This chapter introduces Docker and covers the following:

* After studying the chapter, the reader is able to **run the examples provided** in the following chapters **in a Docker environment**.
* Docker and microservices are nearly synonymous. This chapter **explains why Docker fits so well with microservices**.
* Docker facilitates software installation. Important for this is the Dockerfile, which describes the installation of the software in a simple way.
* Docker Machine and Docker Compose support Docker on server systems and complex software environments with Docker.
* The chapter lays the foundation for an understanding of technologies such as **Kubernetes** **Cloud Foundry**, which are based on Docker.

**Licences and projects**[#](https://www.educative.io/module/lesson/introduction-to-microservices/JP0WmOQxVlK#licences-and-projects)

Docker is under the Apache 2.0 license. It is developed by the company [Docker, Inc.](https://www.docker.com/), among others.

Some core components, such as [Moby](https://github.com/moby/moby), for example, are under an Open Source license and allow other developers to implement systems similar to Docker.

Docker is based on Linux containers, which isolate processes in Linux systems from each other. The [Open Container Initiative](https://www.opencontainers.org/) ensures via standardization of the compatibility of the different container systems.

# Docker for Microservices: Reasons

[Chapter 2](https://www.educative.io/collection/page/10370001/6518081205567488/6272204058656768) defined microservices as separately deployable units. The separate deployment not only results in a decoupling at the architectural level, but also in regard to technology choice, robustness, security, and scalability.

## OS processes for microservices [#](https://www.educative.io/module/lesson/introduction-to-microservices/g21p4oGy65r#os-processes-for-microservices)

If microservices are supposed to have all these characteristics, the question arises as to **how they can be implemented**. Microservices must be scalable independently of each other. In the event of a crash, a microservice must not be allowed to make other microservices unavailable, too, and thus endanger the robustness of the whole system. Therefore, **microservices must at least be separate processes**.

**Scalability can be guaranteed by multiple instances of a process**. When an application is started, the operating system generates a process and allocates resources such as CPU or memory to it. Therefore, more processes can use more resources.

But processes are limited concerning scaling. If the processes all run on one server, then only a limited amount of hardware resources are available. Instead, the microservices should run in a cluster. Kubernetes and Cloud Foundry support running microservices in a cluster.

**With processes, robustness is guaranteed to a certain extent** because the crash of one process does not affect the other processes. However, a server failure still causes a large number of processes, and thus microservices, to fail.

But there are also other problems:

* **All processes share one operating system**. It must provide the libraries and tools for all microservices. Each microservice must be compatible with the operating system version. It is difficult to configure the operating system to support all microservices.
* In addition, the processes must coordinate in such a way that each process **has its own network port**. If you have a large number of processes, it becomes increasingly harder to find unused ports. Also, it’s hard to figure out which ports are used by which process.

## Virtual machines: too heavy-weight for microservices [#](https://www.educative.io/module/lesson/introduction-to-microservices/g21p4oGy65r#virtual-machines-too-heavy-weight-for-microservices)

Instead of a process, **each microservice can run in its own virtual** machine.

**Virtual machines** are simulated computers that run on the same physical hardware. For the operating system and application, virtual machines look exactly like a physical server.

Through virtualization, the microservice has its own operating system installation. Thus, the **configuration of the operating system can be adapted to the specific microservice**, and there is also **complete freedom in choosing the network port**.

### Overhead [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/g21p4oGy65r#overhead)

However, a virtual machine has a **substantial overhead**:

* The virtual machine must give the operating system the illusion of running directly on the hardware. This leads to overhead. Therefore, **performance is poorer** than with physical hardware.
* Each microservice has its own instance of the operating system. This **consumes a lot of memory** in the RAM.
* Finally, the virtual machine has virtual disks with a complete operating system installation. This means that the microservice **occupies a lot of hard disk space**.

So virtual machines have an overhead, making their operation expensive. In addition, operations will have to **manage a large number of virtual servers**. This is **complicated and time-consuming**.

The **ideal solution** would be a **lightweight alternative to virtualization**, which possesses the isolation of virtual machines, but consumes as little resources as processes do and is similarly easy to operate.

## What is Docker? [#](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#what-is-docker)

Docker represents a lightweight alternative to virtualization. Although Docker does not provide as much isolation as a virtualization, it is practically as lightweight as a process.

### Shared kernel [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#shared-kernel)

Instead of having a complete virtual machine of their own, Docker containers share the kernel of the operating system on the Docker host. The Docker host is the system on which the Docker containers run. The processes from the containers, therefore, appear in the process table of the operating system on which the Docker containers are running.

### Isolated network of Dockers [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#isolated-network-of-dockers)

The Docker containers have their **own network interface**. In this way, the **same port can be used in each Docker container**, and each container can use any number of ports.

The network interface is in a subnet where all Docker containers are accessible. The subnet is not accessible from the outside. This is at least the standard configuration of Docker.

The Docker network configuration offers many other alternatives. To still allow external access to a Docker container from the outside, **ports of a Docker container can be mapped to ports on the Docker host**. When mapping the ports of the Docker containers to the ports of the Docker host, be careful because each port of the Docker host can only be mapped to one port of a Docker container.

### Optimized file system [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#optimized-file-system)

Finally, the file system is optimized. There are **layers in the file system**. When a microservice reads a file, it goes through the layers from top to bottom until it finds the data. The containers can share layers.

The drawing below shows this more precisely. The file system layer at the bottom represents a simple Linux installation with the Alpine Linux distribution. Another layer is the Java installation. Both applications share these two layers, which are stored only once on the hard disk, although both microservices use them.

Only the applications are stored in file system layers that are exclusively available to a single container. The lower layers cannot be changed. The microservices can only write to the top layer. The reuse of the layers reduces the storage requirements of the Docker containers.

It is easily **possible to start hundreds of containers on a laptop**. This is not surprising: after all, it is also possible to start hundreds of processes on a laptop. Docker has **no significant overhead** compared to a process. Compared to virtual machines, however, the **performance benefits are outstanding**.

Diagram

Description automatically generated

## One process per container [#](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#one-process-per-container)

Ultimately, Docker containers are highly isolated processes due to their own network interface and file system.

Therefore, **only one process should run in a Docker container**. Running more than one process in a Docker container contradicts the idea of separating processes by means of Docker containers. Because only one process is supposed to run in a Docker container, there should be no background services or daemons in Docker containers.

## Docker image and Docker registry [#](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#docker-image-and-docker-registry)

File systems of Docker containers can be exported as Docker images. These images can be passed on as files or stored in a Docker registry.

Many repositories such as [Nexus](https://www.sonatype.com/nexus-repository-sonatype) and [Artifactory](https://www.jfrog.com/open-source/#artifactory) can store and provide Docker images just like compiled software and libraries. This makes it easy to exchange Docker images with a Docker registry for installation in production. The transfer of images from and to the registry is optimized. Only the updated layers are transferred.

## Supported operating systems [#](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#supported-operating-systems)

Docker **was originally a Linux technology**. For operating systems such as **macOS and Windows, Docker installations are available**. For this purpose, a virtual machine with a Linux installation is running in the background. This is transparent for the user. It seems as if the Docker containers are running directly on a computer.

In addition, Windows, since Windows Server 2016, provides Windows Docker containers. Linux applications run in a Linux Docker container and Windows applications in a Windows Docker container.

## Operating systems for Docker [#](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#operating-systems-for-docker)

Docker changes the requirements for operating systems.

### One process per container [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#one-process-per-container-2)

Only one process is supposed to run in one Docker container. This means that only as much of the operating system is required as is needed to run that process.

* For a Java application, we use the Java Virtual Machine (JVM), which requires some Linux libraries that are loaded at runtime. A shell is not necessary, for example. Therefore, distributions like [Alpine Linux](https://alpinelinux.org/), which are just a few megabytes in size, only contain the most important tools, making them an ideal basis for Docker containers.
* The programming language, Go, can create statically linked programs. In that case, nothing else has to be available in the Docker container besides the program itself, and no Linux distribution is required at all.

### Host does not have to have tools needed in the containers [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#host-does-not-have-to-have-tools-needed-in-the-containers)

The Docker host on which the Docker containers run **only has to run Docker containers**. Many Linux tools are therefore superfluous.

* [CoreOS](https://coreos.com/) is a Linux distribution that can run little more than Docker containers and, for example, considerably simplifies operating system updates of an entire cluster.
* CoreOS can also serve as a basis for Kubernetes.

## Overview [#](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#overview)

The drawing below shows an overview of the Docker concepts.

Diagram

Description automatically generated

* The **Docker host** is the machine on which the Docker containers run. It can be a virtual machine or a physical machine.
* **Docker containers** run on the Docker host.
* The containers typically contain a **process**.
* Each container has its own **network interface** with its own IP address. This network interface is only accessible from the Docker internal network. However, there are also ways to allow access from outside this network.
* In addition, each container has its own **file system**.
* When a container is started, the **Docker image** creates the first version of the Docker file system. When the container has been started, the image is extended by another layer into which the container can write its own data.
* All Docker containers share the **kernel** of the Docker host.

## Does it always have to be docker? [#](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#does-it-always-have-to-be-docker)

Docker is a very popular option for deploying microservices. However, there are **alternatives**. Two alternatives were already mentioned in [this lesson](https://www.educative.io/collection/page/10370001/6518081205567488/6554212332732416): **virtual machines** or **processes**.

## Microservices as WARs in Java application servers [#](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#microservices-as-wars-in-java-application-servers)

However, you can also deploy several microservices as **WAR files** in a Java application server or Java web server. WARs contain a Java web application, and can be deployed separately.

However, the deployment of a WAR may require the server to be restarted. Because microservices should only be deployable separately as per [chapter 2](https://www.educative.io/collection/page/10370001/6518081205567488/6272204058656768), **microservices can be implemented as WARs**.

### Disadvantages [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#disadvantages)

* **Compromises are made with regards to robustness**. A memory leak in a microservice can lead to failure of all microservices. The microservice would allocate more and more memory until an OutOfMemoryError is thrown and the entire Java application server crashes.
* **Separate scalability is also difficult to implement** because each server contains all microservices and, therefore, only all microservices can be scaled together.
  + This makes the scaling more complex than in cases where each server contains only one microservice as unneeded microservices are also scaled.
  + Of course, it would be possible to run each WAR on a separate Java web server and have multiple instances of each of these Java web servers. But then the WARs would no longer run together on one Java web server.
* And finally, all microservices run in one operating system process, which is a **compromise in terms of security**. When a hacker can take over the process, he or she has access to the entire functionality and data of all microservices.

### Advantages [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3w8DYkZXymp#advantages)

* In return, these approaches **consume less resources**. An application server with several web applications requires only one JVM (Java Virtual Machine), only one process, and only one operating system instance.
* Furthermore, there is **no need to introduce a new infrastructure** if application servers are already in use, which can reduce the workload for operations.

# Dockerfiles

**Overview**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#overview)

The creation of Docker images is done via files named Dockerfile. One of Docker’s strengths is that Dockerfiles are easy to write and therefore, the rolling out of software can be automated without any problems.

The typical components of a Dockerfile are:

* FROM defines a base image on which the installation is based. A base image for a microservice usually contains a Linux distribution and basic software, such as the JVM, for example.
* RUN defines commands that execute to create the Docker image. In essence, a Dockerfile is a shell script that installs the software.
* CMD defines what happens when the Docker container is started. Typically, only one process should run in one Docker container. This is started by CMD.
* COPY copies files in the Docker image. ADD does the same; however, it can also unpack archives and download files from a URL on the Internet. COPY is simpler to understand because it does not extract archives, for example. Also, from a security perspective, it can be problematic to download software from the Internet into Docker containers. Therefore, COPY should be given preference over ADD.
* EXPOSE exposes a port of the Docker container. This can then be contacted by other Docker containers or can be tied to a port of the Docker host.

A comprehensive [reference](https://docs.docker.com/engine/reference/builder/) is available on the Internet which contains additional details to the commands in Dockerfile.

**An example for a Dockerfile**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#an-example-for-a-dockerfile)

A simple example of a Dockerfile for a Java microservice looks like this:

1

2

3

4

FROM openjdk:11.0.2-jre-slim

COPY target/customer.jar .

CMD /usr/bin/java -Xmx400m -Xms400m -jar customer.jar

EXPOSE 8080

* The first line defines the base image with FROM. It is downloaded from the public Docker hub. The image contains a Linux distribution and a Java Virtual Machine (JVM).
* The second line adds a JAR file to the image with COPY. A JAR file (Java ARchive) contains all components of a Java application. It has to be available in a sub directory target below the directory in which the Dockerfile is stored. The JAR file is copied into the root directory of the container.
* The CMD entry determines which process should be started when the container is started. In this example, a Java process runs the JAR file.
* Finally, EXPOSE makes a port available to the outside. This is the port under which the application is available. EXPOSE only means that the container provides the port. It is then available on the internal Docker network. Access from outside is only possible when this is enabled at the start of the container.

The Docker image can be built with the command docker build --tag=microservice-customer microservice-customer.

docker is the command line tool with which most functionalities of Docker can be controlled. The created Docker image has the tag microservices-customer as defined by the --tag parameter.

The Dockerfile has to be in the sub directory microservice-customer. The name of this directory is the second parameter.

**File system layers in the example**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#file-system-layers-in-the-example)

The first image in [this lesson](https://www.educative.io/collection/page/10370001/6518081205567488/5176401755897856) shows that a Docker image consists of multiple layers.

Although no layers have been defined in Dockerfile, the image microservices-customer contains multiple layers. Each line of the Dockerfile defines a new layer. These layers are reused. Thus, if docker build is called again, Docker will go through the Dockerfile again. However, it will find that all actions in Dockerfile have already been executed once. As a result, nothing happens.

If the Dockerfile was modified in such a way that, after the COPY, another line with a COPY of another file is inserted, Docker would reuse the existing layer with the first COPY, but the second COPY and all further lines would then create new layers.

In this manner, Docker only re-creates the layers that need to be rebuilt. This not only saves storage space but is also much faster.

**Problem with caching and layers**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#problem-with-caching-and-layers)

A Dockerfile for obtaining a Ubuntu installation with updates looks like this:

1

2

FROM ubuntu:15.04

RUN apt-get update ; apt-get dist-upgrade -y -qq

First, a Ubuntu base image is loaded from the public Docker hub on the Internet. The commands apt-get update and apt-get dist-upgrade -y -qq are used to update the package index and then install all packages with updates. The options ensure that apt-get does not ask the user for permission and outputs only a few messages on the console.

The two commands are separated in the line by a ;. This causes them to be executed one after the other. A new file system layer is created only after both commands have been executed. This is useful for creating more compact images with fewer layers.

However, this Dockerfile also has a problem. If the Docker image is built again, no current updates will be downloaded. Instead, nothing happens because the images are already there.

Layer caching is based only on commands. Docker does not recognize that the external package index has changed. To ignore the existing images and force the rebuilding of the images, the parameter --no-cache=true can be passed to docker build.

**Docker multi-stage builds**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#docker-multi-stage-builds)

Everything needed to build a Docker image can also be found in the Docker image. Therefore, all of it is available at runtime of the Docker container.

If code is being compiled in a Dockerfile, the compiler is also available at runtime. This is unnecessary and can even be a security problem. If the container is compromised, the attacker can compile code inside the container with the original tools, which might allow more attacks.

It is not easy to delete all of the build environment because today there is usually a complex tool chain for building software.

Building the software outside of Docker might also not be an option. Docker is based on Linux; therefore, on macOS, you would need to run a cross compiler to generate Linux binaries.

To solve this problem, Docker has **Multi Stage Builds**. They make it possible to compile the program in one phase of the build in a Docker image, and to transfer only the compiled program to the next phase into a different Docker image.

Afterwards, the build tools are no longer available at runtime. They do not have to be deleted, and they do not have to be installed on the host machine.

[This lesson](https://www.educative.io/collection/page/10370001/6518081205567488/5402146352660480) in the next chapter shows a Docker Multi Stage Build using a Go program as example.

**Immutable server with Docker**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#immutable-server-with-docker)

**Immutable server** is an idea that predates Docker. The idea of an immutable server is that **a server will never be changed**; therefore, the software on the server will never be updated or modified. The server will always be completely rebuilt from scratch.

In this way, the state of the server can be reconstructed cleanly. For each server, an installation script installs all the needed software on a basic OS image.

However, **immutable servers are hard to implement**. It is very cumbersome to completely reinstall a server. The process might take minutes or hours. That is far too much time compared to, for example, just changing a configuration file.

This is exactly where **Docker helps**. Because of the optimizations, only the necessary steps are taken so that *immutable servers* can also be an option from this perspective.

A Dockerfile describes how to create a Docker image starting from a base image. With each build, it will seem as if the complete Docker image is being created. Behind the scenes, however, optimizations ensure that only what is really necessary is built.

For example, if in the very last step a configuration file is added and just that configuration file has been modified, Docker is smart enough to reuse the results of all other installation steps and just add the new configuration file. This just takes a few seconds.

**Docker is conceptually as clear as an immutable server but much more efficient** for actually implementing them.

**Docker and tools: Puppet, Chef or Ansible**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#docker-and-tools-puppet-chef-or-ansible)

Besides immutable servers, there are other ways to handle the installation of software.

**Idempotent installation** means that an installation script provides the same results no matter how often it runs. For an idempotent installation, there are no steps like “install the Java package,” but rather a definition of the desired state: “ensure that the Java package is installed”. If the installation is run on a fresh OS, Java would be installed. If the installation is run on a system that already has Java installed, nothing happens.

Idempotent installation is particularly **useful to enable updates**. For each update, the script would check if all software is installed in the correct version. If that is not the case, the correct version is installed. Tools such as **Puppet, Chef, or Ansible** support the concept of idempotent installation.

A Dockerfile is a very easy way to install software. The use of tools such as Puppet, Chef, or Ansible for the installation of software in a Docker image is possible but does not make a lot of sense. In particular, it is not necessary to use the update functionalities of these tools, because the image is typically freshly built with the *immutable server* approach.

This approach is easier than writing Puppet, Chef or Ansible scripts because defining the desired state is usually quite complex. The Dockerfile only describes how to build an image, whereas the other tools must also enable updates of the servers and are therefore more difficult to use.

**Overview**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#overview)

The creation of Docker images is done via files named Dockerfile. One of Docker’s strengths is that Dockerfiles are easy to write and therefore, the rolling out of software can be automated without any problems.

The typical components of a Dockerfile are:

* FROM defines a base image on which the installation is based. A base image for a microservice usually contains a Linux distribution and basic software, such as the JVM, for example.
* RUN defines commands that execute to create the Docker image. In essence, a Dockerfile is a shell script that installs the software.
* CMD defines what happens when the Docker container is started. Typically, only one process should run in one Docker container. This is started by CMD.
* COPY copies files in the Docker image. ADD does the same; however, it can also unpack archives and download files from a URL on the Internet. COPY is simpler to understand because it does not extract archives, for example. Also, from a security perspective, it can be problematic to download software from the Internet into Docker containers. Therefore, COPY should be given preference over ADD.
* EXPOSE exposes a port of the Docker container. This can then be contacted by other Docker containers or can be tied to a port of the Docker host.

A comprehensive [reference](https://docs.docker.com/engine/reference/builder/) is available on the Internet which contains additional details to the commands in Dockerfile.

**An example for a Dockerfile**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#an-example-for-a-dockerfile)

A simple example of a Dockerfile for a Java microservice looks like this:

1

2

3

4

FROM openjdk:11.0.2-jre-slim

COPY target/customer.jar .

CMD /usr/bin/java -Xmx400m -Xms400m -jar customer.jar

EXPOSE 8080

* The first line defines the base image with FROM. It is downloaded from the public Docker hub. The image contains a Linux distribution and a Java Virtual Machine (JVM).
* The second line adds a JAR file to the image with COPY. A JAR file (Java ARchive) contains all components of a Java application. It has to be available in a sub directory target below the directory in which the Dockerfile is stored. The JAR file is copied into the root directory of the container.
* The CMD entry determines which process should be started when the container is started. In this example, a Java process runs the JAR file.
* Finally, EXPOSE makes a port available to the outside. This is the port under which the application is available. EXPOSE only means that the container provides the port. It is then available on the internal Docker network. Access from outside is only possible when this is enabled at the start of the container.

The Docker image can be built with the command docker build --tag=microservice-customer microservice-customer.

docker is the command line tool with which most functionalities of Docker can be controlled. The created Docker image has the tag microservices-customer as defined by the --tag parameter.

The Dockerfile has to be in the sub directory microservice-customer. The name of this directory is the second parameter.

**File system layers in the example**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#file-system-layers-in-the-example)

The first image in [this lesson](https://www.educative.io/collection/page/10370001/6518081205567488/5176401755897856) shows that a Docker image consists of multiple layers.

Although no layers have been defined in Dockerfile, the image microservices-customer contains multiple layers. Each line of the Dockerfile defines a new layer. These layers are reused. Thus, if docker build is called again, Docker will go through the Dockerfile again. However, it will find that all actions in Dockerfile have already been executed once. As a result, nothing happens.

If the Dockerfile was modified in such a way that, after the COPY, another line with a COPY of another file is inserted, Docker would reuse the existing layer with the first COPY, but the second COPY and all further lines would then create new layers.

In this manner, Docker only re-creates the layers that need to be rebuilt. This not only saves storage space but is also much faster.

**Problem with caching and layers**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#problem-with-caching-and-layers)

A Dockerfile for obtaining a Ubuntu installation with updates looks like this:

1

2

FROM ubuntu:15.04

RUN apt-get update ; apt-get dist-upgrade -y -qq

First, a Ubuntu base image is loaded from the public Docker hub on the Internet. The commands apt-get update and apt-get dist-upgrade -y -qq are used to update the package index and then install all packages with updates. The options ensure that apt-get does not ask the user for permission and outputs only a few messages on the console.

The two commands are separated in the line by a ;. This causes them to be executed one after the other. A new file system layer is created only after both commands have been executed. This is useful for creating more compact images with fewer layers.

However, this Dockerfile also has a problem. If the Docker image is built again, no current updates will be downloaded. Instead, nothing happens because the images are already there.

Layer caching is based only on commands. Docker does not recognize that the external package index has changed. To ignore the existing images and force the rebuilding of the images, the parameter --no-cache=true can be passed to docker build.

**Docker multi-stage builds**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#docker-multi-stage-builds)

Everything needed to build a Docker image can also be found in the Docker image. Therefore, all of it is available at runtime of the Docker container.

If code is being compiled in a Dockerfile, the compiler is also available at runtime. This is unnecessary and can even be a security problem. If the container is compromised, the attacker can compile code inside the container with the original tools, which might allow more attacks.

It is not easy to delete all of the build environment because today there is usually a complex tool chain for building software.

Building the software outside of Docker might also not be an option. Docker is based on Linux; therefore, on macOS, you would need to run a cross compiler to generate Linux binaries.

To solve this problem, Docker has **Multi Stage Builds**. They make it possible to compile the program in one phase of the build in a Docker image, and to transfer only the compiled program to the next phase into a different Docker image.

Afterwards, the build tools are no longer available at runtime. They do not have to be deleted, and they do not have to be installed on the host machine.

[This lesson](https://www.educative.io/collection/page/10370001/6518081205567488/5402146352660480) in the next chapter shows a Docker Multi Stage Build using a Go program as example.

**Immutable server with Docker**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#immutable-server-with-docker)

**Immutable server** is an idea that predates Docker. The idea of an immutable server is that **a server will never be changed**; therefore, the software on the server will never be updated or modified. The server will always be completely rebuilt from scratch.

In this way, the state of the server can be reconstructed cleanly. For each server, an installation script installs all the needed software on a basic OS image.

However, **immutable servers are hard to implement**. It is very cumbersome to completely reinstall a server. The process might take minutes or hours. That is far too much time compared to, for example, just changing a configuration file.

This is exactly where **Docker helps**. Because of the optimizations, only the necessary steps are taken so that *immutable servers* can also be an option from this perspective.

A Dockerfile describes how to create a Docker image starting from a base image. With each build, it will seem as if the complete Docker image is being created. Behind the scenes, however, optimizations ensure that only what is really necessary is built.

For example, if in the very last step a configuration file is added and just that configuration file has been modified, Docker is smart enough to reuse the results of all other installation steps and just add the new configuration file. This just takes a few seconds.

**Docker is conceptually as clear as an immutable server but much more efficient** for actually implementing them.

**Docker and tools: Puppet, Chef or Ansible**[#](https://www.educative.io/module/lesson/introduction-to-microservices/3wkRpqxEBDp#docker-and-tools-puppet-chef-or-ansible)

Besides immutable servers, there are other ways to handle the installation of software.

**Idempotent installation** means that an installation script provides the same results no matter how often it runs. For an idempotent installation, there are no steps like “install the Java package,” but rather a definition of the desired state: “ensure that the Java package is installed”. If the installation is run on a fresh OS, Java would be installed. If the installation is run on a system that already has Java installed, nothing happens.

Idempotent installation is particularly **useful to enable updates**. For each update, the script would check if all software is installed in the correct version. If that is not the case, the correct version is installed. Tools such as **Puppet, Chef, or Ansible** support the concept of idempotent installation.

A Dockerfile is a very easy way to install software. The use of tools such as Puppet, Chef, or Ansible for the installation of software in a Docker image is possible but does not make a lot of sense. In particular, it is not necessary to use the update functionalities of these tools, because the image is typically freshly built with the *immutable server* approach.

This approach is easier than writing Puppet, Chef or Ansible scripts because defining the desired state is usually quite complex. The Dockerfile only describes how to build an image, whereas the other tools must also enable updates of the servers and are therefore more difficult to use.

# Docker Compose

## Overview [#](https://www.educative.io/module/lesson/introduction-to-microservices/3wAkopnNLRr#overview)

A typical microservice system contains more than a single Docker container. As explained in [chapter 2](https://www.educative.io/collection/page/10370001/6518081205567488/6272204058656768), microservices are modules of a system.

It would be good to have a way to start and run several containers together for starting all the modules that the system consists of in one go. This can be done with [Docker Compose](https://docs.docker.com/compose/).

## Service discovery with Docker Compose links [#](https://www.educative.io/module/lesson/introduction-to-microservices/3wAkopnNLRr#service-discovery-with-docker-compose-links)

Coordinating a system of multiple Docker containers requires more than just starting multiple Docker containers. It also requires **configurations for the virtual network** with which the Docker containers communicate with each other. In particular, **containers must be able to find each other in order to communicate**.

In a Docker Compose environment, **a service can simply contact another service via a Docker Compose link** and then use the service name as the hostname. So it could use a URL like http://order/ to contact the order microservice.

Docker Compose links offer some kind of service discovery, that is, a way for microservices to find other microservices. Synchronous microservices require a form of service discovery.

Docker Compose links extend Docker links. Docker links only allow communication. Docker Compose links also implement **load balancing** and set the start order so that the dependent Docker containers start first.

### Ports [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3wAkopnNLRr#ports)

In addition, Docker Compose can bind ports from the containers to the ports of the Docker host where the Docker containers run.

### Volumes [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3wAkopnNLRr#volumes)

Docker Compose can also provide volumes. These are file systems that can be shared by multiple containers. This allows containers to communicate by exchanging files.

## YAML configuration [#](https://www.educative.io/module/lesson/introduction-to-microservices/3wAkopnNLRr#yaml-configuration)

Docker Compose configures the interaction of the Docker containers with a YAML configuration file docker-compose.yml.

The following file comes from a project which implements Edge Side Includes as a way to compose websites from different sources. For this purpose, three containers must be coordinated.

* common is a web application that is supposed to deliver common artifacts.
* order is a web application for processing orders.
* varnish is a web cache to coordinate the two web applications.

version: '3'

services:

  common:

    build: ../scs-demo-esi-common/

  order:

    build: ../scs-demo-esi-order

  varnish:

    build: varnish

    links:

     - common

     - order

    ports:

     - "8080:8080

* The first line defines the used version of Docker Compose – in this case three.
* The second line starts the definition of the services.
* Line three defines the service common. The directory specified in line four contains a Dockerfile with which the service can be built. An alternative to build would be image to use a Docker image from a Docker registry.
* The definition of the service order also specifies a directory with a Dockerfile. No other settings are required for this service (lines 5/6).
* The service varnish is also defined by a directory with a Dockerfile (lines 7/8).
* The service varnish must have Docker Compose links to the services common and order. Therefore, it has entries under links. The varnish service can therefore reach the other services using the host names common and order (lines 9-11).
* Finally, port 8080 of the service varnish is bound to port 8080 of the Docker host, on which Docker containers run (lines 12-13).

### Additional options [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3wAkopnNLRr#additional-options)

Further elements of the YAML configuration are described in the [reference documentation](https://docs.docker.com/compose/compose-file/). For example, Docker Compose supports volumes shared by multiple Docker containers. Docker Compose can also configure the Docker containers using environment variables.

## Docker Compose commands [#](https://www.educative.io/module/lesson/introduction-to-microservices/3wAkopnNLRr#docker-compose-commands)

Docker Compose is controlled by the command line tool docker-compose. It must be started in the directory where the file docker-compose.yml is stored. The [reference documentation](https://docs.docker.com/compose/reference/overview/) lists all command line options for this tool. [This lesson](https://www.educative.io/collection/page/10370001/6518081205567488/4957007581806592) in the Appendix shows an overview of the Docker Compose commands. The most important ones are:

* docker-compose build builds the images for the services. **Try this command in the Docker compose coding environment above**!
* With docker-compose up, all services are started. The command returns the combined standard output of all services. This is rarely helpful, so docker-compose up -d is often the better choice. In this case, the standard output is not returned. **Try this command next**! Notice all the services will be up and accessible at the given link. With docker-compose logs the output of individual containers can be viewed.
* With docker-compose up --scale <service>=<number>, a larger number of containers for a service can be started. In the example from the listing,docker-compose up --scale common=2 could ensure that two containers for the service common are started. **Try this next**.
* docker-compose down shuts down all services and deletes the containers. **Try this to shut down all services**.

Since the examples often require the interaction of several Docker containers, most examples have a docker-compose.yml file to run the containers together.

git clone https://github.com/ewolff/SCS-ESI.git && cd SCS-ESI/scs-demo-esi-order/ && ./mvnw clean package && cd ../docker && docker-compose build && docker-compose up -d

There are not really any fundamental alternatives to Docker:

* **Virtualization** has too much overhead.
* **Processes** are not sufficiently isolated. The required libraries and runtime environments for all microservices must be installed in the operating system. This can be difficult because each process occupies one port, and therefore the allocation of ports must be coordinated.
* Other **container solutions** such as [rkt](https://coreos.com/rkt" \t "_blank) are far less common.

**Clusters**[#](https://www.educative.io/module/lesson/introduction-to-microservices/RMDngq38Gwz#clusters)

For production, **applications should run in a cluster**. Only in this way can the system be scaled across several servers and secured against the failure of individual servers.

Docker Compose can use [Docker Swarm Mode](https://docs.docker.com/engine/swarm/) for **cluster management**. Docker Swarm Mode is built into Docker.

**Kubernetes** is widely used for operating Docker containers in a cluster.

There are also other systems like [**Mesos**](http://mesos.apache.org/). Mesos is actually a system for managing batches for data analysis in a cluster but it also supports Docker containers. Offers in the cloud are also available, such as [ECS](https://aws.amazon.com/documentation/ecs/) (EC2 Container Service).

## Docker without Scheduler [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMDngq38Gwz#docker-without-scheduler)

An alternative is to install Docker containers **directly on a server**. In this scenario, the servers are provided with classic mechanisms – for example, with virtualization. Linux or Windows is installed on the server. The microservice runs in a Docker container.

The only difference to the procedure without Docker is that Docker containers are now used for deployment. But this already makes it much easier to keep production and test environments identical. Concepts such as immutable servers are also easier to implement, as is the technology freedom for microservices.

Traditional virtualization is still responsible for high availability, scaling, and distribution to the servers.

This approach offers some of the advantages of Docker and reduces the effort.

These approaches have **one thing in common**: how the load is distributed in the cluster is decided by the scheduler – that is, by Kubernetes or Docker Swarm Mode. This means that the **scheduler is of crucial importance** for fail-safety and load balancing.

If a container fails, a new container must be started. Likewise, additional containers must be started at times of high loads, ideally on machines that are not too busy.

Schedulers such as Kubernetes solve many challenges, especially with synchronous microservices. It is highly recommended to use such a platform for operating microservices.

But the introduction of microservices requires many changes. The architecture needs to be adapted, and developers need to learn new approaches and technologies, as well as having adopted the deployment pipeline and the tests.

Implementing an additional technology such as a Docker scheduler should, therefore, be well thought out because many other changes also have to be made.

## PaaS [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMDngq38Gwz#paas)

Another alternative is a **PaaS**. A PaaS has a higher degree of abstraction than Docker schedulers because only the application needs to be provided.

The PaaS creates the Docker images. Therefore, a PaaS can be the simpler and therefore better solution.

## Experiments [#](https://www.educative.io/module/lesson/introduction-to-microservices/RMDngq38Gwz#experiments)

* Docker Machine can use clouds like Amazon Cloud or Microsoft Azure with the appropriate [drivers](https://docs.docker.com/machine/drivers/). Create an account with one of the cloud providers. Most cloud providers offer free capacity to a new user.
* Create an account in the [Docker Hub](https://hub.docker.com/). Build a Docker image, for example, based on one of the microservices examples of the following chapters and place it in the Docker hub with docker push.
* Use the [tutorials](https://docs.docker.com/engine/swarm/swarm-tutorial/) to familiarize yourself with the Docker Swarm Mode, which can be used to run Docker in a cluster.

## Why are microservices so important? [#](https://www.educative.io/module/lesson/introduction-to-microservices/JP5JvRrrZ0y#why-are-span-stylecolor8d0327-microservicesspan-so-important)

One of the **strengths** of microservices is that different technologies can be used in each individual microservice.

The technologies in the microservices can be defined as part of the microarchitecture (see [chapter 3](https://www.educative.io/collection/page/10370001/6518081205567488/6218432796164096)).

However, there are **technical challenges** to consider when **selecting technologies** for microservices.

## Chapter walkthrough [#](https://www.educative.io/module/lesson/introduction-to-microservices/JP5JvRrrZ0y#chapter-walkthrough)

This chapter explains **how to deal with the technical microarchitecture:**

* The reader gets to know the **requirements** regarding, e.g., operation or resilience, which the microarchitecture has to fulfill.
* Often microservices are implemented with **reactive technologies**. Thus, the chapter discusses this option in more detail and explains when this approach makes sense.
* As a concrete example of technical microarchitecture, the chapter shows **Spring Boot** and **Spring Cloud**.
* Based on Spring Boot and Spring Cloud, the chapter shows how the **technical requirements the microarchitecture** has to address can be **fulfilled**.
* In addition, the chapter shows how the **programming language Go** in conjunction with appropriate frameworks fulfills the requirements **for implementing microservices.**

A technology for implementing microservices has to fulfill different requirements. The figure below gives us a birds-eye view of what these are.

We’ll be discussing each of these in detail below.

Diagram

Description automatically generated

## Communication [#](https://www.educative.io/module/lesson/introduction-to-microservices/gkpnAWrr6Kr#communication)

Microservices have to **communicate** with other microservices. This requires **UI integration** in the **web UI** or **protocols** such as **REST** or **messaging**.

It is a macro architecture decision which communication protocol is used (see [Architecture Decisions](https://www.educative.io/collection/page/10370001/6518081205567488/4713567020384256)).

However, the microservices have to support the chosen communication mechanism. Therefore,

The macro architecture decision influences the micro architecture.

The technology choices at the micro architecture level have to ensure that the communication protocol defined by the macro architecture can really be implemented in each microservice.

In principle, every modern programming technology can support the typical communication protocols. Therefore, this requirement does not represent a real restriction.

## Operation [#](https://www.educative.io/module/lesson/introduction-to-microservices/gkpnAWrr6Kr#operation)

Operating the microservices should be as easy as possible.

Topics in this area are:

* Deployment
* Configuration
* Logs
* Metrics

Let’s cover these one by one:

### Configuration [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gkpnAWrr6Kr#configuration)

* The microservice has to be adapted to different scenarios. It is possible to use custom code for reading the configuration. However, an existing library can facilitate this task and promote a uniform application configuration.

### Deployment

* The microservice has to be installed in an environment and has to run in this environment.

### Logs [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gkpnAWrr6Kr#logs)

Writing log files is easy. However, the format should be uniform for all microservices.

In addition, a simple log file is not enough when a server has to collect the logs from all microservices and provide them for analysis.

Therefore, technologies have to be in place for formatting the log outputs and for sending them to the server where all logs are stored and analyzed.

### Metrics [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gkpnAWrr6Kr#metrics)

Metrics have to be delivered to the central monitoring infrastructure.

This requires appropriate frameworks and libraries. In principle, different libraries can be used for implementing a macro architecture rule for which instance predefines a log format and a log server.

In this case, the micro architecture has to choose a library for the microservice. Macro architecture rules can also determine the library.

However, this limits the technological freedom of the microservices to those programming languages which can use the chosen library.

## New microservices [#](https://www.educative.io/module/lesson/introduction-to-microservices/gkpnAWrr6Kr#new-microservices)

It should be easy to create new microservices. When a project over time accumulates more and more code, there are two options:

1. The microservices become larger.
2. The number of microservices of constant size increases.

### Option 1: Microservices increase in size [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gkpnAWrr6Kr#option-1-microservices-increase-in-size)

If the microservices increase in size, at some point they will not deserve the name microservice anymore.

### Option 2: Constant size microservices increase in number [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gkpnAWrr6Kr#option-2-constant-size-microservices-increase-in-number)

To avoid the increase in the size of microservices, it is easy to generate new microservices to keep the size of the individual microservices constant over time.

## Resilience [#](https://www.educative.io/module/lesson/introduction-to-microservices/gkpnAWrr6Kr#resilience)

Each microservice has to be able to deal with the failure of other microservices. This has to be ensured when microservices are implemented.

# Reactive Programming

## Reactive programming [#](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#reactive-programming)

One way to implement a microservice is reactive programming. Oftentimes it is stated that microservices must be implemented with reactive technologies.

This section discusses what reactive actually is and determines whether reactive technologies are truly needed for microservices.

Similar to microservices, reactive has an ambiguous definition.

The [Reactive Manifesto](http://www.reactivemanifesto.org/) defines the term “reactive” based on the following characteristics:

### Responsive [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#responsive)

* **Responsive** means that the system **responds as fast as possible**.

### Resilient [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#resilient)

* Because of **resilience** the system **remains available** even if parts fail.

### Elastic [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#elastic)

* The system can deal with different levels of load, for instance by using additional resources. After the load peak subsides the resources are freed again.

### Asynchronous communication [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#asynchronous-communication)

* The system uses asynchronous communication (message-driven).

These characteristics are useful for microservices. They pretty much correspond to the features discussed in chapter 2 as essential characteristics of microservices.

At first sight, it seems that microservices, in fact, must be written with **reactive technologies.**

## Reactive programming [#](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#reactive-programming-2)

However, [reactive programming](https://en.wikipedia.org/wiki/Reactive_programming) means something completely different. This programming concept resembles the data flow. When new data comes into the system, it is processed. A spreadsheet is an example. When the user changes a value in a cell, the spreadsheet recalculates all dependent cells.

## Classical server applications [#](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#classical-server-applications)

A similar approach is possible for server applications. Without reactive programming, a server application typically processes an incoming request in a thread.

If the processing of the request requires a call to a database, the thread blocks until the result of this call arrives.

In this model, a thread has to be provided for each request that is processed in parallel and for each network connection.

## Reactive server applications [#](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#reactive-server-applications)

Reactive server applications behave very differently. **The application only reacts to events**. It must not block because it is waiting, for instance, for I/O. Thus, an application waits for an event such as an incoming HTTP request.

If a request arrives, the application executes the logic and then sends a call to the database at some point. However, subsequently, the **application does not wait for the result of the call to the database** but suspends processing the HTTP request.

Eventually, the next event arrives, namely the result of the call to the database. The processing of the HTTP request then resumes. In this model, only one thread is needed. It processes the respective current event.

The figure below shows an overview of this approach.

Diagram

Description automatically generated

The event loop is a thread and processes one event at a time. Instead of waiting for I/O, the processing of the event is suspended.

Once the results of the I/O operation are available, they are part of a new event which is processed by the event loop.

In this way, a single event loop can process a plethora of network connections. However, processing of the event must not block the event loop for longer unless it is absolutely necessary. Otherwise processing of all events will be stopped.

## Reactive programming and the reactive manifesto [#](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#reactive-programming-and-the-reactive-manifesto)

Reactive programming can support the goals of the Reactive Manifesto:

* **Responsive**: The model can make the application respond faster because fewer threads are blocked. However, whether this really leads to an advantage over a classical application depends on how efficiently the threads are implemented in the system and how efficiently it handles blocked threads.
* **Resilience**: If a service no longer responds, nothing is blocked in reactive programming. This helps with resilience. However, for example, in a classical application, a timeout can avoid a blockage by aborting the processing of the request.
* **Elastic**: With a higher load, more and more instances can be started. This is also possible with the classical programming model.
* **Message-driven**: Reactive programming does not affect the communication between the services. Therefore, communication can or cannot be message-driven in reactive programming as well as in classical applications.

## Reactive programming is not necessary for microservices [#](https://www.educative.io/module/lesson/introduction-to-microservices/gxBvxY11l5r#reactive-programming-is-not-necessary-for-microservices)

The Reactive Manifesto is certainly relevant for microservices. But a microservice does not have to be implemented with reactive programming in order to achieve the goals of the Reactive Manifesto.

Whether or not a microservice is implemented with reactive programming can be different for each microservice.

This can be a micro architecture decision and therefore affects only individual microservices, but not the system as a whole.

It is important to understand the difference, because otherwise the choice of technologies might be limited to reactive programming frameworks even though that is not necessary.

It is perfectly fine to stay with established technologies. In fact, using a technology stack that you are used to might be easier and bring faster results.

At the same time, it is possible to try new technologies like reactive programming in one microservice and then use it in other microservices if it has proven to be useful.

## The Spring framework and the Java community [#](https://www.educative.io/module/lesson/introduction-to-microservices/YMBXW47XDXA#the-spring-framework-and-the-java-community)

The Spring Framework has long been part of the Java community. It has a broad set of features covering most of the technical requirements of typical Java applications. [Spring Boot](https://projects.spring.io/spring-boot/) facilitates the use of Spring.

A minimal Spring Boot application can be found in the directory simplest-spring-boot of the project <https://github.com/ewolff/spring-boot-demos>.

### Java code [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/YMBXW47XDXA#java-code)

The Java code from the project shows how Spring Boot can be used.

/\*

\* The Java code from the project shows how Spring Boot can be used.

\*/

@RestController

@SpringBootApplication

public class ControllerAndMain {

@RequestMapping("/") public String hello() {

return "hello\n";

}

public static void main(String[] args) {

SpringApplication.run(ControllerAndMain.class, args);

}

}

**Line 5 and 7**:

* The annotation @RestController means that the class ControllerAndMain should process HTTP requests.

**Line 6:**

* @SpringBootApplication triggers the automatic configuration of the environment.

The application thereby starts an environment with a web server and with the parts of the Spring framework that are fitting for a web application.

**Line 9:**

* The method hello(), is annotated with @RequestMapping. Therefore it is called upon an HTTP request to the URL "/". The method’s return value is returned in the HTTP response.

**Lines 13 and 14:**

* Finally, the main() method starts the application with the help of the class SpringApplication.
* The application can simply be started as a **Java application** even though it processes **HTTP requests**.

**Note**: A **web server** is required for handling HTTP in the Java world. It is included in the application.

## Compiling the Spring Boot project [#](https://www.educative.io/module/lesson/introduction-to-microservices/YMBXW47XDXA#compiling-the-spring-boot-project)

For compiling the project, Spring Boot supports, among others, [Maven](https://maven.apache.org/). Here is a minimal example of a Maven build configuration file:

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.ewolff</groupId>

<artifactId>simplest-spring-boot</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

</parent>

<properties>

<java.version>10</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

The build configuration inherits settings from the parent configuration spring-boot-starter-parent.

**Maven’s parent configuration** makes it easy to reuse settings for the build of multiple projects.

The version of the **Maven parent** determines which version of Spring Boot is used.

The **Spring Boot version** defines the version of the Spring framework and the versions of all other libraries.

Thus, the developer does not have to define a stack with compatible versions of all frameworks, which is otherwise, often a challenge.

# Spring Boot Starter Web as Single Dependency

The application has a dependency on the library spring-boot-starter-web. This dependency integrates the Spring framework, the Spring web framework, and an environment for the processing of HTTP requests.

The **default** for the processing of the HTTP requests is a **Tomcat server** which runs embedded as part of the application.

Thus, the dependency on spring-boot-starter-web would be enough as a **sole dependency** for the application!

The dependency on spring-boot-starter-test is necessary for **tests**.

**Note:** The code for the test is not part of this course.

## Spring cloud [#](https://www.educative.io/module/lesson/introduction-to-microservices/m2qXWYV35NA#spring-cloud)

[Spring Cloud](http://projects.spring.io/spring-cloud/) is a collection of extensions for Spring Boot which are useful for **cloud applications** and for **microservices**.

Spring Cloud contains **additional starters**. To be able to use the Spring Cloud starters, an entry has to be inserted into the dependency-management section in the pom.xml for importing the information about the Spring Cloud starter.

The pom.xml files in the examples already contain the required import for this.

**The Maven plugin**[#](https://www.educative.io/module/lesson/introduction-to-microservices/m2qXWYV35NA#the-maven-plugin)

The Maven plugin spring-boot-maven-plugin is necessary to build a **Java JAR** that starts an environment with the Tomcat server and the application.

mvn clean package

The above command deletes the old build results and builds a new JAR.

**JAR** is a Java file format which contains **all the code** for an application.

Maven gives this JAR file a name that is derived from the project name. It can be started with:

java -jar simplest-spring-boot-0.0.1-SNAPSHOT.jar

Spring Boot can also generate **WARs** (web archives) which can be deployed on a Java web server like Tomcat or a Java application server.

# Spring Boot for Microservices: Communication

## Communication [#](https://www.educative.io/module/lesson/introduction-to-microservices/x1Br03wm9Bz#communication)

For communication, Spring Boot supports **REST**, the previous listing shows. The listing uses the **Spring MVC API**.

**T H E  S P R I N G   M V C   A P I**

The Spring MVC framework resides pretty well with REST and provides the necessary API support to implement it seamlessly, with little effort.

### The importance of SpringMVC in RESTful web services [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/x1Br03wm9Bz#the-importance-of-springmvc-in-restful-web-services)

**I**. In Spring MVC, a controller handles requests for all the HTTP methods. This serves as a backbone for RESTful web services.

**Example:**

* GET methods can be used to **handle read operations**
* POST methods can be used to **create new** resources
* PUT methods can be used to **update resources**
* DELETE methods can be used to **remove resources** from the server

**II**. The representation of data is crucial in REST. This is why Spring MVC allows us to evade View-based rendering completely by the use of @ResponseBody annotation and many HttpMessageConverter implementations. By this, a response can be sent directly to a client.

**III**. Spring version 4.0’s @RestController added in the controller class applies message conversations to all handler methods in the controller, preventing the need to annotate each method with the @ResponseBody annotation. This also makes our code much cleaner.

**IV**. Spring MVC also provides @RequestBody annotation, which uses HttpMethodConverter implementations to convert inbound HTTP data into Java objects passed into a controller’s handler method.

**V**. The Spring framework also provides a template class, the RestTemplate, which can consume REST resources. You can use this class to test your RESTful web service or develop REST clients.

These were some of the important features of the Spring MVC framework which assist in developing RESTful web services.

Spring Boot also supports the **JAX RS API**. For **JAX RS**, Spring Boot uses the library **Jersey**. JAX RS is standardized as part of the **Java Community Process (JCP)**.

For messaging, Spring Boot supports the **Java Messaging Service (JMS)**. This is a standardized API that can be used to address different messaging solutions from Java.

Spring Boot has starters for the JMS implementations [HornetQ](http://hornetq.jboss.org/" \t "_blank), [ActiveMQ](http://activemq.apache.org/) and [ActiveMQ Artemis](https://activemq.apache.org/artemis/). In addition, there is a Spring Boot starter for [AMQP](https://www.amqp.org/). This protocol is also a standard, but on the network protocol level.

The AMQP starter uses [RabbitMQ](https://www.rabbitmq.com/) as an implementation of the protocol. For AMQP as well as for JMS, Spring offers an API that makes it easier to send messages. In addition, simple Java objects (Plain Old Java Objects, POJOs) with no dependencies on any of the APIs can process AMQP and JMS messages with Spring and also return responses to messages.

Spring Cloud offers [Spring Cloud Streams](https://cloud.spring.io/spring-cloud-stream/) for implementing applications for the processing of data streams. This library supports messaging systems such as Kafka, RabbitMQ (see above) and [Redis](https://redis.io/).

Spring Cloud Stream builds on these technologies and extends them with concepts such as streams and therefore goes beyond just simplifying the use of the technology’s APIs.

The integration of technologies in Spring Boot with Spring Boot starters has the advantage that Spring Boot provides the configuration of the environment.

The example Spring Boot application in this chapter uses an infrastructure such as a Tomcat server to handle HTTP requests. This does not require a separate configuration and no additional dependencies. Spring Boot starters also offer such simplifications for messaging and other REST technologies.

Spring Boot applications can also use technologies without a Spring Boot starter. A Spring Boot application can use any technology that supports Java.

In the end, a Spring Boot project is a Java project and can be extended with Java libraries.

However, it is possible that the configuration can be more complex than a Spring Boot starter.

# Spring Boot for Microservices: Operation

## Operation [#](https://www.educative.io/module/lesson/introduction-to-microservices/R139ByJyABw#operation)

Spring Boot also has some interesting approaches for operation.

### Deployment in Spring [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/R139ByJyABw#deployment-in-spring)

* To deploy a Spring Boot application, it is enough to just copy the **JAR file** to the server and start it. Deploying a Java application can’t be further simplified.

### Configuration in Spring [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/R139ByJyABw#configuration-in-spring)

* Spring Boot offers numerous options for the [configuration](https://docs.spring.io/spring-boot/docs/2.1.2.RELEASE/reference/html/boot-features-external-config.html). For example, a **Spring Boot** application can read the configuration from a configuration file or from an environment variable. **Spring Cloud** offers support for **Consul** as a server for configurations. The examples in this course use application.properties files for configuration because they are relatively easy to handle.

### Logs in Spring [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/R139ByJyABw#logs-in-spring)

* Spring Boot applications can generate [logs](https://docs.spring.io/spring-boot/docs/2.1.2.RELEASE/reference/html/boot-features-logging.html) in many different ways. Usually, a Spring Boot application displays the logs in the console. Output to a file is also possible. A Spring Boot application can also send the logs as **JSON** data to a central server instead of using a simple human-readable text format. JSON facilitates the processing of log data on this server.

### Metrics in Spring [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/R139ByJyABw#metrics-in-spring)

* For metrics, Spring Boot offers a **special starter**, namely the [Actuator](https://docs.spring.io/spring-boot/docs/2.1.2.RELEASE/reference/html/production-ready.html). After adding a dependency to spring-boot-starter-actuator, the application collects metrics, for example about the HTTP requests. In addition, **Spring Boot Actuator** provides **REST endpoints** under which the metrics are available as JSON documents.

## New microservices [#](https://www.educative.io/module/lesson/introduction-to-microservices/qAvEYNAZ8XD#new-microservices)

Creating a new microservice is very easy with Spring Boot. A **build script** and a **main class** are enough, as shown in the example [simplest-spring-boot](https://github.com/ewolff/spring-boot-demos/tree/master/simplest-spring-boot).

To further simplify the creation of a new microservice, a **template** can be created. The template only needs to be adapted for a new microservice.

**Settings** for the configuration of the microservices or for logging can be defined in the template.

Thus, a template simplifies the creation of new microservices and facilitates compliance with macro architecture rules.

A particularly easy way to create a new Spring Boot project is to use <http://start.spring.io/>.

The developer must select the build tool, the programming language, and a Spring Boot version.

In addition, they can select different starters. Based on this, the website then creates a project that can be the basis for the implementation of a microservice.

## Resilience [#](https://www.educative.io/module/lesson/introduction-to-microservices/qAvEYNAZ8XD#resilience)

For resilience, a library like **Hystrix** can be useful.

Hystrix implements typical **resilience patterns** such as **timeouts in Java**. Spring Cloud offers an integration and further simplification for Hystrix.

## Microservices and the increasing popularity for Go [#](https://www.educative.io/module/lesson/introduction-to-microservices/3wZGEQ5nvY4#microservices-and-the-increasing-popularity-for-go)

[Go](https://golang.org/) is a programming language that is increasingly being used for microservices due to its great speed and support for concurrency. Concurrency enhances the efficiency of using multiple machines and cores.

Go also provides a powerful standard library for the creation of web services.

For further details on how Go compares with the other 4 languages commonly used for implementing Microservices, visit this [site](https://rubygarage.org/blog/top-languages-for-microservices).

Similar to Java, Go is based on the programming language C. However, in many areas Go is fundamentally [different](http://dead10ck.github.io/2014/12/15/go-vs-c.html) from C.

package main

import (

"fmt"

"log"

//"time"

"net/http"

)

func main() { http.Handle("/common/css/",

http.StripPrefix("/common/css/",

http.FileServer(http.Dir("/css"))))

http.HandleFunc("/common/header", Header)

http.HandleFunc("/common/footer", Footer)

http.HandleFunc("/common/navbar", Navbar)

fmt.Println("Starting up on 8180")

log.Fatal(http.ListenAndServe(":8180", nil))

}

// Header and Navbar left out

func Footer(w http.ResponseWriter, req \*http.Request) {

fmt.Fprintln(w,

`<script src="/common/css/bootstrap-3.3.7-dist/js/bootstrap.min.js" />`)

}

**Line 3:**

The key word import imports some libraries, among others for **HTTP**.

**Line 10, and 23:**

The program’s main function defines which methods should respond to which **URLs**. For example, the method Footer (**line 23**) returns HTML code.

On the other hand, for the URL /common/css (**line 10**) the application delivers content from files.

It is also very easy to implement a **REST** service with **Go**.

In addition, libraries like [Go kit](https://github.com/go-kit/kit) offer many more functionalities to implement microservices.

## Go build and compilation [#](https://www.educative.io/module/lesson/introduction-to-microservices/3wZGEQ5nvY4#go-build-and-compilation)

Go compilers are particularly well suited for Docker environments because they can create **static binaries**.

**Static binaries** do not require any further dependencies or a specific Linux distribution.

However, the applications must be compiled to **Linux binaries**. This requires a Go environment that can create Linux binaries.

## Docker multi-stage builds [#](https://www.educative.io/module/lesson/introduction-to-microservices/3wZGEQ5nvY4#docker-multi-stage-builds)

The example uses Docker multi stage builds. Such a build divides the build process of the Docker image into several stages.

**First Stage**

The **first stage** can compile the program in a Docker container with a Go build environment.

**Second Stage**

The **second stage** can execute the Go program in a Docker container as a runtime environment that contains only the compiled program.

Consequently, the runtime environment has no build tools and is therefore much smaller.

Docker multi stage builds are not very complicated, as a look at the Dockerfile shows:

FROM golang:1.8.3-jessie

COPY /src/github.com/ewolff/common /go/src/github.com/ewolff/common

WORKDIR /go/src/github.com/ewolff/common

RUN CGO\_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o common .

FROM scratch

COPY bootstrap-3.3.7-dist /css/bootstrap-3.3.7-dist

COPY --from=0 /go/src/github.com/ewolff/common/common /

ENTRYPOINT ["/common"]

CMD []

EXPOSE 8180

### Stage 0 [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3wZGEQ5nvY4#stage-0)

**Line 1**

The **base image** golang contains the Go installation.

**Line 3**

The Go source code is copied and compiled into this image **(line 4/5)**. With that, stage 0 of the build is finished.

### Stage 1 [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3wZGEQ5nvY4#stage-1)

Stage 1 creates a new Docker image.

**Line 7**

The image, scratch, is an empty Docker image.

**Line 8 and 9**

The Dockerfile copies the bootstrap library (line 8) and the compiled Go binary from stage 0 (line 9) into this image.

The option --from=0 indicates that the file common originates from **stage 0** of the Docker build.

### Stage 2 [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/3wZGEQ5nvY4#stage-2)

**Line 10**

Finally, ENTRYPOINT defines the binary that is supposed to be started.

**Line 11**

CMD indicates that no options are to be passed to the binary at the start.

Normally, ENTRYPOINT would be a shell that starts the process that is configured with CMD. However, in the scratch image, there is no shell.

**Line 12**

According to [Docker Documentation](https://docs.docker.com/engine/reference/builder/), “The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP. The default is TCP if the protocol is not specified.” So, port 8180 is specified here.

The criteria from the [second lesson](https://www.educative.io/collection/page/10370001/6518081205567488/5806490545815552) of this chapter for the implementation of microservices can serve as a basis to assess Go’s suitability as a microservices programming language.

## Communication [#](https://www.educative.io/module/lesson/introduction-to-microservices/7AoKzV1VAK1#communication)

Go supports **REST** in the standard libraries. Libraries are also available for messaging systems such as **AMQP**, for example <https://github.com/streadway/amqp>.

There is also a library for messaging with [Redis](https://github.com/go-redis/redis).

Due to the widespread use of Go, there is hardly any communication infrastructure that does not support Go.

## Operation [#](https://www.educative.io/module/lesson/introduction-to-microservices/7AoKzV1VAK1#operation)

Go also offers many options for operation.

### Deployment [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/7AoKzV1VAK1#deployment)

* The **deployment** in a Docker container is very easy with Docker multi stage builds, as already illustrated.

### Configuration [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/7AoKzV1VAK1#configuration)

* Libraries like [Viper](https://github.com/spf13/viper) support the **configuration** of Go applications. This library supports formats such as **YAML** or **JSON**.

### Logs [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/7AoKzV1VAK1#logs)

* Go itself already offers support for **logs**. The Go microservices framework Go Kit contains additional features for [logs](https://godoc.org/github.com/go-kit/kit/log) in more complex scenarios.

### Metrics [**#**](https://www.educative.io/module/lesson/introduction-to-microservices/7AoKzV1VAK1#metrics)

* For **metrics**, [Go Kit](https://godoc.org/github.com/go-kit/kit) supports a plethora of tools such as Prometheus, but also **Graphite** or **InfluxDB**.

## New microservices [#](https://www.educative.io/module/lesson/introduction-to-microservices/7AoKzV1VAK1#new-microservices)

For a new microservice, it is enough to create the Docker build and then write the source code.

## Resilience [#](https://www.educative.io/module/lesson/introduction-to-microservices/7AoKzV1VAK1#resilience)

Go Kit contains an implementation of resilience patterns such as [Circuit Breaker](https://godoc.org/github.com/go-kit/kit/circuitbreaker). In addition, there is a port of the [Hystrix library](https://github.com/afex/hystrix-go" \t "_blank) for Go.

Microservices have to **communicate** with other microservices. This requires a **UI integration** in the **web UI** or **protocols** such as **REST** or **messaging**.

It is a macro architecture decision which communication protocol is used (see [chapter 2](https://www.educative.io/collection/page/10370001/6518081205567488/4998953437233152)).

The technical micro architecture decisions can be made differently for each microservice. But there is a connection with the macro architecture.

The **uniformity** of the operational aspects can be enforced by the **macro architecture**.

If you want to implement a microservice with other technologies in a **Spring Boot** microservices architecture, this can lead to a lot of effort.

A macro architecture decision could be to read out configurations from an application.properties file.

This decision does not restrict the choice of implementation technologies. But for a Spring Boot application, the implementation is very simple because this mechanism is built into Spring Boot and the default for Spring Boot applications.

A **Go** application, on the other hand, would have to be adapted to this requirement.

This effect supports a uniform choice of technology for the microservices because implementing a microservice with Spring Boot is easier, therefore, developers would prefer Spring Boot.

A uniform choice of technology has further advantages. For example, developers are more likely to find their way around in other microservices, and developers of different microservices can help each other out with technology issues.

In order to really treat other technologies as equal, a different macro architecture decision should be made.

Spring Boot offers [many more options](https://docs.spring.io/spring-boot/docs/1.5.6.RELEASE/reference/html/boot-features-external-config.html).

For example, the configuration can be stored in environment variables, transferred via the command line or read from a configuration server.

## Alternatives to Spring Boot [#](https://www.educative.io/module/lesson/introduction-to-microservices/gkv4WYlvzY3#alternatives-to-spring-boot)

In the Java area, there are some alternatives to Spring Boot.

* A classic **Java EE application** with an application server or a web server is also conceivable as an implementation for a microservice. However, in this case, deployment is more complex because the application server has to be installed additionally. Also, application servers and applications must be configured, in some cases even with two different technologies. There are many doubts about the [usefulness of application servers](https://jaxenter.com/java-application-servers-dead-112186.html).
* [thorntail](https://thorntail.io/) provides a simple JAR deployment. However, instead of Spring APIs it implements the standardized Java EE APIs and supplements them with technologies from the microservices area such as Hystrix.
* [Dropwizard](http://www.dropwizard.io/) has long been offering the possibility of developing Java REST services and deploying them as JARs.

Of course, there are many other possible choices for the programming language apart from **Java** or **Go**.

It is impossible to even list them in this course.

Actually, the point this course makes is that the technologies for the implementation of each microservice are not that important.

It is easily possible to implement each microservice with a different programming language and framework, so the decision can easily be changed.

However, it is much harder to change the technologies for communication, integration, and operations that this course focuses on.

The criteria from [lesson 2](https://www.educative.io/collection/page/10370001/6518081205567488/5806490545815552) of this chapter are a yardstick to check the technologies for their suitability for microservices, as [lesson 8](https://www.educative.io/collection/page/10370001/6518081205567488/4799063738286080) does for Spring Boot and [lesson 12](https://www.educative.io/collection/page/10370001/6518081205567488/5334332627484672) for Go. Such an assessment is recommended for each technology used.

In the next lesson, we’ll discuss the advantages of microservices that arise from our discussion in the previous lessons, and then formally conclude this course!