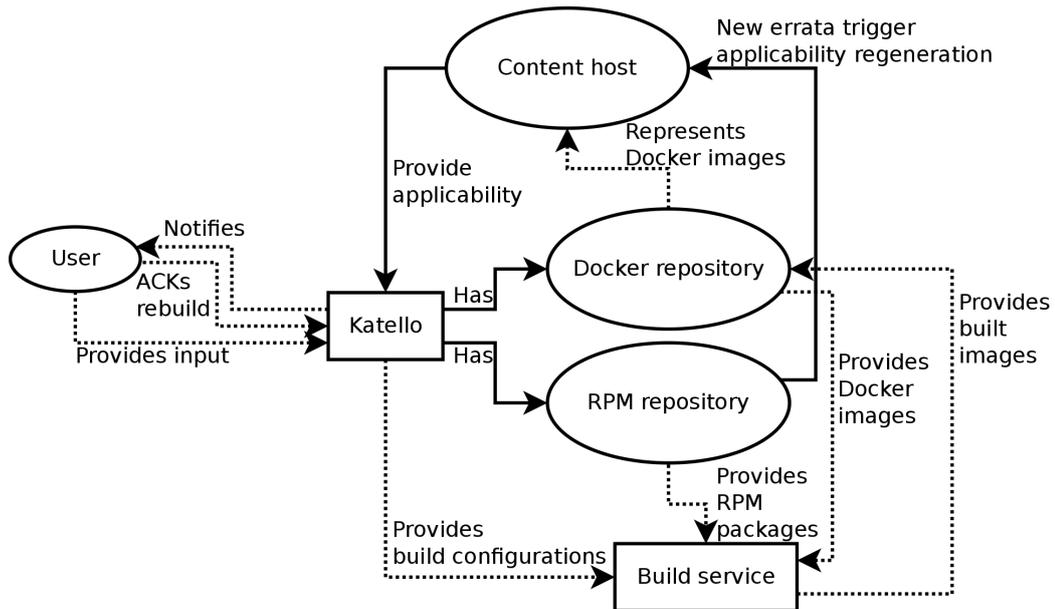


Figure 4.1: System schematic

**Docker Image Build Configuration class** In order to be able to rebuild images, it is needed to save the build configuration. Design of the object responsible for holding all the necessary information, such as git URL, git commit hash, full name of base image, flag whether it should do automatic builds and activation key prefix, is shown in the Figure 4.2.

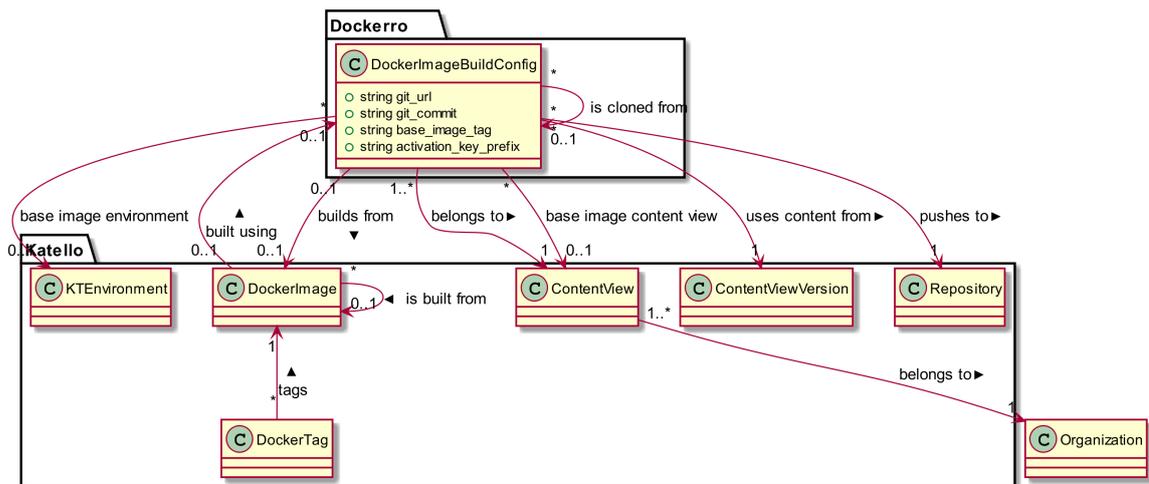


Figure 4.2: Class diagram of Docker Image Build Config.

**Build Resource** Second class designed for use in this project is build resource. Its relation to other objects is shown in the Figure 4.3. It is planned to be used in automated builds, where the user can define which compute resource will be used for the build using this object. It basically marks the compute resource usable in the set Location and Organization. When an automated build is triggered, it would use one of build resources from the content view's organization. The build should fail if there is no build resource present.

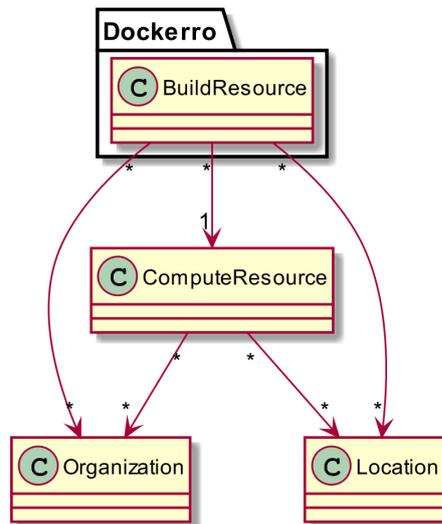


Figure 4.3: Build Resource UML

## 4.2 Build Service

There were long discussions about what should be used as a build service and three possible options. Those were OpenShift, ImageFactory and Dock. These three projects have one thing in common, they build Docker images using Docker daemon in Docker containers.

The principle is that it uses specialized image aimed at building images, launches it in a container on the physical host and inside that container it uses the Docker daemon to build the desired image using the provided configuration. This approach allows to easily scale the whole image building infrastructure by simply launching the builder container on Kubernetes [14] or OpenShift.

*Dock* was selected as the most suitable for the job because it is lightweight yet fully featured. However this does not mean that nothing else can be used. The interface should be flexible enough to allow the developers to create wrappers for their desired build service.

### 4.2.1 OpenShift Origin

OpenShift Origin is Red Hat's open-source platform for building and deploying PaaS <sup>1</sup> applications. It utilizes Docker, Kubernetes and Atomic or Enterprise Linux for managing large scale container deployments. Although its main focus is on container infrastructure orchestration, it does provide the possibility to build Docker images. [7]

### 4.2.2 ImageFactory

ImageFactory is a python program which can be used as a command-line utility or as a daemon communicating through RESTful API. It allows building images for RHEV<sup>2</sup>, VMware vSphere, Amazon EC2, Rackspace, OpenStack and Docker. [5]

### 4.2.3 Dock

Dock is a python library with command-line interface for building Docker images. It supports building images using Docker daemon running on the host, using Docker daemon in privileged Docker container and building images in current environment. Furthermore, it provides the possibility to modify the behavior with pre- and post-build plugins, which can be used for setting repositories for the build and probing the resulting image for package versions.

The first step in using Dock is to create a build container in which other images will be built. The build image contains its own copy of Dock and a wrapper script which is executed when the container is run. There are two main modes of Dock, called privileged and dockerhost. In privileged mode a new instance of Docker is run inside the build container, in dockerhost mode the Docker socket must be provided to the container. No matter how the Docker socket is obtained, it is used to build the new image. Dock can be controlled by command-line arguments or by setting environment variables. [4]

Sequence diagram showing Katello's interaction with Dock can be seen in Figure 4.4. Katello initiates the build by running the build container. To be able to run the build container, Docker pulls an image of the build container and launches it. The started container clones the git repository with Dockerfile, runs its prebuild plugins, pulls the base image for the new image and starts building the Dockerfile. The image being built uses provided activation key to register to consume content from Katello and downloads RPM packages from it. After finishing the build, Dock runs its postbuild plugins and Katello destroys the build container and saves the built image in its repositories.

---

<sup>1</sup>Platform as a Service

<sup>2</sup>Red Hat Enterprise Virtualization

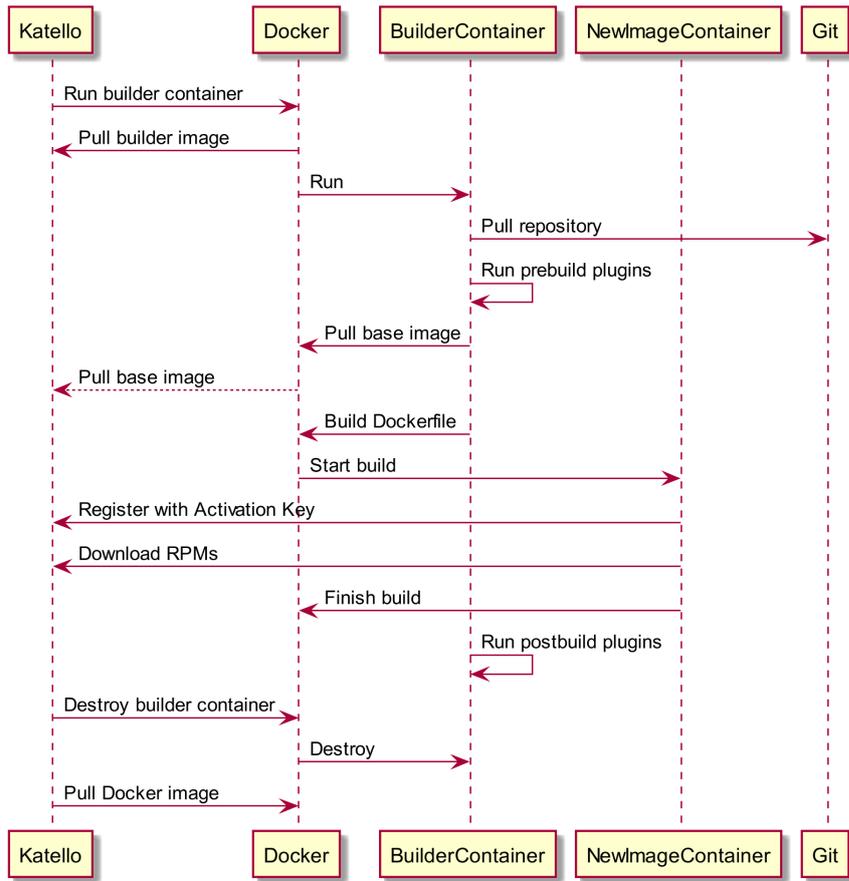


Figure 4.4: Sequence diagram showing Dock usage with Katello for building Docker Images.

## Chapter 5

# Implementation of System for Managing Content on Docker Images

This chapter describes implementation of the system for managing content on Docker images, including details about building Docker images, orchestration of the builds, user interface and code metrics.

### 5.1 Building Docker Images

All the builds are based on Docker image build config class described in Subsection [4.1.2](#). When building Docker images using the workflow described in Subsection [4.1.1](#) the configuration object is only temporary and is not saved. The configuration object is used to generate options and environment variables for the build container and find the right activation key.

Before actually starting the build, the base image is pulled. After that a builder container is launched using the configuration generated by the build configuration object. The builder container clones the git repository containing the Dockerfile and runs its pre-build plugins. In this particular case the pre-build plugins optionally change the FROM declaration in the Dockerfile and add commands for registering the container into Katello using the provided activation key as a first thing to do in the build. Now that everything is prepared, it starts the build of the Dockerfile. When the build has finished, Dock runs the post-build plugins. One post-build plugin is used to force refreshing of package profile of the associated content host. At this moment, control is given back to Katello.

The build proceeds by creating relations between the newly built image and its base image and between the newly built image and the content host representing it. Another thing which needs to be done regarding the content host is to set its bound yum repositories in Pulp, this is an easy task since all the required data is already present in the build configuration.

At the end of the build the builder container is destroyed. Figure [5.1](#) shows how an action for building Docker image looks. The actions from which it is composed are described in Section [5.4](#).

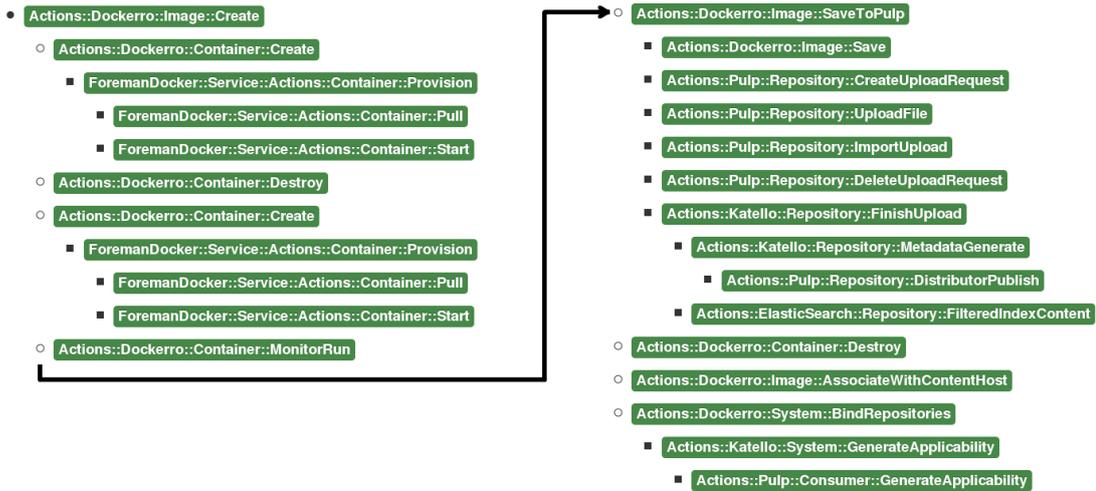


Figure 5.1: Action Structure of Single Docker Image Build

**Building Docker Images from Docker Image Build Configs** There is slight difference between building Docker images using the New Docker image form and building them from the Docker image build configs. When building from existing configuration, there are several extra steps added to the beginning of the build. Instead of using a temporary configuration tailored for a specific environment, the build configuration is cloned for the latest version of its content view present in the Library environment and saved. The build is started with the cloned instance instead. The rest of the build goes as described in Section 5.1.

## 5.2 Bulk Builds of Docker Images

Bulk build of Docker images is basically just a group of regular builds started in a batch with one subtle difference. Because of its nature it cannot be done using temporary Docker image build configs, it has to be saved.

All the selected Docker image build configs are scheduled for building as one task composed of subtasks for each of the configs. The separate builds running in subtasks are done as described in the previous section.

## 5.3 Updating Docker Images

Updating Docker images can be done in two ways, by running update operations using the package manager present on the image and committing the changes or by simply rebuilding the image with newer content. The first approach has one significant drawback. Increasing number of image layers negatively affects the performance and also increases the overall size of the image. Because of this, rebuilds of images are clearly the better choice.

In case of single rebuild all what needs to be done is to trigger another build using the Docker image build config associated with the image. However bulk updates are a bit more complicated topic.

Doing a bulk update entails figuring out which images can be safely rebuilt in parallel and which depend on others. Because Docker images are layered it is important to know

which updates are applicable to the top level of the image and which are applicable to the base image. Pushing the update to the higher layer would lead to wasting space because the package would be present in two versions. Although the version in the higher layer would effectively overwrite the lower one, it would still be there taking up space. This means we cannot update an image and its base image at the same time.

This issue is solved by creating a dependency tree of all images which should be updated. This allows us to divide the images into groups whose updates should not interfere with other ones by starting bulk builds for each level of the dependency tree. This solution is not ideal in terms of performance, because rebuilds of some images may be delayed by waiting for an image, irrelevant to them, to be built. However it is safe and should not lead to errors.

Another thing which needs to be considered is that the newly built images always end up in the Library environment. This can cause problems if we rebuild an image based on an image from an environment other than the Library. If this situation arises, the build fails because it could not be done properly and the user should handle this by updating the base image first, promoting it to the correct environment and updating the dependent images afterwards.

## 5.4 Build Orchestration

Foreman and Katello uses Dynflow<sup>1</sup> for orchestrating actions so it was logical to use it as well. All the actions belong to `::Actions::Dockerro` module.

### 5.4.1 Container Actions

These are the actions belonging to the `::Actions::Dockerro::Container` module. Their role is to provide the user with actions necessary for manipulating containers.

**Create** Action used for creating the build container. Its purpose is to add default values to the user's input. It uses two actions from `foreman_docker`<sup>2</sup> plugin to pull the image for the container and create it afterwards.

**Destroy** Action used for destroying the builder container once the build has finished.

**MonitorRun** An action which starts the container and periodically polls for its status and pulls logs from it. Once the container's state is changed to 'Stopped', it checks its exit code and fails if it wasn't successful.

### 5.4.2 Docker Image Build Config Actions

These are the actions belonging to the `::Actions::Dockerro::DockerImageBuildConfig` module. Their role is to provide the user with actions necessary for manipulating build configs.

**AssociateImage** This action is used to associate built image with a build configuration. Its input are IDs of the build config and base image. It finds the objects having those IDs and creates the association between them.

---

<sup>1</sup><http://dynflow.github.io/>

<sup>2</sup><https://github.com/theforeman/foreman-docker>

**Build** This action is used while bulk building images. It's input is a `build_config`, `compute_resource_id` and `hostname`. If the given `build_config` is a template, it clones it for the latest available version of the associated content view and schedules actions for creating the cloned configuration, creation of Docker image and associating the built image with the created config.

**BuildOnPublish** Action which is planned when `::Actions::Katello::ContentView::Publish` action is run. It lists all Docker image build configs associated with the given content view, selects the ones with automatic flag set and triggers their build.

**Create** This action only saves the given `build_config` into the database.

**CreateAndAttachActivationKey** An action used for creating an activation key for the Docker image build config which also attaches all available subscriptions once the activation key was created.

**Destroy** This action only removes the given `build_config` from the database.

### 5.4.3 Image Actions

These are the actions belonging to the `::Actions::Dockerro::Image` module. Their role is to provide the user with actions necessary for manipulating Docker images.

**AssociateWithContentHost** Action used for creating associations between Docker images and content hosts. It binds a content host with the Docker image it represents.

**Create** Action responsible for creating the Docker image. It plans container creation, monitoring its run, saving the built image to Pulp and destruction of the build container.

**Save** This action is used to pull the built image from Docker and save it to temporary directory as a tarball.

**SaveToPulp** This actions plans action to save the built image as tarball and actions needed for uploading a file to Pulp. It is similar in function to `::Actions::Katello::Repository::UploadFiles`.

**Update** Action used when bulk updating Docker images. It builds a dependency tree of the images and schedules a bulk build for each of the levels.

### 5.4.4 System Actions

The are actions belonging to the `::Actions::Dockerro::System` module. They are used for modifying content hosts.

**BindRepositories** Action which saves into Pulp to which repositories the content host has access so it can compute applicability.

**BindRepositoriesOnPromote** Action run when `::Actions::Katello::ContentView::Promote` action is run. It's purpose is to keep information about bound repositories in Pulp accurate in case a repository was added or deleted in the version being promoted.

### 5.4.5 Dock Plugins

Dock's plugin interface allows to easily create plugins which can be used to easily perform otherwise difficult tasks.

**RunCmdInContainerPlugin** This prebuild plugin allows to run provided command as a first thing during the build. It works by modifying the provided Dockerfile and adding the command as `RUN $command` right after the `FROM` line. It is used to instruct the container to create custom facts to be reported, to generate client certificates for Katello and to register it to consume content from Katello using provided activation key.

Figure 5.2 shows how the Dockerfile shown in Figure 2.3 would look like after being changed by this plugin.

```
FROM centos6-5.v2:5000/default_organization-registry-centos:centos7
RUN yum localinstall -y \
    http://centos6-5.v2/pub/katello-ca\
    -consumer-latest.noarch.rpm && \
    mkdir -p /etc/rhsm/facts && \
    echo '{"dockerro.represents":true, \
        "dockerro.build_config_id":null, \
        "dockerro.build_uuid":"060773163121475d83584d2b34d3fdd7"}' \
    > /etc/rhsm/facts/docker_identification.facts && \
    rm -rf /etc/pki/consumer \
    /etc/pki/entitlement \
    /etc/pki/product && \
    subscription-manager register \
    --org='Default_Organization' \
    --activationkey='dockerro-zoo_2-testing' || true && \
    subscription-manager repos
RUN yum install -y httpd php
RUN echo '<?php phpinfo(); ?>' > /var/www/html/index.php
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
EXPOSE 80
```

Figure 5.2: Dockerfile modified by RunCmdInContainerPlugin

**PostRunCmd** This postbuild plugin creates a temporary container from the built image and forces `subscription-manager` to report its package profile to Katello. This allows us to have accurate and up-to-date list of packages installed on the image.

## 5.5 Web User Interface

Foreman and Katello use Bastion for its web user interface. Bastion is single-page web client based on AngularJS designed specifically for Foreman. For the sake of keeping the interface consistent it was used too.

### 5.5.1 Docker Images

Figure 5.3 shows the form presented to user for creating Docker images. It asks the user for URL to git repository with Dockerfile, optionally git commit hash, content view, environment and target Pulp repository. Furthermore, it allows setting base image which should be used for building images using this configuration. Selection of the base image is done by combination of an environment path picker and a drop box showing images available in the selected environment. Clicking the *Save* button starts the build and shows the user a page with progress bar of the build.

The screenshot shows a web form titled "New Docker Image". The form contains the following fields and controls:

- Git URL\***: A text input field containing "https://github.com/adamruzicka/zoo.git".
- Git commit**: An empty text input field.
- Base Image Environment**: A radio button group with "Library" selected (indicated by a checkmark and a dark blue arrow) and "testing" unselected.
- Base Image**: A dropdown menu showing "centos:7-sm".
- Compute Resource**: A dropdown menu showing "centos6-3.v2".
- Environment**: A radio button group with "Library" unselected and "testing" selected (indicated by a checkmark and a dark blue arrow).
- Content View**: A dropdown menu showing "zoo".
- Target Pulp Repository**: A dropdown menu showing "camel-bear".
- At the bottom, there are two buttons: "Cancel" (grey) and "Save" (blue).

Figure 5.3: New Docker Image form

The part of UI shown in the Figure 5.4 shows listing of Docker image present in current organization. The table has columns which show the ID of the image, its name in the format of *NAME:TAG*, available updates and errata counts, what image was used as its base, whether the base image was replaced by newer one and whether the image can be updated, i.e. if we know how to rebuild it. Furthermore, the table allows to filter out images which we cannot update, this means images synchronized from external repositories or built without a build config.

The column *Image ID* is a link to details about that image shown in Figure 5.5. Column *Available Updates* shows the counts of applicable security, bugfix and enhancement erratas, as well as updates not bound with any errata and updates inherited from its base image.

Updateable						
<input type="text" value="Search..."/>		Showing 9 of 9 (9 Total)		0 Selected		<a href="#">Bulk Update</a> <a href="#">+ New Docker Image</a>
Image ID	Name	Available Updates	Based on	Based on outdated image	Updateable	
<a href="#">14251c727e53925bed4a8c417f391bf2acc8e200023ba20422be4bfe40db8277</a>	rhncfg:zoo_2-testing	0  0  0	centos:7-sm	Unknown	No	
<a href="#">4fc5c6cf40aef76623427c29c223348cc33f75c8943670732dcfc4d36a16f872</a>	rhncfg:zoo_2-Library	0  6  1	centos:7-sm	No	Yes	
<a href="#">f45d60167b498f3383ea17fe0e301ed32a4f102b87bdf67d9f33141037f2c05d</a>	crow:zoo_2-testing	0  0  0	centos:7-sm	Unknown	No	

Figure 5.4: Docker Images listing

On Figure 5.5 we can see the details of a Docker image. These details include what image it was built on, which build config was used to build it, which packages are available for update and what packages are present of the image.

## Docker Image crow:zoo\_2-Library

### Basic Information

**Based on** mammals:zoo\_2-Library  
**Built using** None

### Available Package Updates

Name	Version	Release	Inherited
camel	0.2	1	No
bear	4.3	1	No
camel	0.2	1	Yes
bear	4.3	1	Yes

### Image Content

Name	Version	Release
audit-libs	2.3.3	4.el7
basesystem	10.0	7.el7.centos
bash	4.2.45	5.el7_0.4

Figure 5.5: Docker Image details

### 5.5.2 Docker Image Build Configs

Figure 5.6 shows the form presented to user for creating new Docker image build configs. It is similar to form for creating Docker images show in Figure 5.3.

The difference is this one does not allow setting of environment for the resulting image because it is always built from the Library environment. In addition it allows to set activation key prefix and whether a build using this configuration should be triggered automatically when publishing new version of the content view.

## New Docker Image Build Config

**Git URL\***

**Git commit**

**Activation key prefix**

**Base Image Environment**

Library  testing

**Base Image**

**Content View\***

**Target Pulp Repository\***

**Automatic builds on publish**

Figure 5.6: New Docker Image Build Config form

Figure 5.7 shows the list of present Docker image build configs. It shows only the template build configs with the cloned ones filtered out. This is because user should not be even aware of the clones' presence. The columns in this table show the name, content view, repository and tag of the associated base image.

**Docker Image Build Configs**

Showing 3 of 3 (3 Total)

0 Selected

<input type="checkbox"/>	Name	Content View	Repository	Base image tag
<input type="checkbox"/>	camel-bear	zoo	camel-bear	7-sm
<input type="checkbox"/>	crow	zoo 2	crow	zoo_2-Library
<input type="checkbox"/>	mammals	zoo 2	mammals	7-sm

Figure 5.7: Docker Image Build Config listing

Figure 5.8 shows the UI for bulk builds, where the user can select build configs, a compute resource and trigger the builds.

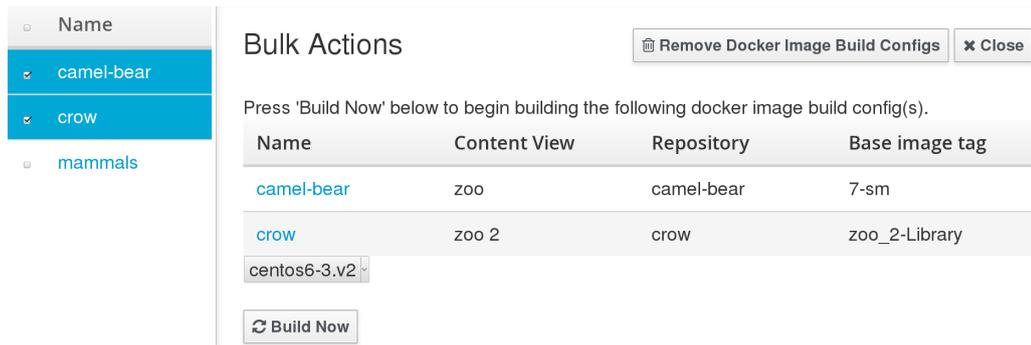


Figure 5.8: Docker Image Build Config bulk build

### 5.5.3 Build Resource

The UI for build resources is really simple, because the object basically has just a name, compute resource and a number of locations and organizations. Form used for creating new build resources can be seen in Figure 5.9. The user can fill a name and select a compute resource available in the current organization. Locations and organizations can be set in the corresponding tabs.

## New build resource

Figure 5.9: New Build Resource Form

### 5.5.4 Permissions and Security

In Katello and Foreman each user can have a number of roles, which allow him to access parts of the web UI and manipulate objects in the system. My project adds several new permissions. Those are permissions for viewing, creating and removing Docker image build config and for creating Docker images. Permission for viewing Docker images was already present in the *foreman-docker* plugin.

## 5.6 Metrics

Source code metrics are shown in Table 5.1. It shows number of lines of code and size in bytes for each of used programming languages.

Table 5.1: Code Metrics

<b>Language</b>	<b>LOC</b>	<b>Size [B]</b>
JavaScript	814	30463
Ruby	1901	66716
HTML	821	31496

# Chapter 6

## Conclusion

The goal of this thesis was to get familiar with current trends in virtualization technologies, projects Foreman and Katello and design and implement a solution for applying content management features of Katello to Docker images. It tries to fill the gap in the market by providing unique content management features. The process of designing and implementing such project was described in previous chapters.

Experimental testing has shown that the solution is able to build Docker images with content defined in Katello's content views, build Docker images based on images stored in Katello, store information about packages installed on built images and update images which have older versions of packages installed.

### 6.1 Ideas for Further Extension

**Using another build service** The project was design in a way that the build service could be easily replaced with another as long as it would follow the same workflow. All which would be needed to do would be to set different builder image name in settings, override methods which generate options, and environment variables for the build container.

**Scaling** To speed the building process it would be good to distribute the builds among several builder machines. One approach to this would be to modify the Dynflow action planning the builds to somehow plan them to run on different compute resources. This could be further improved by introducing some kind of ticketing middleware to Dynflow, which would allow to run only a number of tasks of the same class at a time, thus reducing immediate load on the builder machines. Another approach would be to use a Kubernetes compute resource instead of plain Docker one. This would cause the builds to be distributed among machines without any extra work.

**Using Puppet** Another quite interesting expansion would be to build Docker images with configuration provided by Puppet. There would be a need for a generic Dockerfile which would download a list of selected Puppet modules and apply them to the image. This would need some changes to the Docker image build config class, but would shorten the gap between regular hosts and Docker images by a bit. Another approach would be to create a temporary record on the Puppet master for a certain hostname, run the base image with that hostname, let Puppet apply changes provided by the Puppet master and commit the changes afterwards and remove the record from the Puppet master.

# Bibliography

- [1] Enterprise Manager Ops Center User's Guide v11.1.3.  
[http://docs.oracle.com/cd/E18440\\_01/doc.111/e18415/toc.htm](http://docs.oracle.com/cd/E18440_01/doc.111/e18415/toc.htm).
- [2] Foreman documentation v1.8. <http://theforeman.org/manuals/1.8/index.html>.
- [3] Katello documentation v2.2. <http://katello.org/docs/2.2/>.
- [4] Homepage. Dock readme. <https://github.com/DBuildService/dock>. [upd. 2014-12-16].
- [5] Homepage. ImageFactory. <http://imgfac.org/>. [upd. 2014-12-15].
- [6] Homepage. Linux Containers - LXC.  
<https://linuxcontainers.org/lxc/introduction/>. [upd. 2014-11-20].
- [7] Homepage. OpenShift Origin. <http://www.openshift.org/>. [upd. 2014-12-15].
- [8] Homepage. OpenVZ Linux Containers. [http://openvz.org/Main\\_Page](http://openvz.org/Main_Page).
- [9] Paul Menage. CGroups.  
<https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>.
- [10] FreeBSD Documentation Project. FreeBSD Handbook.  
<http://ftp.freebsd.org/pub/FreeBSD/doc/en/books/handbook/book.pdf.bz2>.
- [11] OpenBSD Project. OpenBSD 5.5 manual pages.  
<http://www.openbsd.org/cgi-bin/man.cgi>.
- [12] Justin Sherill, Grant Gainey, and Todd Warner. Red Hat Satellite 6.0 Core-SOE, 2014. Red Hat internal document.
- [13] Web pages. Introduction to Control Groups (Cgroups).  
[https://access.redhat.com/documentation/en-US/Red\\_Hat\\_Enterprise\\_Linux/6/html/Resource\\_Management\\_Guide/ch01.html](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html). [upd. 2014-12-09].
- [14] Web pages. Kubernetes design overview.  
<https://github.com/GoogleCloudPlatform/kubernetes/blob/master/DESIGN.md>. [upd. 2015-01-10].
- [15] Web pages. Understanding Docker.  
<https://docs.docker.com/introduction/understanding-docker/>.