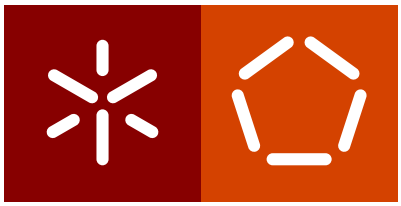


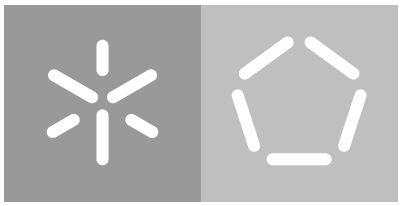
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Miguel Pinheiro Guimarães

Using container-based virtualization on web apps production environment

Dipcode development cycle

February 2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Miguel Pinheiro Guimarães

Using container-based virtualization on web apps production environment

Dipcode development cycle

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertation supervised by
Professor António Luís Sousa

February 2022

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer aos meus pais, que desde sempre me ensinaram que com trabalho árduo, esforço e dedicação é possível alcançarmos tudo aquilo que queremos, e que sempre com uma mão amiga me empurram todos os dias para conseguir chegar mais longe.

Aos meus avós, que apesar de não entenderem os "computadores", sempre me apoiaram com o maior sorriso e palavras de conforto e motivação.

Aos amigos que, durante este anos, partilharam comigo experiências inesquecíveis.

Ao Sandro Rodrigues e à Daniela Costa da Dipcode que, como mentores me apoiam não só no desenvolvimento deste projeto, mas também no meu constante crescimento como profissional.

Ao Professor António Luís Sousa, por tornar este projeto possível.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

With the fast evolution of the internet over the last years, the top priority on software development has shifted from **what?** to **when?**. Reduced time-to-market is now the competitive edge that all companies strive for.

The usage of container-based virtualization technologies keep the multiple environments where a development team works similar enough, that their work is made easier when developing and testing new features, which in turn results in a significantly faster delivery. The nature of this tecnhology also brings numerous advantages when it comes to management, monitoring and maintaining resources, allowing for an ease of adjustment, based on the client needs.

Throughout this dissertation is presented an extended base of knowledge about container technologies, especially Docker, as well as what are the basic techniques to use when building an application inside such infrastructure, from the writing of the Dockerfile to the adaptation of the multiple pipelines responsible to deploy the application.

KEYWORDS Docker, Docker Swarm, Kubernetes, orchestration, containers, containerization, services, build, deploy, web applications.

RESUMO

Com a rápida evolução da internet nos últimos anos, a prioridade máxima no desenvolvimento de software transformou-se de "o quê?" para "quando?". Disponibilizar rapidamente uma aplicação no mercado é agora a vantagem competitiva que todas as empresas ambicionam ter.

A utilização de tecnologias de virtualização através de containers uniformizam os vários ambientes em que a equipa de desenvolvimento opera, facilitando assim o seu trabalho no que diz respeito à adição e teste de novas funcionalidades, o que resulta numa entrega significativamente mais rápida. A natureza desta tecnologia trás inúmeras vantagens à gestão, monitorização e manutenção de recursos, permitindo facilmente aumentar ou reduzir os mesmos baseado nas necessidades dos seus clientes.

Nesta dissertação é apresentada uma extensa base de conhecimento sobre as tecnologias de containerização, em especial o Docker, bem como quais as técnicas base a utilizar quando se pretende construir uma aplicação com uma infraestrutura deste tipo, desde a escrita do Dockerfile, até à adaptação das várias pipelines responsáveis por disponibilizar a aplicação em ambiente de produção.

KEYWORDS Docker, Docker Swarm, Kubernetes, orquestraçã, containers, containerização, serviços, build, deploy, aplicações web.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

A

API Application Programming Interface.

ARP Address Resolution Protocol.

C

CD Continuous Deployment.

CI Continuous Integration.

CPU Central Processing Unit.

D

DI Departamento de Informática.

I

IT Information Technology.

L

LTS Long Term Support.

M

MCP Multicore Processing.

MIEI Mestrado Integrado em Engenharia Informática.

O

OS Operating System.

R

RAM Random Access Memory.

S

SW Software.

T

TLS Transport Layer Security.

U

UI User Interface.

UM Universidade do Minho.

UX User Experience.

V

VM Virtual Machine.

1

INTRODUCTION

This dissertation describes the Master's work developed in the context of *Mestrado Integrado em Engenharia Informática (MiEI)* held at *Departamento de Informática (DI)*, *Universidade do Minho (UM)*.

Dipcode[13] is a Eurotux Group company specialized in tailor-made web development, that is looking to modernize their *Continuous Integration (CI)* system through virtual containerization technologies. Containers are a type of software that can share access to a machine's *Operating System (OS)* kernel, without the use of conventional *Virtual Machine (VM)*. This technology allows the development team to abstract the code from the application and its infrastructure, hence simplifying the version control process, and enhancing the portability of the application to multiple production environments.

The main objectives of this dissertation include the investigation of the best practices to assure security, backups, logs and monitorization of the application.

Dipcodes' current CI system will also need to be adapted to generate, test and submit new images to the repository. Last but not least, the production machines must be prepared to execute those images, updating the *Continuous Deployment (CD)* system to keep the images in production up to date.

Throughout this document will be detailed the lifecycle of a standard web application; how virtualization improves the non-functional aspects of an application; the two types of virtualization: via VM and containers, with a bigger focus on Docker; how conteiners can be deployed through multiple environments using orchestrators; and finnally tying it all together with an example application built from scratch, as well as a special focus on how the set of good practices acquired during the research of the present dissertation has already been utilized within Dipcode.

2

STATE OF THE ART

This Chapter aims to contextualize the focus of the dissertation, starting by describing what web development is, as well as detailing the cycle of developing a web application. Afterwards, we'll be discussing how CI and CD play a role in web application development, as well as what virtualization is, with a special focus on containerization and Docker.

2.1 WEB DEVELOPMENT CYCLE

The discussion in this dissertation focuses heavily on the deployment and production phases of the Web development cycle. As such, in order to understand its context, we will first talk about the web development cycle itself and all its phases[33].

2.1.1 *What is web development?*

Web development, also commonly known as web programming, involves the creation of dynamic web applications that use web browsers and technology to perform tasks over the internet[44]. Nowadays, millions of businesses use the internet as a cost-effective communication channel. It allows them to exchange information with their target market, as well as performing secure and fast transactions. However, effective engagement is only possible when the business is able to capture and store all the necessary data, having a way to process it and present it to the user in an engaging manner.

2.1.2 *Front-end and Back-end*

As a good practice, web developers usually start by dividing their applications in two big categories: the front-end, or client-side development, and the back-end, or server-side development[1].

The front-end refers to constructing what the end user sees when the web application is loaded by the browser. This includes the design, content and the multiple possible

interactions. Essentially, it is responsible for the look and feel of the application, and for how the user interacts with it.

Back-end development controls what happens behind the scenes, managing all the necessary data to be presented by the front-end. It is usually the backbone of the software application, since it stores and manages all the data needed to run as smoothly and efficiently as possible.

2.1.3 *Risks and costs*

At the end of the day, building a web application from scratch is extremely complex and involves a variety of costs and risks to be accounted for. The main components that affect web applications development cost, and that involve risk in one way or another are[29]:

(i) The scope of work;

(ii) Business niche;

(iii) The technical complexity of the project's features;

(iv) *User Interface (UI)*/*User Experience (UX)* design;

(v) Deadlines;

(vi) Non-functional requirements;

(vii) Development company.

We'll quickly sweep through these factors separately and see how they affect the overall cost of the web development process.

Scope of work

The scope of work may be described as a set of features that the application is required to possess and the amount of work required to develop them. Trying to implement all the desirable features at once will not only prolong the development process but also increase the initial costs.

On the other hand, focusing on a smaller set of valuable features and implement additional ones through upcoming updates dramatically reduces the upfront cost. Obviously, using this approach will increase the cost of the development process in the long run, but since the digital world is in constant flux and the best decisions are made based on insight, the client merely pays for the features needed by their business at that given moment, reducing the risks of implementation.

Business niche

A broad business domain means that there are plenty of available people with the ability to turn an idea into reality. A huge pool of contractors means that the price has to be kept in check in favor of healthy competition. If a niche is extremely narrow, the cost of development can skyrocket, as it can be difficult to find developers that know their ways in such a business. A company employed with such a task may be required to hire speacialists, or organize additional training for their staff. All this prolongs the development and increases its cost[19].

Technical complexity of the project's features

There's a set of features that are a standard for web applications to include, that don't require a specific set of skills to implement, and usually companies do not charge much for them. However, other features such as third party integrations can be rather tricky to include in the project.

Usually, this integrations are done via *Application Programming Interface (API)*, that developers normally have had no prior interaction with, and have no control of going further. This third party technologies come with very high risks, as they may present compatibility problems with the system at hand. Providers may also make changes to their API, breaking the application until these conflicts are resolved. This means that the development team needs to thoroughly research the API, as well as extensively test it to determine how the application interacts with it. This, of course, elevates the cost by a considerate margin[19].

UI/UX design

Custom web applications designs can be quite expensive, as it needs to be unique from all the other applications out there, and it needs to capture and reflect the identity and philosophy of the business brand. Pretty effects, awesome animations, company logos, etc. all take additional time, and so cost more money. If a client plans for their app to be used across multiple devices and browsers, those specifications need to be prepared beforehand, and will also increase the cost. Of course, there are cheaper alternatives, such as using a standard design consisting of pre-built templates and themes[19].

Deadlines

The costs of development are greatly affected by the time limits imposed on the development team. A tight deadline usually means more resources need to be dedicated to the process, increasing its costs, as well as higher risks and more effective management[19].

Non-functional requirements

Every software product has a set of functional requirements, i.e., a list of things the application should be capable of doing. All the other requirements can be grouped into a category of non-functional, or quality, requirements[8]. This category includes attributes that describe how the application works. This may include scalability, security, usability, realiability, performance, etc. Their importance lies in the fact that they influence the experience that users have when interacting with the system.

In a web application, one of the most important attributes is the system's ability to deal with a great number of requests. High-load systems are expected to be used by up to thousands, or millions, of users daily, and they should be reliable, secure and present almost no down time. Also, their server needs to run on machines that are far more expensive than your average web application, greatly increasing its costs[19].

Choice of developer

The company entrusted with creating a web application is one of the most important variables that influences the cost of development. The more the company pays its employees, the more it charges its clients. Hourly rates for web application developers depend on three key factors: geographical location, experience and company's size, expertise and reputation.

As such, the development of a web application requires a rigorous process to be executed successfully.

2.1.4 *Development cycle*

The Internet changed software development's top priority from *what* to *when*[3]. Reduced time-to-market has become the competitive edge that leading companies strive for. Thus, reducing the development cycle is now one of software engineering's most important missions.

In this Section, we'll be discussing what the process involves, talking about each phase.

First discussion

Usually, web applications spawn from the client needs. As such, the first phase of development aims to give an overview of what the problem and needs are. If these components are not well understood it will lead to a failure in building the web application. For that purpose, the following questions need to be answered once this first step is done[32]:

(i) What are the underlying needs behind the application being built?

(ii) What problems should this application solve?

(iii) What needs does the application fill?

(iv) What will be its impact on the organization/client business?

(v) How will the application be used?

(vi) What could be the consequences of the delay in building the application for the business?

Requirement analysis phase

Having identified the objectives of the application implementation, the next step is to analyze the requirements. Since the objectives and the business logic can be quite complex, it is important at this stage that this complexity can be broken down into smaller and more manageable tasks that are easier to implement, validate and test.

During this step it's important to:

(i) Reformulate the underlying needs and goals;

(ii) Break down the project into the smaller steps that are needed to complete it;

(iii) Identify each different feature and module;

(iv) Break down every goal into simpler tasks;

(v) Consolidate the different analysis and approaches in a document so that validation can be carried out;

Timeline and cost estimation phase

After completing the requirements document, the timeline and cost need to be estimated. In this phase, experience is a developers best ally. If a similar task has been done before, it becomes easier for one to give an accurate estimate and time tracking becomes a reference. From the previous step, all the pieces needed to put together a project from end to end are now available. For each task, it gets broken down into smaller subtasks if necessary, its complexity is analyzed, and a classification based on priority is given.

The design, conception and planning phase

Since the last step gives an idea of the time needed to complete the whole project, the next phase involves the design, conception and planning phase. During this step the goal is to distribute the tasks between the several developers, and decide the best technical approach to each task at hand. In this phase, the team should:

(i) Order each task according to their priority

(ii) Analyze dependencies between tasks

(iii) Identify each task that can be done in parallel

(iv) Identify the number of resources available and needed to complete the project and elaborate a timeline accordingly

(v) Make an architectural choice

Generally, at this stage, a financial and technical side of the web application development is settled and there is an idea of the necessary budget.

The development phase

The most well known step of the whole process is the development phase, and its start takes place after the requirements document is established. The goal of this phase is to create an application that meets the needs identified in the previous steps. Since the needs generally tend to evolve and new ideas of implementation may arise during the development phase, the use of a flexible and proactive methodology, such as Agile, is advised.

Agile methodologies argue that client satisfaction should be sought through the Continuous Deployment of software that adds value to the final solution, by keeping up constant communication with the client and, also, by focusing on communication between team members. Contrary to previous methodologies, Agile is not characterised by the complete definition of a product, but by a dynamic interaction that allows constant delivery.

The product requirements documentation should be used to implement all features, and an efficient development approach should be chosen in order to make sure the code is behaving as expected by the specification document.

The testing phase

After each iteration of the Agile methodology, commonly called a sprint, a testing phase takes place, where a group of users validates the set of features implemented and report bugs, if any were found. If there are changes to be made, they can be discussed with the project manager who will advise about the best way to take inputs and feedback into consideration. After all features are validated, they are then bundled together and deployed to production. This is usually called a release.

The deployment and post-production phase

Once every feature in a given release is validated and all the reported bugs are fixed, the deployment phase can start. This deployment can be the first launch or an addition of a new release, and it is the moment when the application will run in its real environment.

Some behaviors only appear in a production environment such as load balancer performance. As such, it is important to take time for support and maintenance of the application in order to fix any bugs or performance issues that are discovered after the application went live.

New ideas for extensions may appear once the application is up and running, and post-production phase is the best moment to talk about future needs.

2.1.5 *Continous Integration and Deployment*

The way companies deliver software is going through a wave of change as a result of the environment surrounding them. Market needs and technologies continuously shift, and so there is an increased pressure to adapt and deliver quickly. Companies can no longer afford to make customers wait for several months for a release, and customers expect continuous engagement so that they can provide continuous feedback.

DevOps is a set of practices that tries to bridge developer-operations gap at it's core[42], and covers all the aspects which help in fast, optimized and high-quality software delivery. It extends agile principles to the entire software delivery pipeline.

Continuous Integration

CI refers to early integration, keeping the changes from being isolated in one workspace, and instead promoting its sharing with the team to continuously validate the behavior of the code. This stage of process optimization refers to achieving automation such that as soon as a developer submits a piece of code, a pipeline to test the code is triggered, and the build is posted to a shared repository.

Continuous Deployment

CD can be considered the heart of DevOps[47] and represents the critical piece of software delivery optimization. Setting up the hardware to test a deployment build may take several days, or even weeks and, on top of that, the deployment process is manual, and not consistent. DevOps principles recommend an automation of the deployment process, through an approach that proposes that the entire infrastructure provisioning should be maintained as code in a repository.

2.2 DEPLOYMENT AND POST-PRODUCTION ENVIRONMENT

As stated earlier, this document focuses heavily on the deployment and post-production phases of the web development cycle. It is usually in this phase that the application will be

spawned into its real environment, but what exactly is this so called environment? As it is known, every piece of software needs physical hardware resources or it will be rendered useless. As a standard procedure, companies usually allocate and prepare specific physical machines to host their applications, however there is another viable solution to this problem that will not only heavily reduce costs and wasted resources, but also facilitate the software updates on the post-production phases. That concept is called virtualization[2].

2.2.1 *Basics*

We'll start out by talking about the basic concepts of *Multicore Processing (MCP)* [20], and the two main types of virtualization: via VM's and hypervisors, and via Containers[18].

So, what is MCP ? Just like the name implies, it's computer processing performed using multicore processors. A multicore processor is a single integrated circuit that enables *Software (SW)* applications to run on multiple real processing units, commonly known as cores.

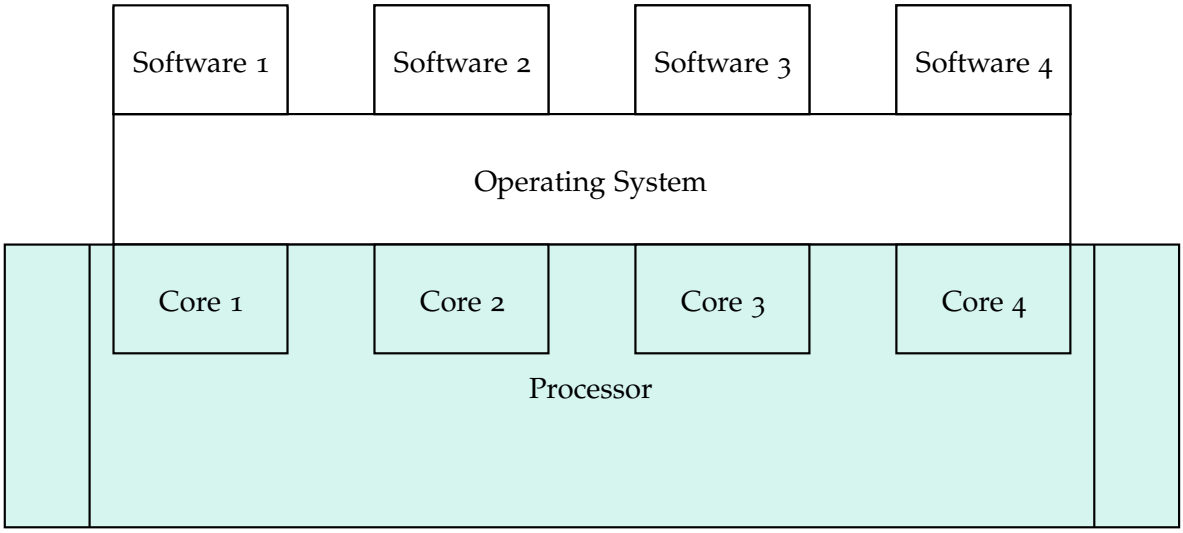


Figure 1: Multicore Processing (MCP) adapted from [20]

Virtualization, on the other hand, is a collection of SW techonologies that can run on virtual hardware (VM's), or virtual operating systems (containers).

2.2.2 *Virtual Machines*

A VM is a software that simulates a machine, i.e., a hardware plaftorm that provides a virtual operating environment for operating systems. To put it simply, this software makes it possible to run what appears to be many separate computers on hardware that is actually just one.

The multiple OS and their applications share hardware resources from a single host server or from a pool of host servers. Each VM requires its own uderlying OS, and the virtualization of its hardware.

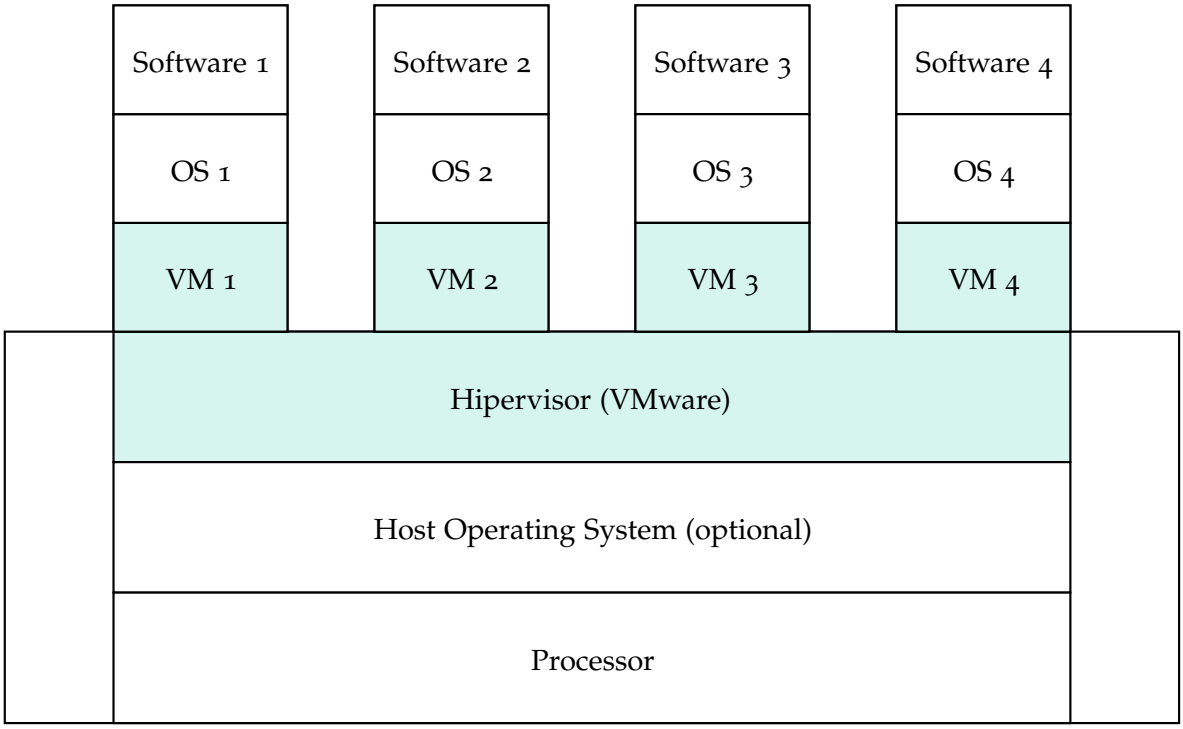


Figure 2: Virtualization via Virtual Machines adapted from [20]

VM's are monitored by a hypervisor, which sits between the VM and the hardware (host machine), and supervises the creation and execution of the operating systems running on the VM.

Although many large and small *Information Technology (IT)* departments have embraced VM's as a way to lower costs and increase efficiency, they can take up a lot of system resources. Each VM runs not only a full copy of an OS, but also a virtual copy of all the hardware that said OS needs to run. This quickly adds up to a lot of *Random Access Memory (RAM)* and *Central Processing Unit (CPU)* cycles. The benefits of VM's include:

(i) All OS resources are available to apps;

(ii) Established management tools;

(iii) Established security tools;

(iv) Better known security controls.

2.2.3 *Containers*

On the other hand, instead of virtualizing the entire underlying machine, containers only virtualize the OS itself. They run atop a single OS kernel, sharing their resources, like binaries and libraries, between them. These sharing of resources significantly reduces the need to reproduce OS code, and it means that a server can run multiple workloads with a single OS installation. Thus, when running the same applications, containers are lighter - their images are just a few megabytes in size and usually take just a few seconds to start.

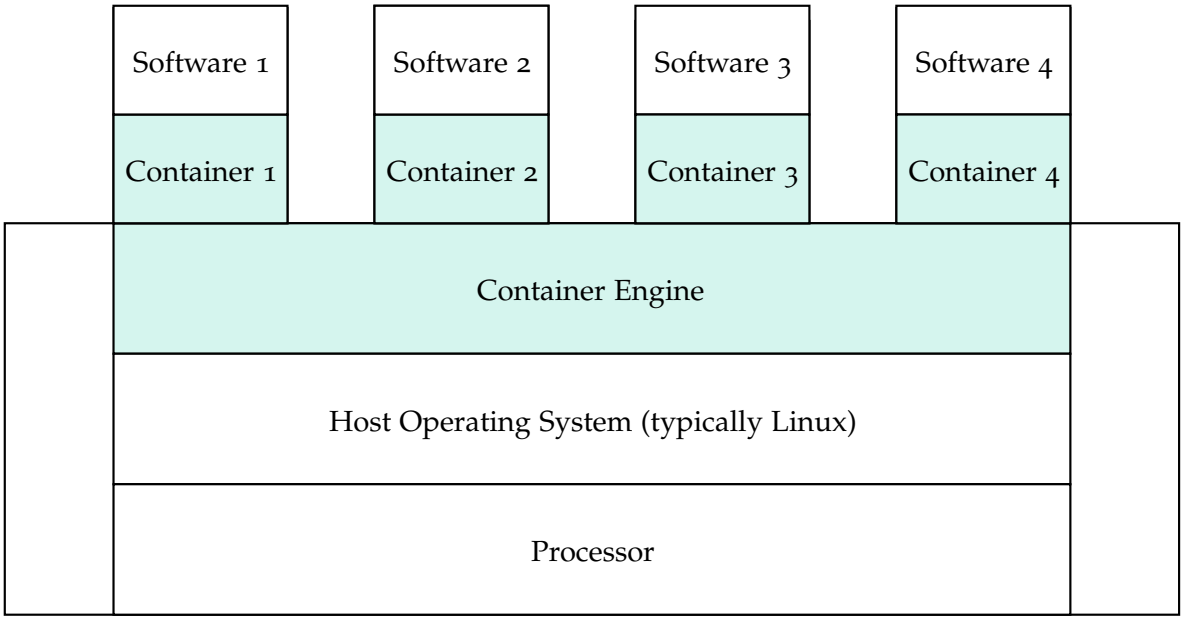


Figure 3: Virtualization via Containers adapted from [20]

In contrast to VM's[17], all that a container requires is enough of an OS to support programs and libraries, and system resources to run a specific program. What this means is you can run two to three times as many applications on a single server with containers, than you can with a VM. In addition, with containers you can create a portable, consistent operating environment for application development, testing and deployment. Some of its benefits include:

(i) Reduced IT management resources;

(ii) Reduced size of images;

(iii) Quicker spinning up of applications;

(iv) Reduced and simplified security updates;

(v) Less code to transfer, migrate, upload workloads.

2.2.4 *Comparison*

Figure 4 provides another way of looking at the three technologies by considering the layer at which each technology exists. MCP provides multiple real hardware plaftorms. Virtualization by VM's can sit on top of MCP and provide multiple virtual hardware plaftorms. Virtualization by containers can sit atop either virtual or physical machines, and it provides multiple virtual operating systems. The figure also shows that:

(i) Multiple software applications can be allocated to a single container

(ii) Multiple containers can be allocated to a single VM

(iii) Multiple VM's can be allocated to a single core

(iv) Multiple cores are contained by a single multiprocessor core

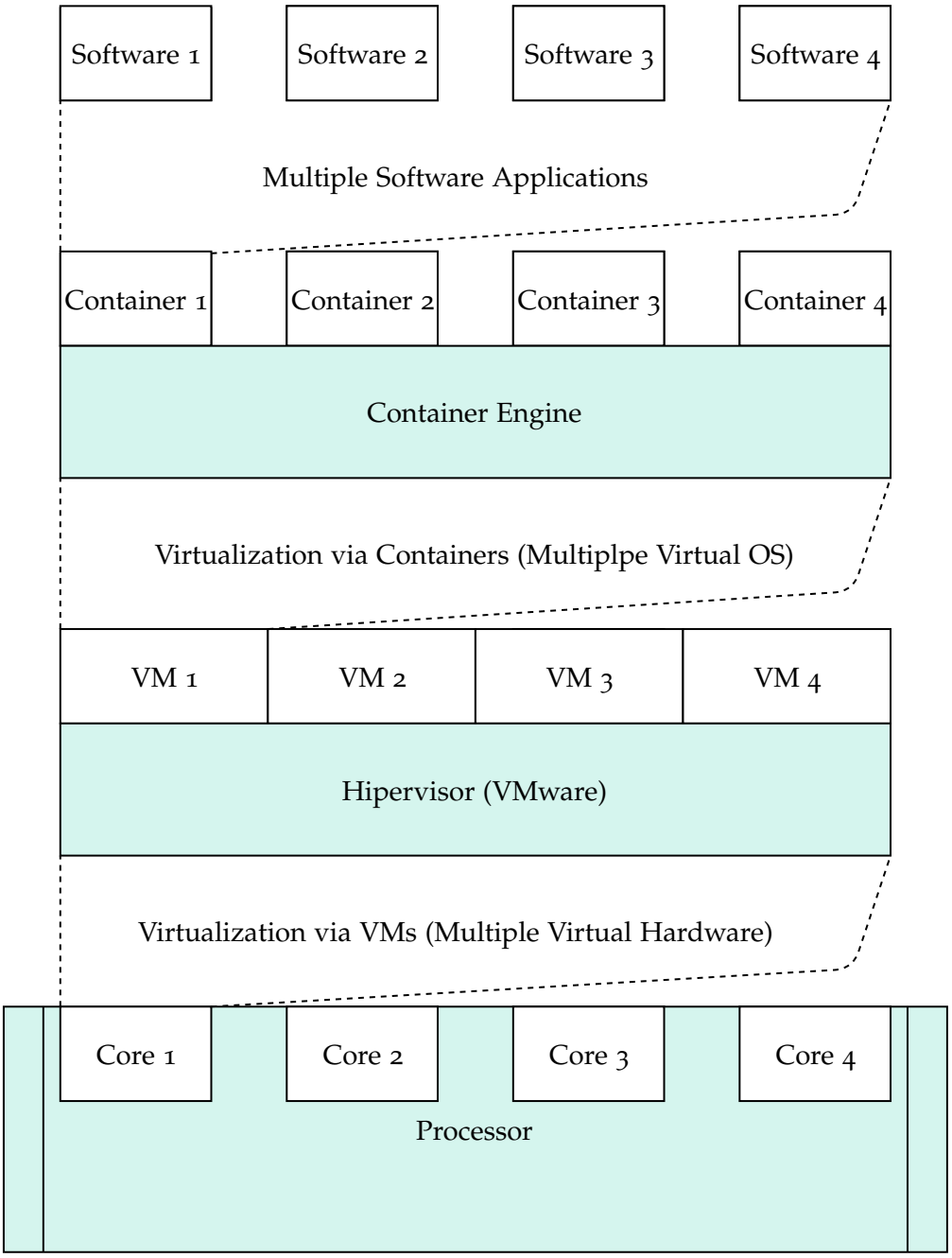


Figure 4: Possible stack of techonologies adapted from [20]

Virtual Machines	**Containers**
– Heavyweight	– Lightweight
– Limited performance	– Native performance
– Each VM runs on its own OS	– All containers share the host OS
– Hardware-level virtualization	– OS virtualization
– Startup time in minutes	– Startup time in seconds
– Allocates required memory	– Requires less memory space
– Fully isolated and hence more secure	– Process-level isolation, possibly less secure

Table 1: Virtual Machines vs. Containers

From the table above, we can see that containers appear to be a good option when it comes to building tailor-made, high-performance, high-responsiveness web applications that can be easily updated and do not use more resources than those necessary. Therefore, from now on, this discussion will be focused on virtualization via containers.

2.3 CONTAINERIZATION

So far, we have compared the functional aspects between the use of containers and virtual machines[5], however that was but the surface of what container-based environments have to offer. In this Section, we will delve into what it means to use containerization throughout the development of an application[7] and how its use could impact the later stages of the application lifecycle.

We will wrap up the Section by addressing Docker, and why it has become the de facto industry standard in terms of containerization solutions so quickly as it will be the technology used to further the investigation outlined in this document.

2.3.1 *Application-Oriented Infrastructure*

It has become apparent over time that containerization benefits go beyond simply allowing higher utilization levels. Containerization turns the data center from machine-oriented to application-oriented. We will be addressing the application environment, containers as the management unit, and security concerns in this Section.

Application Environment

Containers encapsulate the application environment, abstracting many details of machines and OS from the application developer and the deployment infrastructure.

The original purpose of the kernel's cgroup[41] chroot and namespace facilities was to shield applications from disruptive, nosey, and messy neighbors. Combining these with

container images created an abstraction that isolates applications from the OS where they are running. This decoupling of image and OS allows both development and production stages to provide a similar environment, which in turn accelerates growth and increases delivery efficiency by reducing inconsistencies and friction.

The key to making this abstraction work is to have a hermetic container image capable of encapsulating almost all the dependencies of an application into a package that can be deployed into the container. The only local external dependencies, if done correctly, are on the system-call interface of the Linux kernel. While this limited interface improves image portability, it is far from perfect as applications can still be exposed in the OS interface, particularly in the wide area of socket options exposed.

Nevertheless, the container-provided isolation and dependency minimization has been shown to be quite efficient.

Containers as the unit of management

Building software systems around containers rather than machines shift the "primary key" of the data-center from machine to application. This change presents many benefits, as it relieves application developers and operations teams from worrying about specific machine and OS details; it provides the infrastructure flexibility to roll out new hardware and upgrade OS with minimal to no impact on running applications and their developers; and it ties telemetry collected by the management systems (metrics such as CPU and memory usage) to applications rather than machines, which dramatically improves application monitoring and introspection.

Containers provide convenient points to register generic API that enable the flow of information between the management system and an application without either knowing much about the particulars of the other's implementation. The container-management system can communicate information into the container such as resource limits, container metadata for propagation to logging and monitoring, and notices that provide graceful-termination warnings in advance of node maintenance.

Containers can also provide application-oriented monitoring in other ways, such as through Linux kernel cgroups that provide application resource-usage data and can be extended with custom metrics and exported using HTTP API. This data allows the development of generic tools such as an autoscaler that can record and use metrics without understanding the specifics of each application. Because the container is the application, there is no need to demultiplex signals from multiple applications running inside a physical or virtual machine. This is simpler, more robust, and allows a finer-grained reporting and control of metrics and logs. Compare this to having to ssh into a machine to run top. Though developers can ssh into their containers, they rarely need to.

Security Concerns

Docker's security relies on three key components: the isolation of processes at the userspace level managed by the Docker daemon; the enforcement of this isolation by the OS kernel; and the network operations security[9].

Regarding isolation, Docker containers rely exclusively on Linux kernel features like namespaces and cgroups, hardening and capabilities. Namespace isolation and capabilities drop are enabled by default, however, cgroup must be enabled on a per-container basis. Docker's default isolation configuration is strict, its' only flaw being the fact that all containers share the same network bridge, enabling *Address Resolution Protocol (ARP)*[34] poisoning attacks between containers on the same host.

Global security can be increased or decreased based on the options triggered at container launch. Some options give extended access on the host to containers, hence decreasing the overall security and there are even some options that forbid network communication between containers altogether, mitigating the ARP attack described before.

Host hardening through Linux kernel Security Module is a means to enforce security-related limitation constraints imposed on containers, and currently, only a handful of Linux distributions are supported. Default hardening protects the host from its containers, but can not protect containers from other containers. Containers can only be protected from each other by writing specific profiles depending individually on each and every single one.

When it comes to network security, Docker utilizes network resources for image distribution and remote control of the Docker daemon. Concerning image distribution, images downloaded from a remote repository are verified with a hash, and the connection, unless specified otherwise, is made over *Transport Layer Security (TLS)*[40]. Also, the Docker Content Trust architecture allows developers to sign their images before being pushed to the repository. It relies on The Update Framework and was designed to rectify the package manager flaws, with the tradeoff being complex management of keys.

The daemon is remote-controlled through a socket, making it possible to perform any Docker command from another host. By default, this socket is a UNIX socket, but it can be interchangeable with a TCP socket. Access to this socket allows to pull and run any container in privileged mode, therefore granting root access to the host. In the case of a UNIX socket, any user belonging to the docker group can gain root privileges, and in the case of a TCP socket any connection can grant root privileges on the host, therefore this connection must be secured by TLS.

2.3.2 *Docker*

Docker has been mentioned in the previous Section, but why should it be used over the several other existing containerization solutions? Over the years, technologies like

FreeBSD Jails[22], Solaris Containers[12], LinuX Containers[11] and Warden[48] have paved the way regarding the development of containerization solutions. However, it was with the introduction of Docker that caused its popularity to explode.

Docker is a containerization platform that allows users to easily pack, distribute and manage applications within containers[46]. In other words, it is an open-source project that automates the deployment of applications inside software containers.

Nowadays, Docker stands out not only from VM platforms but from alternative container technologies. The reason for this is that it offers several features that are not available elsewhere, including:

(i) Faster startup time. As we have discussed already, compared to VM's, containers are usually faster, to the point that they usually take just a few seconds to startup, whereas VM's can take up to several minutes. Container solutions startup time is all roughly the same, however, they lack many of Docker's other important features.

(ii) Small attack surface. A containerized application that runs using Docker includes a minimal amount of overhead. From a security perspective, this is attractive because it means Dockerized applications have a smaller attack surface.

(iii) Open-source licensing. From the start, most of the Docker stack has been open-source. Specifically, Docker software was licensed mostly under Apache and MIT-style licenses. These licenses, which are more liberal than the GPL (which governs Linux among other older open-source platforms) give developers a great deal of flexibility. They make Docker different not only from closed-source platforms like VMware but also from KVM and LXC, both of which are governed by the GPL or closely related licenses.

(iv) Cross-platform support. While Docker initially was a Linux-only technology, that is no longer the case. Docker containers can now be run in almost any type of environment or OS, which is a tremendously distinctive feature. While alternative deployment features like VMware are cross-platform, technologies such as KVM and LXC remain tied to particular OS.

(v) A flourishing ecosystem. The years following Docker's public release in 2013 saw a thriving ecosystem growing up around itself. Established companies (including Red Hat and Microsoft) and startups alike (from CoreOS to Twistlock and Rancher) contributed code, documentation, standards, marketing efforts, and much more to help build a diverse and dynamic ecosystem surrounding Docker.

3

DEPLOYING APPLICATIONS USING DOCKER

3.1 BASICS OF DOCKER

Docker can build images automatically by reading instructions from a Dockerfile. In a way, the Dockerfile is the base from which the resulting application will be built upon, and it is imperative that this file is solid and does not introduce vulnerabilities in the software. Throughout this Section, we'll be exploring the elementary concepts of Docker using a Python application as a case study, since it is the basis of many applications built and delivered by Dipcode.

3.1.1 *Why most Dockerfile examples should be avoided*

When getting started with a new technology, developers usually begin by searching the web for information. Luckily for anyone getting started with Docker, it provides an extensive and detailed documentation[14] to help begginer and experienced developers alike. Besides the official Docker documentation, due to its open source community, there are countless other sources where one can find information on how to *Dockerize* an application.

Unfortunately, most of the examples found are often broken in multiple ways. Let's consider the following Dockerfile, which is a standard example that can be found when searching for *"Python Dockerization"* examples:

```
FROM python:3

COPY script.py /

RUN pip install flask

CMD ["python", "./script.py"]
```

Listing 3.1: Basic Python Dockerfile

There are a few problems with this Dockerfile, some directly related to security, and some that can break the entire application in a single deploy. The first problem arises from the base image selected. By using the python:3 image, at the time of writing, Python 3.9 will be installed. However, at some point in the future, Python will be upgraded, and Python 3.10 or higher will be installed instead. At that point, the application might not be configured to use the new version of Python, and the deploy will break production.

We'll be delving into the choice of a solid base image further ahead in this dissertation, but for now a simple solution would be to pin the version and operating system in the image, by using the python:3.7.3-stretch image instead.

Similarly to the base image, a dependency like flask is being installed with no versioning, therefore each time the image is rebuilt, potentially a new version of flask (or one of its dependencies) could be installed. A solution, adopted already as a Python good practice, is to pin the dependencies and respective versions in a requirements file, or directory, using pip-tools[26], Poetry[35] or Pipenv[37].

The next problem represented in the example Dockerfile does not come from any faulty line, but rather from the mispositioning of one. In order to secure fast builds, one should take advantage of Dockers' layer caching. It is a mechanism that prevents any line in the Dockerfile to be executed unless it was changed since it was last built. However, if a change is detected, all subsequent layers will be rebuilt again as a result. In this specific case, the code files are copied in before the dependencies are installed. This means that, every time new code is deployed, the dependecies will be installed again, even if they weren't modified at all, and all subsequent layers will be rebuilt.

In sum, layers that are prone to be changed between deploys, like copying in the code, should be pushed down on the Dockerfile, to avoid cache invalidation, and layers that are not likely to change at all should be pushed up, in order to save as much time as possible when building the image.

The last problem regarding the Dockerfile that I would like to address is security related. By default, Docker containers run as root, which in and of itself is a serious security risk, so it is highly advised to create a dedicated user inside the image, so operations that require root permissions are not allowed to be carried on.

By improving on the points above, we can rewrite the Dockerfile:

```
FROM python:3.7.3-stretch

COPY ./requirements /requirements

RUN pip install -r  /requirements/production.txt

RUN useradd --create-home appuser
WORKDIR /home/appuser
```

```
USER appuser

COPY script.py .

CMD ["python", "./script.py"]
```

Listing 3.2: Dockerfile regarding security concers

Even if this resulting image is somewhat better than the one we started with, it is still far from the ideal one that we would want running in a production environment. Throughout this document, we will steadily talk about other important points, and improve our Dockerfile until it is ready to be used in production.

3.1.2 *Choosing a base image*

When building a Docker image for a given application, one has to build on top of an already existing base image, and the Docker Hub[15] provides many possible choices. There are OS images like Ubuntu, CentOS, and many different variations of the Python base image.

Throughout this Section we will be discussing some of the relevant criteria one should follow when choosing a base image, in order to make a reliable decision as to what image should serve as the base for the application moving forward.

What a base image should provide

- **Stability:** With every subsequent build the application goes through, the same basic set of libraries, directory structure and infrastructure should be kept similar. otherwise the application might randomly break;
- **Security updates:** The base image should be well maintained, so that the base security updates are delivered as frequently and as soon as possible;
- **Up-to-date dependencies:** When building a complex application that could be used by possibly hundreds or thousands of daily users, it is important that OS installed libraries and applications (like compilers) are kept up to date;
- **Extensive dependencies:** Depending on the application being built, some less popular dependencies may be required, and a base image with access to a large array of libraries facilitates this;
- **Up-to-date Python:** Having an up-to-date version of Python embedded in the image saves the effort of manually installing it;

- **Small images:** The smaller the chosen image is, the easier it is to build and distribute it.

There are three base images that fit the criteria mentioned above: the latest versions of Ubuntu: ubuntu:18.04 and ubuntu:20.04 that, being *Long Term Support (LTS)* releases, will be maintained and continue to get security updates until 2023 and 2025 respectively. CentOS 8 (centos8), released in 2019, will have full updates until 2024, and maintenance updates until 2029, and Debian (buster), also released in 2019, will be supported until 2024.

Another alternative is to use Docker's own "official" Python image, which comes pre-installed with multiple versions of Python, and has multiple variants, such as Alpine and Debian Buster. Buster comes with many common packages installed, which can make the image itself quite large, so there is a Buster Slim variant that lacks some packages, and the resulting image is smaller.

Alpine, on the other hand, is often recommended as the go-to base image, due to being a smaller image, and faster to build. Most Python packages include pre-compiled binary wheels in order to reduce installation time. However, due to the way Alpine is configured, it does not support the installation of Python wheels, so all the packages need to be compiled after downloading.

Based on the criteria and exploration above, any image with the Debian Buster OS would provide a solid base image, since it is more up to date than something like the Ubuntu 18.04 image, and it is not as recent as Ubuntu 20.04, which could cause some problems down the road, with bugfixes amongst other problems. Overall, any base image based on the Debian Buster distribution can be considered a stable image, that will not have any significant library changes in the foreseable future.

As such, in order to get a full range of Python releases, we will be choosing python:3.x-slim-buster, depending on the version of Python of our choice. The image will provide all the benefits of a Debian Buster dsitro, and will keep the image small.

3.2 SECURITY

In this Section, we'll be covering the security measures that need to be taken in order to ensure that the Docker image generated can be deployed without hindering the entire structure that supports it, protecting both the users of the application, as well as the maintainers and deployers.

As we have discussed previously, Docker's security relies on three key components: the isolation of processes at the userspace level managed by the Docker daemon; the enforcement of this isolation by the underlying OS kernel; and security regarding the network operations.

In the next few sections, we'll be covering on how to achieve this level of security, using a standard Django application as our basis.

3.2.1 *Less capabilites, more security*

One important part of running a container in production is locking it down, to reduce the chances of an attacker using it as a starting point to exploit the whole system. Due to their nature, containers are inherently less isolated than virtual machines, and as such, more effort is needed in order to secure them.

Ensuring this security can be broken down into two main points: first, don't run the container as root, and second, the container should be run with as few capabilities as possible.

Do not run as root

Due to security reasons, running a container as the root user should always be avoided. This means that the running process will inherently have less privileges and, if somehow it becomes remotely compromised, the attacker will have a harder time escaping the container, and compromising the remaining infrastructure.

Not running as root also means that any user inside the container will not try to take actions that require extra permissions. This means that the container can be run with less capabilites, securing it even further.

What exactly is a capability

The term capability refers to Linux capabilities that determine exactly what operations a given process can and can not do. Typical servers run several processes as root: SSH daemons, cron daemon, logging daemons, kernel modules, network configuration tools, amongst other. A container is different, because almost all of those tasks are handled by the infrastructure around the container instead.

SSH access are usually managed by a single server running on the host. The cron daemon should be run as a user process, dedicated and tailored to the application that needs the scheduling service, rather than being a platform-wide facility. Due to the nature of containers, log management should be handed to a third-party service. And lastly, unless the container is engineered to behave like a router of a firewall, network management should be dealt with outside of the container, usually by the host.

This means that in most cases, containers do not need root privileges at all. And therefore, containers can run with a reduced set of capabilities. Let's first take a look at the default list of capabilities available to priviliged processes inside a container:

CHOWN The Linux man page describes chown as the ability to arbitrarily change file UID's (User Identifiers) and GID's (Group Identifiers). This means that root can change the ownership or group of any file system object. This capability should only be needed if

packages need to be installed in the container. Since packages are installed when the image is built, this capability should be dropped in production.

DAC_OVERRIDE According to the man page, dac_override allows root to bypass file read, write and execution permission checks. DAC is short for "discriteonary access control". This means that a root capable process can bypass all security checks on any file of the system, even if the permission and ownership fields would not allow it. Applications like software installation tools would be the only ones who could possibly need access to this sort of capability and, just like chown, should be enabled at build time, but disabled on production.

FOWNER The man page states that fowner allows a root capable process to bypass permission check on operations that require the filesystem UID of the process to match the UID of the file, excluding the ones that are already covered by other capabilites like dac_override and dac_read_search. Being a capability very similar to dac_override, the same set of rules would apply. This capability should be enabled at build time, but not on production.

FSETID The man page reads: "don't clear set-user-ID and set-group-ID mode bits when a file is modified; Set the set-group-ID bit for a file whose GID does not match the filesystem or any of the supplementary GID's of the calling process". Unless an installation is being run inside the container, this capability should not be needed.

KILL A process with this capability can bypass the permission checks for sending signals. This means that a root owned process can send kill signals to non root processes. Since we'll not be running as root inside the container, this capability might be needed for troubleshooting.

SETGID According to the man page, this capability allows a process to make arbitrary manipulations of process GIDs and supplementary GID list. It can also forge GID when passing socket credentials via UNIX domain sockets, and write a group ID mapping in a user namespace. In short, a process with this capability can change its GID to any other GID, which basically alows full group access to all files in the system. Since our container processes are not expected to change UID or GID, this capability can be dropped.

SETUID A process with the setuid capability can "make arbitrary manipulations of process UIDs; forge UID when passing socket credentials via UNIX sockets; write a user ID mapping in a user namespace". Similarly to setgid, a process with this capability can change its UID to any other UID, being granted access to all the files on the system. Since we're running our processes as a non-root user, this capability can also be dropped.

SETPCAP The man page description states that: "add any capability from the calling thread's bounding set to its inheritable set; drop capabilities from the bounding set; make changes to securebit flags". In layman's terms, a process with this capability can change its current capability set within its bounding set. Meaning a process could drop capabilities or add capabilities if it did not have them, but limited by the bounding set capabilities.

NET_BIND_SERVICE This capability allows a process to bind a socket to privilieged ports (the ones below 1024). Unless a container needs to bind to one of these privileged ports, this capability should be dropped as it can introduce severe security risks.

NET_RAW According to the man page, this capability allows the use of RAW and PACKET sockets, as well as binding to any address for transparent proxying. This access allows a process to spy on packets on its network. Most container processes would not need this access, and as such it should be dropped.

SYS_CHROOT This capability allows the use of chroot. It basically allows processes to chroot into a different root filesystem. Unless using the chroot command, this capability should be dropped.

MKNOD The man page states that this capability allows for the creation of special files. Basically, it allows a process to create device nodes. This device nodes are usually provided to containers in the /dev directory, and as such it should also be dropped.

AUDIT_WRITE A process with this capability can write a message to the kernel auditing log. Only a subset of processes even atempt to (login programs, su, sudo), and the ones inside the container are probably not trusted. As such, this capability should be dropped by default.

SETFCAP Finally, this capability allows one to set file capabilities on a file system. This might be necessary during builds, but should be dropped in production.

3.2.2 *How to drop capabilities using Docker*

First, to see what processes have which capabilities, we can run a single command called pscap available in the libcap-ng-utils package.

There are two ways to drop capabilities on the docker containers. We can either drop the capabilities we do not want, like:

```
$ docker run -d --cap-drop=setfcap --cap-drop=mknod
```

Listing 3.3: Dropping capabilities

Or, we can we drop all the capabilities, and add only the ones we need:

```
$ docker run -d --cap-drop=all --cap-add=setuid --cap-add=setgid
```

Listing 3.4: Drop all capabilities, adding only the necessary

The bottomline is that, most containers are being run with more capabilities than they should have, and it is important, security-wise that these capabilities are dropped in a production environment.

3.3 BUILD

Creating efficient docker images using Dockerfile is important when pushing images into production. Images need to be as small as possible in production for faster downloads and lesser surface attack.

Throughout this Section we'll be discussing the build stage and how to optimize it, by keeping the build fast, and the resulting image small.

3.3.1 *How caching can result in insecure builds*

Docker builds can be slow and, as we've discussed previously, and making use of Docker's layer caching can really speed up the process, by reusing previous builds to speed up the next one. However, there is a downside to it: caching can lead to insecure images.

Assuming we're using a stable base image, which means package updates are solely focused on security and severe bugfixes, we can deduce that these updates should be happening on a regular basis, because they are important and unliquely to break the code.

To ilustrate how caching can lead to insecure images, let's consider Dockerfile 3.5:

```
FROM ubuntu:18.04
RUN apt-get update && apt-get upgrade -y && apt-get install -y --no-install-recomends
 python3

COPY myapp.py .
CMD python3 myapp.py
```

Listing 3.5: Basic Python Dockerfile

The first time this image is built, it will download all the ubuntu packages, and the dependencies we need. The following times, however, `docker build` will use the cached layers to speed up the process, and the packages will not be downloaded again, unless the second line in the Dockerfile is changed. So, every time this image is rebuilt, the security updates will not be going through.

This suggests that one would want to bypass the caching mechanism. This can be easily achieved by adding two extra flags on the `docker build` command:

- `–pull`, which pulls the latest version of the base image, instead of using the locally cached one
- `–no-cache`, which ensures that every single layer in the Dockerfile is rebuilt from scratch.

With these, we can ensure that the resulting image will always have the latest package and security updates.

However, Docker's layer caching is something that we want to take advantage of, so a balance must be found between using the cached layers, and the rebuilding of the image itself. So, ideally there should exist two separate build processes: one that rebuilds the image using cache everytime new code is released, and second one that rebuilds the entire image from scratch periodically, according to the organization or project needs.

3.3.2 *Multi-stage builds*

When building a Docker image, the size of the resulting image is a big factor, and it is important to keep it as small as possible, to avoid slow deploys and test runs. Due to the nature of containers, even the smallest amount of code can generate a rather large image. With the use of multi-stage builds, it is possible to reduce the size of the resulting image, at the cost of a large compile and build toolchain.

A Docker image is built as a sequence of layers, with each subsequent layer building on top of the previous one. Layers can do all sorts of tasks, and amongst them they can add or remove files. However, removing one file from one layer, does not remove it from the previous one and, since all layers are required to unpack the image, these files are still inside the resulting image.

Multi-stage builds atempts to prevent these files from being kept in the final image, by separating the build process into multiple stages. Each stage produces an artifact than can be used by the next stage, and all the unnecessary files are kept isolated from the resulting image.

A very simple way to illustrate the concept of multi-stage builds is to divide a regular build into two stages:

1. On the first stage, we build an image to install the compilers and other build tools in order to compile the source code.

2. In the second stage, we install the packages required to run the source code, and copy the artifacts generated in the first stage.

The image generated by the second stage would be the one used to run our application. Using this process means that we can securely rely on Docker's cache to speed up the build on this stage, while also maintaining the image smaller.

When using multi-stage builds, there are additional factors that need to be taken into account, and dealt with carefully in order to optimize our solution.

Since we're dealing with a Django application, we need to adapt the Dockerfile to the Python specifics: this means that we will need to install an extensive array of dependencies, and they need to be then copied to the final image. There are multiple approaches to creating and copying the build artifacts between stages, like virtualenv or using the –user flag.

Using pip install –user option means that the resulting files will be installed in the .local folder of the users home directory. This makes it easier to copy all the files, since they are being stored in the same place:

```
FROM python:3.7-slim as compile-image
RUN apt-get update
RUN apt-get install -y --no-install-recommends build-essential

COPY requirements.txt .
RUN pip install --user -r requirements.txt
COPY setup.py .
COPY myapp/ .
RUN pip install --user .


FROM python:3.7-slim AS runtime-image
RUN addgroup --system myuser \
    && adduser --system --ingroup myuser myuser


COPY --chown=myuser:myuser --from=compile-image \
    /root/.local /home/myuser/.local
COPY --chown=myuser:myuser --from=compile-image \
    . /home/myuser/app

ENV PATH="/home/myuser/.local/bin/
```

Listing 3.6: Dockerfile with dependencies

The main downside of this approach is that some packages that might already be installed as a system-level dependency, will be installed again.

Using virtualenv is another strategy to achieve a single isolated directory of requirements that can be copied to the final image. In this case, the Dockerfile would look something like:

```
FROM python:3.7-slim as compile-image
RUN apt-get update
RUN apt-get install -y --no-install-recommends build-essential

RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY setup.py .
COPY myapp/ .
RUN pip install .


FROM python:3.7-slim AS runtime-image
COPY --from=compile-image /opt/venv /opt/venv
COPY --from-compile-image . /home/app

ENV PATH="/opt/venv/bin:$PATH"
CMD ["myapp"]
```

Listing 3.7: Multi-stage Dockerfile

The second factor that needs to be addressed is the relationship between multi-stage builds and CI/CD pipelines. Due to the nature of CI/CD pipelines, when building any Docker image, it's cache always starts empty. Therefore, in order to take full advantage of Dockers layer caching, we need to pull the latest image before the build starts.

The way to persist the image cache is to integrate in our pipeline a way to pull the image before building the new one, and pushing the new one after it's finished.

In a bash script, it should look something like:

```
#!/bin/bash

set -euo pipefall

docker pull image-registry/image || true
```

```
docker build -t image-registry/image .

docker push image-registry/image
```

Listing 3.8: Script to build multi-stage Dockerfiles

The ||true ensures the build will pass on the first time it is ran, since the pull will fail.

While this is standard procedure when integrating docker builds with any CI/CD system, it won't be enough when using multi-stage builds. Since the final image in a multi-stage build will not contain the first stage of the build, causing it to be rebuilt every time, reducing the speed of our build.

So, in order to optimize this stage, we should be pushing and pulling every single stage of the build, even if they are not needed in production:

```
#!/bin/bash
set -euo pipefall

docker pull image-registry/image:compile-stage || true
docker pull image-registry/image:latest || true

docker build --target compile-image \
        --cache-from=image-registry/image:compile-stage \
        --tag image-registry/image:compile-stage .

docker build --target runtime-image \
        --cache-from=image-registry/image:compile-stage \
        --cache-from=image-registry/image:latest \
        --tag image-registry/image:latest .

docker push image-registry/image:compile-stage
docker push image-registry/image:latest
```

Listing 3.9: Multi-stage optimized build

The key point to take away from this script is that we should explicitly tag and push all the stages, so that the next time the image is rebuilt, it can be used to populate the cache. Usually, the earlier stages will not suffer any change, so the time is only spent in pulling and pushing those images.

3.3.3 *Docker build secrets*

Building a Docker image often involves installing packages or downloading code and, when installing private code, to gain access its often required some kind of secret, be it a password, a private key or a token. This sort of information should not end up in the final image, since anyone with access could extract it.

So, the question becomes: how to feed the build with this information, without compromising security? Before talking about the correct way to do it, let's discuss how we should not do it first.

Environment variables and volumes

Since the goal is to use the secrets as part of the `build` stage, environment variables and volumes are out of question, because they can not be used at this stage.

COPY in the secrets as a file

Doing this means that the secrets file will end up in one of the images layer, even if deleted later. So, anyone with access to the image could easily retrieve it.

Pass the secret in using `–build-arg`

Unfortunately, build arguments are also embedded in the image. By running `docker history –no-trunc image`, an attacker could gain access to the secrets.

COPY in the secret to an earlier stage of a multi-stage build

The idea behind this approach is taking advantage of the fact that everything stored in the previous stages is not carried on to the final image. Therefore, by copying the secrets and installing the packages in an earlier stage, and then copying the artifacts into the final image, the secrets would be secured.

This approach is secure as long as the stage that uses the secrets is never pushed. As we've seen before, not pushing the earlier stages in a multi-stage build will cause the overall build to be slower, since the cache layering is not being used.

A secure and fast solution: the network

When running a container, it gets its own network space in the kernel, with its own network interfaces and corresponding IP addresses. Containers can also chose to join the network of another container. During the build, you can pass an extra agument `–network` that allows the `RUN` step to join a particular network, including that of an existing container.

The idea behind this approach is to run a small webserver with the secrets, so that during the build stage we can retrieve them.

Docker Buildkit

With version 18.09, dockers' build stage was greatly enhanced with the introduction of Buildkit. Enabling this feature is extremely easy and can be done by either setting the environment variable DOCKER_BUILDKIT=1, or it can be enabled by default by setting the daemon configuration in /etc/docker/daemon.json feature to true, and restart the daemon.

By integrating Buildkit, the build stage should see an increase in performance, due to its bottom up approach that enables parallelism, store management, security, and a few other features.

Of those features, one that stands out is the new –secret flag, which allows the user to pass secret information to be used by the Dockerfile and that will not end put stored in the final image.

Just by enabling this feature we can obtain greater performance on the pipeline as well as rid ourselves of creating a separate application to serve this secrets.

3.4 DEPLOY

3.4.1 *Tagging convention*

When one starts working with Docker, one of the first concepts learned is the idea of an image. It was mentioned multiple times so far, alongside another very important concept: the versioning of the images. These versions, or **tags**, is what is used to identify a specific image. As such, defining a convention to tagging our image will be crucial so that the image itself can be properly maintained throughout its lifecycle.

The most common and ideal solution to tagging Docker images that are meant to be publicly distributed is **Semantic Versioning**, also known as SemVer. The concept of SemVer itself is widely known and used across the industry for versioning packages or applications. It is simple, easy to understand and works as follow:

- Each version is defined by three point separated numbers, like x.y.z.
- The first number, in this case being x, represents a **MAJOR** version change, i.e., changes are introduced that can break compatibility.
- The second number, y, represents a **MINOR** version change, i.e., changes that will not break compatibility

- The third and last number, z, represents a **PATCH** version change, usually meant for bugfixes.

Every time a change is added to the code, one of the numbers is increased based on the type of change, and all subsequent numbers are reset. For example, imagine a given package is on it's 1.3.2 version. A **PATCH** like change would increase its version to 1.3.3, while a **MINOR** change would increase it to 1.4.0, and a **MAJOR** change to 2.0.0.

While this tagging convention is widely accepted and understood, ideally we would want our images to be generated indenpendently inside an automated system. A build process inside a CI/CD environment is not able to infer the dimension of a given change, and so that type of information would have to be provided in the build context. And, even although this is possible to achieve, having the tagging process being automated is more sustainable in the long run.

Another strategy that provides just as much benefits as SemVer, especially if used in a context where the images are only being used in an internal environment, is using the Git commit hash value to identify specific versions. These sort of values allows not only to assure the version is unique, but also allows us to trace back what changes led to that specific image.

Since we're on the topic of image versioning, I would like to mention the latest tag, and why it should not be used. The concept of the latest version is often used to define the most up to date version of a given image. However, latest is a tag just like any other, with the only difference being that it is the default tag added to an image if none is provided. This means that a later image created without this tag would not override it, and it could cause misunderstandings surrounding it. The other major problem related to latest is that, when one is using it, there are no indicators that suggest the image has changed, and so, breaking changes could be accidentaly introduced in your application.

3.4.2 *Docker images and the different environments*

One of Docker's many strengths is providing developers a tool that allows for the creation of reproducible and automated environments. Starting with a local image is quite simple, but eventually one wants to spin up a development and test environment, in order to then proceed to a production stage.

But, what about the details? Each one of these environments has a different purpose and, although the underlying application being served is the same, does that mean that one Dockerfile is enough, or should every environment configure its own?

Environment needs

When developing locally, one needs to: visually see the changes made to the code affecting the development server instantly; be able to install new dependencies and packages; and have as much access as possible to multiple debug tools.

In a testing environment, however, direct dev access is not mandatory or required, as it is meant to share results, find bugs and run automated tests. Some testing dependencies and trace tools may be installed, but the important is that the server can be started quickly in order to keep the testing time low.

The staging and production environments should not be mutable. No testing dependencies or direct access is needed, and both of these environments should be as similar as possible, differing only on environment specific variables, like database and third-party API credentials. The most important requirement is that the containers are started quickly and are running as stable as possible.

Multiple Dockerfiles

By analysing the multiple environments and their individual aims and limitations, we can make an educated decision. The staging and production environments should be using the same image, built from the same Dockerfile to ensure that they are as similar as possible.

In the few cases where the size of the final image is irrelevant, and one is sure that the performance will not be hindered, the testing dependencies and development tools can also be included in this Dockerfile, resulting in a single image to be used across all environments.

However, the production environment should not be contaminated by unnecessary features. So, ideally, one should have a Dockerfile to be used in staging and production environments, and a separate one to be used on testing and local development. The effort of maintaining both simultaneously is justified, as they both serve different purposes.

3.4.3 *Orchestration*

The portability and reproducibility of a containerized process allows one to move and scale containerized applications across multiple clouds and datacenters. The container technology guarantees that those applications run the same way no matter where, allowing us to take advantage of all these environments effectively and quickly. Furthermore, as these applications grow, it is important to have access to a set of tools that allow us to automate their maintenance, replace failed containers automatically, and rollout updates and reconfigurations of those containers during their lifecycle.

These set of tools are often called orchestrators, and we will be focusing the discussion on them moving forward. There are several orchestration tools provided by different companies, like Amazon's Elastic Container Service[4], Kubernetes[28] and Docker Swarm[16].

3.4.4 *Amazon ECS*

Amazon ECS (Elastic Container Service) is an orchestration tool developed by Amazon that runs Docker containers on a cluster of Amazon EC2 (Elastic Compute Cloud)[4] virtual machine instances. Just like other orchestrators, it handles installing containers, scaling and monitoring through both API and the AWS Management Console. It also provides a simplified view of the instances resources, like CPU and memory usage. It works with notion of Cluster, which is a group of Container Instances. Each cluster can run multiple services, which in turn can run multiple Tasks. The Tasks are the ones that host the Docker containers. Amazon ECS is a paid service, and could potentially become a valid option in the future of Dipcode.

3.4.5 *Kubernetes*

Kubernetes is an orchestration tool originally developed by Google, that has recently become one of the most popular amongst developers[30]. Just like other orchestrators, it aims to automate everything container related, from its' deploy process to the scaling of resources. It makes use of service meshes like Istio[24], monitoring tools like Promethey[36], command-line tools like Podman[45], distributed tracing from the likes of Jaeger[25] and Kiali[27], enterprise registries like Quay[38], and inspection utilities like Skopeo[10]. Simillarly to Amazon ECS, it uses the notion of Cluster, which can harbor a number of Nodes, which represent the machines. Each cluster can run any number of Pods, which is where one or more containers will be held.

3.4.6 *Docker Swarm*

All of the above have their own approach to deploying containers, with their own advantages and drawbacks, and so, before making a decision, we need to identify what are our business needs and available resources in order to pick the one that suits it the most.

Since this is the first time that Dipcode is delving into container technology to deploy web applications, it's important that the chosen orchestration tool is straightforward to learn and use, can provide all the basic utilities, and can fit seamlessly in the already existing procedures and practices. Of all the solutions listed above, Docker Swarm is the most begginer friendly, allowing for a smooth introduction into the multiple existing concepts,

and paving the way for an eventual transition into more complex solutions, like Kubernetes. Docker Swarm also takes advantage of Docker Compose syntax and structure, and so it should be easily integrated within the development pipeline in place.

Introduced with Docker's 1.12.0 release, on the 28th of July 2016, Docker Swarm is a native orchestration tool for Docker, which allows for one or more (physical or virtual) machines to bind together in order to create a swarm. Since its' release, up until today, it has been maintained and regularly updated by the Docker team, as it is a viable solution to deploying applications. Throughout this Section we'll be exploring Swarms' architecture, key concepts and commands, mirroring it against Dipcodes' reality to see how it can fit together.

Key Concepts

A **Swarm** consists of multiple Docker hosts running a **service** in a **swarm mode**, acting as either managers, or workers. When a service is created, it's **desired state** needs to be defined. The desired state is a set of characteristics that include: the number of replicas of that service that should be running at any given point; the network and storage resources available to that service; which ports of that service should be exposed, among others. The swarm and it's multiple nodes strive to maintain this state, by distributing tasks across multiple nodes, or lauching new nodes if one of them becomes unavailable. The key advantage of services over standalone containers is that the desired state, or configuration, can be updated without the need of a service restart.

A **Node** is an instance of the Docekr engine that is part of the swarm. Multiple nodes can be run on the same physical machine, or cloud server, but ideally, since we're dealing with a distributed service, they should be decentralized, to avoid having a single point of failure.

Roles

As mentioned above, a swarm is a group of physical, virtual machines or cloud servers that have been configured to join together in a cluster. Inside a cluster, each machine can fit into one of three roles:

- the *leader*, assigned when the clusted is established, throught the Raft[39] consensus algorithm, and can only be attributed to one node in the swarm. This node is in charge of all swarm management and task orchestration decision. If, by any chance, this node becomes unavailable, a new one one will be elected using the same algorithm;
- the *manager*, responsible to assign tasks to the worker nodes on the swarm, and that can also carry out some managerial tasks needed to operate the swarm;

- the *worker*, responsible to receive and execute the tasks given to it by manager nodes. By default, all manager nodes are also workers, capable of executing tasks when the resources to do so are available.

Inside a swarm there has to be at least one manager and one worker instance. Since the leader role is determined at swarm creation, and the manager can also be a worker, one machine is enough to build an operational swarm.

A **Service** is the definition of the tasks to be executed on the worker nodes. When a service is created, just like with Compose, the image to run needs to be specified, as well as any other commands that need to be executed inside a running container. There are two types of services: **replicated services**, in which the swarm manager distributes a specific number of replica tasks among the nodes based on the number set on the desired state configuration; and **global services**, in which the swarm runs the task across every available node in the cluster.

A **Task** carries a Docker container and the commands needed to be run inside it. This is the atomic unit of a swarm. Manager nodes assign these tasks to the nodes according to the number of replicas established. Once a task is given to a node, it cannot be moved to another. It can only either run on that node, or fail.

By default, the swarm manager uses **ingress load balancing** to expose the swarm services to the outside. The port can be configured in the desired state, and one is chosen by the manager if none is given. Internally, each node on the swarm is assigned a DNS name that is used by managers to distributed incoming requests.

Creating a Swarm

As of Docker's 1.12.0 release, swarm mode is packaged together with Docker, but is disabled by default. When swarm mode is enabled, Docker starts working with the concept of services, through the `docker service` command. To switch Docker engine to swarm mode, we need to run `docker swarm init`, and a single node swarm is initialized on the current node. Afterwards, the Docker engine sets up the swarm, taking the following actions:

- The current node is switched to swarm mode;
- A new swarm, named `default` is created;
- The current node is designated as the leader of the swarm;
- The manager node is configured to listen on an active network interface on port 2377;
- The current node availability is set to `Active`, meaning that it can receive tasks from the scheduler;

- An internal distributed data store for other Engines participating in the swarm is created. This data storage aims to maintain a consistent view of the swarm and all running services;
- A self-signed root CA is generated;
- Join tokens for managers and workers are generated;
- An overlay network named ingress is created. This network is useful for publishing service ports external to the swarm;
- An overlay default IP adress and subnet mask is created for the networks;

The output of running docker swarm init provides information regarding the expansion of the swarm through the join tokens mentioned above:

```
$ docker swarm init
Swarm initialized: current node (f6pun7exatn38uu6h01tssvi3) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-14xp9154qaqgl6m2qz1azpc243jj2n37dal36xz7ivvkui2qa8-
    f14jtjpioz73epuflsckb2hj7 192.168.1.9:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Listing 3.10: Initializing a swarm

The worker token does not need to be saved for later use, and can be accessed whenever using the command docker swarm join-token worker. Upon requesting to join the swarm:

- The Docker Engine on the current node is switched into swarm mode;
- A TLS certificate is requested from the manager;
- The node is given the machine hostname;
- The current node is added to the swarm at the manager listen address, based on the swarm token;
- The current node availability is set to Active, meaning it can now receive tasks from the scheduler;
- The ingress overlay network is extended to the current node;

Defining a service

Swarm services use a declarative approach to define its desired state, and rely on the Docker engine to maintain it. As mentioned above, the state includes information such as: the image name and tag that the service should run; the number of containers that will execute the service; what ports will be exposed; when should the service be started; how the service must behave upon restarting; amongst other characteristics regarding the nodes where the service is run, such as resources constraints and placement preferences.

The following is the most basic example on how to create a new service:

```
$ docker service create nginx
```

Listing 3.11: Create new service

This command starts an Nginx service on an available node with a randomly generated name and published ports. The only required argument that this command demands is the image name, so it can be pulled. The image can be hosted in a public or private registry, as long as the manager has the required access credentials. After a service is created, its image can not be updated unless the docker service update –image command is executed.

There is an extensive set of additional arguments that can be used alongside the docker service create command that help us define the services behavior. We will be discussing them in this Section, as they will be pivotal in setting up an operational swarm.

Service placement

There are multiple arguments that allow one to control the scaling and placement of services across different nodes. This is crucial, because different nodes will have different hardware specifications, and being able to take advantage of that when deploying the services will increase the swarm efficiency.

First and foremost, swarm services can run in one of two modes: replicated or global. For replicated services, the manager is responsible to schedule the number of replicas across the available nodes. For global services, on the other hand, the scheduler places one task on each available node that meets the service placement constraints and resource requirements. This property can be set through the –mode flag, which defaults to replicated by default, and can be changed to global. For replicated services, the number of replica tasks can be set using the –replicas flag.

The services resource requirements can also be specified, and the service will only be run on the nodes that meet those requirements. The resource requirements that can be specified are the available amount of memory or CPU, through the –reserve-memory and –reserve-cpu flags respectively. If there are no available services that meet the defined criteria, then the service will remain in a pending state until such node is available.

There are also placement preferences that allow one to apply arbitrary labels within a range of values on each node, and spread the services' across those nodes. Similarly to resource constraints, if no node is available, the service will remain in a pending state. This can be achieved through the –constraint flag and equality operator, for example: –constraint node.labels.os=linux or –constraint node.labels.os!=windows.

Placement constraints can be combined with each other, so it is wise to not use settings that will be impossible for nodes to fulfill.

In addition to placement constraints, there are also placement preferences that can be added with the –placement-pref flag. Like placement constraints, the preferences will be taken into account when distributing tasks across the swarm, however if no node fits the given criteria, the task itself will not be blocked. Placement preferences can be added or removed when updating a service, through the –placement-pref-add and –placement-pref-rm flags, and will be ignored when dealing with global services.

Published ports

When creating a service, it's ports can be published to hosts outside the swarm. By relying on the routing mesh when publishing a service, the swarm will make it reachable through the target port on every node regardless of if a service is running on that node. To achieve that, one can use the –publish <PUBLISHED-PORT >: <SERVICE-PORT > flag. Ports can also be published directly on the swarm node where the service is running. This bypasses the routing mesh and increases flexibility, with the added responsibility of keeping track where each task is running, routing the requests and load-balacing across the nodes. This can achieved by adding the mode=host option to the –publish flag.

Connecting to networks

Overlay networks can be used in swarm mode to connect multiple services. First, the network needs to be created with the docker network create –driver overlay ¡network-name¿ command. After creating a network in swarm mode, all managers are aware and have access to it. To add a service to a network, the –network flag should be used. Networks can be easily added or removed from a running service, when updating with the –network-add or network-rm flags.

Accessing secrets

In swarm services, a secret is a blob of sensitive data, like passwords, certificates or ssh keys, that should be kept safe, encrypted and should no be transmitted across the networks. Docker secrets can be used to centralize, securely manage and transmit it to services that have been granted explicit access to it. Secret access can be granted to a service through

the –secret flag, or at update time with –secret-add and secret-rm flags. When a service is granted access to a secret, the decrypted value is mounted onto the /run/secrets/ directory

Configuration data

Configs are very similar to secrets, with the main difference being that they are aimed to be used to store non-sensitive information, as they are not encrypted and are mounted directly into the container filesystem. Configs are helpful in keeping images as generic as possible, being able to use the same config to store data that changes between environments. Config values can be simple strings or binary content of up to 500kB in size. Configs can be set a service creation with the –config flag, and changed at update time with the –config-add and –config-rm flags.

When a config is granted to a service, by default it is mounted on the root (/) directory, is owned by the user running the service (usually root, belonging to the group root), and with world-readable permissions (0444). All these settings can also be changed at config creation.

Configuring update behavior

When a service is running, and we need to apply new changes (ie. new features, or bugfixes), it is important to define how the swarm should apply these changes to the service. There are multiple flags that allows us to specify how the process should be executed, and they can be set at service creation, or given to the docker service update command. –update-delay allows us to configure the time delay between updates to a service task or sets of tasks. The amount of time itself can be described as ThTmTs, in which T represents the number of hours, minutes or seconds, respectively. For example –update-delay 1m10s indicates a 1 minute and 10 seconds delay. By default, the scheduler only updates one task at a time, however more can be updated simultaneously using the –update-parallelism flag.

There are two possible states that a task can assume after being updated: **RUNNING** or **FAILED**. If the task at hand is running, the scheduler waits the time set by the update delay to move on to updating the next one. If, however, the update has failed, the entire process is stopped. If one whishes that the update continues regardless of a single fail, the –update-failure-action can be set to continue. It is also possible to control how many tasks need to fail for the update process to be considered a failure, by setting –update-max-failure-ratio.

In the event of a failure to update, it is important to have a failsafe option, so our application is still operating as expected. Through the –rollback flag, a safety net is created where the service will be rolled back the version that was running before the update action was executed. Flags like –rollback-delay, –rollback-failure-action, –rollback-max-failure-ratio and –rollback-parallelism serve the same purpose as their update counterparts. To mount a volume, one needs to at least provide the source and destination of the volume. In this case, the source determines the name of the volume, and the destination is that path inside the

container where it will be made available: –mount type=volume,src=<VOLUME-NAME >, dst= <CONTAINER-PATH >.

Access to volumes or bind mounts

Since containers, and services by extension, are built to be ephemeral and replaceable, one should avoid writing important data like database or log files inside them. Data volumes or bind mounts should be used instead, as they exist independently from services and anything written on these paths persist outside the container lifecycle. Mounts can be configured with the –mount flag when creating a service, or with the –mount-add and –mount-rm at update time. By default, the type used is set to volume, but it can be changed to type=mount. The main difference between the two is that bind mounts are file system paths that need to exist on the container host, and that are mounted into the container by Docker. Data volumes, on the other hand, are managed completely by the Docker engine, and only the directory inside the container must be provided so that it can be mounted. In this case, volumes present a very clear advantage over their counterpart, allowing for data to be shared between different nodes.

Healthchecks

Healthcheck is a configuration option for both Dockerfile and docker-compose files, that allows the Deamon to verify the condition of a service. They can be helpful, especially during updates, as the containers will only switch image if this healtcheck is successful. For a web application, implementing an healthcheck is simple, as it only needs to expose an endpoint that will successfuly answer if the application and all it's dependencies are already up and running:

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost/alive/"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 40s
```

Listing 3.12: Healthcheck Example

Defining a stack

So far, we have covered how we can turn our Python application into a Docker image to be used as a service on a production environment. We have also explained how that service

can be defined in such environment, by how many replicas we want, what kind of secrets and configurations it has access to and how to update it. However, deploying this single service as a standalone would make our application incomplete. Usually, web applications need multiple layers to work properly, like a database to store our data, a reverse proxy to handle incoming requests from outside the swarm, amongst many others, depending on the application requirements.

Stacks allow us to define the entire application infrastructure in a yaml file, using the Compose syntax from any version equal or higher than v3.0. In fact, this is a significant advantage, since we can use the same file for local development with Compose, and as a basis for a stack to deploy, since Compose specific variables are ignored by the Swarm, and vice versa. To create the stack, we must run the docker stack deploy –compose-file <STACK-FILE ><STACK-NAME >.

To define a stack, we should start by creating a new stack.yml file, and pinning the Compose version on top. Afterwards, we need to define all our different objects that will be available on the stack, like services, volumes, networks, secrets and configs. Let's start by defining our app service, which image is stored in a given repository.

```
version: "3.0"

services:
    app:
        image: <repository>/<image>:<version>
        environment:
            ENV_VAR_1: value
            ENV_VAR_2: value
        secrets:
            - secret1
            - secret2
        deploy:
            replicas: 2
            update_config:
                delay: 10s
                failure_action: rollback
            rollback_config:
                parallelism: 0

secrets:
    secret1:
        external: true
```

```
    secret2:
        external: true
```

Listing 3.13: Docker Swarm basic stack

So far, it looks simple and is no different than running docker service create with all the flags represented above. However, we'll need a reverse proxy to handle incoming requests, and a database to store the data generated by our application, using an nginx image and a mariadb image as an example. Since the reverse proxy and the database do not need to be aware of each other, let's also create two distinct networks: one that encapsules the proxy and the application, designated frontend, and another one to join the application and the database, backend:

```
version: "3.0"

services:
    app:
        image: <repository>/<image>:<version>
        environment:
            ENV_VAR_1: value
            ENV_VAR_2: value
        secrets:
            - secret1
            - secret2
        deploy:
            replicas: 2
            update_config:
                delay: 10s
                failure_action: rollback
            rollback_config:
                parallelism: 0
        networks:
            - frontend
            - backend

    reverse_proxy:
        image: nginx_image:<version>
        environment:
            PROXY_ENV_VAR_!: value
```

```
        configs:
            - source: nginx_conf
              target: /etc/nginx/conf.d/proxy.conf
              mode: 0644
            - source: nginx_inc
              target: /etc/conf.d/common.inc
              mode: 0644
        networks:
            - frontend
        deploy:
            replicas: 1
            update_config:
                failure_action: rollback
                delay: 10s
            rollback_config:
                parallelism: 0

    database:
        image: mariadb:<version>
        secrets:
            - database_password
            - database_root_password
        environment:
            DATABASE_NAME: db_name
            DATABASE_USER: db_user
            DABATASE_PASSWORD_FILE: /run/secrets/database_password
            DATABASE_ROOT_HOST: localhost
            DATABASE_ROOT_PASSWORD_FILE: /run/secrets/
    database_root_password
        networks:
            - backend
        volumes:
            - db-volume:/var/lib/mysql
        deploy:
            replicas: 1

secrets:
    secret1:
```

```
        external: true
    secret2:
        external: true
    database_password:
        external: true
    database_root_password:
        external: true

configs:
    nginx-conf:
        file: <local_path_to_configs>/proxy.conf
    nginx-inc:
        file: <local_path_to_configs>/nginx.inc

networks:
    frontend:
        driver: overlay
    backend:
        driver: overlay

volumes:
    db-volume:
        driver: local
```

Listing 3.14: Docker Swarm basic stack

In a development environment, the reverse proxy server is pointless, as all requests are coming from the development machine, and so there is no need to configure such service at that level, and so everything regarding the reverse proxy service could be moved into a different stack file, only used in production, named stack.production.yml. And image, for the sake of this exercise, that the production environment needs to access an external API that is merely mocked when developing, and so we need to provide an API key. The stack.production.yml file would look something like:

```
version: "3.0"

services:
    app:
```

```
        networks:
            - frontend
        secrets:
            - external_api_key

    reverse_proxy:
        image: nginx_image:<version>
        environment:
            PROXY_ENV_VAR_!: value
        configs:
            - source: nginx_conf
              target: /etc/nginx/conf.d/proxy.conf
              mode: 0644
            - source: nginx_inc
              target: /etc/conf.d/common.inc
              mode: 0644
        networks:
            - frontend
        deploy:
            replicas: 1
            update_config:
                failure_action: rollback
                delay: 10s
            rollback_config:
                parallelism: 0

configs:
    nginx-conf:
        file: <local_path_to_configs>/proxy.conf
    nginx-inc:
        file: <local_path_to_configs>/nginx.inc

networks:
    frontend:
        driver: overlay

secrets:
    external_api_key:
```

```
        external: true
```

Listing 3.15: Docker Swarm production stack

Now, with a stack.yml as our main stack, and a stack.production.yml as our production stack, when deploying both files can be chained together through: docker-compose -f stack.yml -f stack.production.yml config >deploy-stack-vx.yml and they will be merged together, adding new services, secrets, etc. or rewriting previous ones.

3.4.7 *Adapting CI/CD to deploy a stack*

Gitlab CI

The first step to adapting the CI/CD pipeline to this new reality of containers, is the creation and pushing of new images, everytime a new version is available. At Dipcode, our pipelines are integrated with gitlab-ci[23], and so the syntax presented in this Section will be according to said tool. As such, the first step is to create a new stage in the pipeline that is responsible for building and pushing the new images to the registry. Gitlab allows for configuration of variables that will be fed to the pipeline, and that can be only accessed and managed by specific users who have been granted access.

In order to push and pull the images from the repository, at the very least the access password should be stored here. The repository name, the repository user, and the image name can be written directly into the pipeline configuration file, as they alone do not pose a direct security threat. Gitlab also offers some predefined variables like CI_COMMIT_SHA, which represent the current commit revisions for the branch where the pipeline is running. This variable will be important to being able to traceback an image to the source code that originated it.

With the bash script presented back in Section 5.2 as a basis, we can write our docker stage:

```
.docker:
    image: docker:latest
    stage: build
    services:
        - docker:dind
    before_script:
        - echo $CI_REGISTRY_PASSWORD | docker login registry -u user --password
    -stdin
    after_script:
```

```
        - docker logout

.docker-build: &docker-build
  - docker pull image/compile-stage:latest || true
  - docker pull image/app:latest || true

  - docker build
    --cache-from image/compile-stage:latest
    --target compile-stage
    -t compile-stage:latest
    -f path_to_production_dockerfile/Dockerfile .
  - docker build
    --cache-from image/app:latest
    --cache-from compile-stage:latest
    -t app:latest
    -f path_to_production_dockerfile/Dockerfile .


docker:build:
  extends: .docker
  interruptible: yes
  script:
    - *docker-build
  only:
    - merge_requests


docker:release:
  extends: .docker
  script:
    - *docker-build

    - docker tag compile_stage:latest image/compile_stage:latest
    - docker tag app:latest image/app:$CI_COMMIT_SHA
    - docker tag app:latest image/app:latest

    - docker push image/compile_stage:latest
    - docker push image/app:$CI_COMMIT_SHA
```

```
  - docker push image/app:latest

when: manual
only:
  - master
```

Listing 3.16: Gitlab CI pipeline

The remaining takeaways from this stage, is that even although when we have talked before about the latest tag being unreliable, using a consistent tag to represent the most updated version can help speed up the build process in between merge requests, as long as it is managed correctly. When deploying to production, one should never use the latest tag, as it can mask underlying problems when searching for a specific bug.

Deploying the stack

Having covered how to automatically build and publish the updated image version of our application, the next step is to actually force that update on production. This can be achieved manually, by accessing the machine which hosts the swarm manager, update our stack file with the latest image version, and running the docker stack deploy command.

While there is nothing inherently wrong with having a human action be part of a process, it should be avoided, as it becomes more susceptible to faults. As such, this process should also be automated. Luckily for us, and all IT professionals around the world, there are automation tools, like Ansible, that can help us achieve just that.

With the help of automation tools, we can easily configure the deployment of new images. One viable strategy is to keep the version of our images on an environment variable on the machine that hosts the swarm manager, and build our stack file based on those environment variables. This way, the automation process just needs to update the host environment, and force a docker stack deploy.

With some tuning, it is even possible to refine our stack file and automation tools to update the value of our configs and secrets in case they become obsolete, like certificates, or passwords that need to be changed periodically.

3.5 Maintenance post production

Once our application has been deployed and is running in production, there are still actions that need to be taken in order to make sure everything is working as smoothly as possible.

There are multiple ways to monitor an application, and logging represents one of the most common and reliable source of data from the application operations. Extensive monitoring of services can be done through the user of third-party tools as services like Prometheus[36], which is an open-source toolkit that actively monitors the swarm, can be configured to generate alerts, and can be accessed via a web-interface; or Visualizer[21], which provides a simple from the top view of all the nodes, services and replicas available on the swarm, as well as simple metrics such as available space, the last time they were updated and what state they are in at the moment. These services integrate seamlessly with Docker Swarm, and can be a very powerful monitoring ally if configured properly.

Cron jobs, often refered to as simply crons, are scheduled tasks that are executed periodically, and often represent an integral part of a running application simple tasks, like sending e-mails every minute, or more complex and business-suited tasks, like generating reports every day.

Another very important task post-production, as mentioned before, is keeping our application external dependencies up to date with their latest versions, ranging from simple Python dependencies, to the application base Python image, and even our database or proxy image. Currently, Dipcode uses Renovate Bot[49], which is a tool that actively searches for the projects projects dependecies new updated versions, and automatically creates a new merge request with the updated version, for the development team to review, and accept or deny that given update. While, at the moment it is only configured to handle project dependencies, it can be easily changed to search for image updates, making sure the development team is given the chance to update the base images as soon as possible.

3.5.1 *Logging*

As mentioned above, logging the application and maintaining logs files is crucial to make sure everything is working as intended, or for troubleshooting when it does not. When it comes to logging an application running in Docker Swarm, there are multiple options one can choose from, each with their own advantages and drawbacks.

So first, let's discuss how Docker, and by extension Docker Swarm, handle logs. Events generated by single containers are sent to the container log, and events generated by a service, are sent to the daemon log. Both of these logs usually point to stdout and stderr, and Swarm adds service names and replica numbers to help traceback the logs to where they originated from. These logs can then be accessed through the manager node, by using docker service logs <service-name >. By having our application logs being written to stdout and stderr, it is possible to take advantage of Docker engine to manage and store everything log related on the /var/lib/docker/containers/ directory on the host machine. This approach, however, can be hard to manage in the long term, because there will be a log instance per

container, and logs from previous containers will not be accessible through the docker service logs command.

A second approach would be to use data volumes as our log directory, which allows for a greater control of logging in between different containers of the same service, with the clear disadvantage being that replicas running in different hosts would cause the application logs to be decentralized. To centralize logs, an external process responsible for gathering and centralizing these logs on a different location can be easily implemented. This would not be possible if Docker handled logging, since the /var/lib/docker/ directory can only be accessed by the root user.

Another solution is, like with monitoring the swarm, to add a dedicated service responsible for gathering and centralizing these logs. Solutions like Loggly[43] can be integrated, but they usually include a payment plan, which is not justifiable at this stage.

The final possible solution is to simply register application logs into the database. This can have massive repercussions when it comes to space limitations, but it can ease log consultation through database queries.

3.5.2 *Crons*

As forementioned, cron jobs can be integral in the proper functioning of a web application. Routines like sending e-mails every minute, checking if users need to update their password, or a business-specific task like generating reports daily, weekly, or monthly are all tasks that should be taken care by these unix feature.

Usually, these jobs would be configured on the single application host operating system and be executed whenever they were set to. However, by switching to a container infrastructure, we now have multiple application instances, that can be running on just as many hosts. The fact that we have multiple instances of our application running means that the cron jobs can not be transversal to all of them, as they should be executed once at any given time. If they could be defined on the image, and every single instance could execute them, then we would have all our instances busy at midnight generating the report of the previous day, when we only needed one. As this represents a tremendous waste of resources, cron jobs should be configured separately to our application image.

During the investigation, we have come up with a few solutions to the usage of cron jobs, two of which are already being used by Dipcode on the production environment, both with their advantages and drawbacks, and suited to their available infrastructure and business needs.

The first option consists in tying the execution of tasks to an API endpoint that the application itself exposes, and setting up the cron jobs on a separate host. The endpoints are protected with a specific API key that is only used by the cron job scheduler, wich merely

calls those endpoints at the configured time. This approach allows us to ensure that only one instance of the application is running said task, as it then becomes the swarm managers' job to determine which one is available to do so. To further guarantee that only one cron task is running at each time, each endpoint also acquires a lock to prevent a job from running twice, and the lock is release once said job is finished, allowing for next one to be executed. The main drawback we have felt from this approach is that if the instance running the task, or the task itself, unexpectedly fail. These causes the lock to never be released, blocking the task from being executed until it is manually released by the team.

The second alternative used consists in configuring the cron jobs on the swarm manager, where it uses an algorithm to determine which application instance should run the task at hand. The main drawback from this solution is, in the event of the managers' host becoming unavailable, the crontab configuration is not attributed to the next manager chosen by the stack. Overall, this solution is more reliable, but it requires the team to have direct access to the host machine, and it can be tweaked in multiple ways, i.e., if one does not want to clog the existing applications, the manager can simply spin up a new application instance just to execute the task, and then dispose of it. The attributes of containers allows this to be done quickly, and without an immense cost.

3.6 Set of good practices to keep in mind

Taking into consideration what we have outlined so far, let's compile a set of good practices to keep in mind when creating a new application served entirely using container-based technology, or migrating an existing project to this infrastructure.

- Carefully select a stable and secure base image for the main application, keeping it updated;
- Create a new Dockerfile to be used in production, using multiple stages to keep the final image as small, clean and secure;
- Adapt the CI/CD pipeline to build the image, taking into consideration the optimizations achieved with the multi-stage build, and tag the images accordingly;
- Create a docker-compose file that defines the stack with the shared services, volumes, secrets and configurations across all environments;
- Create separate docker-compose files with the stack for each individual environment, with their specific needs in mind;
- Create a script, or a different mechanism to handle the update and deploy of the stack to the different environments.

4

PUTTING IT INTO PRACTICE

Having thoroughly discussed how web applications work, and how Docker and orchestration help to set them up for multiple environments, let's start from the bottom, and create a new Django application, building it up to be production ready using everything we've learned so far.

4.1 CONFIGURING THE APPLICATION

Using the command `django-admin startproject exampleapp`, Django is able to create a new application with just the basics. The default directory structure created is as such:

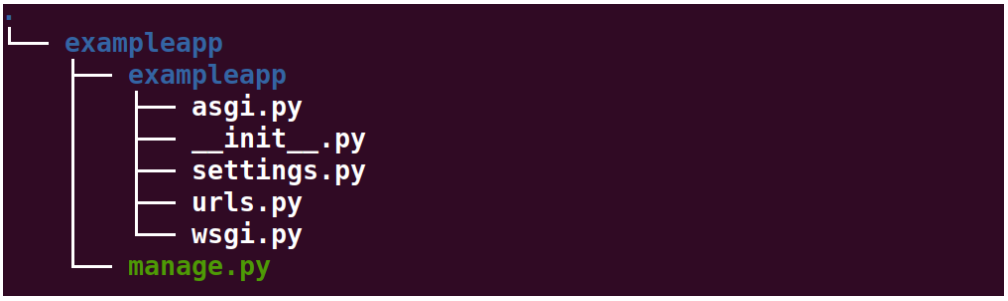


Figure 5: Django standard project structure

Let's start by adding a new directory called `docker`, where we will be storing all our Docker related files, like entrypoints, secrets, certificates and configurations. For now, let's just configure the entrypoints and local configs, so that the project can be run on the development machine.

Locally, we need to create two applications: one to serve our application, and another to serve static files, so we'll need two entrypoint files to start both of them.

Let's also add a `data` directory, responsible to store our media and database files. To generate certificates we'll use `openssl`[31].

```
$ openssl genrsa 2048 > cert.key
$ chmod 400 cert.key
```

```
$ openssl req -new -x509 -nodes -sha256 -days 365 -key cert.key -out cert.crt
```

Listing 4.1: Creating self-signed certificates

The next step is to create a requirements folder, where we will be separating all environment requirements into single files, keeping the shared ones in a common.txt file which will be inherited by all.

Afterwards, let's create our Dockerfile and docker-compose.yml file to run the application locally. For local development purposes, a multi-stage build is not necessary, and so using a standard Dockerfile to run the application will be enough.

```
FROM python:3.9-slim-buster

USER root

RUN apt-get update && apt-get install -y --no-install-recommends && apt-get install -y postgresql-
    client

RUN addgroup --system app && adduser --system --ingroup app app

USER app

COPY --chown=app:app requirements /home/app/requirements
COPY --chown=app:app docker /home/app/docker

ENV PYTHONPATH $PYTHONPATH:/home/app/src/exampleapp

RUN python -m pip install --user -U --ignore-installed -r /home/app/requirements/docker.txt

WORKDIR /home/app/src
```

Listing 4.2: Development Dockerfile for an example application

As for our local development docker-compose file, we'll need four services: the application and a service to provide static files, as mentioned before, and a database and caching with redis:

```
version: '3.5'

services:
  db:
    image: postgres:12.4-alpine
    restart: unless-stopped
```

```
    container_name: exampleapp-db
    environment:
      - POSTGRES_DB=db
      - POSTGRES_USER=dbuser
      - POSTGRES_PASSWORD=dbpass
    ports:
    - "5432:5432"
    volumes:
      - ./data/db:/var/lib/postgresql/data

  redis:
    image: redis:6.0.8
    restart: unless-stopped
    container_name: exampleapp-redis
    volumes:
      - ./data/redisdata:/data
    ports:
      - "6379:6379"

  static:
    image: node:10-slim
    restart: unless-stopped
    container_name: exampleapp-static
    command: /bin/sh /home/node/docker-entrypoint.sh
    environment:
      - PATH=$PATH:/home/node/app/node_modules/.bin/
    volumes:
      - ./exampleapp/static/:/home/node/app
      - ./docker/docker-entrypoint-static.sh:/home/node/docker-entrypoint.sh
    ports:
      - "4200:4200"

  app:
    build: .
    restart: unless-stopped
    container_name: exampleapp-app
    env_file: docker/config.env
    ports:
```

```
    - "8000:8000"
  volumes:
    - ./data/static:/data/static
    - ./data/media:/data/media
    - .:/app
  depends_on:
    - db
    - redis
```

Listing 4.3: Development docker-compose.yml

By now, we have configured our application to run Docker locally, and the project should look like:

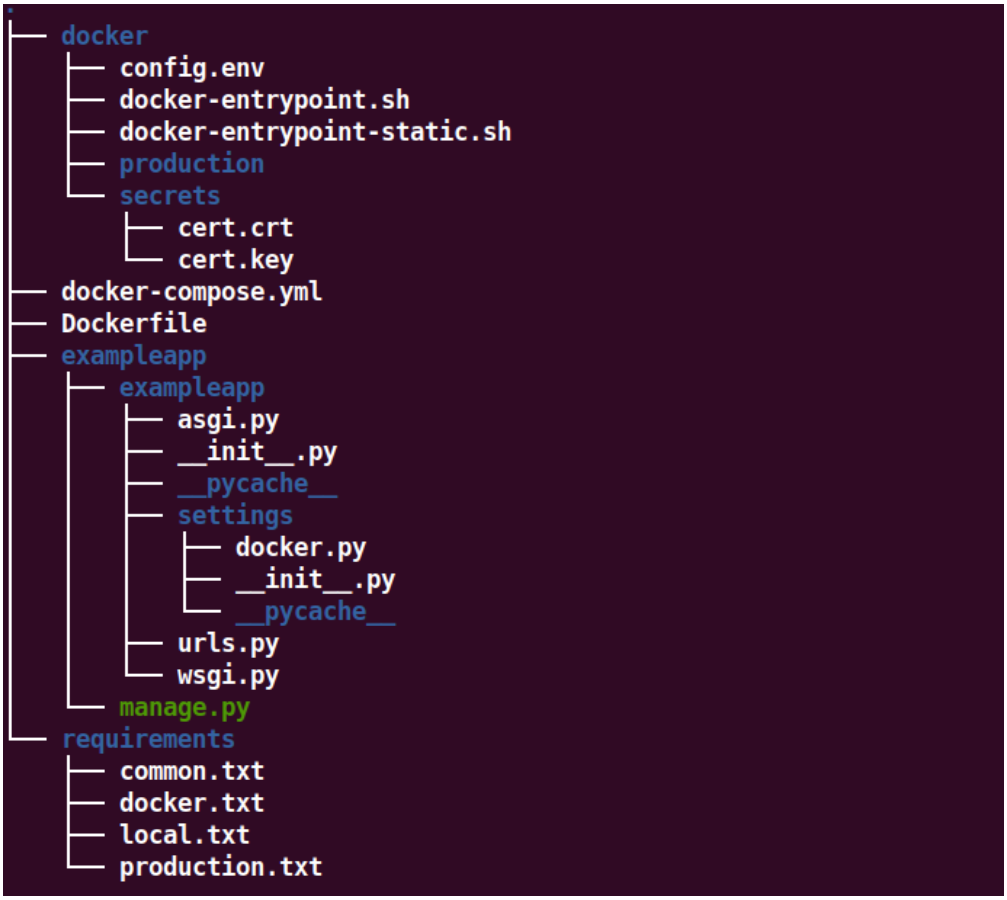


Figure 6: Django standard project structure

4.2 CONFIGURING MULTIPLE ENVIRONMENTS

With the application ready and running locally, it's time to set it up to run in staging and production environments, as the needs are different. Let's start by creating an optimized version of the Dockerfile, now using multi-stage builds to improve the performance of builds:

```
###################################
###### PYTHON COMPILE IMAGE #########
###################################
FROM python:3.9-slim-buster as compile-image

ENV PYTHONUNBUFFERED=1

RUN apt-get update
RUN apt-get install -y --no-install-recommends build-essential libpq-dev

COPY ./requirements requirements

RUN pip install pip-tools
RUN pip-sync --pip-args '--force-reinstall --compile --user --no-warn-script-location'
    requirements/local.txt

###################################
##### PYTHON RUNTIME IMAGE #########
###################################
FROM python:3.9-slim-buster as runtime-image

RUN apt-get update && apt-get install -y --no-install-recommends
RUN apt-get install -y postgresql-client

RUN addgroup --system appuser && adduser --system --ingroup appuser appuser
# Copy compiled dependencies
COPY --chown=appuser:appuser --from=compile-image /root/.local /home/appuser
    /.local

# Copy application code
COPY --chown=appuser:appuser ./exampleapp /home/appuser/exampleapp
```

```
COPY --chown=appuser:appuser ./docker/docker-entrypoint.sh /home/appuser/
    entrypoint.sh
COPY --chown=appuser:appuser ./docker/cert.crt /home/appuser/cert.crt
COPY --chown=appuser:appuser ./docker/cert.key /home/appuser/cert.key

RUN chmod +x /home/appuser/entrypoint.sh
RUN python -m compileall /home/appuser/exampleapp -l

ENV PATH=/home/appuser/.local/bin:$PATH
ENV PYTHONPATH=$PYTHONPATH:/home/appuser/exampleapp
ENV PYTHONUNBUFFERED=1

USER appuser
WORKDIR /home/appuser

ENTRYPOINT [ "/home/appuser/entrypoint.sh" ]
```

Listing 4.4: Production Dockerfile for an example application

Let's also create a new folder called deploy to store the yaml stack files, as well as all the environment and configuration files needed to deploy the stack. For the sake of trying to mimic a real application as much as possible, we'll be creating stack files for staging and production environments, that way we can store all the common characteristics in a stack.yml file, and environment specific needs on stack.staging.yml and stack.production.yml files, respectively. What changes between the staging and production stacks is usually permissions, configurations, which networks each service belong to, the number of replicas or the volumes.

For our application stack, we'll need three services: the application, the database, and an nginx proxy that will also serve the static files. The nginx image being used was pulled from Bunkerity[6], which is an image that easily integrates into Swarm and automatically applies security best practices:

```
version: '3.8'

services:
  app:
    image: ${API_IMAGE_URI}:${API_IMAGE_TAG:-latest}
    env_file:
      - configs/all/enviroment.env
    networks:
```

```
      - frontend
      - backend
    secrets:
      - secret_key
      - database_uri
    healthcheck:
      test: >
        curl --fail http://localhost:8000/api/alive -H 'Host: "$$SERVER_NAME"' ||
    exit 1
      start_period: 60s
    volumes:
      - static-volume:/home/appuser/exampleapp/static:wr
    deploy:
      replicas: 1
      update_config:
        order: start-first
        failure_action: rollback
        delay: 10s
      rollback_config:
        order: stop-first
        parallelism: 0
      restart_policy:
        condition: any
        window: 120s

  proxy:
    image: bunkerity/bunkerized-nginx:1.3.2
    configs:
      - source: proxy_nginx_conf
        target: /etc/nginx/conf.d/proxy.conf
        mode: 0644
      - source: proxy_common_inc
        target: /etc/nginx/conf.d/common.inc
        mode: 0644
    environment:
      MODSEC_RULE_ENGINE: DetectionOnly
    ports:
      - "80:80"
```

```
    networks:
      - frontend
    healthcheck:
      test: curl -f http://localhost/nginx_status || exit 1
    volumes:
      - static-volume:/usr/share/nginx/html/static:ro
    deploy:
      replicas: 1
      update_config:
        order: start-first
        failure_action: rollback
        delay: 10s
      rollback_config:
        order: stop-first
        parallelism: 0
      restart_policy:
        condition: any
        window: 120s

  db:
    image: postgres:12.4-alpine
    stop_grace_period: 30s
    volumes:
      - data-volume:/var/lib/postgresql/data
    networks:
      - backend
    secrets:
      - database_password
    healthcheck:
      test: mysqladmin ping -u $$MYSQL_USER -p$$MYSQL_PASSWORD -h localhost
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

networks:
  frontend:
    driver: overlay
```

```
  backend:
    driver: overlay

volumes:
  static-volume:
    driver: local

configs:
  proxy_common_inc:
    file: configs/all/common.nginx.inc

secrets:
  database_password:
    external: true
  database_uri:
    external: true
  secret_key:
    external: true
```

Listing 4.5: Common Docker stack file for an example application

The deploy configurations are generic:

- Each service will have one replica;
- When failing to update, they are allowed to rollback, keeping the environment stable;
- The application and proxy service are allowed to be restarted under any condition, however the database should only restart if there is a failure, preventing the loss of data as much as possible.

Notice that healthchecks were also added to smooth the update between images, as we've discussed before.

The changes to the staging stack are merely configuration and environment related, and in production the application is allowed to have two replicas, instead of one.

```
version: '3.8'

services:
  db:
    env_file:
```

```
      - configs/staging/postgres.env

configs:
  proxy_nginx_conf:
    file: configs/staging/proxy.nginx.conf
```

Listing 4.6: Docker stack for staging environment

```
version: '3.8'

services:
  app:
    deploy:
      replicas: 2

  db:
    env_file:
      - configs/production/postgres.env

configs:
  proxy_nginx_conf:
    file: configs/production/proxy.nginx.conf
```

Listing 4.7: Docker stack for production environment

By making use of Gitlab CI[23], it is possible to configure a pipeline that automatically builds and pushes images with new code to a given repository. It's important that this pipeline makes use of multi-stage builds to increase performance, and that the images are correctly tagged, as they can be used to traceback what changes were added in each.

```
image: git-registry.eurotux.com/dipops/images/python-dev

include:
  - project: "dipops/gitlab-ci-templates"

stages:
  - build
  - deploy
```

```
# Docker build and release

.docker:
  image: docker:latest
  stage: build
  before_script:
    - echo "$CI_REGISTRY_PASSWORD" | docker login $CI_REGISTRY -u
    $CI_REGISTRY_USER --password-stdin
  after_script:
    - docker logout

.docker-build: &docker-build
  - docker pull $CI_REGISTRY_IMAGE/compile-image:latest || true
  - docker pull $CI_REGISTRY_IMAGE/runtime-image:latest || true

  - docker build
    --cache-from $CI_REGISTRY_IMAGE/compile-image:latest
    --target compile-image
    -t compile-image:latest
    -f ./docker/production/app/Dockerfile .
  - docker build
    --build-arg VERSION=$CI_COMMIT_SHORT_SHA
    --build-arg BRANCH=$CI_COMMIT_REF_SLUG
    --cache-from $CI_REGISTRY_IMAGE/runtime-image:latest
    --cache-from runtime-image:latest
    -t runtime-image:latest
    -f ./docker/production/app/Dockerfile .


docker:prebuild:
  extends: .docker
  interruptible: true
  script:
    - *docker-build
  only:
    - merge_requests
```

```
docker:release:staging:
  extends: .docker
  script:
    - *docker-build

    - docker tag compile-image:latest $CI_REGISTRY_IMAGE/compile-image:latest
    - docker tag runtime-image:latest $CI_REGISTRY_IMAGE/runtime-image
    :$CI_COMMIT_SHORT_SHA
    - docker tag runtime-image:latest $CI_REGISTRY_IMAGE/runtime-image:latest

    - docker push $CI_REGISTRY_IMAGE/compile-image:latest
    - docker push $CI_REGISTRY_IMAGE/runtime-image:$CI_COMMIT_SHORT_SHA
    - docker push $CI_REGISTRY_IMAGE/runtime-image:latest
  allow_failure: false
  when: manual
  only:
    - dev

docker:release:prod:
  extends: .docker
  script:
    - *docker-build

    - docker tag runtime-image:latest $CI_REGISTRY/prd-runtime-image
    :$CI_COMMIT_SHORT_SHA
    - docker tag runtime-image:latest $CI_REGISTRY/prd-runtime-image:latest

    - docker push $CI_REGISTRY/prd-runtime-image:$CI_COMMIT_SHORT_SHA
    - docker push $CI_REGISTRY/prd-runtime-image:latest
  allow_failure: false
  when: manual
  only:
    - master
```

Listing 4.8: Gitlab CI pipeline to build images

4.3 DEPLOY

Having defined our stacks for the different environments, all that's left is to write a script to deploy it to those same environments. Keep in mind that there are several different ways of deploying a stack. In fact, Dipcode has set up an entirely different project focused on deploying the stack, making use of of Gitlab CI[23] to automatically trigger the pipelines that build the images alongside Ansible roles to trigger the deploy and store secrets on a Key Vault.

The script needs to access the host machine, generate the stack file given the environment configurations and secrets, and then deploy the stack.

```
#!/bin/sh

# SSH into the host machine

docker-compose -f stack.yml -f stack.$ENV.yml config > stack-v$VERSION.yml
docker stack deploy --compose-file stack-$VERSION.yml app-stack

# Exit the host machine
```

Listing 4.9: Generic Swarm deploy script

5

CONCLUSIONS AND FUTURE WORK

Throughout this dissertation we were able to create a solid knowledge foundation for both creating and adapting projects to a container-based infrastructure.

At this point, we can contrast the results of the investigation with the objectives that were initially set.

By hand-picking our base image, creating a dedicated user, and adapting the capabilities, we can ensure that the **security** of the application is ensured right from the start, i.e., the Dockerfile.

Volumes, and multiple third-party tools, coupled with a tight and attentive management can unsure that the application data is **backed up**, the application **logs** are stored appropriatly, and can be easily consulted, and that we can easily have a from the top monitoring of the resources our application is consuming.

Additionally, we have seen how to adapt the existing CI/CD pipeline to make full use of Docker images caching features to build images fast, and how an automation tool can help deploy newly generated images quickly into production.

Last, but not least, an important objective was acquired during the investigation: making sure that periodic tasks (cron jobs) were able to be executed succesfully, and without consuming too much resources from the infrastrucure.

During the period of researching and writing of the present dissertation, Dipcode has used this knowledge in two different projects. The first project has its entire infrastructure outside of Dipcode, and is using Microsoft's Azure Devops as an orchestrator, however it is currently using multi-stage builds on the production Dockerfile to fasten its' building and deploy time.

The second project mentioned is being completely managed and hosted by Dipcode, and is using most of the practices described throughout this document, as well as new and improved techniques to serve an application using entirely Docker Swarm.

5.1 FUTURE WORK

Although the work presented in this dissertation provides an extended guide on how to migrate and build container-based applications from the ground up, there is a lot of work that can be done in the future to assist Dipcode in this process.

There are amazing third-party tools to help the development team monitor the application, and keep logging files. However, as we have tried to keep our presented solutions maleable and able to adapt to the project needs, having our own made service that can make use of Docker's API to monitor the application usage of resources would be ideal.

Furthermore, since at Dipcode we have a wide array of projects, the development of a tool responsible to collect the generated logs to a centralized location, and that would also catalog them cronologicaly, would be incredibly helpful to all developers.

BIBLIOGRAPHY

[1] H. M. Abdullah and A. M. Zeki. Frontend and backend web technologies in social networking sites: Facebook as an example. In *2014 3rd international conference on advanced computer science applications and technologies*, pages 85–89. IEEE, 2014.

[2] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the internet impasse through virtualization. *Computer*, 38(4):34–41, 2005.

[3] M. Aoyama. Web-based agile software development. *IEEE software*, 15(6):56–65, 1998.

[4] A. AWS. Amazon ec2 container service. https://aws.amazon.com/ecs/. Accessed: 10-07-2021.

[5] R. Bower. What's the diff: Vms vs containers. https://www.backblaze.com/blog/vm-vs-containers/, 2018. Accessed: 05-11-2019.

[6] Bunkerity. Bunkerized nginx. https://hub.docker.com/r/bunkerity/bunkerized-nginx/. Accessed: 10-07-2021.

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[8] L. Chung and J. C. S. do Prado Leite. On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications*, pages 363–379. Springer, 2009.

[9] T. Combe, A. Martin, and R. Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3:54–62, 09 2016.

[10] Containers. Skopeo. https://github.com/containers/skopeo. Accessed: 10-07-2021.

[11] L. Containers. Linux containers. https://linuxcontainers.org/. Accessed: 10-07-2021.

[12] S. Containers. Solaris containers. https://www.oracle.com/solaris/technologies/solaris-containers.html. Accessed: 10-07-2021.

[13] Dipcode. About us. https://dipcode.com/about-us/, 2019. Accessed: 29-09-2019.

[14] Docker. Docker docs. https://docs.docker.com/. Accessed: 10-07-2021.

[15] Docker. Docker hub. https://hub.docker.com/search?q=&type=image. Accessed: 10-07-2021.

[16] Docker. Docker swarm. https://docs.docker.com/engine/swarm/. Accessed: 10-07-2021.

[17] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614. IEEE, 2014.

[18] M. Eder. Hypervisor-vs. container-based virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 1, 2016.

[19] Existek. Cost of the web application development. https://existek.com/blog/web-application-development-cost-estimation/. Accessed: 10-12-2021.

[20] D. Firesmit. Multicore and virtualization: An introduction. https://insights.sei.cmu.edu/sei_blog/2017/08/multicore-and-virtualization-an-introduction.html, 2017. Accessed: 01-11-2019.

[21] B. Fisher. Docker swarm visualizer. https://github.com/dockersamples/docker-swarm-visualizer, 2016-2021. Accessed: 10-07-2021.

[22] FreeBSD. Freebsh jails. https://docs.freebsd.org/en/books/handbook/jails/, 1995-2021. Accessed: 10-07-2021.

[23] Gitlab. Gitlab ci/cd. https://docs.gitlab.com/ee/ci/. Accessed: 10-07-2021.

[24] Istio. Istio. istio.com. Accessed: 10-07-2021.

[25] Jaeger. Jaeger. https://www.jaegertracing.io/. Accessed: 10-07-2021.

[26] Jazzband. pip-tools. https://pypi.org/project/pip-tools/, 2012-2021. Accessed: 10-07-2021.

[27] Kiali. Kiali. https://kiali.io/. Accessed: 10-07-2021.

[28] Kubernetes. Kubernetes. https://kubernetes.io/. Accessed: 10-07-2021.

[29] O. Nesterov. How much does a web application cost: A proven way to determine the real price. https://www.mindk.com/blog/how-much-does-a-web-application-cost/. Accessed: 12-10-2019.

[30] Opensource.com. What is kubernetes? https://opensource.com/resources/what-is-kubernetes?extIdCarryOver=true&sc_cid=701f2000001OH8HAAW. Accessed: 10-07-2021.

[31] OpenSSL. Open ssl. https://www.openssl.org/. Accessed: 10-07-2021.

[32] V. Osetsky. Web application development proces flow. https://medium.com/existek/web-application-development-process-flow-5645e2683b5d, 2018. Accessed:15-10-2019.

[33] J. Patel. Web development life cycle. https://www.monocubed.com/web-development-life-cycle/. Accessed: 10-12-2021.

[34] D. C. Plummer et al. Ethernet address resolution protocol: Or converting network protocol addresses to 48. bit ethernet address for transmission on ethernet hardware. *RFC*, 826:1–10, 1982.

[35] P. Poetry. Poetry. https://python-poetry.org/, 2018-2021. Accessed: 10-07-2021.

[36] Prometheus. Prometheus. https://prometheus.io/. Accessed: 10-07-2021.

[37] pypa. Pipenv. https://pypi.org/project/pipenv/, 2017-2021. Accessed: 10-07-2021.

[38] Quay. Quay. https://quay.io/. Accessed: 10-07-2021.

[39] Raft. Raft consensus algorithm. https://raft.github.io/. Accessed: 10-07-2021.

[40] E. Rescorla, N. Modadugu, et al. Datagram transport layer security, 2006.

[41] R. Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186, 2013.

[42] SDArchitect. Understanding devops - infrastructure as code. https://sdarchitect.blog/2012/12/13/infrastructure-as-code/, 2012. Accessed: 26-01-2020.

[43] Solarwinds. Loggly. https://www.loggly.com/. Accessed: 10-07-2021.

[44] A. Taivalsaari and T. Mikkonen. The web as a software platform: Ten years later. *Proceedings of the 13th International Conference on Web Information Systems and Technologies*, 2017.

[45] D. Tidwell. Podman - the next generation of linux container tools. https://developers.redhat.com/articles/podman-next-generation-linux-container-tools?intcmp=701f2000000tjyaAAA&extIdCarryOver=true&sc_cid=701f2000001OH8HAAW, Nov. 19 2018. Accessed: 10-07-2021.

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

[47] M. Virmani. Understanding devops bridging the gap from continuous integration to continuous delivery. *Fifth international conference on Innovative Computing Technology (INTECH 2015)*, 1, 2015.

[48] Warden. Warden. https://docs.warden.dev/. Accessed: 10-07-2021.

[49] WhiteSource. Renovate bot. https://www.renovatebot.com/. Accessed: 10-07-2021.