BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CONTAINERIZATION OF DATABASE DETECTORS

KONTEJNERIZACE DETEKTORŮ NAD RELAČNÍMI DATABÁZEMI

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MICHAL OBERREITER**
AUTOR PRÁCE

SUPERVISOR **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Oberreiter Michal**
Program: Informační technologie
Název: **Kontejnerizace detektorů nad relačními databázemi**
Containerization of Database Detectors
Kategorie: Analýza a testování softwaru

Zadání:

1. Prostudujte projekty db-reporter a db-detectors v rámci platformy Testos pro detekci dat v relačních databázích. Nastudujte technologii Docker.
2. Analyzujte požadavky na zapouzdření detektorů dat nad databázemi do Linuxových kontejnerů. Navrhněte řešení kontejnerizace detektorů v technologii Docker.
3. Implementujte rozhraní REST API k zapouzdřeným aplikacím. Navrhněte pro uživatele přehledný výstupní formát detekce a implementujte export výsledků detektorů do tohoto formátu.
4. Správnost funkcionality podpořte automatizovanými integračními testy.

Literatura:

- Kropáč, F.: Nástroj pro analýzu obsahu databáze pro účely testování softwaru. 2017. Bakalářská práce FIT VUT v Brně.
- Ochodek, M.: Nástroj pro analýzu obsahu databáze pro účely testování softwaru. 2017. Bakalářská práce FIT VUT v Brně.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstract

This thesis deals with containerization of command-line applications including containerization of existing tools for database content analysis. The thesis is a part of Testos platform, which aims at software testing automation. The goal was to design and implement a solution that would be both universally usable for command-line applications and at the same time flexible enough to accommodate database detectors and their specific requirements. Docker was chosen as the containerization platform, on which a management system was built. This system provides both a graphical user interface and an application programming interface. The result allows for easy application management and output retrieval. The primary contribution of this thesis is the streamlining and simplification of running command-line applications with specific dependencies. These features come in form of abstracting the underlying mechanisms and providing a graphical user interface.

Abstrakt

Tato práce se zabývá kontejnerizací aplikací pro příkazové řádky, konkrétně pak analyzátorů obsahu databáze. Práce je řešena v kontextu platformy Testos, která cílí na automatizaci softwarového testování. Cílem řešení je navrhnout a implementovat univerzálně použitelný nástroj, který by také vhodným způsobem řešil specifické požadavky databázových detektorů. Pro účely kontejnerizace byl zvolen nástroj Docker, nad kterým byl postaven zapouzdřující systém. Dále bylo vytvořeno webové uživatelské rozhraní komunikující s API. Výsledné řešení umožňuje snadno spravovat aplikace příkazové řádky a získávat z nich relevantní výstupy. Přínosem této práce je usnadnění práce s aplikacemi, které vyžadují své specifické závislosti. Usnadnění spočívá v zapouzdření specifik nástroje Docker pod obecnější model práce a také ve vytvoření uživatelsky přívětivého grafického rozhraní.

Keywords

containers, containerization, microservices, Testos, Docker, REST, .NET Core, Flask

Klíčová slova

kontejnery, kontejnerizace, microservices, Testos, Docker, REST, .NET Core, Flask

Reference

OBERREITER, Michal. *Containerization of Database Detectors.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšířený abstrakt

Nutnost vytvářet portabilní softwarová řešení ve věku cloudových řešení a on-demand služeb se značně zvyšuje. Vývojáři či výzkumníci v mnoha případech nejsou schopni replikovat korektní chování daných nástrojů ve svém prostředí a jsou nuceni zdlouhavě diagnostikovat tyto problémy. Tyto a mnohá jiná úskalí jsou důvodem, proč kontejnerizace nabývá na významu.

Tato práce se zabývá kontejnerizací aplikací pro příkazové řádky a snaží se tak usnadnit spouštění těchto aplikací. Jejím cílem bylo navrhnout a implementovat univerzálně použitelný nástroj, který by také vhodným způsobem řešil specifické požadavky analyzátorů obsahu databáze – detektorů a zároveň poskytoval grafické i HTTP rozhraní. Tento nástroj – *Detection Containers* (*DeCon*) – je řešen v kontextu platformy *Testos* vyvíjeného na Fakultě Informačních Technologie Vysokého Učení Technického v Brně. Testos cílí na automatizaci softwarového testování a jeho nástroje se snaží kombinovat různé úrovně testování – od jednotkového po akceptační. Nástroje *db-detectors* a *db-reporter*, které jsou součastí platformy Testos, jsou předmětem kontejnerizace a integrace.

Při návrhu systému DeCon byl kladen důraz na zjednodušení práce s kontejnerizovanými aplikacemi. Abstrakce spočívá ve vytvoření modelu práce podobnému testovacím případům a spuštěním. Obdoba testovacích případů v DeConu jsou *configurations* a *jobs* jsou ekvivalentem testovacích spuštění. Jako kontejnerizační systém byl použit *Docker*, který mimo jiné poskytuje robustní rozhraní jak z příkazové řádky, tak i HTTP. DeCon byl navrhnut právě na základě těchto Docker kontejnerů. Samotný DeCon je pak koncipován jako sada navzájem komunikujících *microservices*. Oproti *monolitickým aplikacím* se ty, které užívají principu microservices, vyznačují lepší škálovatelností, jasnějším oddělením závislostí a také možností kombinovat více technologií v jednom softwarovém řešením.

Klíčovými komponenty – službami – řešení jsou *Configuration service* a *Job service*. První zmíněná zajišťuje správu nastavení pro jednotlivé kontejnerizované aplikace, zatímco druhá orchestruje jejich spouštění a poskytuje informace o získaných datech a aktuálním stavu. Spouštění *jobs* je realizováno skrze *Docker service*, která abstrahuje jednotlivá vybraná volání do Dockeru. *Docker service* zpracováním požadavku spouští samotný Docker kontejner, ve kterém okamžitě startuje tzv. *Application wrapper*, který zajišťuje odchytávání výstupu (*standard out* i *standard error*) z dané zapouzdřené aplikace. Zachycený výstup je zasílán do *Job service*, kde je uložen do databáze. Zprávy o chybách či nevalidních datech jsou zasílány do *Logging service*, která tyto záznamy ukládá paralelně do textového souboru i databáze. Přístup k těmto službám je realizován pomocí *Gateway*, která odděluje privátně a veřejně dostupná volání služeb a také poskytuje funkcionalitu přepínání editovatelnosti *configurations*. Tato vlastnost byla vyžadována z důvodu potenciálního využití pro veřejné demonstrační účely.

REST API poskytované službou *Gateway* je konzumované webovou aplikací, jejímž účelem je poskytnout uživatelsky přívětivou správu systému DeCon. Aplikace klade důraz na maximální jednoduchost a snaží se nebýt překážkou v možném budoucím týmovém workflow, tudíž neimplementuje autentifikaci ani autorizaci. Uživatelské rozhraní jako jediné obsahuje specializace pro databázové detektory (všechny služby jsou stavěné obecně) ve formě přizpůsobeného zadávání parametrů a exportu zpracovaných výsledků ve formátu *JSON*. Pro každou zapouzdřenou aplikaci uživatelské rozhraní nabízí možnost náhledu stavu aplikace s automatickou aktualizací, přehledem a exportem surových výstupů. V přehledu je možné filtrovat i pomocí regulárních výrazů.

K dosažení maximální flexibility v možnosti zapouzdření různých aplikací DeCon implementuje podporu pro publikování portů z kontejnerů, specifikaci složky k zpřístupnění,

nastavení časového limitu pro násilné ukončení kontejneru a také dodání vlastního Dockerfile pro instalaci potřebných závislostí aplikace.

Služby DeConu jsou implementovány v multiplatformním frameworku *.NET Core* s využitím jazyka C# (*Configuration service*, *Job service*, *Logging service*, *Application wrapper*) a také *Flask API* v Pythonu (*Gateway*, *Docker service*). V rámci výběru implementační technologie byly zohledněny faktory, jakými jsou například: množství aplikační logiky, požadavky na práci s vlákny a asynchronní operace atd. Webová aplikace využívá technologie *React* v jazyce JavaScript.

Výsledek práce byl podroben automatizovanému testovaní s užitím různých druhů testů. Pro otestování kódu na úrovni bloků byly vytvořeny jednotkové testy. Testování na úrovni služby je realizováno komponentními testy, jejichž vytvoření bylo možné díky využitý principů vkládání závislostí a programování vůči rozhraní. K otestování celkové funkcionality posloužily integrační testy, které komunikovaly přímo s *Gateway* a testovaly tak systém z pohledu vnějšího aktéra.

Implementovaný nástroj DeCon splňuje kladené požadavky a skýtá potenciál pro budoucí širší nasazení v rámci platformy Testos.

Containerization of Database Detectors

Declaration

I hereby declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Aleš Smrčka, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Michal Oberreiter
May 14, 2019

Acknowledgements

I would like to express my gratitude for the assistance and support that I have received from my supervisor Ing. Aleš Smrčka, Ph.D. Also I would like to thank him for the time dedicated to the regular Testos meetings.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

In the age of cloud computing, importance of creating environment independent solutions becomes more apparent. Oftentimes developers manage to get their tools working on their local machines but others struggle to reproduce expected behavior due to unforeseen differences in these environments. Also, the increased difficulty of testing out any dependence-heavy application contribute to the rise of containerization.

This thesis aims to provide a solution for containerization of database detectors and other command-line applications by building a container management system on top of Docker. This system *DeCon* – Detection Containers – offers ability to easily setup and run user-specified command-line applications via an included graphical user interface or an application programming interface. DeCon is tailored to the specific requirements of the Testos platform. Users do not need to possess any prior knowledge of the Docker platform for performing basic tasks in DeCon. However for advanced users, DeCon offers a great deal of customizability in terms of application dependencies.

Containerization of database detectors is just one of the few included demonstration examples that aims to highlight the features of DeCon. Database detectors are treated as any other command-line application everywhere, except the web user interface, where customized controls and result parsing is added. Results collected from database detectors can be exported for additional processing.

In order to properly define and describe technologies used to implement this system, Chapter 2 lists related topics which include virtualization, containerization, Docker platform, web APIs and microservices. Chapter 3 discusses design decisions made during the design process of DeCon and attempts to give a technology-independent description of the system. The specifics of the implementation are explained in Chapter 4, where author reasons technology choices, gives examples of how the technologies were used and describes the measures that were undertaken to verify the functionality.

1.1 Glossary

DeCon	Detection Containers
Testos	Test Tool Set platform developed at FIT BUT [22]
db-detectors	Database detectors by Marek Ochodek [14]
db-reporter	Database reporter by František Kropáč [10]
API	Application programming interface, see Section 2.4
Configuration (DeCon)	Entity based on which jobs are spawned
Job (DeCon)	Entity holding information about application run

Chapter 2

Background

In this chapter, the author describes basic concepts of virtualization while highlighting its modern-day usage in enterprise solutions. Specifically, author tries to present an ecosystem-wide overview of the Docker platform while noting differences between the implementations across different operating systems. Author views these topics as important to this thesis subject.

Description of virtualization in comparison to containerization in Section 2.1 is needed to convince the reader of the importance of containerization today's application development and deployment. Section 2.2 deals with Docker description and its technical implementation. Docker's cross-platform implementation similarities and differences provide a view into possible future integration over different platforms. HTTP protocol is outlined in Section 2.3. Section 2.4 deals with web APIs and tries to briefly describe the basics of these omnipresent technologies, while underpinning the fundamentals for understanding the inner workings of DeCon. Service-oriented architecture and microservices in Section 2.5 help to understand design principles and choices that have been made when designing DeCon.

2.1 Virtualization

Virtualization is a process of running a virtual instance of a computer system in a layer abstracted from an actual hardware [15]. It is commonly used for running multiple simulated and isolated environments on a single system. Three main types of virtualization are:

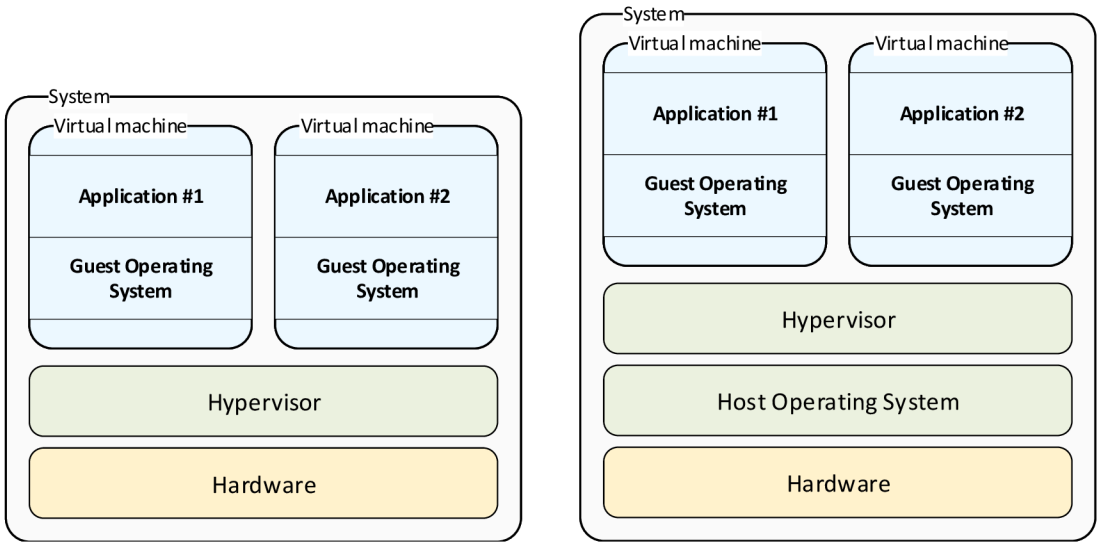
- full virtualization
- paravirtualization
- operating-system-level virtualization

2.1.1 Full Virtualization

In case of full virtualization, a virtual machine[1](VM) simulates enough hardware to allow for running the operating system inside the VM [26].

A *hypervisor* (or a virtual machine monitor – VMM) [16, pg. 413] is used to provide a layer between the host and guest environments and connection to the actual hardware via the host system. Hypervisor is responsible for creating and running virtual machines.

[1]Emulation of a computer system.



(a) Bare-metal (b) Hosted

Figure 2.1: Comparison of hypervisor types

Any program or operating system running inside a VM should exhibit the same behavior as if it was run on the native system and given the same system resources [16, pg. 413]. Hypervisors, as shown in Figure 2.1, can be classified into two types [16]:

- Type-1 – native hypervisors

  This type of hypervisors (also called bare-metal hypervisors) runs directly on the host's hardware, meaning that instructions are executed without any dependency on the host OS; thus minimizing a potential attack surface. Examples include: Microsoft Hyper-V[2], VMware ESX/ESXi[3], Xen[4], XboX One system software[5] etc.

- Type-2 – hosted hypervisors

  Hypervisors of this type use the host operating system to execute instructions; therefore degrading the guest system performance by introducing latency caused by instruction interpretation. This type is also vulnerable to threats caused by security issues of the host system. Examples of Type-2 hypervisors: VMware Workstation[6], VirtualBox[7], QEMU[8], bhyve[9], KVM[10] etc.

Virtualization enables many use-cases with which a regular user comes into contact without even realizing. With the emergence of cloud computing, which builds upon securely isolated environments sharing the same hardware, virtualization became a focus point from security and performance perspectives.

[2] https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/
[3] https://www.vmware.com/cz/products/esxi-and-esx.html
[4] https://xenproject.org/
[5] https://www.xbox.com
[6] https://www.vmware.com/cz/products/workstation-pro.html
[7] https://www.virtualbox.org/
[8] https://www.qemu.org/
[9] http://bhyve.org/
[10] https://www.linux-kvm.org/page/Main_Page

Cloud computing technology is built around features of virtualization [28]. *Platform as a Service* (PaaS) model utilizes this technology to run client-provided applications on a vendor's virtualized platform. Similarly, *Infrastructure as a Service* (IaaS) model lets clients manage an operating system running inside a virtual machine. PaaS and IaaS services are provided by many enterprise vendors such as: Amazon (AWS), Microsoft (Azure) or Google. The model known as *Software as a Service* (SaaS) provides the whole package (infrastructure, platform and software) as a cloud service. Regular users may come across SaaS when using cloud storage services, such as Dropbox[11] or cloud-enabled software applications like Office 365[12] or Google Suite[13].

Paravirtualization is a technique that aims to improve performance and efficiency compared to full virtualization by modifying the guest OS kernel to replace non-virtualizable instructions with calls communicating directly with the hypervisors' virtualization layer [23, pg. 5]. Figure 2.2 outlines the system architecture with multiple paravirtualized guest systems. For example the Xen project hypervisor supports paravirtualized Linux kernel apart from supporting full virtualization.

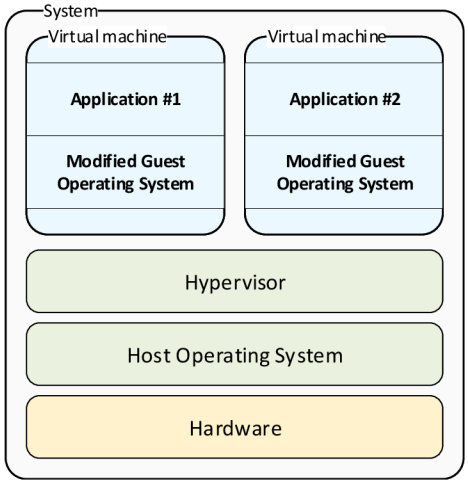


Figure 2.2: Paravirtualization

2.1.2 Containerization

Containerization or operating-system-level virtualization is a feature of operating system's kernel that allows for existence of multiple isolated user spaces [24], see Figure 2.3. A container is then an isolated user space, which shares the host operating system's kernel but has a restricted access to the resources of that host. Advantages of containerization in comparison to full virtualization are [2, pg. 2]:

- slimness – container does not include the OS, so it offers a higher environment density
- quick start – since containers share kernel with the host system, no booting is required
- performance – sharing kernel gives containers performance of a native application

[11] https://www.dropbox.com/
[12] https://www.office.com/
[13] https://gsuite.google.com/

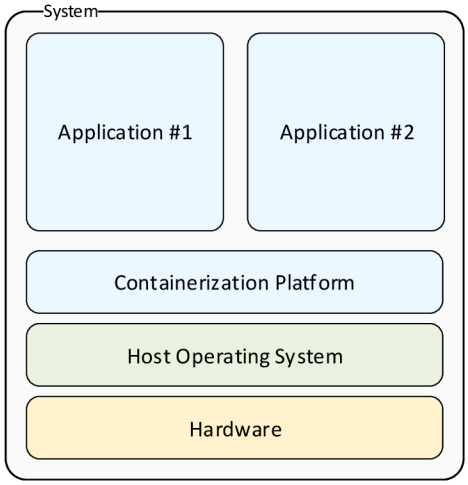


Figure 2.3: Containerization

However, the main concern with containers is their security, since sharing kernel with the host system may allow an attacker to gain access or compromise the system from inside the container using a security issue inside the shared kernel [2]. An another disadvantage also comes from sharing the kernel – containers cannot run applications which are not compatible with the host kernel.

Containers' primary use-case is to create and deploy lightweight encapsulated environments that are independent of the host system configuration or currently installed libraries and versions. Also, resource allocation and management can be easier with containers compared to VMs [27, pg. 5]. Over the years many containerization solutions have been released. Those released include:

- chroot[14] (1982) (change root) is a command on UNIX systems that allows for changing root directory for the current process and its children.
- FreeBSD jail[15] (2000) improves upon chroot by providing virtualized access to the file system, users and networking. Jailed processes cannot break free on their own, however an unprivileged user on the host can in cooperation with the jailed user can obtain elevated access in the host environment.
- OpenVZ[16] (2005) is a containerization technology that focuses on sharing resources of a physical server across multiple isolated environments called *Virtual Private Servers* or *Virtual Environments*.
- LXC[17] (2008) (Linux Containers) and its wrapper LXD are technologies for creating and managing isolated environments – containers. Prior to Docker v0.9 it was used as and underlying container management technology but since was replaced by Docker's own *runc*[18] project.

[14] http://man7.org/linux/man-pages/man2/chroot.2.html
[15] https://www.freebsd.org/doc/handbook/jails.html
[16] https://openvz.org/
[17] https://linuxcontainers.org/
[18] https://github.com/opencontainers/runc

- Docker[19] (2013) aims to provide a high-level solution for mainly application containerization targeted at developers and DevOps. Unlike other competitors, Docker is available across multiple platforms (Linux, FreeBSD, Windows, macOS). Further description and technical overview is available in Section 2.2.
- rkt[20] (2014) (Rocket) is a competing solution to Docker, which hopes to solve Docker's privilege issues (Docker Engine runs as the `root` user) by allowing for more control by an unprivileged user, as well as image signing by default. Rocket is also able to fetch, convert and execute existing Docker images.

The aforementioned technologies by themselves only aim at running containers on a single physical machine and not deploying and running them across compute clusters consisting of multiple nodes. This issue is solved by *orchestration* which aims to abstract the host infrastructure and make deploying to a cluster environment transparent to users [4]. Nodes across the cluster are managed by the scheduler, which orchestrates the whole cluster as shown in Figure 2.4. Container orchestration is a process that automates deployment, management, scaling, networking, and availability of container-based applications.

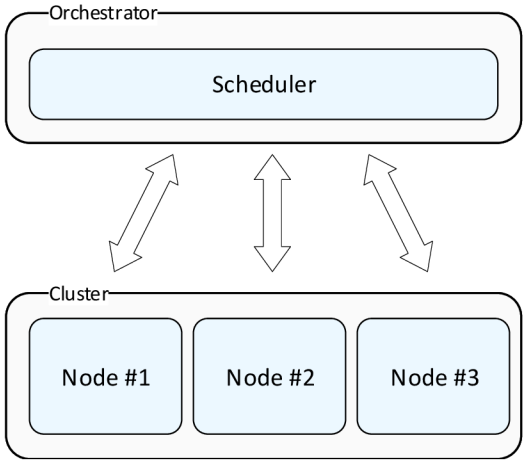


Figure 2.4: Orchestration

Orchestration solutions include:

- Docker Swarm[21] – a native Docker solution for managing container deployments to clusters. Each machine inside a cluster hosts a full Docker Engine which is controlled from a swarm manager.
- Kubernetes[22] – originally a Google project aimed at providing a platform for automating deployment and scaling of containers. Kubernetes can be integrated with Docker containers by overriding the default Docker orchestrator – Docker Swarm.
- Amazon ECS[23] (Elastic Container Service) – a scalable, high-performance orchestration service for Docker containers available on Amazon's own cloud computing platform AWS (Amazon Web Services).

[19] https://www.docker.com/
[20] https://coreos.com/rkt/
[21] https://docs.docker.com/engine/swarm/
[22] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[23] https://aws.amazon.com/ecs/

2.2 Docker Platform

Docker is a software tool for creating and managing containers[24]. In contrast to full virtualization, Docker performs virtualization on the operating system level, meaning that all containers share the same operating system kernel. Docker as a tool was first released in 2013. It is commonly used in conjunction with orchestration tools, such as Docker's own Swarm tool or Kubernetes.

2.2.1 System overview

Docker as a platform is composed of three integral components: client, host and registry. Figure 2.5 illustrates the interaction between these components. Docker Engine (also Docker Daemon) is the backbone of Docker, it manages all images and containers related operations. Docker Engine can *pull* remote *images* from a registry and run them locally or used them to create new user-defined images. Clients connect to Docker Engine via the provided API.

A Docker image is a template for instantiating containers. Given image is either pulled from a registry or can be built by specifying a *Dockerfile*. A Docker container is an instance of an image, which can be started and managed. Container management is performed via a Docker command-line interface (CLI) or an application programming interface (API). In its default configuration[25] all changes made to the container during runtime are lost upon container removal. The isolation level can be changed, so that network connections and/or sharing files with the host system is enabled. Container outputs (standard output and standard error) are stored to logs via a logging driver which determines a storage method.

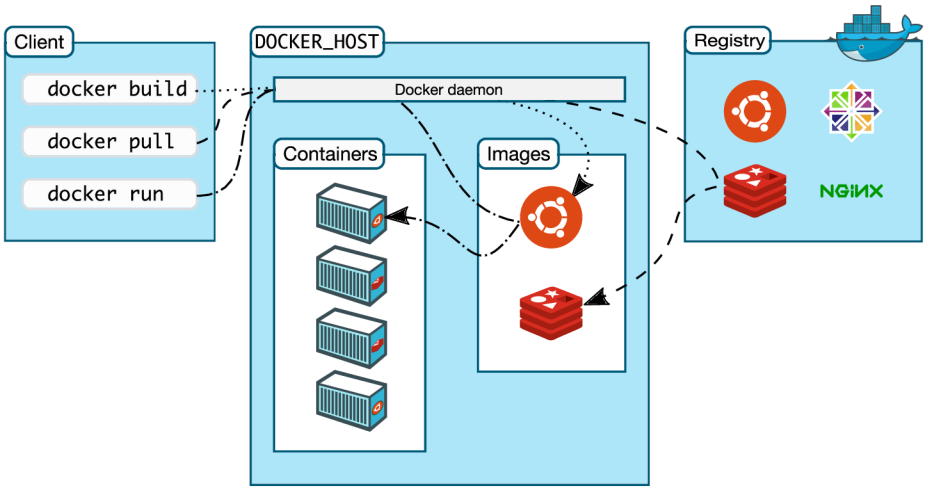


Figure 2.5: Docker overview, source: [5]

Dockerfile is a text file which defines instructions to be executed against a parent image. Instructions provided extend the parent image. Each instruction represents a layer. Layers are read-only (during runtime) except the last one which is called *container layer*. An example of the layer system in Docker is demonstrated in Figure 2.6. This layer is write-enabled – it stores all the changes made during container runtime.

[24] https://www.docker.com/

[25] A volume can mounted, then files in the volume are shared between the host and the container.

Dockerfile must specify either an *entry point* (a program or a script that will be always executed by `run` or `start` command) or the `cmd` instruction which can be used for specifying entry point arguments and also for setting an overridable entry point[26].

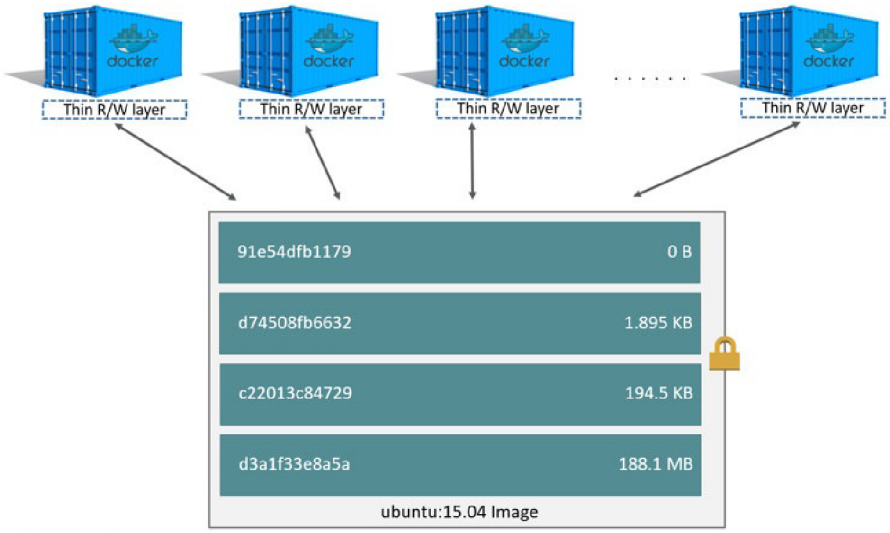


Figure 2.6: Container layers, source: [6]

If an image has no parent image then it is called a base image. Unlike base images, parent images are distinguished by having `FROM` directive in their own Dockerfile.

2.2.2 Technical overview

Docker uses a number of technologies to deliver containerization features across multiple platforms. As depicted in Figure 2.7, on Linux-based systems Docker Engine uses *runc*[27] to spawn and run containers in accordance with the *Open Containers Initiative* (OCI) standard. Container management is done through *containerd*[28], which handles image operations, storage and network management. These tools enable cross-platform and cross-engine container compatibility. In the past, Linux Containers (LXC) were used for container management (prior to the release of Docker v0.9). In case of Linux-based systems, Docker uses many kernel features such as these:

- *namespaces* – encapsulates a global system resource in a way that is invisible to a process within the namespace. Namespaces used by Docker Engine:
  - `pid` – process isolation
  - `net` – networking
  - `ipc` – interprocess communication
  - `mnt` – mount points
  - `uts` – Unix Timesharing System – isolation of kernel and version identifiers

[26] Overriding can be performed for example with `run` command.
[27] https://github.com/opencontainers/runc
[28] https://containerd.io/

- *cgroups* – a feature that organizes processes into hierarchical groups allowing for hardware resource management (e.g., memory and CPU usage)
- *UnionFS* variants – union mount file systems enable image layering by presenting multiple file systems as one virtual directory (e.g., *AUFS*' branches translate to Docker's layers). Docker can use multiple different storage drivers, such as *AUFS*, *btrfs*, *vfs* or *DeviceMapper*. These drivers use stackable image layers and copy-on-write technique.

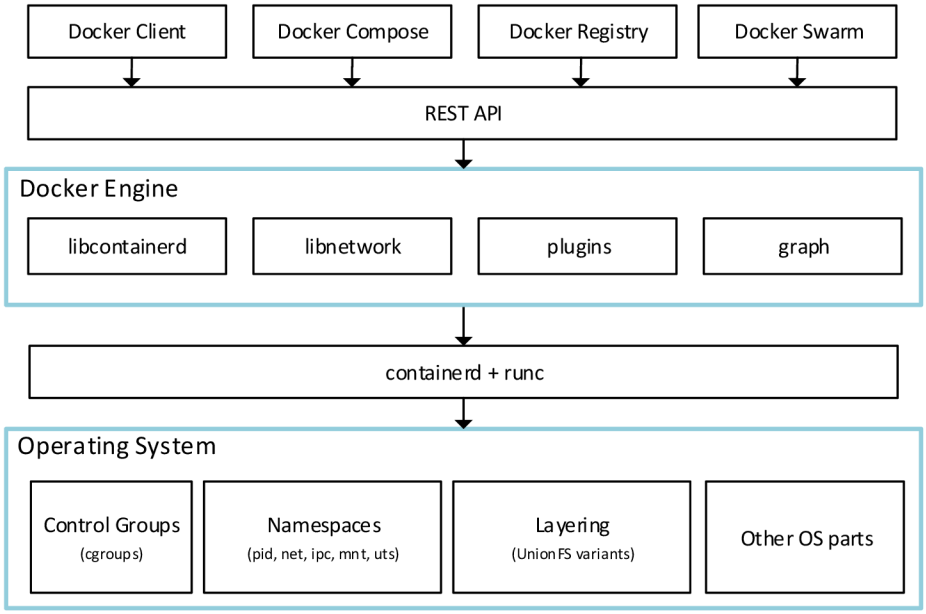


Figure 2.7: Docker architecture on Linux

2.2.3 Docker security

Security in the Docker ecosystem is important in order to prevent attackers from gaining control or damaging the host system from inside the container. On Linux-based systems security can be hardened using *Linux Security Modules* (LSM) [2]. Docker supports AppArmor and SELinux LSMs which both provide Mandatory Access Control (MAC).

MAC is an access control system where access to all resource objects is mandated by a central authority – system administrator – and cannot be overridden, unlike in Discretionary Access Control (DAC), where access permissions can be changed by users.

SELinux[29] (Security-Enhanced Linux) is originally a US National Security Agency (NSA) project which was later picked up by the SELinux community. Access control is implemented via labels which are present on every system object (e.g., file, directory, process). The role of system administrator is then to define associations between processes and system objects.

The relationship between Docker and SELinux revolves around securing isolation between containers and isolation from the host. SELinux's *Type Enforcement* rules are based on a process type label, which restricts read/write operations on some system objects inside

[29] https://selinuxproject.org/page/Main_Page

containers. Other SELinux feature used by Docker is *Multi-Category Security* (MCS) enforcement which is able to isolate containers from each other by creating unique container identifier on startup.

AppArmor[30] is an LSM that uses MAC system to restrict program's access to resources. It is currently maintained by Canonical.

Unlike SELinux which uses labels, AppArmor's behavior is defined by profiles, which limit process capabilities. AppArmor supports two modes of behavior: *complain* and *enforcement*. In the complain mode, all policy violations are permitted but also logged. In contrast, the enforcement mode prohibits these violations. The complain mode can be used for defining new or customizing existing profiles [2, pg. 6]. Docker uses AppArmor to deny access to key parts of the host kernel. If no profile is specified, Docker uses its default profile.

Capabilities on Linux provide a fine-grained control over permissions; thus eliminating the need for the `root` user in cases where only a specific subset of permissions is needed. Capabilities are a per-thread attribute. Some of the capabilities are:

- `NET_ADMIN` – network administration
- `SYS_ADMIN` – system administration
- `SYS_TIME` – time manipulation
- `WAKE_ALARM` – system wake up

Capabilities can be used with Docker to give containers additional permissions. When `privileged` flag is supplied to `docker run` command, container will run in privileged mode, which gives it by default number of capabilities (`SETPCAP`, `AUDIT_WRITE`, `NET_RAW`, `KILL` etc). Additional capabilities can be provided by `cap-add` and dropped by `cap-drop` as seen in Listing 2.1.

```
$ docker run --cap-add=ALL --cap-drop=MKNOD ...
```

Listing 2.1: Example of `run` command with capabilities

Docker Registry[31] is a scalable server-side application which hosts and enables distribution of Docker images. The Docker ecosystem has its own public registry – Docker Hub[32] – which hosts many[33] official images from authors themselves. Registries can also be hosted on private servers.

Docker Machine[34] is a part of the Docker ecosystem that allows for Docker Engine hosts management. It is used for provisioning Docker hosts on remote systems (installing Docker Engine, configuring client etc.). Alternatively, it can serve as a way to run Docker Engine on non-compatible Windows and MacOS versions, which was in the past the only way to run Docker on non-Linux operating systems.

2.2.4 Interfaces

Docker Engine API (also called Docker REST API and Docker Remote API) exposes the Docker Engine functionality via HTTP based API as shown in Figure 2.8. On top of the

[30] https://gitlab.com/apparmor/apparmor/wikis/home/
[31] https://docs.docker.com/registry/
[32] https://hub.docker.com/
[33] For example: Ubuntu, MySql, NGINX, MongoDB, NodeJS
[34] https://docs.docker.com/machine/overview/

Engine API Docker provides a command-line interface which serves as a wrapper of the API calls. Most of the command-line calls map directly to the API ones, with exception of `docker run` command which consists of `create` and `start` API calls. Listings 2.2 and 2.3 depict a difference between these types of calls[35].

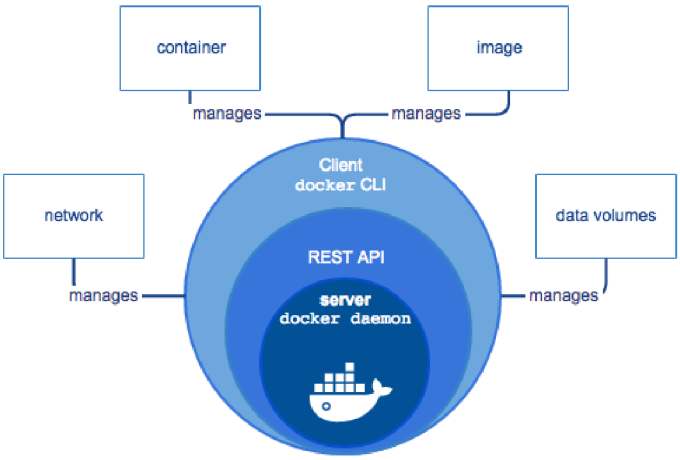


Figure 2.8: Docker components, source: [5]

```
$ docker build . -t api
$ docker run api
```

Listing 2.2: Docker's command-line interface

```
POST /build
POST /containers/create
POST /containers/start
```

Listing 2.3: Docker's HTTP interface

Docker Compose[36] is a tool in the Docker ecosystem for creating and managing multi-container environments. Definition of the composure is done by specifying a *YAML* file `docker-compose.yml`, example of which can be seen Listing 2.4. A Dockerfile usually contains list of services, volumes and networks and their respective configurations. Services can specify features such as port exposures, dependency on other services[37], commands to be executed etc. Docker Compose is typically used to host web applications with their databases. In such environment one container hosts the web application and the second one is used for a *database management system* (DBMS). These two are by default connected to their internal network, on which both can be accessed by their respective service names (e.g., `mongodb://mongodb:27017`).

[35] It should be noted that command-line interface also has the `create` and `start` commands.

[36] https://docs.docker.com/compose/overview/

[37] Dependency ensures the startup order of services, however does not wait for the actual service to be ready.

```
version: "3"
services:
  web:
    build: .
    ports:
       - '${WEB_PORT}:8080'
    depends_on:
        - db
  db:
    image: mongodb
```

Listing 2.4: Dockerfile example

2.2.5 Docker on Windows

Docker on Windows (DoW) can run both native Windows containers and Linux containers. Each of these types has its specifics and can use multiple technologies to achieve the desired level isolation and performance.

Windows containers can be categorized by their level of isolation. First type is *Windows Server Containers* which architecturally resembles container technology present on Linux-based systems. As a replacement for `containerd` and `runc`, Windows implements *Hosted Compute Service* (HCS) for low-level container manipulation (Figure 2.9). HCS is also able to create Hyper-V isolated containers [20].

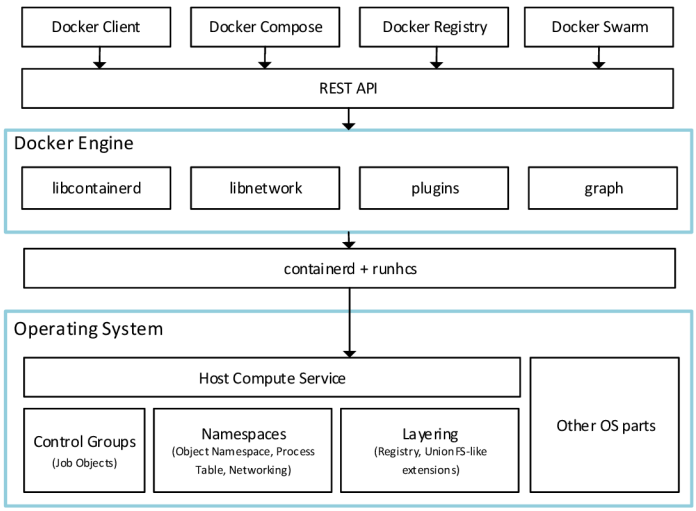


Figure 2.9: Docker architecture running natively on Windows

Unlike Windows Server Containers, the second type – *Hyper-V isolation* – runs containers inside a virtual machine; therefore gaining security benefits of a virtual machine, while still retaining some advantages of containers.

Windows containers are based on the *Nano Server* or *Server Core* images. Both images are derivatives of the Windows Server operating system, though Nano Server is much more slimmed-downed version, claiming 93 % lower virtual hard-disk size and 80 % fewer reboots than Server Core [21].

Docker on Windows is also capable of running Linux containers. This is beneficial in cases when users want to use[38] Docker inside *Windows Subsystem for Linux*[39], which at the moment does not natively support Docker. The only way is to connect a Docker client to the Docker on Windows via an HTTP bridge.

Linux containers can run on Windows in two different ways [3]. First of them consists of having a full *Moby* virtual machine (Docker's own virtual machine inside Hyper-V, i.e. Linux container host) whose kernel is shared with all Linux containers. In this mode only a chosen type of containers can be run at the same time and a reconfiguration is needed to switch between Linux and Windows containers.

For use-cases when Windows and Linux containers need to run at the same time or Hyper-V isolation is needed between the individual Linux containers (and not only on the Linux container host level), users can choose to enable experimental feature (as of Spring 2019) to run Linux containers directly without Moby VM, as shown in Figure 2.10. This feature is called *Linux containers on Windows* (LCOW).

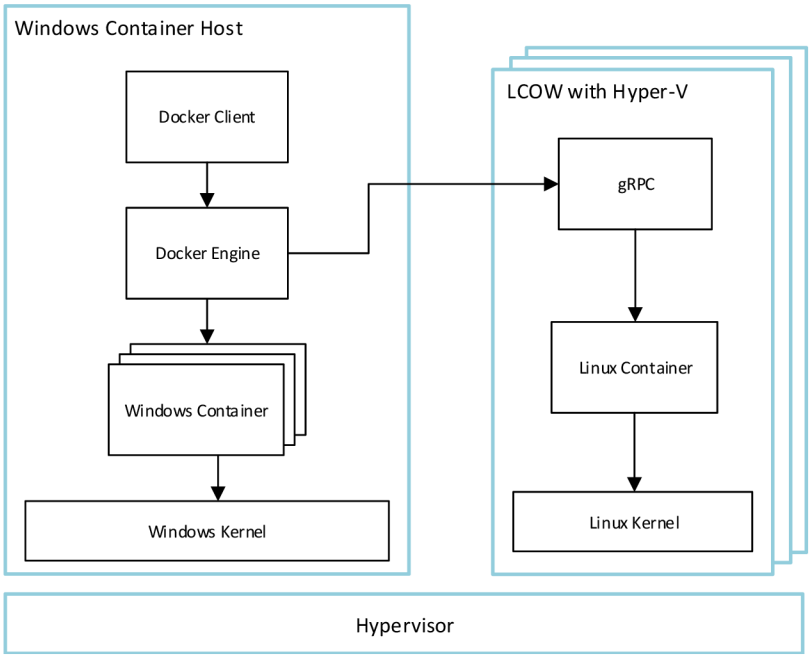


Figure 2.10: DoW concurrently running Windows and Linux containers using Hyper-V

This approach spawns a new Hyper-V isolated environment for each container; thus sandboxing the container inside its own virtual machine. A Linux kernel with minimal dependencies is present inside the VM, which performs container management via receiving calls through *gRPC* – a remote procedure call framework[40]. The embedded Linux distribution is built using *LinuxKit*[41].

[38] Most of DeCon was developed using Docker on Windows.
[39] https://docs.microsoft.com/en-us/windows/wsl/about
[40] https://grpc.io/
[41] https://github.com/linuxkit/linuxkit

2.2.6 Docker on MacOS

Similar to Windows, Docker for Mac uses virtualization to run Linux containers. In case of MacOS, Docker includes its own hosted hypervisor *Hyperkit*[42], which is based on *bhyve* – a BSD hypervisor[43]. The distribution of Linux inside the virtual machine is also based on LinuxKit.

2.3 Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems [8].

HTTP is a stateless protocol running in a client-server model. *Requests* made by clients are processed and *responses* are sent by servers. Each request has to adhere to the format specified by the HTTP standard [8, pg. 35]. A request is composed of a request-line (HTTP *method*, URI of the resource and HTTP version), headers, an empty line and an optional message body. Listing 2.5 shows an example of such request. An HTTP method indicates the desired action to be performed for a given request. Table 2.1 depicts an example of HTTP methods mapping to the CRUD[44] operations.

HTTP method	CRUD operation
GET	Read resource
POST	Create resource
PUT	Update/replace (complete resource needed)
PATCH	Update/modify (only changes required)

Table 2.1: Example of HTTP methods mapping to CRUD

```
POST /api/values HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Host: www.testos.org
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

name=John&surname=Doe
```

Listing 2.5: Example of an HTTP request

Each response contains a status-line (HTTP version, HTTP *status code* and *reason phrase*) followed by zero or more headers, an empty line and an optional message body (Listing 2.6). A status code, a three digit number, indicates the outcome of the request and can be categorized into 5 groups: informational, success, redirection, client error and server error. Reason phrases give human-understandable information about status codes, for example: 200 – OK, 404 – Not Found.

[42] https://github.com/moby/hyperkit
[43] http://bhyve.org/
[44] CRUD – Create, Read, Update, Delete – basic operations with a resource.

```
HTTP/1.1 404 Not Found
Date: Fri, 10 May 2019 23:49:20 GMT
Server: Apache/2.4.39 (Win64)
Content-Length: 42
Connection: Closed
Content-Type: text/html; charset=iso-8859-2
```

Listing 2.6: Example of an HTTP response

2.4 Web APIs

Application programming interface (API) is a set of functions and procedures that allows for accessing features of an operating system, application, service or a software library. In a web environment, APIs can be used for manipulating documents, fetching data from the server, manipulating graphics [12] etc.

Many web APIs adhere to principle of REST. Thus it is important to present an overview of REST API principles, since this thesis deals with such APIs.

2.4.1 Representational state transfer

Representational state transfer (REST) is an architectural style that conforms to a set of constraints [7, pg. 76–85]. These constraints are:

- *Client-Server* – separation of concerns, meaning that user interface can be decoupled from the data storage; therefore improving portability, scalability and enabling independent redeployment of both components.
- *Stateless* – the client-server communication must be stateless – all data needed to understand a request must be a part of the request and no communication context can be stored on the server. Statelessness contributes to visibility (all data is in the request), reliability (no state can be corrupted on the server) and scalability (server does not need to keep any context for the individual clients).
- *Cache* – responses can be marked as cacheable or non-cacheable, so clients can reuse responses for later equivalent requests. This results in an improvement in efficiency, scalability and client performance.
- *Uniform interface* – uniformity of interfaces decouples implementations from the provided services. Such interface is defined by these constraints: resource identification (e.g., URI system), resource manipulation through representations, self-descriptive messages, **h**ypermedia **a**s **t**he **e**ngine **o**f **a**pplication **s**tate (HATEOAS, dynamic discoverability of links to other actions which hold contextual data – similar to links on a website).
- *Layered system* – constraints of layered system enable for existence of network-based intermediaries to be deployed between client and server in order to provide security, caching or load balancing features.
- *Code on demand* – an optional constraint for extending functionality by executing a downloaded code like Java applets or a JavaScript code.

Data elements of REST can be summarized to [7, pg. 89]:

- Resource – abstraction of information
- Resource identifier – typically a URL
- Resource metadata
- Representation – JSON, XML, HTML, JPEG etc.
- Representation metadata – content type, alternates
- Control data – usually HTTP headers

2.4.2 REST APIs

Web services can expose themselves through web APIs to clients. Web APIs that adhere to the principles of REST are called REST APIs, then a web service with a REST API is a RESTful service [11, pg. 6].

2.4.3 HTTP-based REST APIs

Though REST was outlined with web in mind, it is predominantly but not exclusively[45] used with HTTP. HTTP offers features such as URIs, HTTP methods, caching headers that directly map to properties of REST. Usage of HTTP with REST is based on conventions and design guidelines [11].

2.5 Service-Oriented Architecture

Service-oriented architecture (SOA) is an approach to software application design [9]. Instead of developing single monolithic application, multiple smaller components are used to achieve the same functionality. This improves maintainability and ensures better separation of concerns when implemented correctly.

A common pattern in SOA is an *enterprise service bus* (ESB), which handles point-to-point communication. ESBs attempt to decouple service from each other by a standardized way of communication. ESBs are in nature similar to message buses such as D-Bus[46] or Testos Bus[47]. Data storage is usually shared between all services, which limits the scalability of such storage.

SOA pattern is typically used in large enterprise solutions, where services can represent whole legacy applications or in cases where monoliths cannot be split due to the existing infrastructure.

2.5.1 Microservices

Microservices are a software development style for designing and running small loosely coupled autonomous services with bounded contexts – domain boundaries. Richards [17] defines microservices as a specific approach to the service-oriented architecture, because SOA in general does not provide specifics on how to split services so that the outcome produces

[45] See https://github.com/swagger-api/swagger-socket
[46] https://www.freedesktop.org/wiki/Software/dbus/
[47] https://pajda.fit.vutbr.cz/testos/testos-bus

desired benefits over a single monolithic application. In contrast to SOA, microservices limit sharing of data storage. Microservices are usually smaller in size, compared to SOA services. Main principles of microservices as stated by Newman [13, pg. 246]:

- hidden internal implementation details
- decentralized
- deployed independently
- failure isolation
- highly observable
- business concept as the focus point
- automation of tasks

Motivation for using microservices comes from the need for highly scalable and agile infrastructure and development. Traditionally used monolithic approach to building applications is not suited for solving these issues. Scaling a monolithic application means that the entire application host hardware needs to be scaled, while with microservices only specific services could be scaled.

Another big concern with developing monoliths is their high-risk deployment. With every code modification, the entire application has to be built, tested and then deployed. Therefore if large applications are in question, this process can be time consuming; thus reducing the ability to quickly iterate over new versions. Redeploying monolith also comes with a risk of breaking the functionality since many changes are put into production at once.

Deconstructing monolith into microservices also comes with the benefit that small developer teams can own the whole lifecycle (from development and testing to deploying) of their service. Having small autonomous services also makes them immune to implementation changes in other services. Microservices usually do not make use of a shared codebase as it is considered anti-pattern, since it creates a tight coupling among the services which defeats the purpose of microservices. This also means that *DRY*[48] principle is not strictly enforced across services.

Handling databases in microservices does not follow the same principles as in monolithic applications. To ensure loose coupling, a shared database should not be used, as it couples the connected services with current database schema and any change to the schema can potentially break the services. Instead database-per-service pattern is more suitable and better scalable [19], since many instances of the same service can be active at the same time.

The absence of a shared database prohibits the *ACID* (Atomicity, Consistency, Isolation, Durability) principles to be retained. This fact poses challenges summarized in Brewer's CAP theorem [1]:

- *Consistency* – consistency equivalent to having a single up-to-date copy of the data
- *Availability* – high availability of that data without guaranteeing that the information is updated

[48] DRY – Don't Repeat Yourself

- *Partitions* – tolerance to network partitions (delays or lost messages)

At most two of these properties can be applied to any distributed data system. In microservices, high availability directly contradicts consistency (e.g., multiple services involved in a transaction). Therefore *eventual consistency* is introduced, which provides *BASE* semantics: Basically Available, Soft state, Eventual consistency. BASE consistency is achieved via convergence that is usually implemented by data replication across the services.

The use of technology-agnostic inter-service communication protocols (e.g., HTTP) achieves decoupling any specific technology from the actual implementation. Language and framework choice can vary service by service. By using different technologies, developers can choose the one that the best.

By maintaining multiple small isolated services, errors and failures can be contained in a way that no other services will be affected; therefore preventing cascade failures. Microservices on its own do not guarantee this feature but a proper robust design can help with mitigating such system-wide failures. A technique of *circuit breakers* can be applied to „fail fast“ in case of repeated service failures so that failures do not cascade over to other services [18].

In real life, microservices are used for variety of applications, ranging from e-shops to on-demand video platforms, such as Netflix[49]. For implementing aforementioned features, mainly scaling, orchestration tools like Kubernetes in conjunction with Docker can be used.

2.6 Testos

Testos (Test Tool Set) [22] is a platform developed in Faculty of Information Technology at Brno University of Technology. Testos supports automation of software testing. Tools within the platform (Fig. 2.11) combine different levels of testing (from unit to acceptance testing) with different categories of testing, such as model-based testing, requirement-based testing, GUI testing, data-based testing and execution-based testing with dynamic analysis.

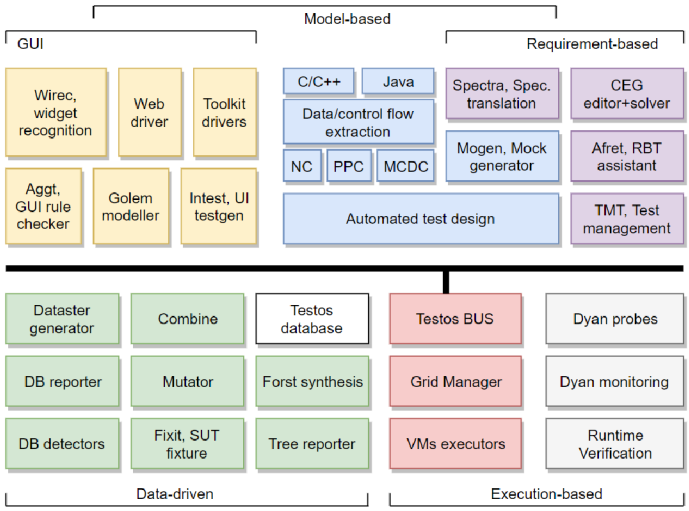


Figure 2.11: Testos platform, source: [22]

[49] https://www.netflix.com/

2.6.1 Database detectors

Database detectors or *db-detectors* is a tool for database content analysis authored by Marek Ochodek [14]. The goal was to detect data restrictions in an already created relational database by implementing a set of detectors for database exploration. Database detectors communicate with the database reporter which orchestrates the process of detection. As stated by the author, its current implementation is limited by only one database per program instance and suffers from poor error handling [14, pg. 33].

2.6.2 Database reporter

Database reporter or *db-reporter* a is tool for orchestrating database content analysis authored by František Kropáč [10]. Reporter aimed to provide means to organize, schedule and manage lifecycle of database detectors. Communication with the detectors is implemented using D-Bus[50] – a low level message bus. Kropáč states that the main shortcomings are: limit to one database per instance and lack of code documentation [10, pg. 31].

[50] https://www.freedesktop.org/wiki/Software/dbus/

Chapter 3

Analysis and Design

Design process of any software product poses challenges, requiring a precise and concrete analysis of existing solutions and requirements on the new solution. The need for analysis is further accentuated in cases where the solution includes numerous interacting parts, such as this thesis. In this chapter author attempts to offer an insight into the thought process concerning the architecture of this system – DeCon.

3.1 Design Goals

The original purpose of DeCon was to containerize database detectors and reporter using Docker containers. After discussions with the supervisor, the goal was extended to present a user-friendly means of interacting with containers and include the ability to run arbitrary command-line applications inside the containers. Author tried to follow design principles of microservices and general rules concerning design and implementation of REST APIs, so that the overall system adheres to the most recent trends in designing complex systems.

Since Testos is a collection of testing tools and, in the future, DeCon is expected to containerize some of these tools, it seemed only logical to design DeCon in resemblance to test cases and test runs. DeCon's *configurations* are counterparts to test cases and *jobs* are equivalent to test runs.

3.2 Target Product

The final product should be composed of multiple microservices that are, through a gateway, accessible via a web API. This API will be consumed by a simple graphical user interface, which should provide a user-friendly way to interact with DeCon. Users should be able to start and manage jobs, review, filter and download logs. Publicly facing DeCon API and the microservices should be designed without distinction between database detectors and other command-line applications. The only specialized part for the detectors will be the GUI, which should provide customizations to enable easier and more streamlined user experience. No knowledge of Docker should be required for basic operations. However, advanced users should be able to provide their own custom Dockerfiles. DeCon will adapt to hosting scenarios (local hosting or public server) when appropriate startup flags are specified.

3.3 Existing Solutions with Similar Functionality

Since DeCon builds on top of Docker, some similar solutions already exist. These solutions focus on providing a user interface on top of Docker, which, in principle, share similar functionalities with DeCon. However, it should be noted that DeCon is not merely a GUI to Docker containers. Its backend architecture, test case-like job execution and customized parsing are custom-tailored to the needs of Testos platform. In spite of this, author regards mentioning these existing solutions as important.

3.3.1 Portainer

Portainer[1] is an open-source Docker management tool available on Linux, Windows and MacOS. It runs directly on top Docker Engine API and exposes the Docker functionality through a graphical interface as shown in Figure 3.1. Portainer supports Docker features such as networking, volumes, secrets, Swarm mode etc. Feature-wise it covers most of Docker. Therefore users can mostly avoid the command-line interface.

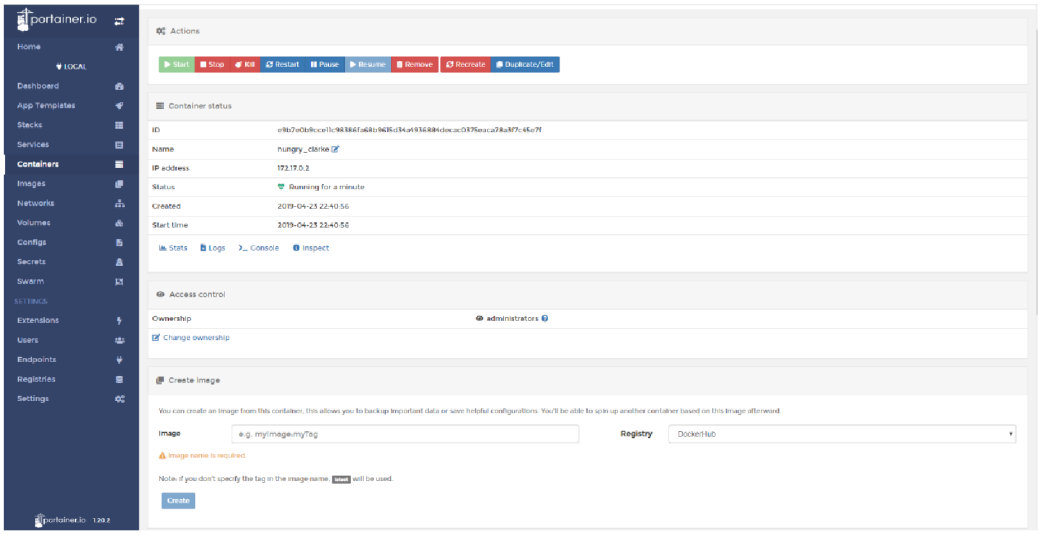


Figure 3.1: Container management in Portainer

3.3.2 Kitematic

Kitematic[2] is an open-source container management tool acquired by Docker. This solution focuses solely on container management. Unlike Portainer, it offers a greatly simplified user interface (Figure 3.2). Thus it loses some of the functionality of Portainer, but it makes the user experience easier for users with no previous experience with the Docker platform.

[1] https://www.portainer.io/
[2] https://kitematic.com/

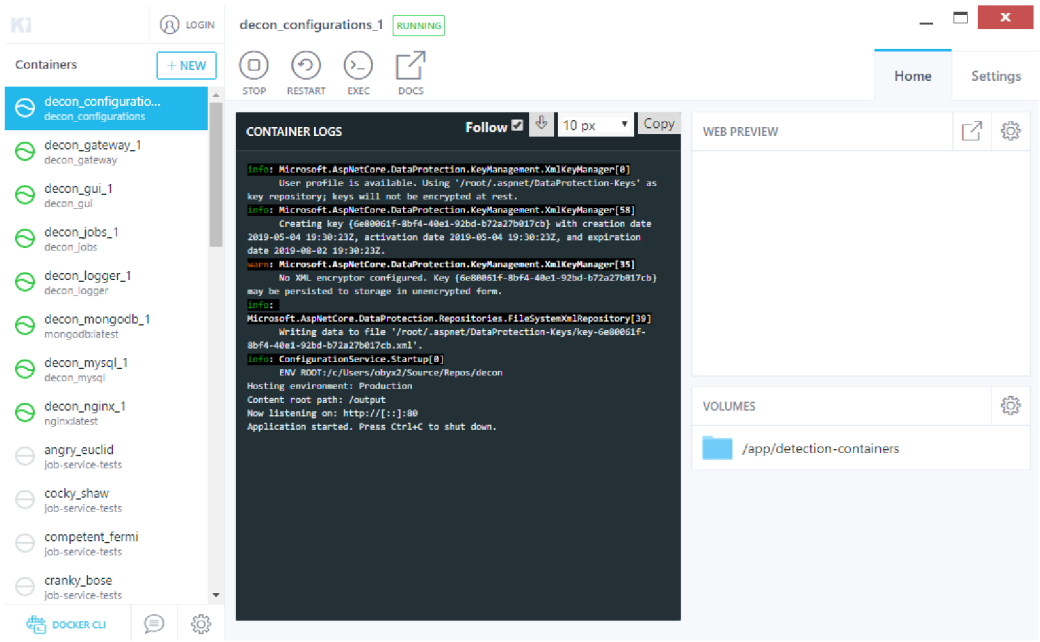


Figure 3.2: Kitematic's container output

3.4 Requirements

Many of the requirements were outlined prior to design and implementation of DeCon. From these requirements a detailed analysis was conducted. Both functional[3] and non-functional[4] requirements were specified.

Identifier	**Name**	**Category**
req_auth	**Token based authorization**	**Security**
Authorization will be token based.		
req_logging	**Logging**	**Reliability**
Relevant information and exceptions will be logged.		
req_code_style	**Code style consistency**	**Code**
Code style will be consistent and will adhere to language specifics.		
req_code_doc	**Code documentation**	**Code**
Code will be documented in a way usual for the given programming language.		
req_testing	**Testing**	**Code**
Testing will be done where deemed necessary.		
req_unit_tests	**Unit tests**	**Code**
Key parts of the codebase will be covered by unit tests.		
req_integration_tests	**Integration tests**	**Code**
Testing across services will be performed.		
req_code_struct	**Code structure**	**Code**
Code will be structured in a clear and easy to understand way.		
req_code_patterns	**Design patterns**	**Code**

[3]Requirements describing actual system behavior or features.
[4]Requirements defining general system characteristics.

Services will use well-known design patterns where applicable.		
req_request_time	**Fast request response**	**Performance**
No request will take longer than it is needed for the relevant response to be returned.		
req_parallel_jobs	**Parallel jobs**	**Performance**
Multiple command-line application can run at the same time.		
req_log_persistence	**Job output persistence**	**Reliability**
Output produced by the command-line applications has to persist Docker cache, container and images wipes.		
req_install_instructions	**Install and run instructions**	**Documentation**
Clear and concise instructions will be given on how to run DeCon.		
req_http	**HTTP communication**	**Interoperability**
All web APIs will be HTTP based.		
req_portability	**Platform independence**	**Portability**
All services should be platform independent.		
req_api_gui	**GUI and API**	**Functionality**
DeCon will provide both an API and a GUI to manage jobs.		
req_job_configurations	**Job configurations**	**Functionality**
Configurations will hold general information about the command-line applications.		
req_mounting	**Mounting**	**Functionality**
Additional folder/file can be mounted for application to access.		
req_locking	**Lock mode**	**Functionality**
Configuration editing could be disabled by an option for demo purposes.		
req_arbitrary_cli	**Arbitrary applications**	**Functionality**
DeCon will be able to run both database detectors and any arbitrary command-line application		
req_gui_export_import	**Session state export/import**	**Functionality**
State of user's session can be exported to a file and imported back.		
req_cli_specialization	**Specializations in GUI**	**Functionality**
Only the GUI will include features specific to the detectors, other components will make no distinction between detectors and any other command-line application.		
req_percentage_report	**Progress reporting**	**Functionality**
Command-line applications can report their progress by sending messages in a specified format.		
req_detectors_output	**Parsed output of detectors**	**Functionality**
Output from the detectors will be parsed and presented in a clear and understandable way.		
req_containers	**Docker containers**	**Functionality**
Docker containers will be used for service containerization.		

Table 3.1: Requirements

3.5 Architecture

Given the requirements imposed on the final product, it was clear from the initial design phase that it would be impractical to develop DeCon as a monolithic application. This emerges primarily from the following reasons:

- having both command-line and graphical interface inside any monolith could result in an inconsistent behavior
- applications that have long execution time (or indefinite) would be unmaintainable in a monolithic web application – a background service is needed
- combining different technologies and frameworks would be hard to achieve

Instead, DeCon was designed as a set of services running inside Docker containers. Service-oriented architecture was considered and at one point even partially implemented. SOA solution would split the system into a few separate services; thus resolving the aforementioned issues of a monolithic approach. However, for the actual implementation, microservices were chosen as a result of certain later identified key disadvantages of the SOA approach. These included insufficient service granularity and public/private service separation. Facing the requirements in question, the following advantages offered by microservices are of great importance:

- separation to small independent units that are easier to containerize and orchestrate via Kubernetes or similar tools
- communication between services is encouraged to be simple, usually using HTTP-based REST APIs
- adding and testing new features, as they are conceived, is a relatively low-risk operation in terms of impact on other services

In practical applications, correct data separation in microservices might prove difficult, as data are usually stored in databases. As mentioned in Section 2.5.1, sharing databases between multiple microservices could be considered an anti-pattern, since database creates a dependency between services and a possible single point of failure. In order to mitigate these hazards, DeCon uses one database per service. However, this introduces problems of eventual consistency, that have to be managed manually[5] by making calls to related services. Advantages of this approach are mainly in better separation, ability to change database schemas independently and distribute the load more evenly, since database does not have to bear the load of multiple services. This is greatly beneficial in cases where services sharing the databases are heavily loaded with incoming traffic.

Communication between services is implemented by HTTP REST APIs (`req_http`). HTTP was chosen for its ubiquity and support by application frameworks. Since HTTP usually[6] runs on TCP/IP protocol, communication is considered reliable on the packet level. Communication however can fail when one or more services go offline. In these circumstances, DeCon availability will be affected, however, where possible, system will apply appropriate measures (job timeout, optimistic container killing – when not sure,

[5] See Section 3.5.2, deletion of a configuration has to trigger cascade deletion of jobs.
[6] See HTTP/3 over QUIC https://http3-explained.haxx.se/en/h3.html

DeCon will attempt to kill the container, even when host system was restarted and no such container exists) to eventually achieve consistency.

All services are designed with portability in mind (`req_portability`), so all of them should be able to run under any supported operating system[7]. Since Docker is supported on all mainstream OS types (Windows, Linux, MacOS), DeCon can be run on any of these systems. However, it should be noted that inconsistencies[8] exist across Docker implementations.

When dealing with authentication and authorization in applications where the end user is expected to be either developer (self-hosted scenario) or a team member (hosted scenario), it is important consider the obtrusiveness of the authentication and authorization process, since introducing a complex authentication and authorization system could prove as redundant. In accordance with the requirement (`req_auth`), authorization will be token based, meaning that only an unguessable *universally unique identifier* (UUID) will be used. The knowledge of the token will authorize users to perform any action allowed given the current DeCon configuration (e.g. modifiability of job configurations). Authentication on the API is not present, only in the case of web application, browser's local storage is used to store known job and configuration identifiers/tokens.

It is important to mention why author chose not to use native Docker logs and instead resolved to sending the outputs via HTTP directly to a DeCon service. The reasons for doing so were:

- native logs and status (see `docker inspect`) retrieval is request based

  Given that a container would be started, no subsequent status request would be made and container would be deleted, all logs would be lost, since they were present only in Docker. Having a wrapping application that listens for the outputs and actively streams data back to a DeCon service ensures that data are stored in a permanent storage almost immediately.

- every request for a job status would result in a call to Docker for new logs

  Calling Docker with every request (especially when the GUI or large number of clients would make frequent requests) could cause noticeably load on Docker Engine. By sending the output data, as they are captured, to a DeCon service, Docker is spared the load. Under this architecture the service has to only retrieve logs from its database.

As for regular logging of events and errors from DeCon services, author also chose to implement custom logging mechanisms. The reasons for custom logging were mainly:

- different technologies used across services

  Having a diverse set of technologies used in DeCon and not having single point for logging, would greatly increase difficulty of searching the logs.

- lack of a standard log format

  As mentioned in the first point, diverse technologies produce different log formats, which would further decrease searchability of the logs.

[7] Supported operating system of the multi-platform framework.

[8] For example DNS record for the host system differs, however future versions of Docker are expected to fix this issue.

- inconsistent for custom logging attributes

  Having custom logging attributes can help services to determine whether any errors occurred during the job execution.

When designing DeCon an important choice had to be made – where to differentiate between generalization and specialization in terms of database detectors. According to the requirement (`req_cli_specialization`) a choice was made to specialize only in GUI. The differences between detectors and any other command-line application are in the means of data input and presentation, namely to what extent the inputs and outputs are user-friendly. The rest of the process of managing jobs is identical, therefore it was decided to keep the specializations in the GUI. In the future, if DeCon would integrate other Testos tools, having a generalized backend infrastructure might prove helpful to avoid unnecessary issues.

DeCon components (Fig. 3.3)

- *Gateway* – public endpoint
- *Job service* – handling jobs and their lifecycle
- *Configuration service* – managing configurations
- *Logging service* – log storage and retrieval
- *Application wrapper* – capturing output from the command-line application
- *Docker service* – wrapper around Docker commands specific for DeCon
- *Web application* (the GUI) – user interface for managing DeCon

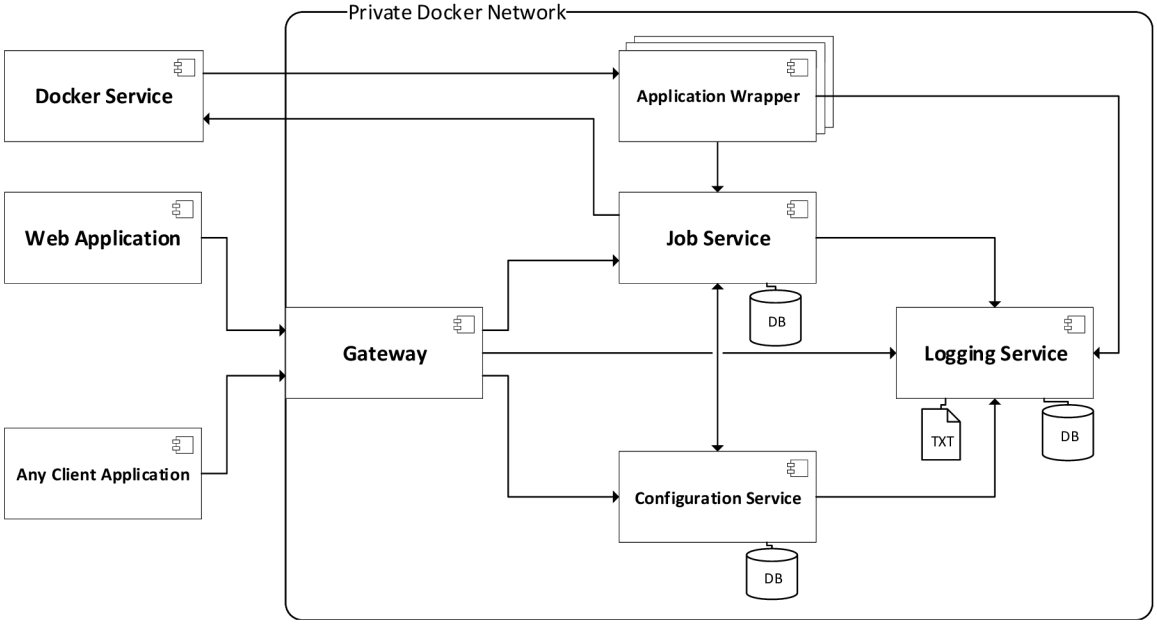


Figure 3.3: DeCon architecture

Name	URL	HTTP method
Create a configuration	`/configuration/`	POST
From the passed model a new configuration will be created.		
Get a configuration	`/configuration/{token}`	GET
Configuration with a matching token will be returned.		
Update a configuration	`/configuration/{token}`	PUT
Configuration with a matching token will be updated.		
Delete a configuration	`/configuration/{token}`	DELETE
Configuration with a matching token will be deleted along with all associated jobs.		
Start a job	`/job/`	POST
Job will be created and started according to the passed model.		
Get a job	`/job/{token}`	GET
Job with a matching token will be returned.		
Delete a job	`/job/{token}`	DELETE
Job with a matching token will be deleted.		
Kill a job	`/job/kill/{token}`	GET
Job with a matching token will be killed, if it is running.		
Get status of a job	`/job/status/{token}`	GET
Status information about a job matching the token will be returned.		
Configuration modifiability	`/modifiable`	GET
Returns a boolean value depending on whether configurations can be modified.		

Table 3.2: Gateway actions

3.5.1 Gateway

Gateway is the only publicly accessible API. Its role is to abstract internal API calls which may or may not directly 1:1 to the public ones. The available actions/calls are listed in Table 3.2. Having the API as the singular point for public access also simplifies access control, since private actions are simply not exposed through Gateway. It interacts with Job and Configuration services, depending on the type of request. Logging information are sent to Logging service for further processing.

To be able to run DeCon on a production server or in a public demonstration environment, DeCon has to mitigate the threat of remote code execution, because by giving users access to any executable present on the host system, while having an internet connection, would effectively open the container and the host system to any malevolent entity. The danger lies in the users' ability to add new configurations; thus toggleability of this feature is required (`req_locking`). Since this is a purely permission related issue, it is a part of Gateway.

3.5.2 Configuration service

Configuration service handles retrieving and managing configurations – templates from which jobs are spawned (`req_job_configurations`). Configuration is identified by its unique name. Optionally a mount to a directory or a file can be specified (`req_mounting`). Mounting is done via Docker Volumes. Deleting a configuration will cause a call to Job

Name	URL	HTTP method
Create a configuration	`/`	POST
Get a configuration	`/{token}`	GET
Update a configuration	`/{token}`	PUT
Delete a configuration	`/{token}`	DELETE

Table 3.3: Configuration service actions

service for all jobs created from this configuration and then another call for deleting them along with this configuration. Table 3.3 shows actions of this service.

Configuration attributes

- `Name` – unique configuration name
- `FilePath` – path to the executable
- `WorkingDirectory` – working directory of the executable
- `Mount` – file or directory that will be mounted to `/app/mount`
- `Dockerfile` – optional, path to a Dockerfile for installing dependencies
- `ContainerPort` – port inside the container which will be exposed
- `JobTimeout` – timeout after which job (without any activity for that amount time) will be killed

DeCon can optionally expose a port from the container to the host system. This can enable usecases where the command-line application would be accessible from the host system and act as a server. When creating a configuration, the port can be specified to indicate which port from the container will be exposed. Mapping of this port to the host is individually specified when creating jobs. However, it should be noted, when an already used port is specified, *job will fail to start*. This can be avoided by setting the port to `0` (or null) thus Docker will pick any port available, then the actual port is updated on the job entity. If no port is specified with the configuration, container will not be exposed.

Job timeouts are essential when dealing with potentially unstable applications that could cause unwanted resource consumption due to hangups. By default, job will be killed after 900 seconds of no standard output and error activity. This can be configured by specifying the timeout when creating new configurations. If timeout `0` is entered, no timeout will be applied. This feature is handled at Configuration service level, since it is expected that this setting is the same for all jobs under given configuration.

3.5.3 Job service

Job service is responsible for managing all activities associated with running and managing jobs. It manages the whole job lifecycle, see Table 3.4.

For a job to be started, a configuration name must be provided, optionally any number of command line arguments can be passed as a string array or a port on the host system

can specified for exposing the command-line application. If a configuration with a matching identifier is found, then the job is scheduled for execution.

Since this service indirectly works with Docker containers (whose startup time can be noticeable), it adheres to `req_request_time` requirement by making calls to *Docker service* asynchronous (if a request to Docker service would fail, job timeout would cancel the job after the specified interval). The number of concurrently running jobs is not limited (`req_parallel_jobs`). Though the host system will eventually run out of the system resources, since with every job a new container is created.

Name	URL	HTTP method
Start a job	`/`	POST
Get a job	`/{token}`	GET
Delete a job	`/{token}`	DELETE
Kill a job	`/kill/{token}`	GET
Get a status of a job	`/status/{token}`	GET
Get all job IDs by a conf.	`/listbyconfiguration/{conf}`	GET
Add a new status update	`/log`	GET

Table 3.4: Job service actions

All jobs can report their current progress in a percentage value by outputting a specialized message (`req_percentage_report`). This feature enables the GUI to render the progress bar given that the command-line application supports this reporting. Regular expressions for parsing the progress percentage can be specified by a text file. Expressions are expected to be delimited by a UNIX line ending. When parsing the output lines, Job service checks each line whether any of the provided regular expressions matches the current line. If a match is found, it is parsed to `double`. If successful, job's `ProgressPercentage` property is updated with the parsed value. If multiple matches are found, only the first one is parsed, the rest is ignored. The file with regular expressions can contain for example lines depicted in Listing 3.1.

```
(?<=\[PROGRESS\]).+
(?<=\[PROGRESSX_SOMETHING_ELSE\]).+
```

Listing 3.1: Example of a file for progress reporting specification

Job entity attributes

- `Id` – universally unique identifier
- `Created` – timestamp when the entity was created
- `Started` – timestamp when the job `State` switched from *Scheduled* to *Running*
- `Finished` – timestamp when the job `State` switched from *Running* to *Success*
- `State` – job state, see *Job state lifecycle*
- `Configuration` – name of the associated configuration
- `Arguments` – serialized command-line arguments

- `StdOut` – collection of `StreamEntries` sent from Application wrapper
- `StdErr` – collection of `StreamEntries` sent from Application wrapper
- `LastActivity` – the most recent timestamp received
- `ProgressPercentage` – reported progress of the command-line application
- `ExitCode` – exit code of the command-line application

StreamEntry entity

- `Timestamp` – timestamp when the entry was captured by Application wrapper
- `Value` – actual line value

Job state lifecycle (Fig. 3.4)

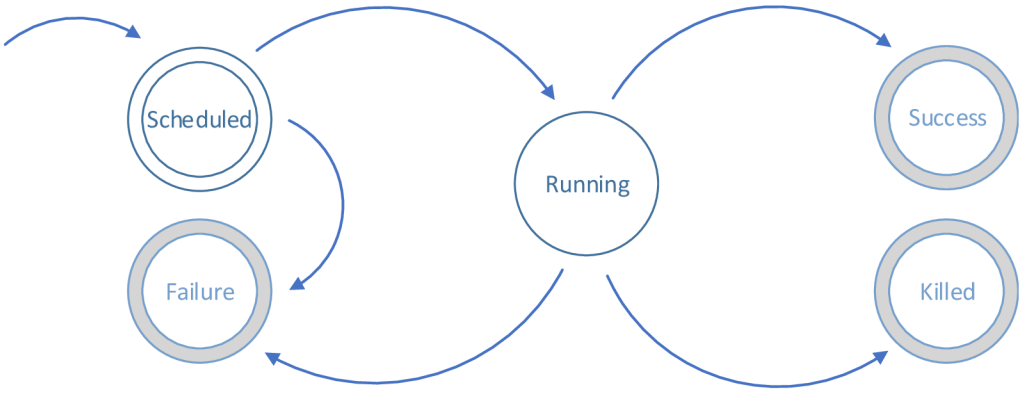


Figure 3.4: Job state diagram

- *Scheduled* – job is waiting to be started (Docker container is being started or the Application wrapper is in process of starting the command-line application). If any errors occur during startup, the job changes its state to *Failure*.
- *Running* – the command-line application is running and the Application wrapper is capturing the output. Any error directly from the Application wrapper terminates the job. Subsequently the presumably running container is killed and the job is marked as a *Failure*.
- *Success* – the application has exited.
- *Killed* – job was killed explicitly or implicitly after a set timeout (default 900 seconds, see Section 4.6) when no activity has been detected.
- *Failure* – an internal error occurred or the Docker container failed to start.

Interactions when starting a new job (Fig. 3.5)

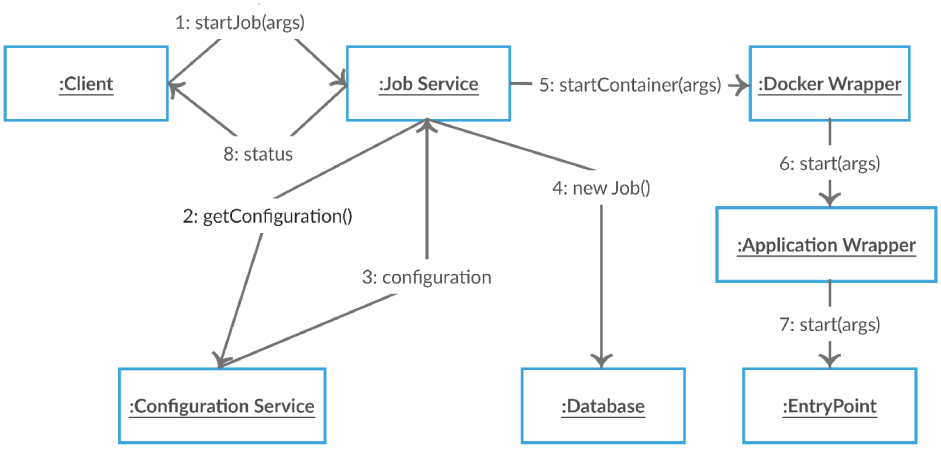


Figure 3.5: Collaboration diagram of a job start

1. A request is made to start a new job.
2. A request for the specified configuration is made to Configuration service.
3. The configuration is returned.
4. The job entity is created in the database.
5. A request for starting a new Docker container is made (asynchronous to 3.).
6. The container is started with the Application wrapper inside (asynchronous to 3.).
7. Application wrapper starts the command-line application (asynchronous to 3.).
8. Job Status is passed back to the client (on a `startJob` request Job service returns only a HTTP status code, Gateway then makes a `getStatus` request (see Figure 3.6) which then adds actual job status information).

Interactions when getting a job status (Fig. 3.6)

Figure 3.6: Collaboration diagram of a job status retrieval

1. A request is made by the client for the job status.
2. The job entity is retrieved from the database.
3. The job status is passed back to the client.

Interactions when sending output data (Fig. 3.7)

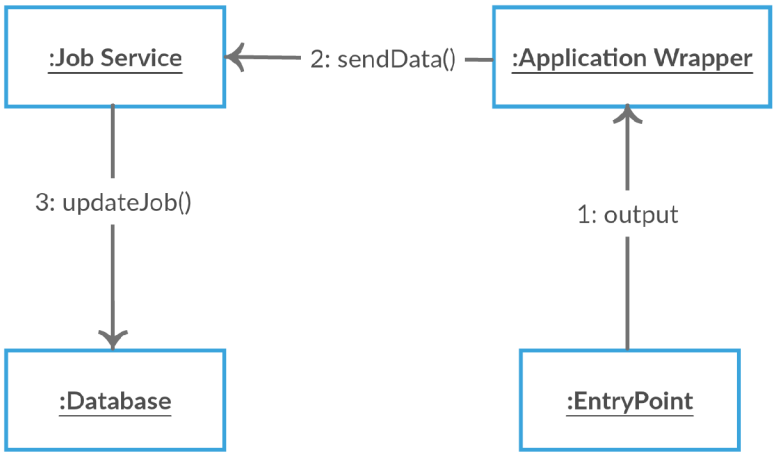


Figure 3.7: Collaboration diagram of a job update

1. Standard output or error is produced and captured by Application wrapper.
2. Captured data are sent to Job service.
3. Job service updates the associated job entity.

3.5.4 Logging service

Logging service provides a simple log storage for all DeCon services with the option of DeCon specific logging information and filtering (`req_logging`). Service saves the logs to both a database and a text file. Actions supported by this service are listed in Table 3.5.

Log entry format

- `Id` – universally unique identifier
- `LogLevel` – log severity
- `EventId` – identifier associated with the log entry in the source service
- `Name` – point of origin (service or part of a service)
- `Message` – actual message
- `Timestamp` – timestamp when the log entry was created in the source service

Name	URL	HTTP method
Get logs	`/`	GET
Add a log	`/{token}`	POST

Table 3.5: Logger service actions

3.5.5 Application wrapper

Application wrapper encapsulates command-line applications (`req_arbitrary_cli`). It provides ability to start a command-line application with given arguments and capture the output streams line by line. Each captured line is then sent to Job service, where it is stored with the corresponding job entity. Message types originating from application wrapper:

- `StdOut` – line containing standard output message
- `StdErr` – line containing standard error message
- `Error` – internal error (f.e. failed application start, job switches to *Failure*)
- `Start` – successful start (causes job to go from *Scheduled* to *Running*)
- `Exit` – successful exit (causes job to go from *Running* to *Success*), exit code is returned

This application is started with every new job and receives a token (job identifier), a file path to the executable and a working directory for the executable. When the started executable exits, Application wrapper also terminates, subsequently the encapsulating container is also terminated. If a kill request is made, the encapsulating container is terminated.

3.5.6 Docker service

Docker service is the only service running outside of Docker containers, since its purpose is to execute only predetermined Docker commands (`req_containers`). Having this service outside of Docker was necessary because it has to be able to access all of the host file system, since it needs to copy working directories inside containers. This was the only factor preventing this service from being hosted inside a Docker container.

Service receives requests for starting/killing (Table 3.6) containers from Job service, which are then parsed and transformed to Docker commands for execution. Docker service is able to build and run any Docker container; therefore it is completely independent of other DeCon services.

Name	**URL**	**HTTP method**
Start a Docker container	`/`	POST
Kill a Docker container	`/{token}`	DELETE

Table 3.6: Docker service actions

3.5.7 Web application

The graphical user interface for DeCon was designed in order to provide a user-friendly way of interaction with DeCon (`req_api_gui`). Implementation as a web application was chosen because web interface provides many benefits, such as:

- no additional user environment dependencies
- faster development time than comparable solutions (desktop or mobile application)
- accessibility across operating systems

The design was focused on communicating all the information clearly, simply and in a modern-looking web environment. Prior to creating wireframes, author studied solutions containing features such as test automation or integration of console-like interfaces in web applications. These solutions included Microsoft Azure[9] and GitLab[10].

As mentioned in Section 3.2, web application is the only specialized part of DeCon. Specialization was done in order to simplify argument input for database detectors. Since DeCon's authorization (`req_auth`) is purely token based, users intentionally cannot obtain lists of existing entities. Instead they rely on other users providing them with tokens to configurations/jobs or creating the entities by themselves. Therefore, all tokens known to the current user are stored in his browser's local storage, from where they can be exported and later imported back. Users can share their tokens via the share functionality which creates a hyperlink which will import the token to the recipient's local storage.

Both job and configuration can be deleted. In addition to deleting them permanently, user can choose to delete them locally; thus allowing other users to still access them.

General layout of the application is conceived as single-page website, meaning that no complete page reload is necessary when performing interactions with the interface.

The navigation bar is placed in the top part of the interface and contains the application name and a link to the API documentation. The left side is occupied by the menu with buttons to add a new configuration, export and import the session state. This menu is collapsable; thus, it can allow for better output readability from the main window. The window houses all of the job meta information and outputs. In the upper part, buttons for job management are placed along with the job name and configuration name. Actions, such as job killing, sharing, refreshing or forking (creating a new job with the same arguments), can be performed through these buttons. Meta information is outputted to a clearly formatted table. If the currently shown job is running and DeCon is able to determine percentage of progress[11], an animated progress bar is rendered as seen in Figure 3.8.

The main output panel is situated beneath the meta information section. If the job is based on database detectors, a specialized tab called *Results* is rendered, otherwise only a single panel with stream toggling and filtering is shown. Filtering works in two modes – standard and regular expressions. Users can toggle between these modes and only lines matching the filter will be shown. The panel below renders outputted standard output and error line along with timestamps. The entire output can be scrolled through. When the auto-refreshing of the job window is enabled, the panel is automatically refreshed. Per user request, the entire output can be exported to a standardized format shown in Listing 3.2, where `STREAM_TYPE` is either `STDOUT` or `STDERR`, `TIMESTAMP` is in the ISO 8601 format and `VALUE` represents actual output value.

```
STREAM_TYPE TIMESTAMP: VALUE
STDOUT 2019-04-16T21:17:10.689Z: [INFO] Starting detectors...
```

Listing 3.2: Output format of an exported job

The Results tab contains the parsed output from database detectors (`req_detect_out`) as seen in Figure 3.9. Output logs are dynamically parsed, so that easy-to-understand representation can be rendered. For each table, all the detected columns are shown, along

[9] https://azure.microsoft.com/en-us/

[10] https://about.gitlab.com/

[11] Supported with db-detectors and any other command-line application that implements progress reporting.

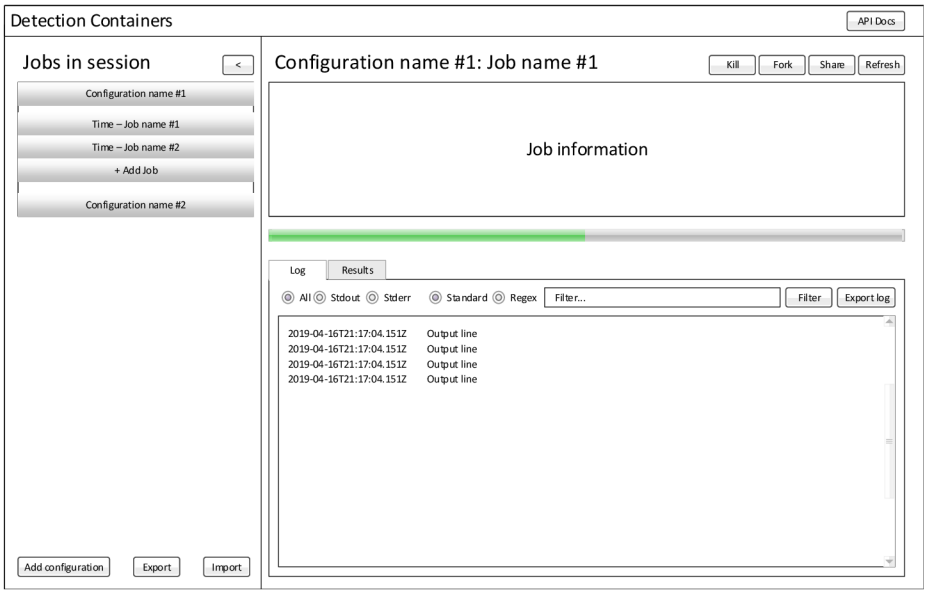


Figure 3.8: User interface showing a running job

with their respective column type and detector names that returned the highest weight[12]. If multiple detectors report the same weight, all values are displayed. When the result is expanded, all weights reported are shown. The parsed result can be exported in the format seen in Listing 3.3.

```
{
  "table_name": {
    "column_name: {
      "type": "COLUMN_TYPE",
      "weights": [
        {
          "weight": 0.9,
          "detector": "Detector#1"
        },
        {
          "weight": 0.2,
          "detector": " Detector#2"
        }
      ]
    }
    ...
  }
  ...
}
```

Listing 3.3: Export format of a database detectors result

[12]Weight is converted to percentage.

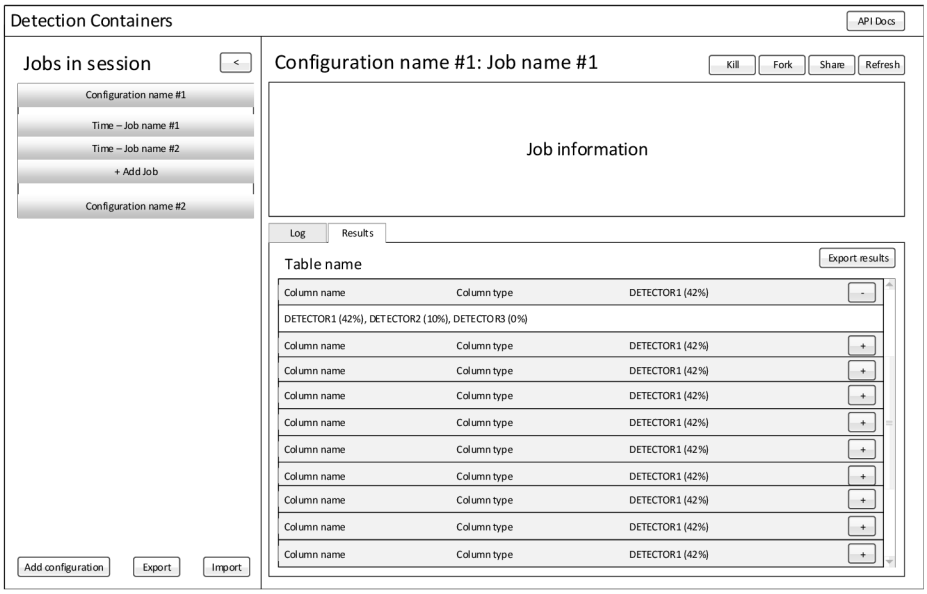
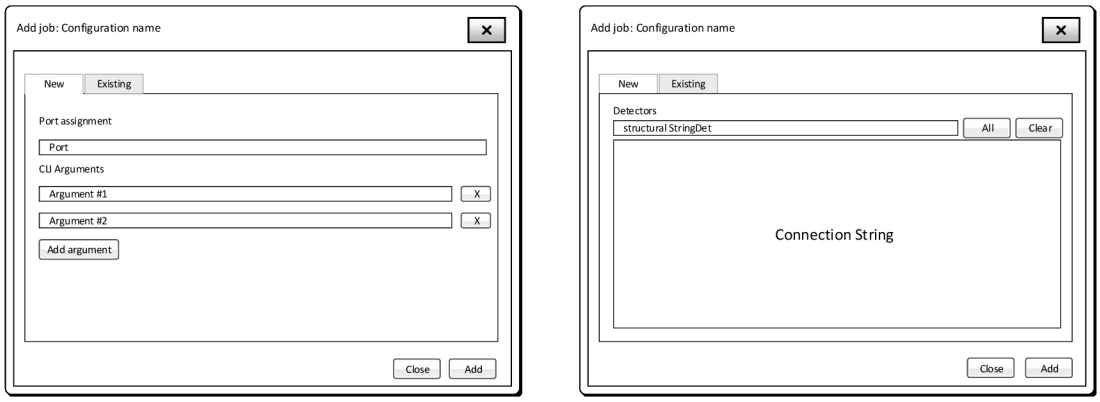


Figure 3.9: User interface showing a parsed and displayed result of database detectors



(a) Job creation for a generic CLI application

(b) Job creation for db-detectors

Figure 3.10: Comparison of modals for job creation

Modals or modal windows are child windows of their parent window and their primary function is to communicate new information or to provide additional functionality to the application. Modals are used in DeCon for creating configurations, starting jobs, sharing etc.

Notable example of modal window in DeCon is the Add job modal. This modal greatly differs based on whether it is created for database detectors or not. For a regular command-line application, users can input command-line arguments one-by-one or specify a port on which the command-line application will be exposed on the host system. In the case of detectors a specialized user interface is displayed (`req_cli_specialization`). This interface contains a list box of known detectors, which can be selected (or any custom text can be inputted and then confirmed via the enter key) and set of text boxes and drop-downs for specifying the connection string to the target database. The difference is shown in Figure 3.10, see Appendix B for screenshots of the actual user interface implementation.

Chapter 4

Implementation Details of DeCon

In this chapter author will attempt to describe details of DeCon implementation with focus on technology-specific aspects of this solution. Notable implementation details will be mentioned and integration of database detectors will be discussed.

4.1 Technology Choices

When making decisions regarding choices of implementation language or framework, ease-of-use, productivity and suitability should be considered. Since DeCon is implemented using microservice architecture, it allows for usage of diverse technologies. Among the services, two subgroups with different requirements were identified:

- services using databases and having similar APIs (Configuration, Job and Logging services)
- simple services not using databases (Gateway and Docker service)

For the first set of services ASP.NET Core[1] Web API framework using C# language was chosen. This framework was selected mainly because it provides a robust API building system based on a language that has well-designed threading and asynchronous support. Unlike the full .NET framework, .NET Core is compatible with multiple operating systems; therefore it can run on Linux-based systems. The database type of choice for these services was MongoDB which was selected for its simplicity and flexibility. Third-party libraries (NuGet packages) used in .NET services:

- MongoDB driver[2] – interface to MongoDB
- NSwag[3] – Swagger support
- CommandLineParser[4] – parsing command-line arguments (Application wrapper only)
- Json.NET[5] – JSON serialization/deserialization framework

[1] https://dotnet.microsoft.com/apps/aspnet
[2] https://docs.mongodb.com/ecosystem/drivers/csharp/
[3] https://github.com/RicoSuter/NSwag
[4] https://github.com/commandlineparser/commandline
[5] https://www.newtonsoft.com/json

- RestSharp[6] – simple HTTP API client
- Moq[7] – library for creating mock objects
- XUnit[8] – unit testing framework

The second type of services did not require some of the mentioned features and codebase was expected to be much smaller. Therefore, Flask API on Python was the framework of choice, because it contains virtually no boilerplate code[9] and much smaller container footprint (only Python and a few packages required). Packages used in Flask APIs:

- Flask-RESTPlus[10] – Swagger support
- Flask-CORS[11] – Cross-origin resource sharing support
- Requests[12] – simple HTTP request library

Both frameworks are compared on a short code sample in Appendix C.

For Application wrapper, .NET Core was used, as it offered greater simplicity as far as capturing of standard output and process error is concerned, and an easily configurable HTTP client.

For the web application multiple approaches were considered. Eventually a frontend-only web application was implemented, because the singular role of the web application is to consume DeCon's public API; therefore no backend code was required. From all the frontend technologies, React[13] (a JavaScript library) was selected since it offers a fast virtual DOM and can be easily combined with other JavaScript libraries, such as jQuery[14].

A reverse proxy[15] was used to forward the requests to Gateway and the user interface. NGINX[16] was chosen as the reverse proxy functionality provider.

4.2 General Implementation Principles

The implementation across different services follows the same standards. Code style and formatting adheres to the usual practices and standards of given programming language (`req_code_style`). All relevant sections including all public method signatures are documented (`req_code_doc`). The DeCon implementation, where applicable, tries to follow the SOLID[17] design principles (`req_code_struct`). SOLID is manifested in DeCon mainly by using dependency injection[18] to inject implementations of interfaces. Where applicable, design patterns (such as repository pattern) are used (`req_code_patterns`).

[6] http://restsharp.org/
[7] https://github.com/Moq/moq4/wiki/Quickstart
[8] https://xunit.net/
[9] Code required to setup the framework, often no changes are made to the template.
[10] https://flask-restplus.readthedocs.io/en/stable/
[11] https://flask-cors.readthedocs.io/en/latest/
[12] https://2.python-requests.org/en/master/
[13] https://reactjs.org/
[14] https://jquery.com/
[15] Type of a proxy server that retrieves resources on behalf of client [25].
[16] https://www.nginx.com/
[17] Single responsibility principle, Open-closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle.
[18] Technique that injects object's dependencies usually by passing them to a constructor.

4.3 Project Structure

Each service is implemented as a separate project. In case of ASP.NET-based services a shared project is included in the service. This could seem as a violation of one the microservices principles – no dependencies should exist between microservices. However, in this case, the shared project is included inside the service folder as a Git[19] *submodule*. Therefore it enables control over *pulling* new changes for each service independently. The shared project is not a set of particular files shared between services but instead it is a version-controlled snapshot of the included files. The project includes a common infrastructure used across the ASP.NET services, such as a database access via repository pattern or a base controller with support for logging and exception handling.

4.4 User Interface Functionalities

The behavior of the web application is built around the concept of browser local storage. It stores all the visible configuration names and job identifiers – everything else is dependent on the API. Data are stored in JSON and their format is equivalent to the one that of the exported state. In fact, importing and exporting of the state only replaces/downloads the actual value of the local storage. By default, the storage is initialized to include all demos (db-detectors, demo-mount, demo-flask etc.). To reset the local storage to the default state, users can import the initializing file located in `/data/gui_initial_state.dcx`. With every import, a backup file is downloaded, which contains the application state before it was replaced by the uploaded one.

Separation of local and server storage was done because of the expectation of DeCon deployment and sharing between multiple users, for example a project team. Because every browser's storage has its state and the exported files can be shared, DeCon tries not to interfere with the actual team workflow; therefore maximizing its adaptability to potential future deployments.

4.5 Use of Docker Features

Services are placed each in its separate Docker container which is then managed by Docker Compose utility. Apart from using Docker to run the DeCon services, Docker is also used to start new jobs. This process is implemented in Docker service which encapsulates Docker calls.

Container startup is implemented via 5 separate calls to Docker – `build`, `create`, `copy` to container, `start` and `port` retrieval, see Listing 4.1 for example calls. An alternative, more common approach, is to call `start` directly, but since the requirement (`req_parallel_jobs`) states that multiple containers need to run in parallel, the usage of Docker volumes for the command-line application directory would not be sufficient. Volumes are problematic, as they are essentially only mounts to the host directory. Therefore file locking and concurrent write access to files could potentially result in an unexpected behavior. This problem is solved by copying the entire working directory to the container. Copying files however comes both with upsides and downsides. Benefits of this approach are that all changes made to the command-line application are reflected on every new job start and no conflicts occur between the running jobs on the file system level. However,

[19] https://git-scm.com/

copying also has some downsides. Slower startup time (files need to be copied) and loosing file changes in the working directory[20] are the main ones. By default, files from the working directory are copied, but users also can opt-in to mount the working directory as a volume. This comes with the mentioned downsides but can also allow for usecases where user would edit the files and in real time changes were propagated to the running application[21]. All the created containers are automatically removed by Docker when they exit, in order to free the system resources.

Persistence of DeCon databases is implemented using Docker volumes, which provide a permanent storage. These volumes are mounted to the data directory inside MongoDB containers. This ensures that database data persist Docker image and container wipes

```
$  docker build -f=DOCKERFILE -t=IMAGE_NAME SOURCE_FOLDER
$  docker create --rm --network=NETWORK_NAME --name=CONTAINER_NAME
    --volume=SOURCE_FOLDER:TARGET_FOLDER IMAGE_NAME
$  docker cp SOURCE_FOLDER CONTAINER_NAME:/app/cli/
$  docker start CONTAINER_NAME
$  docker port CONTAINER_NAME
...
$  docker stop CONTAINER_NAME
```

Listing 4.1: Used Docker commands for container management

Container creation arguments

- `network` – name of a Docker network (for DeCon it is `decon_internal`) to which all DeCon services are connected
- `name`– container name, DeCon uses the job identifier (UUID)
- `volume` – file/folder to mount, mount point specified in job's configuration, mounted as `/app/mount/` in the container[22]
- `rm`– remove the container after it exits

Arguments for Application wrapper passed with `create` command

- `token` – job identifier with which Application wrapper will send output back to Job service
- `file` – executable to start
- `working directory` – working directory of the executable
- `arguments` – command line arguments encoded

4.5.1 Custom Dockerfile

DeCon provides an option to customize the container environment. The default Dockerfile uses the official Debian image[23] with preinstalled `build-essential` package and Python.

[20] Configuration's mount argument can be used to preserve the file changes.
[21] For example running a Flask API with debug mode enabled.
[22] Configuration's mount option is designed for reading/writing to shared files, not for executables.
[23] https://hub.docker.com/_/debian

If the command-line application needs additional dependencies, custom Dockerfile can be provided. Examples of base and custom Dockerfiles are listed in Appendix D. The custom Dockerfile (see Listing 4.2 for a basic template) must follow these rules:

- the result image must be based on `decon-app-wrapper` image or the entire `/app/` folder has to be moved to the desired container
- `/app/app-wrapper/publish/ApplicationWrapper` has to be the container entry-point[24]

```
FROM decon-app-wrapper

... # Your commands

ENTRYPOINT ["/app/app-wrapper/publish/ApplicationWrapper"]
```

Listing 4.2: Basic template for custom Dockerfiles

Provided Dockerfile has its build context set to the specified working directory. In the final image the working directory will be copied to `/app/cli/` directory during the container start. Since Application wrapper is a self-contained .NET Core application that targets `linux-x64`, moving the binary to a different distribution is possible but functionality is not always guaranteed. For each configuration a new image is created, but the build starts only after the first job request is made. During the build, job is stuck in *Scheduled* state. Build can take up to a several minutes (depending on the system performance and Dockerfile complexity). When the build time exceeds 15 minutes, the started job will fail, however when the build is finished you can start a new job that will begin its execution almost immediately.

4.6 Running DeCon

To run DeCon following dependencies need to be installed on the host machine:

- Docker 18.03+
- Python 3.7+
- Doxygen

The host machine can use any operating system supported by Docker 18.03+ versions. DeCon's start is encapsulated in the startup script `run.sh` which builds and starts all the necessary services. Complete install and run instructions are specified in the `README.md` file (`req_install_instructions`).
Startup script usage (Listing 4.3)

```
$  ./run.sh [--build] [--port=PORT] [--docker-service-port=DOCKER_PORT]
    [--os-type=TYPE] [--preserve-jobs] [--no-configuration-modifiable]
    [--db-recreate] [--job-progress-regex-file=PATH_TO_FILE] [--help]
```

Listing 4.3: Argument list of DeCon the startup script

[24]Executable that will be started inside the container.

- `b | build` – forces to build all dependencies
- `p | port` - port on which DeCon GUI and API will run, default 80
- `d | docker-service-port` - port on which Docker service will run, by default 6002 (to prevent possible conflicts with port assignment on the host, since Docker service runs directly on the host machine)
- `o | os-type` - operating system on which Docker Engine runs, defaults to value from `docker info` – `OperatingSystem`, other possible values: `linux`, `windows`, `macos`
- `n | no-configuration-modifiable` – configurations editing, enabled by default
- `r | db-recreate` - recreate all databases
- `g | job-progress-regex-file` – regular expressions values for matching progress reports, delimited with newline, defaults to `/data/job-progress-regex.txt`
- `j | preserve-jobs` – running jobs are not killed upon exit, by default killed
- `h | help` – help is displayed

During startup, services are built (if `build` argument is specified or DeCon was not previously built) and Docker Compose is started. While Docker Compose is running, `Ctrl+C` command can be sent to gracefully terminate the services (to forcefully kill services, another `Ctrl+C` can be used). After startup, the web application is available at `http://localhost/` and API at `http://localhost/api/`. Doxygen[25] documentation is located in `doc/html/index.html`. Swagger specification is available at the following URLs:

- Gateway – `http://localhost/api/swagger` (Swagger UI)
- Job service – `http://localhost/swagger/job/`
- Configuration service – `http://localhost/swagger/configuration/`
- Logger service – `http://localhost/swagger/logger/`
- Docker service – `http://localhost:6002/swagger/` (Swagger UI)

4.6.1 Included examples

In order to easily demonstrate DeCon functionality, a database with sample data is provided for running database detectors. Connection information for this database is listed in Table 4.1. Database is hosted inside a MySQL container which is part of DeCon's internal Docker network. Listing 4.4 depicts an example of a request body for starting a new job based on database detectors.

Database type	`MySQL`	Host name	`mysql`	Port	`3306`
Database name	`testdb`	Username	`root`	Password	`root`

Table 4.1: Connection information for the included database

[25] `http://www.doxygen.nl/`

```
{
  "configuration": "db-detectors",
  "Arguments": [
      "structural FloatDet StringDet",
      "{\"type\":\"mysql\", \"host\":\"mysql\", \"port\":\"3306\",
          \"name\":\"testdb\", \"user\":\"root\", \"pass\":\"root\",
          \"path\":\"\"}"
   ]
}
```

Listing 4.4: Job creation model for running the detectors on the included database

Other included examples

- demo-mount - example of the mounting functionality, optionally arguments will be displayed
- demo-progress - demonstration of the progress bar reporting from C code
- demo-dockerfile - demonstration of adding a custom Dockerfile
- demo-flask - Flask API running from a custom Dockerfile with exposed port to the host system, optionally the configuration can be changed to use volumes and live script reloading can be performed

All examples are loaded into the Configuration database on the first start and then every time the database is recreated. Between recreations all these examples can be deleted. Database is initialized by `services/configuration_service/initial.json` file, which contains an array of configurations.

4.7 Integration of Database Reporter and Detectors

As mentioned in the previous chapters, DeCon's API does not distinguish between database detectors and any other containerized command-line application, only specialization happens in the user interface with specialized argument inputting and output parsing. The detectors had to be integrated into DeCon in a seamless way. Key integration parts were:

- database detectors configuration is by default[26] present in the database
- database detectors configuration is hard-coded to the default state of the GUI

To allow for this smooth integration, number of changes had to be made to db-detectors and db-reporter projects. First of all, an integration `bash` script (`run_detectors.sh`) was implemented, because no such script was part of either thesis and as stated the submitted code was not tested together. The necessary knowledge for creation of this script had to be extracted from the provided *undocumented* files, since little or no instructions were given in any of the theses.

[26] This configuration can be deleted, however via the `db-recreate` option, db-detectors and other examples will be restored

Actions performed by the startup script

- starting db-reporter
- starting db-detectors
- sending a D-Bus message to db-reporter

During the integration, it was discovered that db-detectors and db-reporter both expected that the other party would provide the database connection string, therefore it had to be manually inserted to the D-Bus communication as if the message originated from db-detectors [10, pg. 20] [14, pg. 14].

Furthermore, the format of the structural detector result differed between thesis' text and actual implementation – this caused db-detectors to fatally crash. In addition to these disparities, UUID detector contained an incorrect method call and faulty condition.

These errors were not the only ones, since after integration it became apparent that only few detectors were actually working. Out of the 28 detectors, only 5 of them are working (string, integer, float, UUID, datetime), the rest is unable to start. After a deep dive into the code, author of this thesis concluded that the issue is most likely inside the dependency manager, since only the detectors which depend only on the structural detector are working. Trying to identify the exact root problem without a proper technical documentation was not possible.

From all the supported database types[27], only MySQL and MariaDB is verified to work, since it was the only database type that was tested by the authors of db-detectors and db-detector.

It was also necessary to refactor the way how the results were collected. Previously they were written into a file, but because DeCon is focused on retrieving the console output[28], db-reporter was modified to output the results to standard output in a format seen in Listing 4.5.

```
[RESULT][DetectorName]ActualResult
[RESULT][StringDet]{"name": "col_mac_address", "table_name": "MOCK_DATA", "type":
    "VARCHAR(500)", "weight": 1.0}
```

Listing 4.5: Database detectors output

This format is later parsed in the user interface for the purposes of knowledge extraction. Apart from modifying the output, author decided to partially refactor db-detectors, since some logic errors were identified in the code and overall code maintainability was considered inadequate. Refactoring also introduced configurable log verbosity and streamlined log messages which were previously hard to understand without a prior complete code knowledge.

4.8 Verification of Functionality

In accordance with the requirement for testing (`req_testing`), multiple suites of tests were created in order to properly verify the functionality of DeCon. These automated tests range from testing individual code blocks to testing the complete functionality on the system level.

[27] MySQL (MariaDB), SQLite, PostgreSQL, Oracle, SQL Server
[28] Writing to files requires a configured mount point.

4.8.1 Unit testing

Unit tests were created to verify the functionality on the method level. (`req_unit_tests`) Since most of the DeCon services do not include any considerable amount of code, the only service, where it was deemed necessary to create unit tests, was Job service. Unit tests in Job service focus on testing custom method extensions, model validation and correct return codes. These tests are implemented using xUnit framework, Listing 4.6 depicts an example test in this framework.

```
using Xunit;

namespace Tests
{
    public class ExampleTests
    {
        [Fact]
        public void OrwellTest()
        {
            Assert.Equal(5, (2 + 2));
        }
    }
}
```

Listing 4.6: Unit tests example in xUnit

4.8.2 Component testing

Unlike unit testing, component testing aims at testing individual services separately without an actual interaction with any external entity. In an environment, where the services depend on others for their correct functionality, new requirements surfaced for the code structure and usage of design patterns.

Because only with patterns like dependency injection and use of interfaces, communication with external services can be achieved without modifying the code that is being tested. Configuration of dependency injection is done in the `Startup` class, where application services are set up. Dependencies can be added, as seen in Listing 4.7, by specifying an interface and a class that implements the interface. Thus all controllers expecting the interface in their constructor are provided with an instance of the specified class (Listing 4.8). This instance can be either shared across all controllers – *singleton* – or unique to that given controller – *transient*.

```
public void ConfigureServices(IServiceCollection services) {
    ...
    services.AddTransient<IConnector, HttpConnector>();
    ...
}
```

Listing 4.7: Configuration of dependency injection

```
[ApiController]
public class JobController : BaseController {
    public JobController(IConnector connector) : base(connector) {
        ...
    }
}
```

Listing 4.8: Injected dependency on a controller

The component tests take advantage of this feature by using ASP.NET `TestServer` package, which starts the API and creates a client. The API is started using a modified `Startup` class which uses mock interface implementations, as seen in Listing 4.9. Component testing in Job service includes verification of correct routing, response checking and chaining multiple requests together.

```
client = new TestServer(new
    WebHostBuilder().UseStartup<MockStartup>()).CreateClient();
var statusId = await GetStatusId();
var response = await client.GetAsync($"/kill/{statusId}");
Assert.Equal(HttpStatusCode.OK, response.StatusCode);
```

Listing 4.9: Component tests using a test client

4.8.3 Integration testing

For testing the overall functionality of DeCon, a set of integration tests were created (`req_integration_tests`). These tests are used to verify the correct behavior of Gateway service (and all of the services behind) including the NGINX reverse proxy. Integration tests are implemented using *Test service* in Python. After the initialization of DeCon, Test service begins its execution by sending requests to the DeCon public API. Tests are performed from the host network. Testing includes creating new configurations, updating and deleting them, job creation, execution and output retrieval. Actual execution of database detectors is also performed. Database detectors target the included MySQL database – see section 4.6.1.

4.8.4 Running the tests

All of the previously described tests can be executed using `tests.sh` script which shares the argument options with `run.sh` script. This script requires DeCon dependencies to be installed (Docker, Python, Doxygen), since the integration tests need to start all the DeCon services to verify the expected functionality. Lisitng 4.10 shows how to start the tests.

```
$ ./tests.sh
```

Listing 4.10: Testing script usage

In the first step unit and component test are executed. After that Docker Compose starts all services and integration tests begin their execution. After each step, a summary is printed with information about the test results. Overall DeCon includes over 70 tests.

Chapter 5

Conclusion

The goal of this thesis was to design and implement containerization of command-line applications including database detectors. The final product – DeCon – offers abstraction over underlying container technology by simplifying the setup and providing configurations and jobs – similar to test cases and test runs. For basic tasks DeCon does not require any knowledge of Docker. However if needed, DeCon offers a great deal of configurability in terms of required dependencies.

DeCon was implemented as a set of microservices communicating with each other using HTTP-based REST APIs. On top of the services, a light-weight web user interface was created. This user interface was not part of the original thesis assignment and adds benefits of user-friendliness. All the microservices do not make distinctions between database detectors and any other command-line applications, therefore any detectors specific logic is limited to the user interface, where users have customized dialogs, result parsing and exporting features. Despite the limitations of the current implementation of database detectors, best efforts were made to integrate the detectors, even if some of the functionality does not work as expected.

The final product was subjected to testing by a number of automated tests which ranged from unit, component to integration tests. These tests helped to ensure the correct functionality of the final product. To demonstrate this functionality, examples of different types of applications were included.

In the future DeCon can be modified to run in clustered environments where it would greatly benefit from the microservices architecture in terms of scalability. Furthermore, the API and the user interface could be expanded to accommodate more fine-grained job management, however these and other future improvements would arise when DeCon would be used to containerize other Testos tools. Subsequently DeCon could be used to a great effect in integrating the whole Testos platform.

Bibliography

[1] Brewer, E.: *Towards Robust Distributed Systems.* In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing.* ACM. July 2000. page 45. doi:10.1145/343477.34350.

[2] Bui, T.: *Analysis of Docker Security. arXiv preprint arXiv:1501.02967.* January 2015.

[3] Cooley, S.; Lang, P.; Mastrean, A.; et al.: *Linux Containers on Windows.* [Online; accessed 30.04.2019].
Retrieved from: https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers

[4] *Container Orchestration — Devopedia.* [Online; accessed 30.04.2019].
Retrieved from: https://devopedia.org/container-orchestration

[5] *Docker Overview — Docker Documentation.* [Online; accessed 04.05.2019].
Retrieved from: https://docs.docker.com/engine/docker-overview/

[6] *About Storage Drivers — Docker Documentation.* [Online; accessed 04.05.2019].
Retrieved from: https://docs.docker.com/storage/storagedriver/

[7] Fielding, R. T.: *Architectural Styles and the Design of Network-based Software Architectures.* PhD. Thesis. University of California, Irvine. 2000.

[8] Fielding, R. T.; Gettys, J.; Mogul, J. C.; et al.: *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. RFC Editor. June 1999.
Retrieved from: https://www.rfc-editor.org/rfc/rfc2616.txt

[9] *Service-Oriented Architecture (SOA) — IBM Knowledge Center.* [Online; accessed 30.04.2019].
Retrieved from: https://www.ibm.com/support/knowledgecenter/en/SSMQ79_9.5.1/com.ibm.egl.pg.doc/topics/pegl_serv_overview.html

[10] Kropáč, F.: *Nástroj pro analýzu obsahu databáze pro účely testování softwaru.* Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. 2017.
Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=19446

[11] Masse, M.: *REST API Design Rulebook.* O'Reilly Media. October 2011. ISBN 978-1449310509. 116 pp.

[12] *Introduction to Web APIs — Mozilla Developer Network Web Docs.* [Online; accessed 30.04.2019].

Retrieved from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction#What_can_APIs_do

[13] Newman, S.: *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media. February 2015. ISBN 978-1491950340. 280 pp.

[14] Ochodek, M.: *Nástroj pro analýzu obsahu databáze pro účely testování softwaru*. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. 2017.
Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=19259

[15] *What Is Virtualization? — OpenSource.com*. [Online; accessed 30.04.2019].
Retrieved from: https://opensource.com/resources/virtualization

[16] Popek, G. J.; Goldberg, R. P.: *Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM*. vol. 17. July 1974: page 412–421. doi: 10.1145/361011.361073.

[17] Richards, M.: *Microservices vs. Service-Oriented Architecture*. O'Reilly Media. April 2016. ISBN 978-1491975657. 55 pp.

[18] Richardson, Chris: *Pattern: Circuit Breaker — Microservices.io*. [Online; accessed 30.04.2019].
Retrieved from:
https://microservices.io/patterns/reliability/circuit-breaker.html

[19] Richardson, Chris: *Pattern: Database per Service — Microservices.io*. [Online; accessed 30.04.2019].
Retrieved from:
https://microservices.io/patterns/data/database-per-service.html

[20] Slack, J.: *Introducing the Host Compute Service (HCS) — Microsoft Tech Community*. [Online; accessed 30.04.2019].
Retrieved from: https://techcommunity.microsoft.com/t5/Containers/Introducing-the-Host-Compute-Service-HCS/ba-p/382332

[21] Snover, J.; Mason, A.; Back, A.: *Microsoft Announces Nano Server for Modern Apps and Cloud — Windows Server Blog*. [Online; accessed 30.04.2019].
Retrieved from: https://cloudblogs.microsoft.com/windowsserver/2015/04/08/microsoft-announces-nano-server-for-modern-apps-and-cloud/

[22] Testos Group: *Testos*. [Online; accessed 30.04.2019].
Retrieved from: http://testos.org/

[23] VMware: *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. Technical report. VMware. March 2008.
Retrieved from: https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html

[24] *OS-level Virtualisation — Wikipedia, The Free Encyclopedia*. [Online; accessed 30.04.2019].
Retrieved from: https://en.wikipedia.org/wiki/OS-level_virtualisation

[25] *Reverse Proxy — Wikipedia, The Free Encyclopedia.* [Online; accessed 01.05.2019]. Retrieved from: https://en.wikipedia.org/wiki/Reverse_proxy

[26] *Virtual Machine — Wikipedia, The Free Encyclopedia.* [Online; accessed 30.04.2019]. Retrieved from: https://en.wikipedia.org/wiki/Virtual_machine

[27] Zhang, Q.; Liu, L.; Pu, C.; et al.: *A Comparative Study of Containers and Virtual Machines in Big Data Environment. arXiv preprint arXiv:1807.01842.* July 2018.

[28] Zissis, D.; Lekkas, D.: *Addressing Cloud Computing Security Issues. Future Generation Computer Systems.* vol. 28. March 2012: page 583–592. doi: 10.1016/j.future.2010.12.006.

Appendices

Appendix A

Contents of the CD

Directory structure of the included CD:

- `/decon` DeCon source files
- `/text` Source files for the thesis text
- `xoberr00-decon.pdf` Text of the thesis

A.1 Building and Running DeCon

It is recommended to follow instructions in the `README` file. However, given that all dependencies (Docker, Python, Doxygen – see `README`) are installed, DeCon can be started using this commands:

```
$ cd ./decon
$ ./run.sh
```

If no prior build was performed, the script starts the build operation. Depending on the system performance, this operation could take more than 30 minutes to complete. During the build, Docker base images are downloaded, DeCon images are built and databases initialized. After the build is finished, DeCon services are started and users can access the web application at `http://localhost/`.

Appendix B

Web Application

The screenshots represent the final implementation of the DeCon graphical user interface.

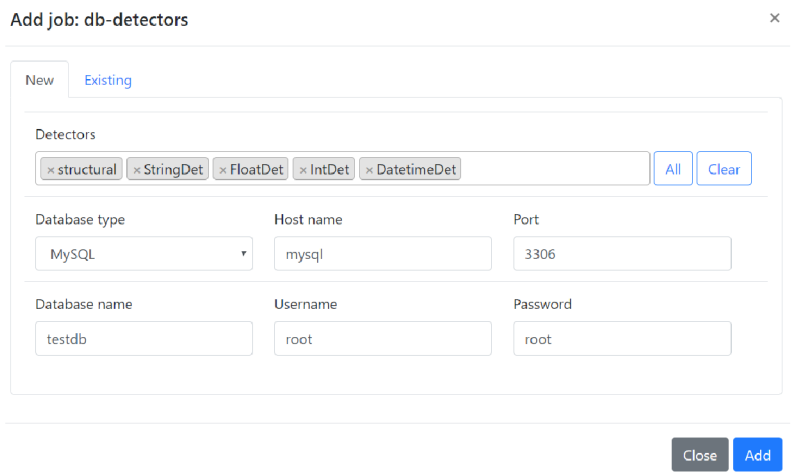


Figure B.1: Modal window for adding a new job for database detectors

Figure B.2: Modal window for adding a new job

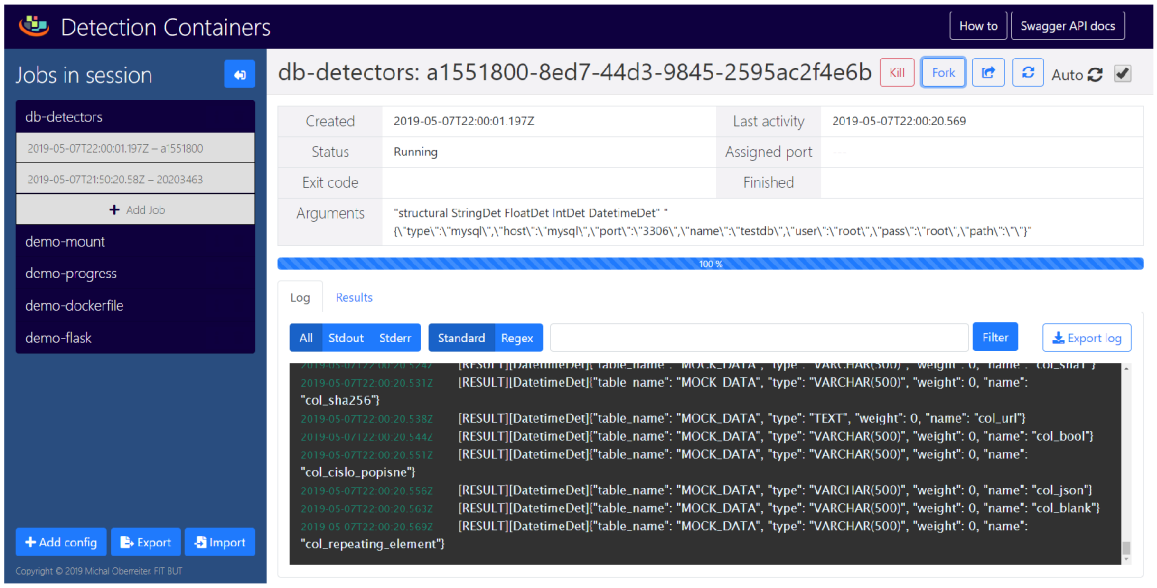


Figure B.3: DeCon running a database detectors job while sending data to the output console

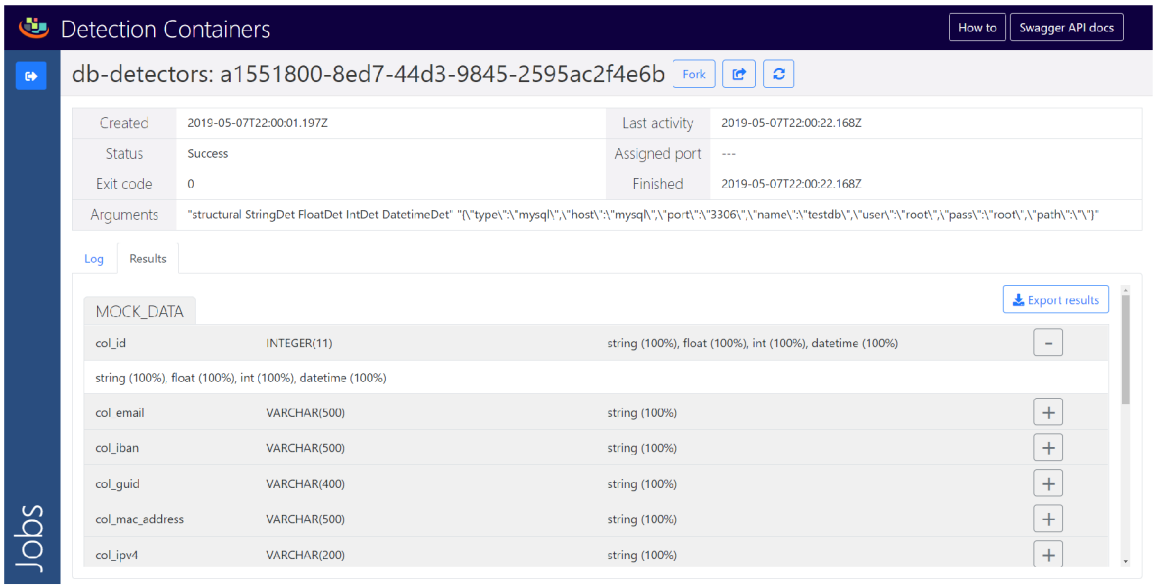


Figure B.4: Parsed results of a database detectors job

Appendix C

Code Samples

The following examples compare controller definition in ASP.NET Core Web API and Flask API. Both examples define Swagger attributes that are used for generation of Swagger specification.

```
/// <summary>
/// Controller for Configuration requests
/// </summary>
[ApiController]
public class ConfigurationController : BaseController {
    /// <summary>
    /// Configuration controller ctor
    /// </summary>
    /// <param name="repository">Repository patter</param>
    /// <param name="logger">Logger</param>
    /// <param name="connector">HTTP connector</param>
    /// <param name="contextAccessor">HTTP context accessor</param>
    public ConfigurationController(IRepository repository,
        ILogger<ConfigurationController> logger, IConnector connector,
        IHttpContextAccessor contextAccessor) : base(repository, logger,
        connector, contextAccessor) {
    }
    /// <summary>
    /// Retrieve configuration by name
    /// </summary>
    /// <param name="configuration">Configuration</param>
    /// <returns>Configuration model</returns>
    [Route("{configuration}")]
    [HttpGet]
    [SwaggerResponse(200, typeof(ConfigurationExternal), Description =
        "Configuration")]
    [SwaggerResponse(400, typeof(void), Description = "Invalid model state")]
    [SwaggerResponse(500, typeof(void), Description = "An exception has occurred
        during retrieval of configuration")]
    public ActionResult<ConfigurationExternal> GetCongfiguration([FromRoute]
        string configuration) {
        return Ok(...);
    }
}
```

Listing C.1: Example of controller implementation in ASP.NET Core

```
@configurations.route("/<id>")
@api.doc(params={'id': 'ConfigurationID'})
class ConfigurationsParam(Resource):
    """
    Configuration actions with parameters
    """
    @api.response(200, 'Success', configurationModel)
    @configurations.doc(responses={
        200: 'Success',
        404: 'Not found',
        500: 'Error retrieving configuration'
    })
    def get(self, id):
        """
        Configuration retrieval
        """
        return "", status.HTTP_200_OK
```

Listing C.2: Example of controller implementation in Flask

Appendix D

Docker Examples

These examples represent usage of Docker in this thesis and give samples of Dockerfiles used to build DeCon images.

```
$  docker create --rm --network=decon_internal
    --name=d548e996-4eda-4343-9405-f5a2e87d411e --volume=:/app/mount/
    decon-image-dbdetectors --token "d548e996-4eda-4343-9405-f5a2e87d411e" --file
    "/app/cli/db-reporter/run_detectors.sh" --workingDirectory "/app/cli/"
    --arguments "EncodedArguments"
$  docker cp /home/user/decon/db-detection/
    d548e996-4eda-4343-9405-f5a2e87d411e:/app/cli/
$  docker start d548e996-4eda-4343-9405-f5a2e87d411e
...
$  docker stop d548e996-4eda-4343-9405-f5a2e87d411e
```

Listing D.1: Concrete example of container management in DeCon

```
FROM microsoft/dotnet:2.2-sdk as dotnet-builder
WORKDIR /app/app-wrapper/
COPY ./ApplicationWrapper.sln /app/app-wrapper/
COPY ./src/ApplicationWrapper.csproj /app/app-wrapper/src/
RUN dotnet restore -r linux-x64
COPY . /app/app-wrapper/
RUN dotnet publish "ApplicationWrapper.sln" --no-restore --self-contained -c
    release -o /output

FROM debian
LABEL maintainer="xoberr00@stud.fit.vutbr.cz"

RUN apt-get update && apt-get install -y --no-install-recommends apt-utils
    build-essential python3-pip python3-setuptools python3-dev
RUN pip3 install wheel

RUN mkdir -p /app/app-wrapper
COPY --from=dotnet-builder /output/ /app/app-wrapper/publish/

WORKDIR /app/app-wrapper/publish
ENTRYPOINT ["/app/app-wrapper/publish/ApplicationWrapper"]
```

Listing D.2: Definition of the default Dockerfile for command-line applications

```
FROM mysql
LABEL maintainer="xoberr00@stud.fit.vutbr.cz"

RUN apt-get update && apt-get install -y --no-install-recommends apt-utils
RUN apt-get install -y dbus
RUN apt-get install -y dbus-x11
RUN apt-get install -y python3
RUN apt-get install -y python3-pip
RUN apt-get install -y python3-dev
RUN apt-get install -y wget
RUN apt-get install -y telnet
RUN apt-get install -y iputils-ping

RUN apt-get install -y build-essential libdbus-1-dev nlohmann-json-dev
    libspdlog-dev libglib2.0-dev
RUN apt-get install -y python3-levenshtein python3-mysqldb libdbus-1-dev
    nlohmann-json-dev libspdlog-dev libglib2.0-dev unixodbc-dev
RUN pip3 install lxml pandas fuzzywuzzy python-dateutil SQLAlchemy NumPy pyodbc

RUN mkdir -p /app/cli/db-reporter/
COPY ./db-reporter /app/cli/db-reporter/
RUN cd /app/cli/db-reporter/ && make

COPY --from=decon-app-wrapper /app/ /app/
ENTRYPOINT ["/app/app-wrapper/publish/ApplicationWrapper"]
```

Listing D.3: Custom Dockerfile example (demo-dockerfile)

Appendix E

API Models

Examples of the models used for communication with clients or between microservices.

E.1 Gateway Models

```
{
    "Id":"20203463-a9ce-4a56-ac9f-ad4be8ceb4a7",
    "Created":"2019-05-07T21:50:20.58Z",
    "Started":"2019-05-07T21:54:01.925Z",
    "Finished":"2019-05-07T21:54:15.957Z",
    "State":"Success",
    "ProgressPercentage":100,
    "Configuration":"db-detectors",
    "Arguments":"EncodedArguments",
    "lastActivity":"2019-05-07T21:54:15.957Z",
    "StdOut":[
        {
            "timestamp":"2019-05-07T21:54:14.042Z",
            "value":"[RESULT][StringDet]{\"type\": \"VARCHAR(500)\", \"name\":
                \"col_iban\", \"table_name\": \"MOCK_DATA\", \"weight\": 1.0}"
        }
    ],
    "StdErr":[

    ],
    "ExitCode":0,
    "HostPort":null
}
```

Listing E.1: Full Job entity model example

```
{
  "Id": "72c894b3-0036-4e06-92f4-e067e5914807",
  "Started": "2019-04-22T18:51:28.113Z",
  "State": "Running",
  "LastActivity": "2019-04-22T18:51:28.113Z",
  "ProgressPercentage": 42
}
```

Listing E.2: Job status model example

```
{
    "Configuration":"demo-mount",
    "Arguments":[
        "arg1",
        "arg2"
    ],
    "HostPort":null
}
```

Listing E.3: Job creation model example

```
{
    "Name":"db-detectors",
    "FilePath":"/decon/apps/db-detection/run_detectors.sh",
    "WorkingDirectory":"/decon/apps/db-detection/",
    "Mount":null,
    "Dockerfile":"/decon/apps/db-detection/Dockerfile",
    "ContainerPort":null,
    "JobTimeout":300,
    "IsVolumeMounted":false
}
```

Listing E.4: Configuration model example

E.2 Other Models

```
{
   "Id":"41ffb383-80b2-4487-a11c-704ca8a65654",
   "LogLevel":"Error",
   "EventId":"5e819f2c-8d29-47a8-aa9e-cd1a73a2afdd",
   "Name":"DetectionContainersAPI.Assets.ProcessWrapper",
   "Message":"Container start failed",
   "Timestamp":"2019-04-16T12:09:13.7370525+00:00"
}
```

Listing E.5: Log entry model example

```
{
    "Image":"decon-image-dbdetectors",
    "Dockerfile":"/decon/apps/db-detection/Dockerfile",
    "Cmd":[
        "--token",
        "7a71d0de-f6bf-4136-8e63-8da3f0910351",
        "--file",
        "/app/cli/run_detectors.sh",
        "--workingDirectory",
        "/app/cli/",
        "--arguments",
        "EncodedArguments"
    ],
    "HostConfig":{
        "NetworkMode":"decon_internal"
    },
    "ContainerId":"7a71d0de-f6bf-4136-8e63-8da3f0910351",
    "SourceFolder":"/decon/apps/db-detection/",
    "TargetFolder":"/app/cli",
    "PortMapping":"0:6000",
    "IsVolumeMounted":"false"
}
```

Listing E.6: Docker service container start model example