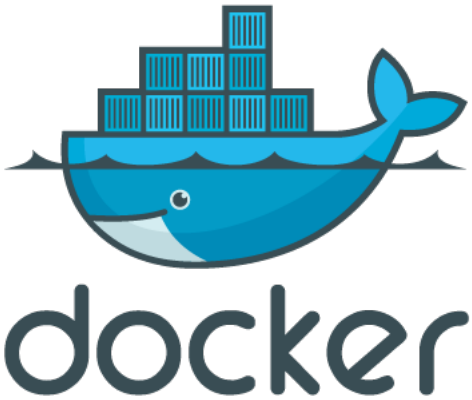
University of Groningen

Software Patterns
Team 2

Pattern-based Recovery & Evaluation: Docker



Authors:
Putra, Guntur
Fakambi, Aurélie
Schaefers, Joris
Menninga, Wouter

Monday 11th January, 2016

Version 0.3

Authors

Name	E-Mail
Putra, Guntur	G.D.Putra@student.rug.nl
Fakambi, Aurélie	A.Fakambi@student.rug.nl
Schaefers, Joris	J.Schaefers@student.rug.nl
Menninga, Wouter	W.G.Menninga@student.rug.nl

Revision History

Version	Author	Date	Description
0.1	Menninga	12-12-15	Added stakeholders and concerns.
	Schaefers	11-12-15	Added stakeholder concern table
	Schaefers	11-12-15	Added key drivers
	Putra	13-12-15	First draft of introduction chapter.
	Fakambi	11-12-15	System context
0.2	Menninga	29-12-15	Improved stakeholders section
	Menninga	02-01-16	Added Client-Server pattern
	Putra	03-01-16	Revised the introduction.
	Menninga	03-01-16	More additions client-server pattern + added security to stakeholders
	Putra	03-01-16	Added layers pattern documentation.
	Menninga	04-01-16	Added Plugin pattern.
	Menninga	04-01-16	Some improvements to System Context.
	Schaefers	04-01-16	Changed the names of the stakeholders concerns to include both the main- and sub quality
	Schaefers	04-01-16	Added image showing the relation between the stakeholders and the concerns
	Schaefers	04-01-16	Changed the key drivers, added references, changed the explanations
0.3	Putra	08-01-16	Revised introduction
	Menninga	09-01-16	Finished/improved Plugin and Client-Server
	Menninga	09-01-16	Improvement stakeholder concerns
	Putra	10-01-16	Revised layers pattern
	Menninga	10-01-16	Client server in logical view
	Menninga	10-01-16	Added proxy pattern
	Putra	10-01-16	Added few text of logical view and raw intro of evaluation
	Menninga	10-01-16	Security as key driver
	Fakambi	10-01-16	Shared Repository,Event-Driven and Direct Authentication Patterns
	Menninga	11-01-16	Intro for pattern documentation
	Schaefers	11-01-16	Fixed the .lib to make the references resolve again
	Schaefers	11-01-16	Added a start to the logical view in the architecture
	Schaefers	11-01-16	Figure updated about docker registry in chapter 5

Contents

List of Figures

List of Tables

1 Introduction

This document presents architecture recovery of Docker[1] by identifying software patterns and performing an evaluation of the architecture based on the identified patterns. This document is part of the Software Pattern assignment at the University of Groningen.

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux [16].

The structure of the document is explained as follows. Chapter 2 gives brief explanation with regard to Docker. Chapter 3 elaborates on the stakeholders involved in the Docker project and its corresponding key-drivers. Patterns discovered in the Docker project are documented in chapter 5. Evaluation is presented in chapter 6. Chapter 7 gives several recommendation for the Docker project. Lastly, a conclusion is drawn in chapter 8.

This project utilizes the IDAPO[2] process to recover the architecture [17]. The PBAR[3] approach is used to perform the evaluation [12].

There are 12 steps in IDAPO. Those steps are (1) identifying the type and domain of the product, (2) identifying used technologies, (3) studying used technologies, (4) identifying candidate patterns, (5) reading patterns literature, (6) studying the documentation, (7) studying the source code, (8) studying components & connectors, (9) identifying patterns and variants, (10) validating identified patterns, (11) getting the feedback from community, and (12) register pattern usage.

Chapter 2 and Chapter 3 correspond step (1) of IDAPO. Step (2) to (5) are related to Chapter 4, where used technologies are presented in the architecture. Chapter 5 corresponds to step (6) to (12), where the result of those steps is the pattern documentation.

While IDAPO recovers the patterns inside Docker, PBAR process determines the high-level process of pattern-based recovery and the evaluation of the architecture. The evaluation is documented in Chapter 6.

[1] `https://www.docker.com/`
[2] Identifying Architectural Patterns in Open Source Software
[3] Pattern-Based Architecture Reviews

2 System Context

In this chapter the context of Docker is presented.

2.1 System Context

In this subsection an overview and global information about Docker project and history are presented.

Docker is a open-source-software designed with the purpose of making the deployment of distributed application easier. 'Docker provides an integrated technology suite that enables development and IT operations teams to build, ship, and run distributed applications anywhere'[7].

Docker helps to automate the deployment of applications inside so-called 'containers' by making use of operating-system-level virtualization.

Operating-system-level virtualization is a virtualization method where the kernel of an operating system allows the isolation of multiple instances of the user space of the operating system. Leveraging this type of virtualization over full virtualization (i.e. hardware emulation) offers a significant performance increase, because there is no need for emulation of the hardware.[18]

Figure 2.1 illustrates the difference between virtualization using hardware emulation and operating-system-level virtualization. As opposed to a virtual machine, a container does not have a guest operating system. Additionally, it does not need a hypervisor for emulating virtual hardware and trapping privileged instructions. This saves a lot of performance overhead.

One tool that makes use of operating-system-level virtualization is Docker. Docker can be used to package applications and dependencies in a 'container'. Running this container using Docker will run it within a separate user space using operating-system-level virtualization.
Docker containers are instances of running Docker images. The process of creating a Docker image is relatively uncomplicated. It involves the creation of a so-called 'Dockerfile', which is a type of Makefile, containing instructions that are to be executed to install the application and dependencies (using the `RUN` instruction), and copy relevant data into the container (using `ADD`). Every Dockerfile extends another image (using the `FROM` instruction), which is either a previously created image or a base image (often containing the files of a specific operating system).

Docker images use a layered file system. Every command which is executed during the build-process creates

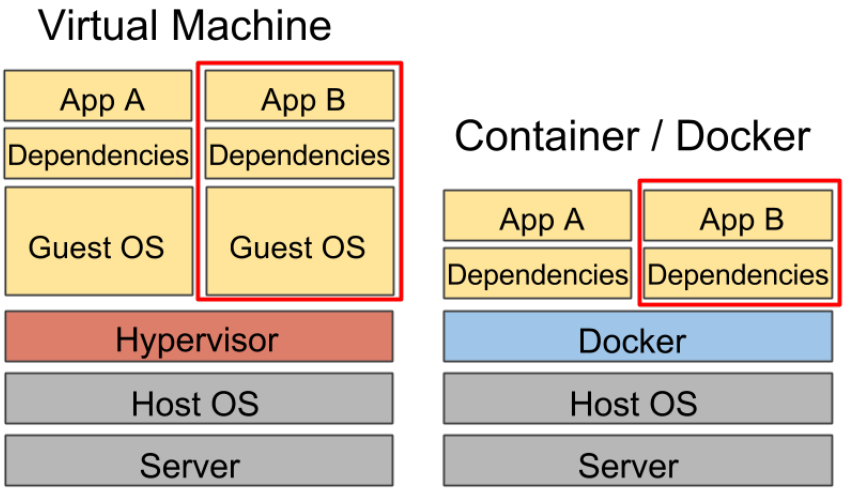


Figure 2.1: Virtual machines compared to containers[15]

a new layer, where all modifications to the file system are stored. Docker supports exporting images to tar-archives, which can be imported between similar versions of Docker.

The first open-source version of Docker was released in March 2013. Docker is written in the Go Programming language

2.2 Community

In this subsection the developpers programming docker and the community around them are presented.
The creator of Docker is Solomon Hykes.

Docker started as an internal project within dotCloud, a platform-as-a-service company with four main developers.
The following organizations are the main contributors to Docker: the Docker team, Red Hat, IBM, Google, Cisco Systems and Amadeus IT Group.

Docker has an active community and the managers encourage developpers around the world to contribute. *https://docs.docker.com/opensource/project/who-written-for/*

There are several ways to get in contact with the community of Docker:

- IRC – Here, the most knowledgeable Docker users reside. At the `#docker` and `#docker-dev` groups on `irc.freenode.net`.
- Google Groups – The developers are active in the `docker-dev` Google group.
- Stack Overflow – Questions about Docker itself can be asked on the Stack Overflow.

3 Stakeholders and Concerns

This section defines all stakeholders of the Docker system and describes the concerns of these stakeholders. A stakeholder is a person, group of persons, or organization that are involved in Docker. This section uses the quality attributes as described in ISO-25010[14].

3.1 Stakeholders

This section presents the stakeholders and their concerns. We list the main concern in bold and, where applicable, the specific concern in parenthesis.

For Docker, we identified the following stakeholders:

- Software developers
- Software Maintainers
- Cloud providers
- The open container initiative
- Docker developers
- Plugin developers

Software Developers

The developers are those that use Docker to create their software systems. Docker is used by developers to deploy software and create architectures consisting of multiple containers. Developers using Docker for their system, expect it to work according to the documentation and have a good usability.

Concerns

Usability	Since the software developers are using Docker, they want it to have good usability. This means it is easy to learn how to use Docker and it is not hard to work with. In fact, this is one of the main benefits of Docker, since it makes it easier to containerize applications, which could be done before, but was very difficult to do in practice.
Functional Suitability *(Functional correctness)*	Software developers care about the functional correctness, since bugs in the Docker software makes the development of their software more difficult.

Software Maintainers

Software maintainers are responsible for deploying the software and keeping the software product running. Docker is often used for software deployment (especially to the cloud). The software maintainers expect Docker to be reliable, portable, efficient and secure.

Concerns

Reliability	Software maintainers are responsible for keeping the software working while it is deployed. They will only use Docker if it is reliable. It has to be tolerant against failing containers.
Portability	The software maintainers want to be able to run Docker and its containers on a variety of different environments.
Performance efficiency	The software maintainers want Docker to have a good performance. They want the resources of their servers to be used as efficiently as possible and therefore want Docker to have almost no overhead, like Virtual Machines often do have.

Security	Software maintainers want Docker to be secure, so they can run their applications inside the containers without Docker negatively affecting the security of their application.

Cloud providers

There are numerous cloud providers offering services which are based on Docker[1]. These cloud providers offer Container-based cloud computing, sometimes referred to as CaaS (Containers-as-a-Service).

Concerns

Portability *(Installability)*	The cloud providers want to integrate Docker into their cloud computing architecture.
Compatibility *(Co-Existence)*	Docker has to share the environment with the existing architecture of the cloud providers.
Reliability	The customers using the cloud providers' services expect a good reliability. If cloud providers are to implement Docker in their architectures, they need Docker to have good reliability.
Security	Cloud providers want to offer a secure cloud computing environment to their customers.

The Open Container Initiative

The Open Container Initiative[2] was formed with the purpose of creating an open industry standard for container formats and runtime in June 2015.

They are interested in creating a formal, open, industry specification around container formats and runtime. This specification should be independent of particular clients/orchestration stacks, commercial vendors or projects and should be portable across a wide variety of operating systems, hardware etc.

Docker has donated its container format and runtime, known as 'runC' to this initiative and is one of the members of the initiative, together with a lot of other members (including competing technologies, such as 'rkt' from CoreOS)[3].

Concerns

Portability	The main goal of the initiative is to standardize the container format and runtime used also by the Docker project. This allows users of Docker to use the Docker containers with other container runtimes.
Compatibility *(Interoperability)*	Docker has to be able to work with the containers from the initiative.

Docker developers

The Docker developers are the developers contributing to the Docker code base. Some of these developers work on Docker in their free time, others work on it as part of their job at a company with an interest in Docker and some developers work for Docker Inc itself[2].

Concerns

Maintainability *(Modifiability)*	These developers contribute new features and bugfixes to the existing code base. Therefore, they want the project to have good modifiability, such that additions and improvements can be realized without to much effort.
Security	The Docker developers care about the security of the product they are working on.[4]

[1] `https://www.docker.com/partners#/service`
[2] `https://www.opencontainers.org/`
[3] A list is available at `https://www.opencontainers.org/about/members`
[4] `https://github.com/docker/docker#security-disclosure`

Plugin developers

Docker allows extending its capabilities with plugins[5]. These plugins are created by the plugin developers.

Concerns

Portability *(Adaptability)*	The developers of the plugins want to extend the functionality of Docker in an effective and efficient way.

Overview

The stakeholders and their related concerns is visualized in Figure 3.1 below.

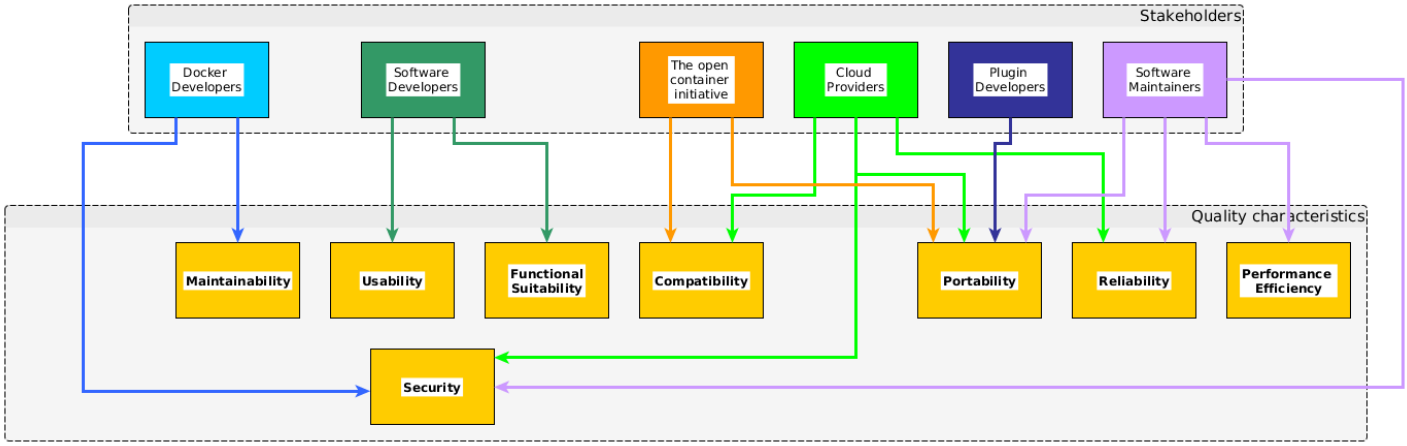


Figure 3.1: Stakeholders and the quality attributes they are concerned about

3.2 Key Drivers

Docker has the following key drivers

- KD- 1 – Portability
- KD- 2 – Security
- KD- 3 – Reliability

3.2.1 Portability

Looking at Figure 3.1, which visualizes the quality concerns of the stakholders, it is apparent that portability is the quality that is most desired among the stakeholders. One of the main features, if not the main feature of Docker, is to provide "agility and control for Development and IT Operations teams to build, ship, and run any app, anywhere"[3].
Docker wants the containers running the applications to be "hardware-agnostic" and "platform-agnostic" [10], meaning that Docker containers do not know anything about the hardware, nor the platform it is run on, thus making it very portable. This allows developers to focus on the actual application, without having to worry about the underlying hardware or platform. The applications the developers create could then be run everywhere where docker is supported.
To provide this quality attribute, Docker is also focused on increasing the performance. By making the Docker containers small, yet having a high performance, the containers become usable by a wide variety of systems and are able to replace the need to use any virtual machines that would otherwise be used to achieve this level of portability[10].
To increase the portability even more, the Docker Hub makes it very easy for Docker containers to be shared and used on other machines.

[5] `https://blog.docker.com/2015/06/extending-docker-with-plugins/`

3.2.2 Security

The Docker daemon executes software inside containers and exposes an API. If somebody would gain unauthorized access to this API, this attacker would be capable of executing any program on the vulnerable host machine. Therefore, security is an important key driver for Docker.

Additionally, since Docker does not use separated operating systems, but instead operating-system-level virtualization, the security and isolation of the containers is a serious concern if software inside the containers is not trusted. For example, for a cloud provider running containers for clients, isolation of these containers is an important aspect of the security, since clients should not be able to access data of other clients.

3.2.3 Reliability

The main function of a Docker container is to run a certain application. However, a system usually needs to run a set of different applications and it should not be the case that running these applications in Docker compromises the systems reliability.
Dependencies often lead to problems[10], decreasing the reliability of software. The Docker files are text files containing a small set of instructions that will run an application. This means that the instructions needed to run an application are conveniently located in a single place, the Dockerfiles. This allows viewing and the dependencies of all the different applications and making them easily maintainable.
Docker increases the reliability of the applications it runs, by increasing the testability of the containers and applications.
To make sure Docker itself stays reliable, it provides a test framework and test functions that thoroughly test the Docker software.

4 Software architecture

This section discusses the architecture of Docker. Three views from the 4+1 Architectural View Model will be presented : the logical, the development view and the process view.

4.1 Logical View

Docker is made using the the go language created by Google. Docker is an application categorized as “Operating-system-level virtualization”. It consists of:

- 2979205 lines of go code (allmost 3 million).
- 1782 ’.go’ files
- 267 different packages

If the main “/docker” package is build, the dependencies can be visualized using a visualization tool. When restricting the depth to be 1, it produces the image below, Figure 4.1. The green node is the "main" node for which the dependencies are shown.

Figure 4.1: A high-level overview of the Docker architecture.

Looking at the official documentation, their image (Figure 4.2) shows to be largely similar.

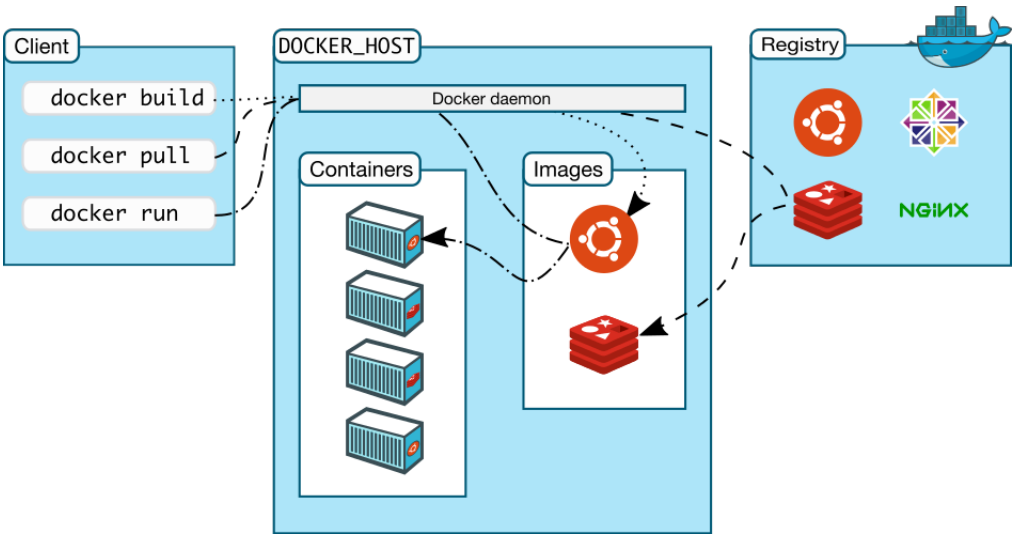


Figure 4.2: A high-level overview of the Docker architecture. Source: [6].

Docker uses a client-server architecture. The client (a command-line tool) acts as the primary user interface and talks to the Docker daemon. The daemon is a background process which dfig:dockerdep1oes all the heavy lifting, e.g. the building and running of the containers.

The most important part of docker will be discussed in the following sections below.

4.1.1 Client

The client is a small binary and acts as the primary user interface to Docker. The user enters command into the client and the client forwards these commands to the Docker daemon, which executes commands. The Docker client is capable of connecting to daemons running on the local machine, as well as connecting to daemons running on remote machines over the internet.

4.1.2 Server / Daemon

The Docker daemon is a process running on the host machine (as can be seen in Figure 4.2). This process is started when the host machine starts and runs in the background. It exposes a REST interface and listens for requests coming from clients on the same or remote hosts.

4.1.2.1 Images

Docker images can be interpreted as a read-only template from which Docker containers are started. A Docker image consists of a stack of layers which are bonded by union file systems. These layers support reusability and sharing.

Docker images can be build from text files ('Dockerfiles'), which contain instructions like installing software and copying files.

4.1.2.2 Containers

Docker containers are based on Docker images. This image acts as a read-only layers. Then, Docker container adds a thin writable layer on top of it to perform operations. Docker containers basically consist of the files of the operating system, user-added files, application, and some information. Multiple containers may run based on a same image. In this case, Docker will not create separate copies for each containers. Instead, they will share same image with their own writable thin layer.

To run Docker containers, Docker uses the `libcontainer` library, which is part of the Open Container Initiative.

4.1.3 Registry

A Docker registry is a place where Docker images can be stored and retrieved. The Docker daemon fetches desired Docker images from this repository. This registry may be a public or a private one, such as one behind firewall. Docker Hub[1] is one example of a docker registry hosted by Docker. A registry provides an easy way to distribute images between different hosts.

4.2 Process View

This section discusses the system processes, how they communicate and the runtime behaviour of the system.

4.2.1 Plugins

Docker supports extending its capabilities using third-party plugins. At the moment of writing, Docker allows extending the functionality of the volumes and network subsystems, which are responsible for storing files outside containers and inter-container communication over the network respectively. This will be extended in the future[4].

A plugin runs in its own separate process and communicates with the Docker daemon using the REST API. In order for the Docker daemon to know about the plugins existence, a file has to be placed in a pre-defined directory.

In Figure 4.3 the process of using a plugin can be seen.

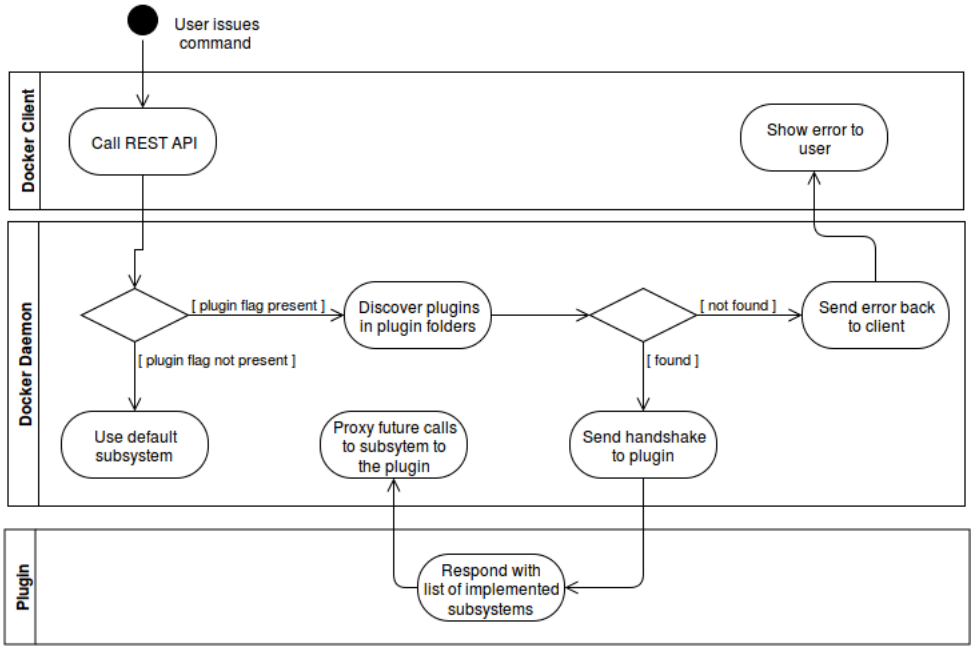


Figure 4.3: An activity diagram showing the process of using a plugin (simplified)

A user indicates that he/she wants to use a plugin by passing a command-line parameter to the Docker client when starting a container. If this parameter is present, then the daemon will start looking for the plugin in the plugin directory. If the plugin is not found, it returns an error to the user. When the plugin is found, the daemon will send a handshake to the plugin using a UNIX socket and the plugin will reply with a list of subsystems it implements. The Docker daemon will then use the `PROXY` pattern to forward calls to this subsystem to the plugin process.

[1] `https://hub.docker.com/`

5 Pattern Documentation

This chapter describes the patterns used in Docker identified by us. We documented the patterns as described by Harrison et al.[11], with added information about where the pattern was found (traceability).

5.1 Client-Server

Traceability

The Client-Server pattern can be deducted from the online documentation[6].

The Client-Server pattern can be deducted from the 'What is Docker's architecture?' section of the online documentation[6].

Source

Architectural patterns revisited – a pattern language, P. 29 [1]

Issue

It should be possible for Docker containers to be controlled remotely and a single interface should be able to control containers on multiple hosts (e.g. in the cloud).

Additionally, certain operating systems lack the underlying technologies necessary for running containers. For those OSes, it should be possible to call remote daemons running on operating systems which are supported.

Solution

Docker uses a `CLIENT-SERVER` architecture. The client, a binary supplying a command-line interface, act as the primary interface for the user. The user enters commands into this client, which are then send to a server: the Docker daemon.

Assumptions/Constraints

- The versions of the client binary and the server binary should match. Different versions can cause problems.
- All the services offered by the daemon have to be made available to the client using a REST interface.

Rationale

The daemon is a background process, which supplies the requested services to the client. The daemon exposes a REST interface.

For Docker, the client can be configured to connect to other daemon processes than the one running on the local machine. It can be configured to connect to remote Docker daemons as well, allowing the user to issue commands to daemons running remotely.

By separating the client and server it is possible to use the same client to issue commands to different daemons, running on different hosts. It is also possible to use the client on operating systems that do not support running containers.

Implications

The use of the `CLIENT-SERVER` pattern results in two different executable binaries: a daemon and a client.

The use of the `CLIENT-SERVER` pattern increases the interoperability, since the client can send commands to daemons running on remote machines and the local machine.

Additionally, the portability is increased, since the client can run on Operating Systems that cannot run containers themselves.

Related Patterns

5.2 Layers

Traceability
Layers is mentioned in the Docker architecture in the online documentation, specifically in the explanation of Docker images[6].
Furthermore, the user guide of Docker also elaborates on layers in the Docker images[5].

Source
Architectural patterns revisited – a pattern language, P. 29 [1]

Issue
Docker emphasizes on developing containerization with almost no overhead. Running multiple instances of Docker containers with the same image may end up in high overhead. A sharing and reuse mechanism must be implemented to prevent overhead and to save more space/resources.

Assumptions/Constraints
An application running in a Docker container must have a CLI[1] (bash), as Docker images are based on a Linux distribution.

Solution
Each Docker image consists of a stack of layers, which is read-only. The first layer in the stack is a base image from a Linux distribution. Docker also implements copy-on-write strategy. In this strategy, system processes that need the same data share the same instance of data rather than having their own. A copy will be made when a specific system process makes a change to the data (copy-on-write).

Rationale
Layers are very good in terms of sharing and reusability. In this way, overhead can be avoided by sharing resources. A new image that uses same stack of layers may reuse the layers rather than making a separate copy. This may result in smaller sizes of Docker images.

Implications
A Docker container will use Docker image as the base of it. This will remain as read-only layers. Then, a Docker container will create another thin writable layer on top of the image. Multiple containers that run based on the same Docker image will not make their own copy of the image to prevent duplication.

This will end up in more efficient memory allocation (better performance efficiency).

Related Patterns

- Client-server
- Shared repository

5.3 Shared repository

The docker registry is considered as a Shared Repository.
Two alternatives exist:DockerHub and Docker Trusted Registry.

Docker has a feature wich can be configured : the Notification Sytem which makes the Shared Repository an active Repository. *https://docs.docker.com/registry/notifications/*

Traceability
The Shared Repository pattern can be deducted from the online documentation : *https://docs.docker.com/registry/*
"The Registry is a stateless, highly scalable server side application that **stores** and lets you **distribute** Docker images. A registry is a storage and content delivery system."

Source
Architectural Pattern Revisited - A Pattern Language, P.13 [1]

Issue
Docker provided a way for the user to conrol the storage and distribution of images.
The user wants to be alerted of new events happening in the registry through notifications.

[1]Command Line Interface

Solution
Users interact with a registry by using docker push and pull commands.

Rationale
After the integration of the Shared Repository Pattern each has one central repository containing their images and they can it access using a loggin.

Related Patterns
Event-driven Messaging

Related Quality Attributes/Key drivers
Reusability, Changeability, Maintainability, Integrability

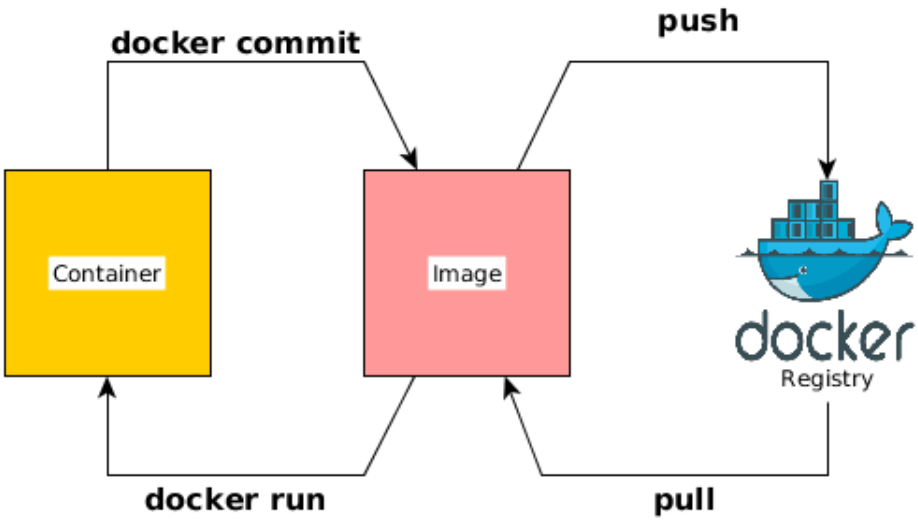


Figure 5.1: Docker Registry

5.4 Event-Driven Messaging

Traceability
The notification mechanism of the Docker Registry uses the Even-Driven Messaging Pattern.

Source
http://soapatterns.org/designpatterns/eventdrivenmessaging

Issue
The user wants to be alerted about certain events occuring in his registry. The notification system needs a mechanism in order to send the events to the user.

Assumptions/Contrainst
This pattern is used at a lower level in the Docker Registry/Active Repository Pattern.

Rationale

> "Notifications are sent to endpoints via HTTP requests. Each configured endpoint has isolated queues, retry configuration and http targets within each instance of a registry. When an action happens within the registry, it is converted into an event which is dropped into an inmemory queue. When the event reaches the end of the queue, an http request is made to the endpoint until the request succeeds. The events are sent serially to each endpoint but order is not guaranteed."

Related Patterns
Active Repository Pattern

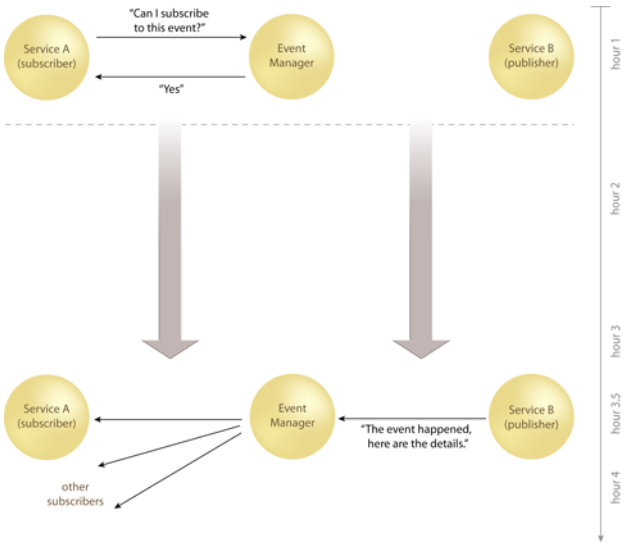


Figure 5.2: Event-Driven Messaging

5.5 Direct Authentication

Traceability

The Direct Authentication pattern can be deducted from the source code through the "auth.go" file from the Docker Github Repository.
The notification mechanism of the Docker Registry uses this pattern.

> Since securing access to your hosted images is paramount, the Registry natively supports TLS and basic authentication.

Source

https://msdn.microsoft.com/en-us/library/ff647715.aspx

Issue

The Docker Registry needs an authentication system in order to ensure security for the user through a login.

Solution

By using the Direct Authentication the access to the Docker Registry is controlled for each user.

Rationale

Related Patterns

Shared/Active Repository Pattern

Related Quality Attributes/Key drivers

Security

5.6 Plugin

Traceability

The existence of Docker Plugins becomes apparent from it's documentation at [4].

Additionally, the directories `docker/pkg/plugins/` [2] and `docker/daemon/graphdriver/plugin.go` [3] (among others) in the project's repository contain the code for discovering plugins and the interfaces the plugins should implement.

[2] `https://github.com/docker/docker/tree/master/pkg/plugins`
[3] `https://github.com/docker/docker/blob/master/daemon/graphdriver/plugin.go`

Source
Patterns of Enterprise Application Architecture, P. 499 [9]

Issue
The users of Docker want to have customization, by extending Docker with third party custom-built tools. This customization means that third parties should be able to write plugins that extend Docker's core functionality.[13]

The implementation of such plugins is only available at runtime.

Assumptions/Constraints

- Plugins can only extends the functionality of the components of Docker, that have an interface that plugins can implement.

Solution
Docker uses the Plugin pattern to link the implementation of the interfaces of several extendable components with third-party implementation at runtime.

Rationale
Docker discovers plugins by looking for .sock, .spec or .json files in the plugin directories on the host system. These files describe how Docker can communicate with the plugins using the REST API (usually via a Unix socket).

The plugins themselves run as separate processes on the same host as the Docker daemon and implement an HTTP server listening for requests from the Docker daemon. After a user requires a plugin (this is indicated e.g. as a command-line parameter when starting a container using the Docker client) Docker uses the discovery algorithm (see also Section 4.2.1). After that, Docker sends a handshake to the plugin and the plugin returns a list of which subsystems this plugin implements.

For these subsytems Docker will replace the default implementation by a Proxy, that forwards all calls over the REST interface to the plugin process.

Implications
The use of the Plugins pattern means that the adaptability increases.

Related Patterns

- Proxy

5.7 Proxy

Traceability
The use of the `PROXY` pattern becomes apparent from the source code in the repository on GitHub. For example, the proxy for the Volumes plugin can be found in `docker/daemon/graphdriver/proxy.go` [4].

Source
Pattern-oriented Software Architecture - Volume 4, P.290 [8]

Issue
The plugins are separate processes than the daemon and are only available at runtime. Therefore, it is impossible for the daemon process to access the services of the plugins directly.

Assumptions/Constraints

- The plugins have to implement a server listening for requests from the daemon.

Solution
Let the daemon only communicate with the plugins through a proxy. This proxy implements all the 'housekeeping' functionality, like sending API requests and authentication. It has the same interface as the plugin.
Whenever a call is done from the daemon to the plugin, it goes via the proxy, which communicates this call using a REST API to the plugin process.

[4] `https://github.com/docker/docker/blob/master/daemon/graphdriver/proxy.go`

Rationale

Using the PROXY pattern allows the daemon to communicate with the plugins, without requiring direct access to these plugins.

Also, because the subsystem component has the same interface as its proxy, the implementation of software using this component does not depend on whether the proxy is used or the original subsystem.

Implications

The use of the PROXY pattern allows communication with the plugins, which increases the extendability. Because the communication with the process is not direct, there is some performance overhead.

Related Patterns

- Plugin

6 Evaluation

This chapter presents the evaluation of documented patterns and overall system against the key drivers of the architecture which already presented in Chapter 3. PBAR[12] method is used to carry out the evaluation. Force Resolution Maps is also utilized to present the result.

6.1 Patterns

6.1.1 Client-Server

6.1.2 Layers

6.1.3 Shared repository

6.1.4 Event-Driven Messaging

6.1.5 Direct Authentication/Brokered Authentication

6.1.6 Plugin

6.1.7 Proxy

6.2 Overall system

7 Recommendations

8 Conclusion

8.1 Pattern-Based Architecture Reviews

8.2 IDAPO

8.3 Final Words

A Time Tracking

Week 1

Person	Task	Hours
Putra	Coaching session, researching about Docker, working on first draft of introduction	3.5
Fakambi	Coaching session, Research about Docker, Work on the system context	3.5
Schaefers	Coaching session, setting up initial layout and new git repository, defining key drivers, research.	6.5
Menninga	Coaching session, stakeholders and concerns	4.0

Week 2

Person	Task	Hours
Putra	Catching up after winter break, reading Docker docs, revising intro, and writing layers pattern documentation	6
Fakambi	Coaching session, Meeting for discussing the patterns	3
Schaefers	Creating scripts to find patterns, building and running docker, creating various dependency graphs, changing the key driver section, adding stakeholders and concern image	8.5
Menninga	Searching patterns, improving stakeholders, client-server and plugin patterns. Improvements to System context.	7.5

Week 3

Person	Task	Hours
Putra	Coaching session, reading docker documentation and source code, revising intro and layers pattern, small changes in logical view and evaluation	14.5
Fakambi	Coaching Session, small modifications System Context, Work on the Shared Repository, Event-Driven and Direct Authentication	12
Joris	Researching and generating dependency graphs. Coaching session (via skype). Adding a software architecture begin.	11
Menninga	Coaching session, improvements after feedback, process/logic view, client-server, plugin, proxy, changed key drivers	15

Bibliography

[1] Paris Avgeriou and Uwe Zdun. Architectural patterns revisited – a pattern language. 2005.

[2] Chris Dawson. Who are the docker developers? `http://thenewstack.io/who-are-the-docker-developers/`, 2014. [Online; accessed 9th January 2015].

[3] Docker. Docker - build,ship, and run any app, anywhere. `https://www.docker.com/`, 2015. [Online; accessed 13-December-2015].

[4] Docker. Docker - understand docker plugins. `https://docs.docker.com/engine/extend/plugins/`, 2015. [Online; accessed 3rd January 2015].

[5] Docker. Docker - understand images, containers, and storage drivers. `https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/`, 2015. [Online; accessed 10th January 2016].

[6] Docker. Docker - understanding the architecture. `https://docs.docker.com/engine/introduction/understanding-docker/`, 2015. [Online; accessed 30th December 2015].

[7] Docker. What is docker? `https://www.docker.com/what-docker`, 2016. [Online; accessed 3-January-2016].

[8] K Henney F Buschmann and D.C. Schmidt. *Pattern-oriented Software Architecture - Volume 4*. John Wiley & Sons, Ltd, 2007.

[9] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[10] GitHub. Docker - the open-source application container engine. `https://github.com/docker/docker`, 2015. [Online; accessed 13-April-2015].

[11] N. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *Software, IEEE*, 24(4):38–45, July 2007.

[12] NeilB. Harrison and Paris Avgeriou. Using pattern-based architecture reviews to detect quality attribute issues - an exploratory study. In James Noble, Ralph Johnson, Uwe Zdun, and Eugene Wallingford, editors, *Transactions on Pattern Languages of Programming III*, volume 7840 of *Lecture Notes in Computer Science*, pages 168–194. Springer Berlin Heidelberg, 2013.

[13] Adam Herzog. Extending docker with plugins. `https://blog.docker.com/2015/06/extending-docker-with-plugins/`, 2015. [Online; accessed 3rd January 2015].

[14] ISO. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. `http://iso25000.com/index.php/en/iso-25000-standards/iso-25010`, 2011.

[15] W.G. Menninga. Benchmarking Tool for the Cloud. Bachelor Thesis, University of Groningen, 2015.

[16] Maureen OǴara. Ben golub, who sold gluster to red hat, now running dotcloud. `http://maureenogara.sys-con.com/node/2747331`, 2013. [Online; accessed 13-December-2015].

[17] Klaas-Jan Stol, Paris Avgeriou, and Muhammad Ali Babar. Design and evaluation of a process for identifying architecture patterns in open source software. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 147–163. Springer Berlin Heidelberg, 2011.

[18] John Paul Walters, Vipin Chaudhary, Minsuk Cha, Salvatore Guercio Jr, and Steve Gallo. A comparison of virtualization technologies for hpc. In *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*, pages 861–868. IEEE, 2008.