

Introduction

In this lesson, we'll get a quick introduction to this course.

WE'LL COVER THE FOLLOWING ^

- Why this course?
- Basic principles
- Concepts
- Recipes
- Source code

Why this course?

Microservices are one of the most important software architecture trends. A number of detailed guides to microservices are already currently available, including the [microservices-book](#) written by the author of this course. **So, why do we need yet another course on microservices?**

It is **one thing to define an architecture, quite another to implement it.** This course presents technologies for the implementation of microservices while highlighting their advantages and disadvantages.

The **focus** rests specifically on **technologies** for entire microservices systems. Each individual microservice can be implemented using different technologies.

Therefore, the technological decisions about frameworks for individual microservices are not as important as the decisions at the level of the overall system. For individual microservices, the decision about a framework can be quite easily revised. However, the technologies chosen for the overall system are difficult to change.

Compared to the [microservices-book](#), this course talks primarily about

technologies. We have another course called [An Introduction to](#)

[Microservice Principles and Concepts](#) that does discuss architecture and reasons for and against microservices.

Basic principles

To become familiar with microservices, an introduction into microservices-based architectures and their benefits, disadvantages, and variations is essential.

[An Introduction to Microservice Principles and Concepts](#) explains the basic principles to the extent required for understanding the **practical implementations**.

Concepts

Microservices require **solutions for different challenges**. Among those are concepts for **integration** (*frontend integration, synchronous, and asynchronous microservices*) and for **operation** (*monitoring, log analysis, tracing*).

Microservices platforms such as *PaaS* or *Kubernetes* represent exhaustive solutions for the operation of microservices.

Recipes

The course uses **recipes as a metaphor for the technologies**, which can be used to implement the different concepts you will learn. Each approach shares a number of features with a recipe.

- Each recipe is described in *practical terms*, including an example technical implementation. The most important aspect of the examples is their *simplicity*. Each example can be easily followed, extended, and modified.
- The course provides the reader with a *plethora of recipes*. The readers have to select a specific recipe from this collection for their projects, akin to a cook who has to select a recipe for their menu. The course shows different options because, in practice, nearly every project has to be dealt with differently. The recipes build the basis for this.
- *Variations* exist for each recipe. After all, you can cook a recipe in a variety of ways. This is also true for the technical principles described in this course.

variety of ways. This is also true for the technologies described in this

course. Sometimes the variations are very simple so that they can be immediately implemented as experiments in an executable example.

Each recipe includes an associated *executable example* based on a concrete technology. The examples can be run individually as they are not based on each other. This allows the readers to concentrate on the recipes that are interesting and useful for them and to skip the examples that are less relevant for their work. In this manner, the course provides easy access for obtaining an *overview* of the relevant technologies, thereby enables the readers to select a suitable technology stack. Subsequently, the readers can use the links supplied in the course to acquire in-depth knowledge about the relevant technologies.

Source code

Sample code is provided for almost all the technologies presented in this course. If the reader wants to really understand the technologies, they should browse the code. It also makes sense to look at the code to understand how the concepts are actually implemented.

In the next lesson, we'll have a quick look at the structure of the course.

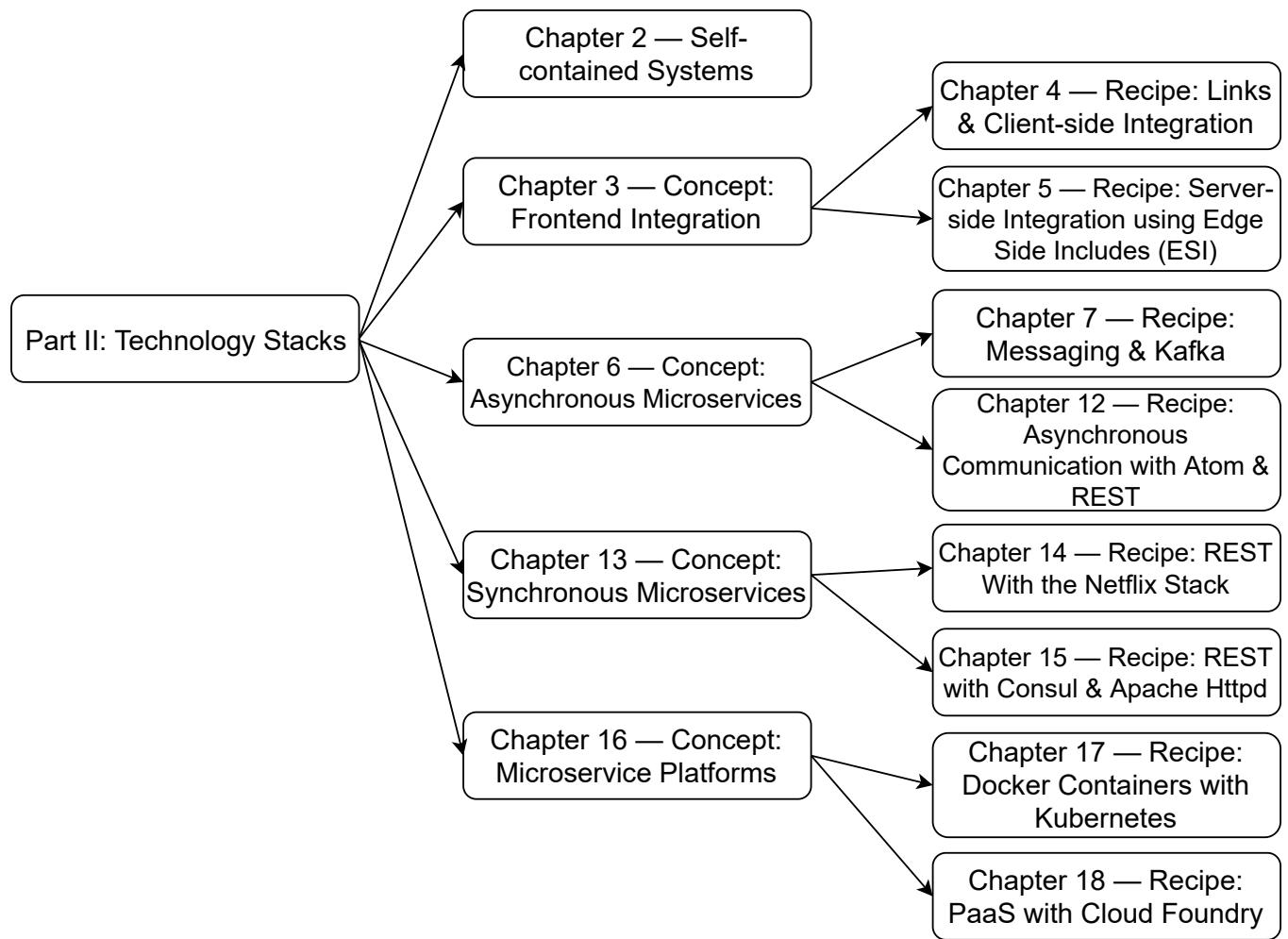
Structure of the Course

In this lesson, we'll get a quick overview of the structure of this course.

WE'LL COVER THE FOLLOWING ^

- Appendix
- Target Group
- Prior Knowledge
- Quick start
- Acknowledgements

This course is **part II** of a series of courses and focuses on *technology stacks*.



- Chapter 2 explains **self-contained systems (SCS)** as an especially useful approach for microservices. It focuses on microservices that include a UI as well as logic.
- One possibility for integration for SCS in particular is **integrating at the web frontend** ([chapter 3](#)). Frontend integration results in a loose coupling between the microservices and a high degree of flexibility.
- The recipe for web frontend integration presented in [chapter 4](#) capitalizes on **links and JavaScript** for dynamic content loading. This approach is easy to implement and utilizes well-established web technologies.
- On the server, integration can be achieved with **ESI (Edge Side Includes)** ([chapter 5](#)). ESI is implemented in web caches so that the system can attain high performance and reliability.
- The concept of **asynchronous communication** is the focus of [chapter 6](#). Asynchronous communication improves reliability and decouples the systems.
- **Apache Kafka** is an example of an asynchronous technology ([chapter 7](#)) for sending messages. Kafka can save messages permanently and thereby enables a different approach to asynchronous processing.
- An alternative for asynchronous communication is **Atom** ([chapter 8](#)). Atom uses a REST infrastructure and thus is very easy to implement and operate.
- [Chapter 9](#) illustrates how to implement **synchronous microservices**. Synchronous communication between microservices is often used in practice although this approach can pose challenges regarding response times and reliability.
- The **Netflix Stack** ([chapter 10](#)) offers Eureka for service discovery, Ribbon for load balancing, Hystrix for resilience, and Zuul for routing. The Netflix Stack is widely used in the Java community.
- **Consul** ([chapter 11](#)) is an alternative option for service discovery. Consul

contains numerous features and can be used with a broad spectrum of technologies.

- [Chapter 12](#) explains the concept of **microservices platforms**, which support operations and communications of microservices.
- The **Kubernetes** ([chapter 13](#)) infrastructure can be used as a microservices platform and is able to execute Docker containers, as well as having solutions for service discovery and load balancing. The microservice remains independent of this infrastructure.
- **PaaS (Platform as a Service)** is another infrastructure that can be used as a microservices platform ([chapter 14](#)). Cloud Foundry is used as an example. Cloud Foundry is very flexible and can be run in your own computing center as well as in the public cloud.

Appendix

The appendix contains a local set up of coding environments.

Target Group

This course explains basic principles and technical aspects of microservices. Thus, it is interesting to different audiences for a variety of reasons.

For **developers**, this course offers a **guideline for selecting a suitable technology stack**. The example projects serve as a basis for learning the foundations of the technologies.

The microservices contained in the example projects are written in Java using the Spring Framework. However, the technologies used in the examples serve to integrate microservices. So additional microservices can be written in different languages.

This course also introduces technologies for deployment such as **Docker**, **Kubernetes**, or **Cloud Foundry** that also solve some operational challenges.

Managers are presented with technical details that will benefit them in their careers.

Prior Knowledge

The course assumes the reader to have basic **knowledge of software architecture and software development**. All practical examples are documented in such a way that they can be executed with little prior knowledge.

This course focuses on technologies that can be employed for microservices using different programming languages. However, the examples are written in Java using the Spring Boot and Spring Cloud frameworks so that **changes to the code require knowledge of Java**.

Quick start

The examples are explained in the following sections:

Concept	Recipe	Lesson
Frontend Integration	Links and Client-side Integration	Example
Frontend Integration	Edge Side Includes (ESI)	Example
Asynchronous Microservices	Kafka	Example
Asynchronous Microservices	REST and Atom	Example
Synchronous Microservices	Netflix Stack	Example
Synchronous Microservices	Consul and Apache httpd	Example

All projects are available on [GitHub](#). The projects always contain a [how-to-](#)

[RUN.md](#) file showing step-by-step instructions on how the demos can be

installed and started.

The examples are independent of each other, so you can start with any one of them.

Acknowledgements

I would like to thank everybody who discussed microservices with me, who asked me about them, or worked with me. Unfortunately, these people are far too numerous to name them all individually. The exchange of ideas is enormously helpful and also fun!

Many of the ideas and their implementation would not have been possible without my colleagues at INNOQ. I would especially like to thank Alexander Heusingfeld, Christian Stettler, Christine Koppelt, Daniel Westheide, Gerald Preissler, Hanna Prinz, Jörg Müller, Lucas Dohmen, Marc Giersch, Michael Simons, Michael Vitz, Philipp Neugebauer, Simon Kölsch, Sophie Kuna, Stefan Lauer, and Tammo van Lessen.

Also, Merten Driemeyer and Olcay Tümce who provided important feedback.

Finally, I would like to thank my friends and family, whom I often neglected while writing this course – especially my wife. She also did the translation into English.

Of course, my thanks also go out to the people who developed the technologies, which I introduce in this book and thereby created the foundation for microservices.

I also would like to thank the developers of the tools of <https://www.sofcover.io/> and Leanpub.

Let's get started!

Introduction

In this lesson, we'll look at a quick introduction to self-contained systems.

WE'LL COVER THE FOLLOWING



- Definition
- Reasons for the term self-contained systems
- Chapter walkthrough

Definition

A **self-contained system (SCS)** is a type of microservice that **specifies elements of a macro architecture**. SCSs do not represent a complete macro architecture. The area of operation, for instance, is completely missing.

The idea behind SCS is to provide microservices that are self-contained, providing everything needed for the implementation of a part of the domain logic.

This means an SCS includes **logic, data, and a UI**. That also means if a change impacts all technical layers, it can still be contained in one SCS, making it easier to perform the change and put it into production.

So, an SCS for a microservice **payment** would store all information relevant to payment – that is, it would implement a **bounded context**. But it would **also implement the UI** – web pages to show the payment history or make a payment. **Data** about customers or ordered items would need to be replicated from other SCSs.

Reasons for the term self-contained systems

There is no uniform definition for microservices. **Self-contained systems, however, are precisely defined**. You can read the definition of SCSs on the

<http://scs-architecture.org> website. The content of the site is provided under a creative commons license, so that the material on the site may be reused by anyone if the source is named, and the materials are distributed under the same license terms.

The content of the website is available as source code at <https://github.com/innoq/SCS> so that everyone can make changes. The website contains links to articles about experiences with SCSs from different projects and companies.

Self-contained systems are best practices that have proven their usefulness in various projects.

While microservices do not provide many rules about how systems should be built, SCSs have precise rules based on proven patterns. Thus, SCSs give a point of reference as to how microservices architecture can look.

Although SCSs are a collection of best practices, the SCS approach is not the best architecture for every situation. Therefore, it is important to understand the reasons for the SCS rules. By doing so, teams can choose variations of this approach or even completely different approaches that might be better adapted to the respective project.

Chapter walkthrough

This chapter covers the following:

- First, it describes the reasons for using SCSs.
- SCSs determine different macro architecture decisions. This chapter explains the reasons for these macro architecture decisions and discusses the advantages to each decision.
- SCSs are a variation of microservices. This chapter addresses the differences between the terms “SCS” and “microservice.”
- This chapter then discusses the benefits of the SCS approach.
- Finally, challenges connected to the development of an SCS system are described and potential solutions are discussed

described and potential solutions are discussed.

SCSs include a UI and focus on UI integration. Therefore, SCSs are a good motivation for the UI integration techniques that the next chapters explain.

QUIZ

1

Suppose a microservice contains the data and logic but the UI is implemented from another microservice. Is this microservice an SCS?

COMPLETED 0%

1 of 2



In the next lesson, we'll discuss a more thorough definition of self-contained systems.

Definition

In this lesson, we'll further discuss and define microservices.

WE'LL COVER THE FOLLOWING



- Rules for communication
- SCS databases
- Rules for the organization
- Rule: minimal common basis
 - Avoid shared business logic
 - Avoid common infrastructure

Self-contained systems make different macro architecture decisions.

- Each self-contained system is an **autonomous web application**, meaning SCSs contain web UIs.
- There is **no common UI**. An SCS can have HTML links to other SCSs or can integrate itself by different means into the UI of other SCSs. But every part of the UI belongs to an SCS. [Chapter 3](#) describes different options for the integration of frontends. This means each SCS generates a part of the UI. There is no separate microservice that generates all of the UI and then calls logic in the other microservices.
- The SCSs can have an **optional API**. This API can, for example, be useful if mobile clients or other systems have to use the logic in the SCS.
- The **complete logic** and **data** for the domain are contained in the SCS. This is what SCSs were named for. An SCS is self-contained because it contains UI, logic, and data.

These rules ensure that an SCS completely implements a domain. This means that **a new feature causes changes to only one SCS** even if the logic, data,

and the UI is changed. These changes can be rolled out with a single deployment.

If several SCSs shared a UI, many changes would affect not only the SCSs but also the UI. Then two well-coordinated deployments and a closely coordinated development would be necessary.

Rules for communication

The communication between SCSs has to follow a number of rules:

- Integration at the **UI level** is ideal as the coupling is very loose. The other SCS can display its UI as required. Even if the UI is changed, other SCSs are not affected.
 - For example, if HTML links are used, the integrated SCS doesn't even have to be available. The link is displayed even if the system is unavailable; there will be an error if the user clicks on the link. This helps with resilience because the failure of an integrated SCS does not affect other SCSs.
- The next option is **asynchronous communication**. The advantage of this is that if the integrated SCS fails, the requests are delayed only until the failed SCS is available again. However, the calling SCS will not fail because it has to be able to deal with longer latency times.
- Finally, integration with **synchronous communication** is also possible. In such cases, precautions must be taken to deal with the potential failure and slow responses of the integrated SCSs. In addition, the response times add up if you have to wait for the responses of all synchronous services.

These rules focus on a very loose coupling between the SCSs to avoid error cascades in which one system fails and consequently the dependent systems fail.

SCS databases

These rules mean that an SCS **replicates data**. The SCS has its own database, and since it is primarily intended to communicate asynchronously with other SCSs, it is a challenge if the SCS must use data from another SCS to process a request. If the data is requested during the processing of the request, the

communication is synchronous.

For asynchronous communication, the data must be replicated beforehand so that it is available in the SCS when processing the request.

Consequently, SCSs are **not always consistent**. If a change to the dataset has not yet been passed to all SCSs, then the SCSs have different datasets which may not be acceptable in some situations. For particularly high-consistency requirements, the SCSs must use synchronous communication. In this scenario, one SCS receives the current state of the data from the other SCS.

Rules for the organization

SCSs provide macro architecture rules for the organization that they are being used in. An SCS belongs to **one team**.

The team does not necessarily have to make all the changes to the code, but it must at least review, accept, or reject changes. This enables the team to direct and control the development of the SCS.

A team can handle several SCSs. However, it isn't a good idea for an SCS to be changed by more than one team.

Thus, self-contained systems **use strong architectural decoupling to achieve organizational advantages**.

- The teams do not need to coordinate very much and can work in parallel on their tasks.
- Requirements can usually be implemented in one SCS by one team because an SCS implements a domain. Coordination in this respect is therefore hardly necessary.
- Because the technical decisions mostly concern only one SCS, there are hardly any arrangements necessary.

For SCSs, as well as for microservices and other types of modules, the division is aimed at enabling independent development. Having several teams change the same module is obviously not a good idea.

The modules allow independent development, but when several teams are working on one module, close coordination is still necessary. The advantage of

modularization is not exploited then. This is why equal joint development of an SCS by several teams is not allowed.

Rule: minimal common basis

Because self-contained systems aim at enabling a high degree of independence, the common basis should be minimal.

Avoid shared business logic

Business logic must not be implemented in code used by more than one SCS. *Shared business logic* leads to a close coupling of the SCSs which should be avoided. Otherwise, a change to one SCS could require changes to the shared code.

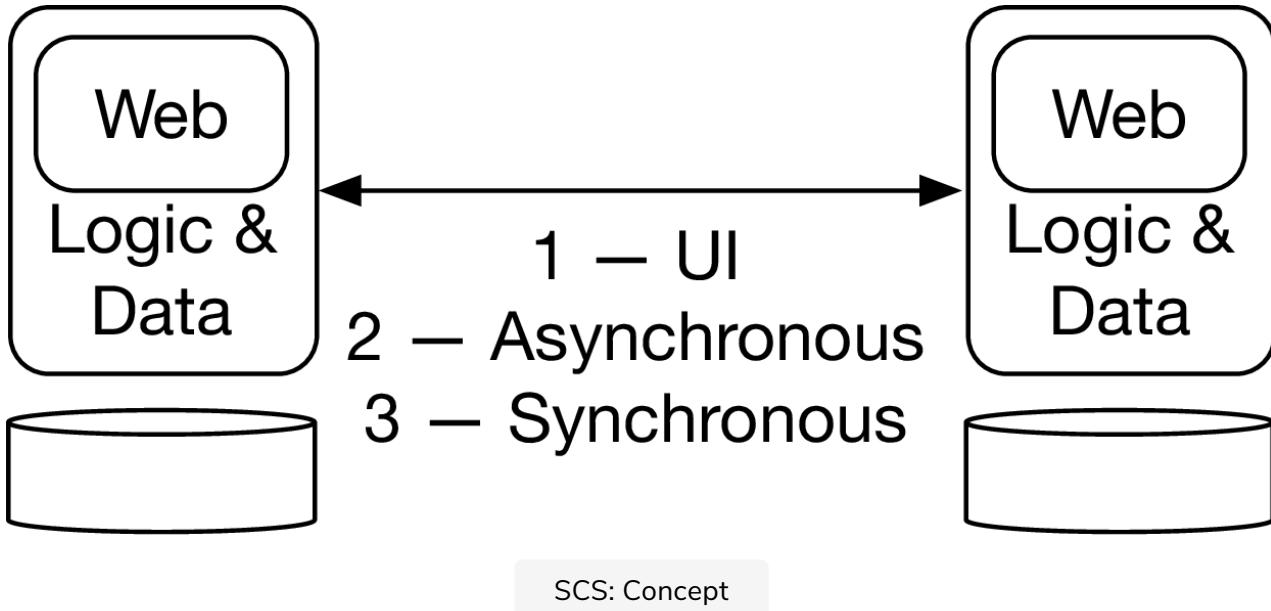
Such changes must be coordinated with other users of the code which creates a tight coupling that SCSs should avoid. In addition, common business code indicates a bad domain macro architecture. Business logic should be implemented in a single SCS. The business logic in an SCS can, of course, be called and used by another SCS via the optional interface of the SCS.

Avoid common infrastructure

Common infrastructure should be avoided. SCSs should not share a database, otherwise, the failure of the database could lead to a failure of all SCSs.

However, a separate database for each SCS requires a considerable effort. For this reason, compromises are conceivable if the robustness of the system is not quite so important. The SCSs could have a separate schema in a common database. A shared schema would violate the rule that each SCS should have its own data.

The drawing below provides an overview of the most important features of the concept of self-contained systems.



Here are some shorter rules to keep a minimal common basis:

- Each SCS contains its own *web UI*.
- In addition, the SCS contains the *data* and the *logic*.
- Integration is *prioritized*. UI integration has the highest priority, followed by asynchronous and finally synchronous integration.
- Each SCS ideally has its *own database* to avoid a common infrastructure.

Q U I Z

1

Which of the following is the best for inter-SCS communication?

In the next lesson, we'll look at an example of an SCS architecture.

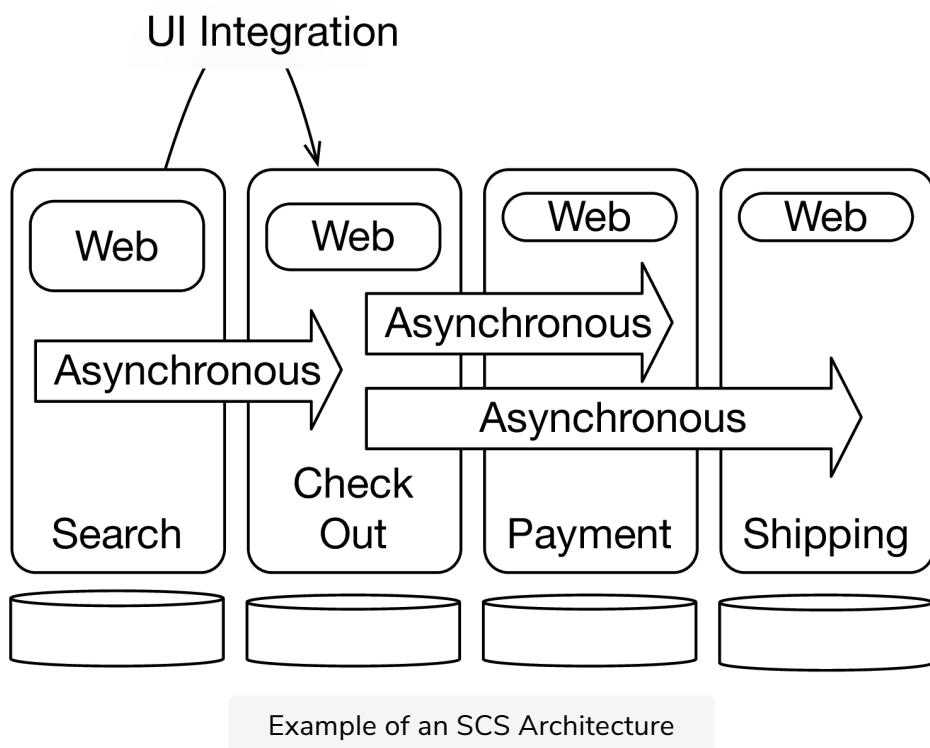
An Example

In this lesson, we'll look at an example of an SCS.

WE'LL COVER THE FOLLOWING ^

- E-commerce example
- Communication

E-commerce example



The drawing above demonstrates how an e-commerce system divided into the bounded contexts **search**, **check-out**, **payment**, and **shipping** can be implemented with SCSs.

1. One SCS implements the **search** for products.
2. At **check out**, a filled shopping cart is turned into an order.
3. **Payment** ensures that the order is paid and provides information about

the payment.

4. **Shipping** sends the goods to the customer and offers the customer the possibility to obtain information regarding the state of the delivery.

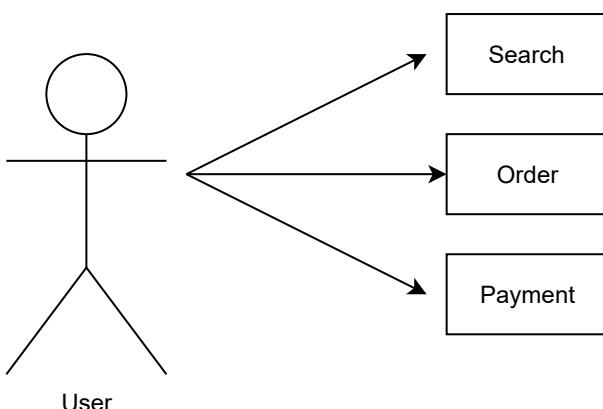
Each SCS implements a bounded context with its own database, or, at least, one schema in a common database. In addition to the logic, each SCS also contains the web UI for the respective functionalities.

An SCS does not necessarily have to implement a bounded context, but this achieves a high degree of independence of the domain logic.

Communication

A user sends an HTTP request to the system. The HTTP request is then usually processed by a single SCS because all the necessary data is available in the SCS.

This is good for performance and also for resilience. The failure of one SCS does not lead to the failure of another SCS, because the SCS does not call any other systems when processing the requests. Slow calls of other SCSs via the network are also avoided, which promotes performance.



The user communicates directly with all SCSs. They generally do not communicate with each other.

Of course, the SCSs must still communicate with each other, they are, after all, part of an overall system. One reason for communication is the **lifecycle of an order**.

1. The customer searches for products in **search**
2. Orders them in **check out**
3. Pays them in **payment**

4. Tracks them in **shipping**

When moving from one step to the next, the **information about the order must be exchanged between the systems**. This can be done **asynchronously**. The information does not have to be available in the other SCSs until the next HTTP request; therefore, **temporary inconsistencies are acceptable**.

In some places, close integration seems to be necessary. This means that *search*, for example, must display not only the search results, but also the contents of the shopping cart.

However, the shopping cart is managed by *check out*. UI integration can be a solution to challenges like these. In that case, the *check out* SCS can decide how the shopping cart is displayed, and the representation will be integrated in the other SCSs. Even changes to the logic rendering the shopping cart will be implemented only in one SCS, although many SCSs will display the shopping cart as part of their web pages.

QUIZ

1

Is it possible to display a component of an SCS in other SCSs?

COMPLETED 0%

1 of 2



In the next lesson, we'll study the differences between self-contained systems and microservices.

SCSs and Microservices

In this lesson, we'll study the relationship between SCSs and microservices.

WE'LL COVER THE FOLLOWING



- One SCS can be split into several microservices
- Summary

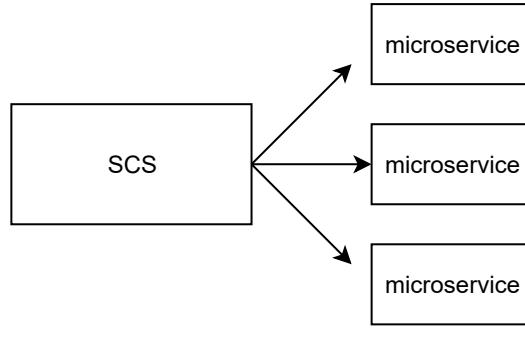
SCSs stand out from deployment monoliths in the same way as microservices. A deployment monolith would implement the entire application in a single deployable artifact while **SCSs divide the system into several independent web applications**.

Because an SCS is a separate web application, it can be deployed independently of the other SCSs. SCSs are also modules of an overall system. Therefore, **SCSs are independently deployable modules** and consequently correspond to the definition of microservices.

One SCS can be split into several microservices

However, an SCS may be split into several microservices. If the payment in the *check out* SCS of an e-commerce system causes a particularly high load, then it can be implemented as a separate microservice which can be **scaled separately** from the rest of the *check out* SCS. So, the *check out* SCS consists of a microservice for payment and contains the rest of the functionality in another microservice.

Another reason for separating a microservice from an SCS is the **security advantage offered by greater isolation**. Additionally, domain services that calculate the price of a product or quotation for all SCSs can be a useful shared microservice. The microservice should be assigned to a team like an SCS in order to avoid too much coordination.



An SCS can be split into microservices

Summary

In summary, microservices and SCSs differ in the following ways:

- Typically, **microservices are smaller than SCSs**. An SCS can be so large that an entire team is busy working on it. Microservices can potentially have only a few hundred lines of code.
- **SCSs focus on loose coupling**. There is no such rule for microservices, although tightly coupled microservices have many disadvantages and therefore should be avoided.
- In any case, **an SCS must have a UI**. Many microservices offer only a technical interface for other microservices but no user interface.
- **SCSs recommend UI integration or asynchronous communication**; synchronous communication is allowed but not recommended. Large microservices systems such as Netflix also focus on *synchronous communication*.

QUIZ

1

If an SCS can be separated into several microservices, it is still a good idea for the original team to work on all the microservices.

COMPLETED 0%

1 of 2



In the next lesson, we'll look at some challenges pertaining to using SCSSs.

Challenges

In this lesson, we'll look at some challenges pertaining to the use of self-contained systems.

WE'LL COVER THE FOLLOWING ^

- Limitation to web applications
- Single page app (SPA)
 - Disadvantages
 - SCS based challenges
- Mobile applications
 - Web application
 - Web application with framework
 - Native app
- Look and feel

SCS describes an architectural approach that is more narrowly defined than microservices. Therefore, **SCS cannot be the approach for solving all problems.**

Limitation to web applications

The first limitation of SCSs is that **they are web applications**. Thus, SCSs are not a solution when a web UI is not required.

However, some aspects of SCSs can still be implemented in a scenario that does not involve web applications: the clear separation of domains and the focus on asynchronous communication.

If a system is to be developed that offers only an API, an architecture can be created that provides at least some of the advantages of SCSs.

Single page app (SPA)

A single page app (SPA) is usually an application written in JavaScript that runs in the browser and it often implements complex UI logic. Applications such as Google Maps or Gmail are examples of applications that are highly complex and must be very interactive. SPAs are ideal for these cases.

Disadvantages

However, SPAs also have **disadvantages**:

- Since logic can be implemented in an SPA, in practice **business logic often also leaks into the UI**. This makes further development difficult, because logic is now implemented on the server and the client and thus at two different points in two, usually different, programming languages.
- The **load times of an SPA can be higher than the load times of a simple website**. Not only must HTML be displayed, but the JavaScript code must be loaded and started. Loading times are very important in some areas like e-commerce, because the user behavior of customers depends on loading times.

SCS based challenges

With SCSs, these problems are complemented by **further challenges** such as:

- An **SPA for the entire system** would mean that there is a **common UI**. This is forbidden in SCSs.
- An **SPA per SCS** is one possibility for providing each SCS with its own UI.
 - In this case, switching from one SCS to another means starting and loading a new SPA, which can take some time.
 - Moreover, it is not easy to split one SCS into multiple SCSs, because the SPA also needs to be split up. However, a division can be important for the development of the system in order to adapt to the architecture.

Due to the high popularity of SPAs among developers, the contradictions between SPAs and SCSs in practice are a major reason for difficulties in implementing SCSs.

An alternative is [ROCA](#). It establishes rules that stand for classic web applications and are much easier to use with the SCS idea.

Mobile applications

SCSs are not suitable as a backend for mobile applications. The mobile application is a separate UI from the backend so that UI and logic is separated and the concepts of SCS are violated.

Of course, it is still possible to implement an SCS with a web UI that also provides a backend for a mobile application.

There are different **alternatives**. Let's look at a few.

Web application

A *web application* can be developed instead of a mobile application. It can be responsive so that the layout adapts to a desktop, tablet, or smartphone.

Compared to a native app, the advantage is that there is no need to download and install an app from the app store. The number of apps that a typical mobile user has is quite low. Therefore, the installation of an app can be a hurdle.

Thanks to HTML5, JavaScript applications can now use many of the mobile phone features. Websites such as <https://caniuse.com/> show which features are provided by which browser. When the decision is made in favor of a web interface, real SCSs can be implemented.

Web application with framework

A *web application can be implemented with a framework* such as [Cordova](#) that takes advantage of the smartphone's specific features.

There are other solutions based on Cordova. Therefore, you can still create a real SCS, but have the app be downloaded from the app store, use all the features of the smartphone, and have it behave like a native app.

Of course, it is possible to implement parts of an app with such a framework and implement the rest as a truly native app.

Native app

Finally, a *native app* can be created where, for example, the backends can offer a REST interface. If the backends do not also implement a web interface, they are not SCSs. Even in this case, it is possible to divide the logic into largely independent services and use asynchronous communication between the services in order to achieve many advantages of the SCS architecture.

Look and feel

Dividing a system into several web applications quickly raises the question of a uniform look and feel. **A uniform look and feel can only be achieved with a macro architecture decision.** This also applies to SCSs, of course.

QUIZ

1

Suppose you are developing an app that has no UI. Is it possible to migrate it from a monolithic system to one that leverages the advantages of an SCS-based one?

COMPLETED 0%

1 of 2



In the next lesson, we'll study the benefits of using SCSs.

Benefits & Variations

In this lesson, we'll look at some benefits of self-contained systems and some variations to the approaches discussed here.

WE'LL COVER THE FOLLOWING ^

- Benefits
- Variations
- Typical changes
- Possibilities for combinations

Benefits

SCSs are a logical extension of the idea of bounded contexts.

A **bounded context** contains the logic on a specific part of the domain.

SCSs make sure that the UI and the persistence are part of one single microservice so that the **split by domain that domain-driven design advocates are further improved by SCS**.

Ideally, one feature should lead to one change. In an SCS system, the chance that this actually happens is quite high: the change will likely be in one domain. It can be contained in one SCS even if the UI and the persistence need to change, and the change can be put into production with one deployment. Therefore, **SCSs provide better changeability**.

Also, **testing is easier** because all the required code for the domain is in one domain. So, it is easier to do meaningful tests of the logic together with the UI.

Due to the focus on the UI, SCSs make it **easier to implement integration in the UI**. SCSs are a logical extension of the idea of bounded contexts. This is a simple

the UI. So, SCSs open up additional integration options. This is particularly

useful for heterogeneous UI technology stacks. With the rate of innovation in UI technologies, it is unrealistic to assume a uniform UI technology stack.

Variations

Instead of implementing an SCS with UI, logic, and data there are different variations. These architectures are not SCS architectures; however, they might still be sensible in certain contexts.

- Domain microservices with logic and data but *without UI* can make sense when an API or a backend is to be implemented.
- Microservices without logic or data, which consequently implement *only a UI*, can be a good approach for implementing a portal or a different kind of frontend.

Both of these variations are sensible when the project is a pure frontend or backend and does not allow the full implementation of an SCS.

Typical changes

Good architecture should **limit a change to one microservice**. The basic assumption of the SCS architecture is that a change typically goes through all layers and is limited to a domain implemented in one microservice. This assumption has been confirmed in many projects.

Still, if a change in a project usually affects only the UI or only the logic, it may be better to implement a division by layers. A new look and feel or new colors can be implemented in the UI layer, while logic changes can be implemented in the logic layer – of course, only if they do not affect the UI.

Possibilities for combinations

Self-contained systems are easy to combine with other recipes.

- Frontend integration (see [chapter 3](#)) is preferred when integrating SCSs.
- SCSs focus on asynchronous communication ([chapter 6](#)). SCSs are web applications so that the use of Atom ([chapter 8](#)) for asynchronous REST

communication is especially easy because it also builds on HTTP. Kafka

([chapter 7](#)), on the other hand, has a different technical foundation; even though it can still be combined with SCSs, it represents an additional technical effort.

- Synchronous communication ([chapter 9](#)), even though possible, should be avoided.

Thus, UI integration, asynchronous communication, and synchronous communication between SCSs are possible, but a clear prioritization exists. Of course, SCSs can also communicate with microservices or other systems via these mechanisms.

QUIZ

1

Suppose you are implementing a system where one change always affects the UI or the logic. How would this system best be divided?

COMPLETED 0%

1 of 2



In the next lesson, we'll look at the conclusion to this chapter!

Chapter Conclusion

We'll conclude this chapter with a quick summary of what we learned.

WE'LL COVER THE FOLLOWING ^

- Summary

Summary

- Self-contained systems represent an approach for the implementation of microservices that have proven their usefulness in numerous projects.
- Self-contained systems provide best practices on how to implement successful architectures with microservices.
- However, Self-contained systems also restrict the general approaches of microservices and are therefore more specialized.
- Nevertheless, aspects such as asynchronous communication and division into different domains are a helpful approach in many situations.

That's it for this chapter! The next one is about frontend integration.

Introduction

In this lesson, we'll look at a quick introduction to frontend integration.

WE'LL COVER THE FOLLOWING ^

- Chapter walkthrough

Chapter walkthrough

This chapter explains how microservices can be integrated in the web frontend.

- First, this chapter discusses *why* the web frontend of a microservices architecture should be modularized.
- *Single page apps (SPAs)* are difficult to modularize.
- *ROCA (resource-oriented client architecture)* is an approach for web UIs which supports modularization.
- Furthermore, this chapter covers the available *options* for integration.
- Finally, the *benefits* and *disadvantages* of the different integration options are discussed.

This chapter demonstrates how frontend integration can be implemented and for which scenarios this is the best approach.

In the next lesson, we'll look at how a monolithic frontend compares to a modular frontend.

Frontend: Monolith or Modular?

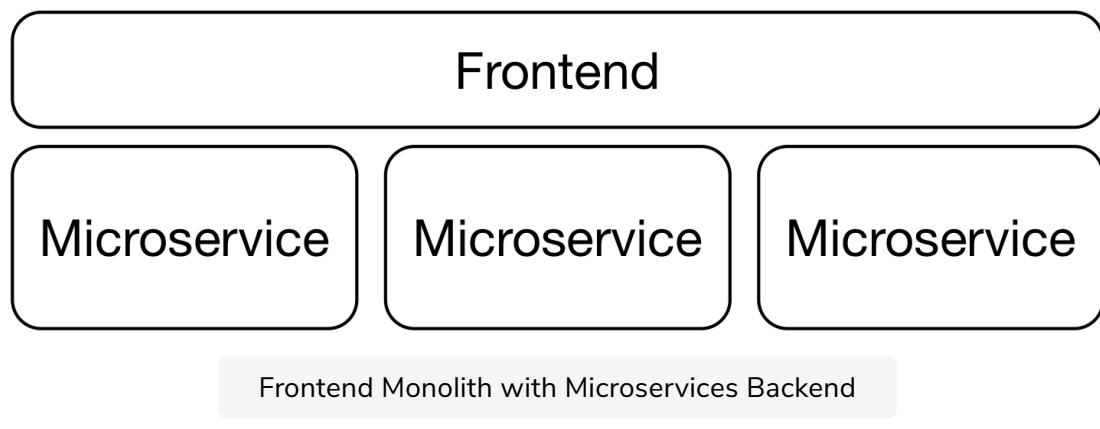
In this lesson, we'll debate the pros and cons of a monolithic frontend and a modular frontend.

WE'LL COVER THE FOLLOWING



- Option: monolithic frontend and backend
- Option: modularly developed frontend
- Reasons for a frontend monolith
 - Native mobile & rich client apps
 - SPAs
 - Single team
 - Migration
- Modularized frontend
 - Advantages
 - Modularized frontend and frontend integration

The following drawing shows a frontend monolith which serves as the frontend for multiple microservices.



Option: monolithic frontend and backend

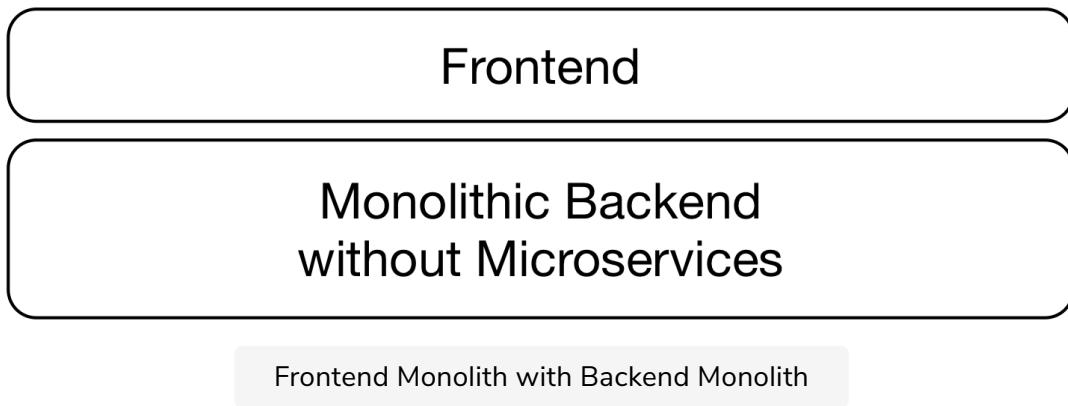
Doing without modularization in the frontend is inconsistent. On the one hand, a backend is modularized into microservices which involves a great

rather, a backend is modularized into microservices which involves a great deal of effort.

On the other side is a monolithic frontend, which is an architecture that must be questioned. The result does not have to be a modularized frontend. It may also turn out that a monolithic frontend together with a monolithic backend also meets the requirements. Then you can save the effort for the modularization of the backend into microservices.

This can be the case, for example, if the advantage of separate deployment and further decoupling in other areas is not important after all.

The following drawing shows a frontend monolith which serves as frontend for a monolithic backend.

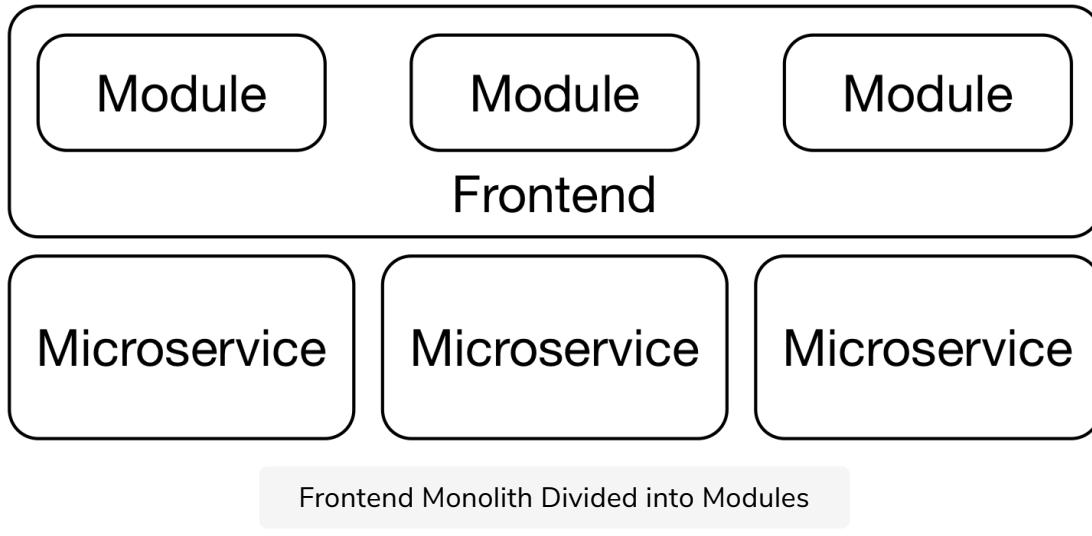


Option: modularly developed frontend

Of course, a **frontend deployed as a monolith can be developed modularly**. Unfortunately, experience shows that modular development often still leads to an unmaintainable, non-modular system in the end because the boundaries between the modules dissolve over time. The boundaries between modules that can be deployed separately, such as microservices, are not easy to circumvent so that modularization is secured in the long term.

Nevertheless, each microservice can, for example, be assigned a module in the frontend in order to decouple and parallelize the development. However, in that case, the frontend monolith still has to be deployed as a whole. Separate deployment is one advantage of microservices over other modularizations.

The following drawing shows a frontend monolith that is divided into modules.



Reasons for a frontend monolith

A frontend monolith can be a good choice in some circumstances.

Native mobile & rich client apps

Native mobile applications or rich client applications are **always deployment monoliths**. In other words, they can be delivered only as a whole.

Mobile applications even need to pass a review process in the app store in case of an update, so deployment takes even longer. However, this mechanism can be “undermined” to a certain extent. An app can display web pages. These can be provided by a microservices system that uses web frontend integration.

Frameworks such as [Cordova](#) even allow web applications to take advantage of proprietary mobile phone features or to offer a web application for download from the app store. This means that compromises can be implemented between native applications and web applications.

SPAs

Single page apps (SPAs) can also only be deployed as a whole. There are enough alternatives to SPAs for web applications to completely modularize an application in the frontend.

The boundaries are fluid; SPAs can contain links to other sites or other SPAs and display HTML generated by a different system. In this way, SPAs can be integrated into the frontend. But in practice, **SPAs typically lead to a**

Integrated into the frontend. But in practice, SPAs typically lead to a **frontend deployment monolith** despite these theoretical possibilities.

Single team

One reason to develop a monolithic frontend would be if there was *a team* that was dedicated to frontend development. Each team should be responsible for one component.

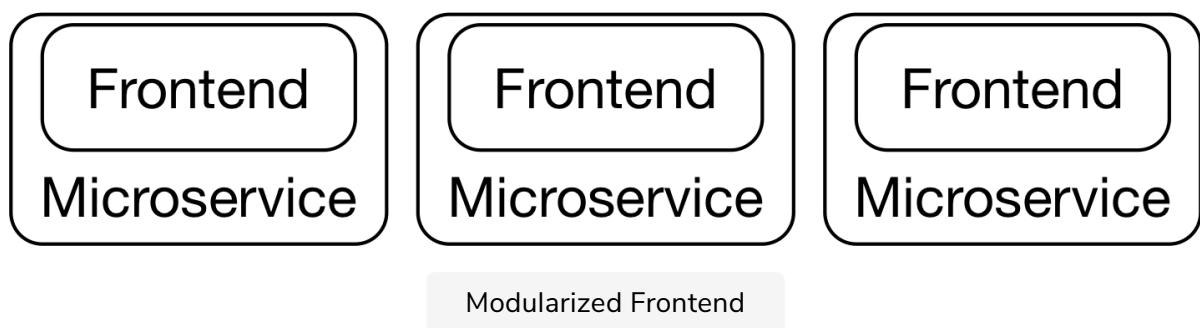
If you do not want to break up the frontend team because it is too big an organizational change, or because the team is working in a different location than the other teams, a monolithic frontend can be the best approach.

Migration

Finally, the *migration* of an existing system might be particularly easy if the monolithic frontend remains intact.

Modularized frontend

The alternative to a monolithic frontend is a fully modularized one. The drawing below shows a modularized frontend where each microservice has its own frontend. Like the backend, the frontend is part of the separately deployable microservices.



Advantages

Such a modularization of the frontend has many advantages.

- Microservices and self-contained systems can be **independent concerning their domain logic**.
- If the microservices contain a part of the modularized frontend, then a **change in a domain can be implemented by modifying and deploying just one microservice**, even if changes are necessary at the frontend.

• If the UI is a monolith, on the other hand, many changes to the

- If the UI is a monolith, on the other hand, many changes to the domain logic require modifications to the UI monolith so that the UI monolith becomes a main focus of change

Modularized frontend and frontend integration

To combine the separately deployed frontends into a complete system, the **frontends must be integrated**.

Modularization is intended to decouple development. Nevertheless, an integrated system must be created. In other words, a frontend modularized into different microservices is only possible if an approach for frontend integration has been chosen.

For this, different technical approaches are possible, which are the focus of this and the following chapters.

QUIZ

1

What is the argument against a monolithic frontend for multiple microservices?

COMPLETED 0%

1 of 4



In the next lesson, we'll discuss a few options for frontend integration.

Options

In this lesson, we'll discuss various options for frontend integration.

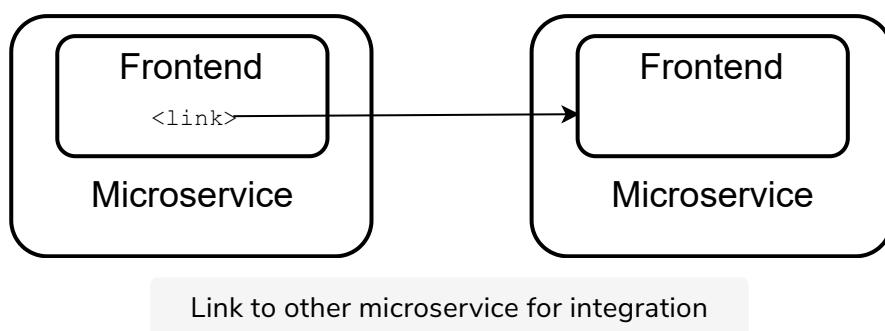
WE'LL COVER THE FOLLOWING ^

- Links
- Redirects
- Transclusions

There are different options for frontend integration.

Links

The easiest options are links. **One frontend displays a link that another frontend handles.** The World Wide Web (WWW) is based on precisely this mechanism, a system creating a link to another system.



Redirects

Redirects represent another option.

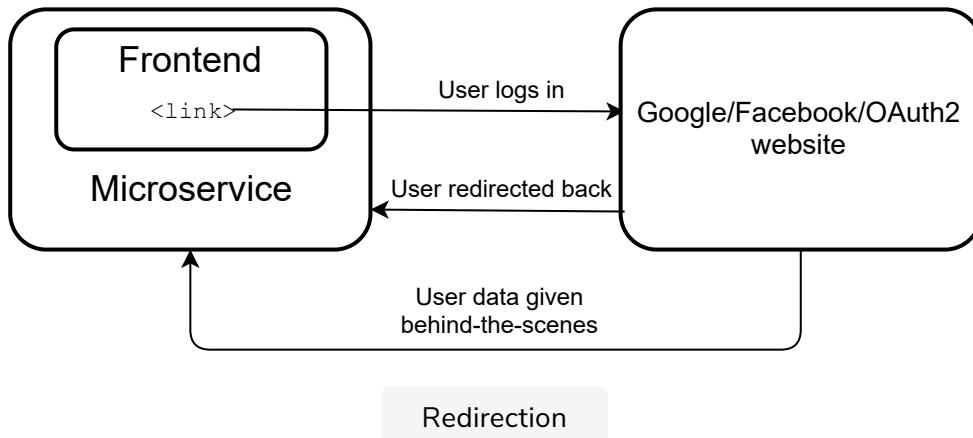
For example, **OAuth2** uses this approach:

- A website provides a link to an OAuth2 provider such as Facebook or Google.
- The user enters their password and confirms that the website is allowed to access their information.

to access certain information.

- The user is then redirected back to the original website by another redirect. Behind the scenes, the website receives the user's data.

Redirects can combine frontend integration with data transfer in the background.



Transclusions

Finally, there are various kinds of *transclusion*. This involves combining the content of a website with the content of another website.

This can be done either on the server or on the client. [Chapter 4](#) shows an example where transclusion is implemented on the client with JavaScript. [Chapter 5](#), on the other hand, shows transclusion on the server-side with ESI (Edge Side Includes).

The blog article at <https://www.innoq.com/en/blog/transclusion/> gives an overview of further possibilities.

These options can, of course, be **combined**, however, this leads to a high level of technical complexity. Therefore, you should first try to get by with just links because they have very low complexity, and only add more options if needed.

QUIZ

1

What are the differences between redirects and links? (Need author's opinion here)

COMPLETED 0%

1 of 2



In the next lesson, we'll discuss resource-oriented client architecture.

Resource-oriented Client Architecture (ROCA)

In this lesson, we'll learn all about ROCA.

WE'LL COVER THE FOLLOWING



- ROCA principles
 - Adherence to REST principles
 - Accessible information
 - Logic on server
 - Authentication in HTTP request
 - Cookies
 - No server-side sessions
 - Browser controls
 - No layout information in HTML
 - JavaScript only for enhancement
 - No redundant logic

Modularization and integration in the frontend have an impact on the architecture and the technologies in the frontend. SPAs (single page apps) are not well suited for integration in the frontend. Therefore, the question arises as to **which frontend architecture is better suited for integration**.

ROCA ([Resource-oriented Client Architecture](#)) is an approach for implementing web applications. It focuses on established technologies such as HTML and leads to an architecture that comprises many benefits for frontend modularization and integration.

ROCA principles

ROCA has a few principles. Let's discuss each.

ROCA has a few principles. Let's discuss each.

Adherence to REST principles

The **server adheres to the REST principles**:

1. All resources have an unambiguous URL.
2. Links to web pages can be sent by e-mail and then accessed from any browser if the necessary authorizations are given.
3. HTTP methods are used correctly. For example, GETs do not change data.
4. The server is stateless.

Accessible information

Resources identified by URLs can have other representations besides HTML, such as JSON or XML. This means that the data is not only available to people, but also to applications.

Logic on server

All logic is on the server. Accordingly, JavaScript on the client-side only serves to optimize the user interface. Not just browsers but also other clients should be able to access the system.

Logic on the server is desirable for security reasons because the client could be manipulated. Changing the logic is also easier because it is implemented in one location making it unnecessary to update a large number of clients.

Authentication in HTTP request

The **authentication information is included in the HTTP request**. HTTP basic, digest, client certificates, or cookies can be used for this purpose. In this way, authentication and authorization can take place solely on the basis of information contained in the HTTP request. Therefore, no server-side session is required for the authentication information.

Cookies

Cookies may be used only for:

1. Authentication
2. Tracking
3. As protection against cross-site request forgery

Therefore, they may not contain business information.

No server-side sessions

There must not be any server-side session. The use of sessions contradicts the ideas of HTTP because the communication is no longer stateless.

This condition makes it difficult to implement failover and load balancing. The only exception is in data that is required for authentication purposes in addition to the authentication information in the HTTP request.

Browser controls

The browser controls such as the back, forward, or refresh button should work. This should be the norm, but many web applications with JavaScript logic have difficulties with it or have to do a lot of work in order to ensure the functioning of these buttons.

No layout information in HTML

The **HTML may not contain layout information** and should be accessible via a screen reader. The layout is defined by CSS so that layout and content are separate.

JavaScript only for enhancement

JavaScript can be used only in the form of *progressive enhancement*. The application can even be used without JavaScript – just not as easily and comfortably. The goal is not to avoid JavaScript completely, but to rely on the fundamental architecture and technologies of the web – HTTP, HTML, and CSS.

No redundant logic

Logic must not be implemented redundantly on client and server because the business logic is implemented on the server and it shouldn't be implemented again on the client.

In the end, **ROCA applications are perfectly normal web applications**. They use web principles as they were originally intended.

1

Which of the following best describes ROCA?

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss some benefits that this architecture provides.

Benefits of ROCA

In this lesson, we'll discuss the benefits of ROCA.

WE'LL COVER THE FOLLOWING ^

- Clean architecture
- Web features can be harnessed
- Low bandwidth consumption
- High speed
- Resilience
- JavaScript not necessary
- ROCA versus SPAs
- Integration options

ROCA has a number of **advantages**:

Clean architecture

The applications have a *clean architecture*. The logic is on the server. Changes to the logic can easily be rolled out by a new server version.

Web features can be harnessed

All *features of the web* can also be used.

- URLs can be sent to other people (for example, by email) because they uniquely identify resources.
- HTTP caches can be used if, say HTTP GETs are not allowed to change data.
- Optimizations in the browsers are exploited. Browsers implement many tricks to show users the first parts of a website and to allow interactions

as quickly as possible.

Low bandwidth consumption

Applications get by with *little bandwidth* as a result of usually having to transfer no more than HTML – and only for the web pages that are actually visited.

In an SPA, on the other hand, often the entire application must be transferred before any interaction is possible at all.

Modern SPA technologies optimize this by loading separate modules rather than all the code. However, initializing the application and making it react to user interaction still takes some time. This is easier with a simple web application.

Modern browsers are optimized to make user interaction with simple web applications as fast and responsive as possible.

High speed

The solution is *fast*, especially for mobile devices; the speed of JavaScript implementations often leaves much to be desired. ROCA applications require a minimum of JavaScript.

Resilience

Finally, an *error in JavaScript* or in the transfer of the JavaScript code due to a problem on the network only leads to hard-to-use, though still available applications. If logic were implemented in JavaScript, this would not be the case and the application would have less resilience.

JavaScript not necessary

Users with *JavaScript switched off* can still use the application. Nowadays, these users are practically non-existent so this advantage is irrelevant.

ROCA versus SPAs

ROCA is an alternative to SPAs (single page apps) where the browser is used only to execute JavaScript code.

- ROCA **takes advantage of the browser's optimizations** for HTML and HTTP, and also of the other advantages of a real web application.
- Of course, a ROCA system can also use JavaScript as discussed previously. The goal of ROCA is not to avoid JavaScript completely, but to **limit it** to areas where it makes sense or where there are no alternatives.
- **SPAs might be a better fit for complex UI** – for example, games or map applications on the web.
- However, for other types of applications, such as e-commerce applications, SPAs not only provide unneeded flexibility but make it harder to achieve some features easily, like search engine optimizations (SEO). For a lot of applications, SPAs are actually not that great a fit.
- SPAs seduce developers to implement **more logic on the client**. Although this might make the system more responsive, it can also lead to redundant implementation on the client and server that is harder to maintain.
- Also, it is hard to export the logic in the SPA as a REST service so that other clients can use it.

[Chapter 4](#) shows an example of a ROCA application that is quite comfortable to use.

Integration options

ROCA generally applies to good UI layer design. It makes sense as a guideline for implementing browser applications, monolithic or modularized.

However, ROCA does facilitate **simplified integration**. HTML is the focus in the frontend while JavaScript serves only to improve usability. Thus, the system can be modularized by providing HTML pages with links to other HTML pages that may originate from other microservices. Or, the HTML pages can even be composed of several parts. Each component can come from a different microservice, making it easy to modularize a ROCA UI.

Because ROCA supports all options for frontend integration, ROCA is a good

basis for frontend integration and for a microservices architecture.

QUIZ

1

How is less bandwidth consumed in a ROCA application?

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss some challenges that frontend integration presents.

Challenges

In this lesson, we'll discuss some challenges pertaining to frontend integration.

WE'LL COVER THE FOLLOWING



- UI infrastructure
 - Uniform look and feel
- Interfaces in frontend integration
- UI changes impact multiple modules

Frontend integration means that the frontend is composed of different systems. This causes some challenges.

UI infrastructure

For UI integration to work, some infrastructure has to be provided. That might be common CSS or JavaScript code, but it can also include server infrastructure for server-side transclusion.

There could be a need for **UI parts that do not belong to any specific microservices** – for example, the home page or a navigation bar. These have to be developed and maintained.

It is important to **not put too much into the UI infrastructure**. If the microservice relies too much on the generic UI infrastructure, it will depend heavily on it and that contradicts the goal of independent development.

UI integration leads to a certain degree of dependencies on the code level, which **leads to tight coupling**; therefore, the dependencies should be limited.

For example, it makes little sense to provide all the styling, UI frameworks, and UI code for all microservices in one single component.

This would be a problem in particular for migration to new technology stacks.

The stack is the same for all microservices, and it is, therefore, hard to migrate stepwise to a new stack.

Uniform look and feel

For example, it is not easy to achieve a uniform look, feel, and design of the overall application.

A uniform look and feel is usually associated with the sharing of artifacts. Multiple microservices must share the CSS, fonts, or JavaScript code required to implement the design.

Interfaces in frontend integration

Transclusion generates a subtle type of interface definition. Normally, an interface is defined by data types and operations. This is obviously not the case for frontend integration. Still, there is a kind of interface definition.

For example, HTML code has to integrate itself into another page if it is displayed in another frontend. To do this, it may be necessary to have common CSS classes, and so all the frontends must possess the same CSS selectors. If JavaScript is used in the displayed HTML, the common JavaScript code and the JavaScript libraries used must be available in the other web pages.

Overall, these requirements form a kind of interface definition that ensures that HTML can actually be displayed. When only links or redirects are used, this challenge does not exist; only the URL must be known. The linked page can use completely different CSS and JavaScript.

Transclusion therefore couples the systems more strongly than links do.

UI changes impact multiple modules

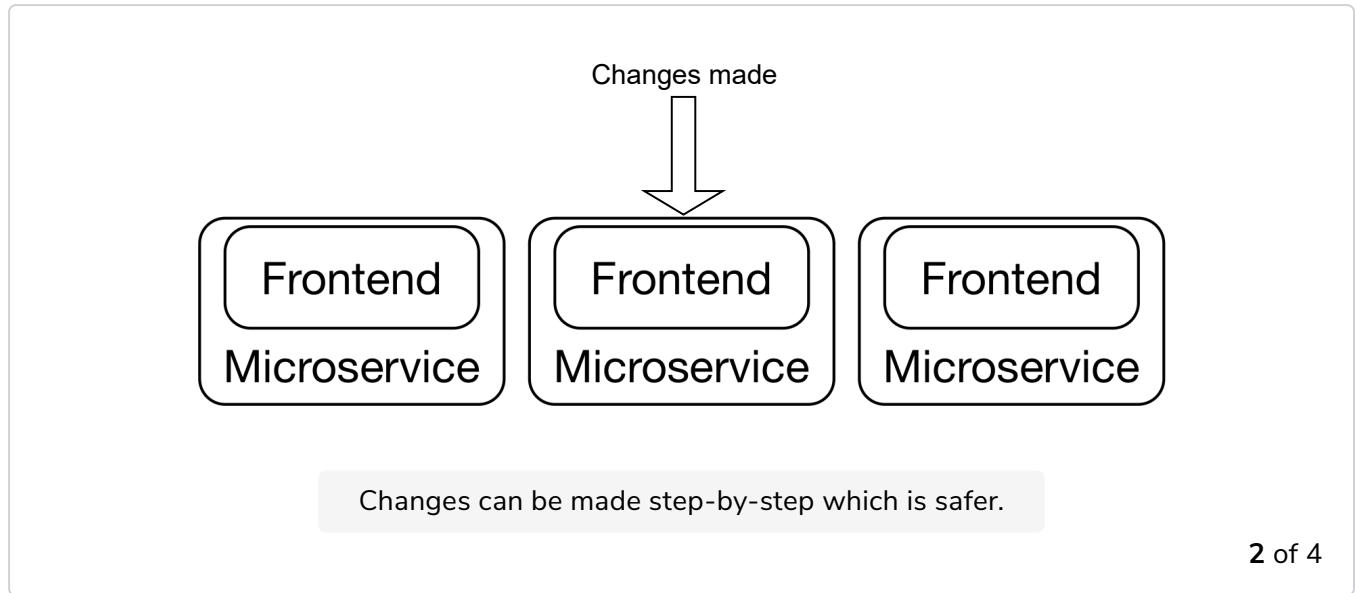
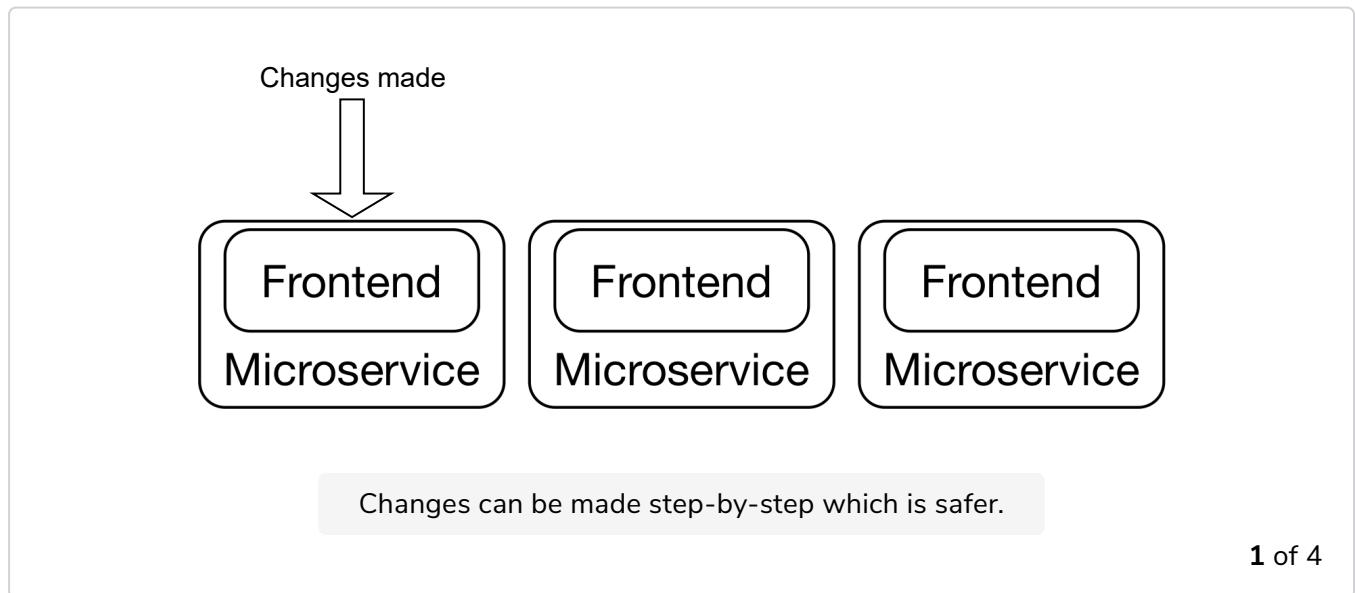
If the changes to a system typically only require changes to the UI, this can be more complicated with frontend modularization.

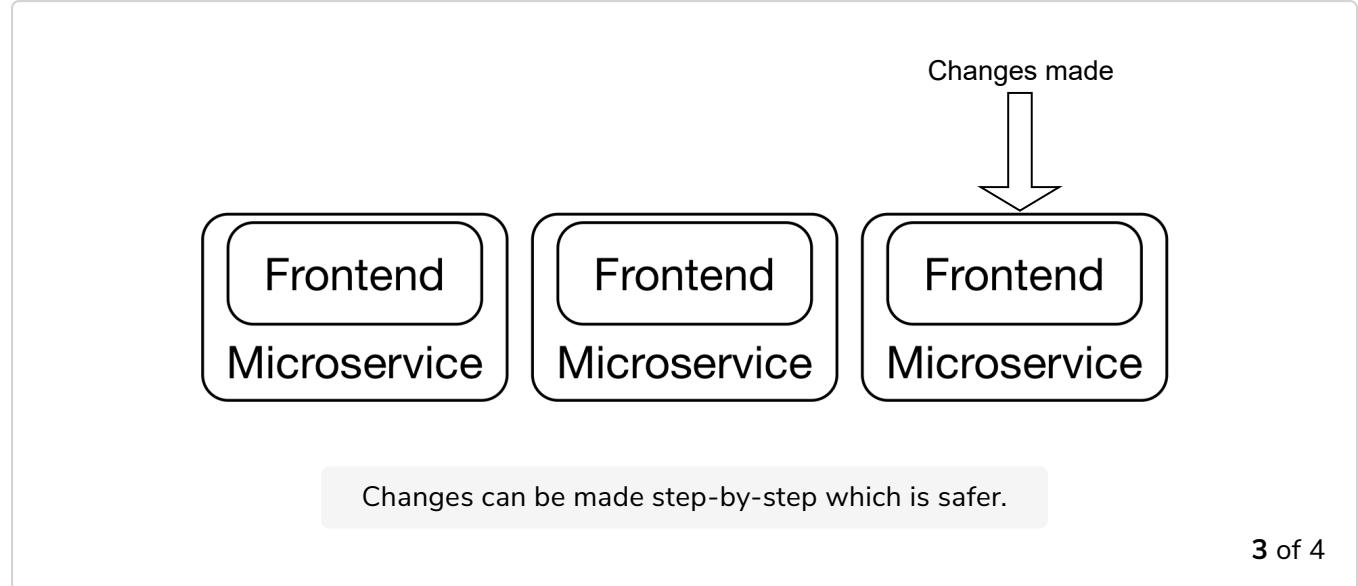
The code for the UI is distributed across the different frontends, so that a change means that all frontends have to be modified. Therefore, if the UI is constantly redesigned or if the CSS is constantly being changed, then frontend

constantly redesigned or if the CSS is constantly being changed, then frontend modularization can increase the effort.

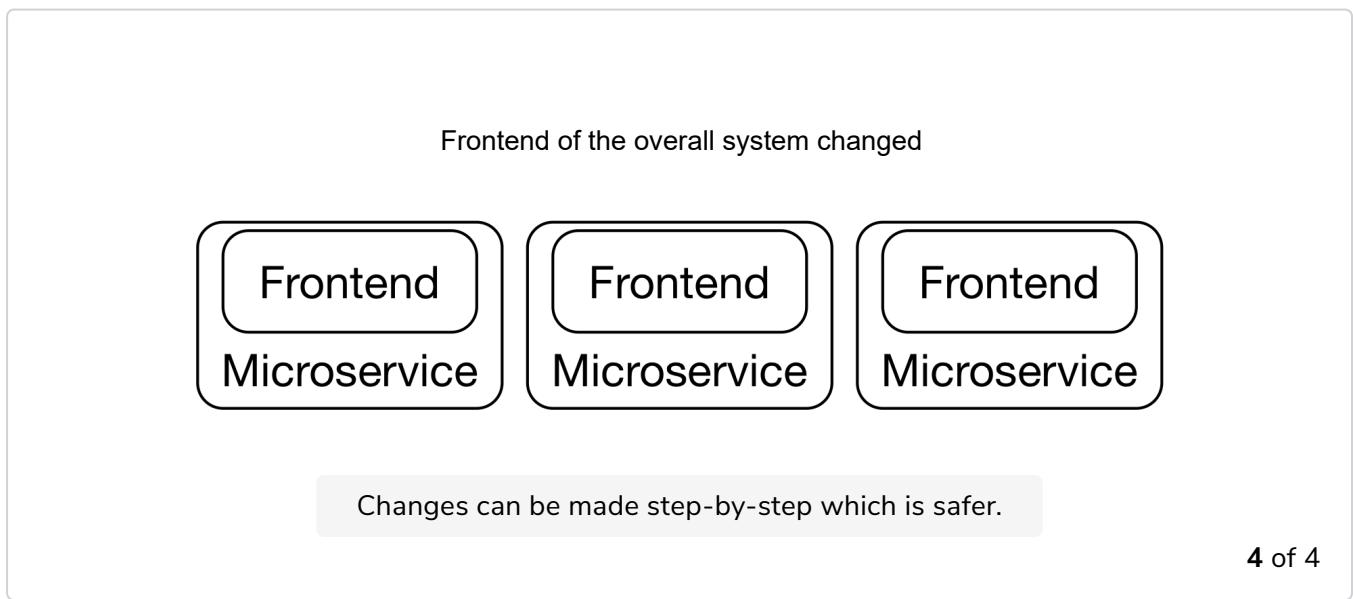
However, frontend modularization has an advantage, even in the case of such changes. The changes can be made step by step by changing only one frontend at a time, **thus minimizing the associated risk**.

In addition, changes that take place only in the UI should be far less frequent than domain-based changes that affect all layers. So, while it is harder to **change all of the UI**, those kinds of changes **should not be happening too often**. Making UI-based harder and making more frequent domain-based changes easier at the same time seems like a good trade-off.





3 of 4



4 of 4



Q U I Z

Q

Suppose the management of a website have asked the developers to implement two changes:

1. Migrate to a microservices system with modularized UI.
2. A complete redesign of the UI.

Which change should the developers make first if they want to save effort?

COMPLETED 0%

1 of 1



In the next lesson, we'll discuss the benefits of frontend integration.

Benefits

In this lesson, we'll discuss the benefits of frontend integration.

WE'LL COVER THE FOLLOWING ^

- Loose coupling
- Logic and UI in one microservice
- Free choice of frontend technologies

Frontend integration offers a number of benefits which make the approach attractive.

Loose coupling

Integration in the frontend creates loose coupling. For example, if links are used for integration, only the URL has to be known for integration to occur. What is behind the URL and how the information is displayed doesn't matter and can be changed without impacting other frontends. So, a change can be limited to one frontend, even if the page looks completely different.

Logic and UI in one microservice

This is advantageous from the architecture's point of view. All logic for a certain functionality is implemented in a single microservice. For example, a microservice can be responsible for maintaining and displaying a to-do list, even if the list is displayed as integrated in the UI of another microservice. If you want to display additional information in the to-do list, such as a priority, you can implement the logic, persistence, and UI by changing only one microservice, even if another microservice integrates the rendered to-do list.

Free choice of frontend technologies

Another plus for frontend integration are the frontend technologies. Frontend

Another plus for frontend integration are the frontend technologies. Frontend technologies especially are subject to many innovations; there are constantly new JavaScript frameworks and new ways to create beautiful and easy-to-use interfaces being made.

An important advantage of microservices is the freedom of technology. Each microservice can choose its own technologies. If technology freedom should also apply to the frontend, then **each microservice must have its own frontend that can use its own technology**.

For this, **the frontends must be integrated accordingly**. In particular, care must be taken to ensure that integration does not restrict the use of frontend technologies.

For example, if the integration forces a certain JavaScript library, this can limit the technology selection as it is not possible to use another version of this library in parallel.

For example, the client-side integration in [chapter 4](#) requires that each frontend uses a certain jQuery version or provides its own custom code.

QUIZ

1

Frontend integration **always** leads to loose coupling _____. (need author's opinion here)

COMPLETED 0%

1 of 2



In the next lesson, we'll discuss some variations to the approaches we've already discussed.

Variations

In this lesson, we'll touch on some variations of the approaches discussed in this chapter.

WE'LL COVER THE FOLLOWING ^

- Self-contained systems
- Links & transclusion
- Edge side includes

Self-contained systems

Self-contained systems (SCS) (see [chapter 2](#)) particularly focus on frontend integration.

Links & transclusion

The next chapters discuss the different variations for frontend integration. [Chapter 8](#) shows how links and transclusion work on the client side.

Edge side includes

[Chapter 5](#) demonstrates the integration on the server-side with ESI (Edge Side Includes).

Typically, a microservices system also uses synchronous communication ([chapter 9](#)) or asynchronous communication ([chapter 6](#)) in addition to frontend integration. The use of several integration approaches is common and easily possible.

Ultimately, the browser only needs to access the various backends via HTTP. This means that **frontend integration puts few constraints on the technologies used**.

The next lesson will be a quick conclusion to this chapter.

Chapter Conclusion

Here's a quick conclusion of the chapter.

WE'LL COVER THE FOLLOWING ^

- Summary

Summary

In general, **integration at the frontend level should always be considered when possible**. Too often, microservices are implemented only as an approach to the backend, although by focusing on the frontend, the coupling can be looser, and the system can be simpler and more flexible. It can also be ensured that the entire logic, including the logic for the UI, is actually implemented in the microservice.

Frontend integration **allows loose coupling of microservices**. Already the use of links and of some JavaScript code can be enough to integrate the frontends of different microservices. It is therefore important not to immediately define a complex technology stack, but to first find out what can be achieved by simple means.

This leads to another advantage: **technical complexity** of the solution is **not particularly high**; only web applications are used. There are more of them than usual, but there are no new technologies.

But even when using server-side integration, the integration in the frontend is still not particularly complex and still leads to a loose coupling.

That's it for this chapter. In the next chapter, we'll discuss frontend integration with links and JavaScript.

Introduction

In this lesson, we'll look at a quick overview of what this chapter holds.

WE'LL COVER THE FOLLOWING ^

- Chapter walkthrough

Chapter walkthrough

This chapter provides an example of a frontend integration with links and JavaScript.

The reader learns:

- The scenarios for which a simple approach based on links and JavaScript for frontend integration is feasible.
- An example, which is not a self-contained system (SCS), which shows that frontend integration does not only make sense in the context of SCS but also in other scenarios.
- The example shows how an integration with links and JavaScript can be implemented.
- Mechanisms with which a uniform look and feel can be implemented with such an integration.

QUIZ

Q

Frontend integration only makes sense in the context of self-contained systems.

COMPLETED 0%

1 of 1



Great, let's officially start off with an overview of the app that will be the focus of this chapter and how it was integrated with links in the next lesson!

Overview

In this lesson, we'll look at a quick overview of an app and how its components were integrated.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Search
- Postbox
- Structure of the application
 - Main application
 - Assets
 - Damage
 - Letter
 - Postbox
 - Backend simulator

Introduction

The example in this chapter is a classic assurance application designed to help office workers interact with customers.

It was created as a **prototype for a real insurance company**. The example shows how a classic application can be implemented as a web application with frontend integration.

The example shows **the division of a system into several web applications** and the integration of these applications. ROCA is used as the basis for this. This is an approach for implementing web applications (see the [last chapter](#)), which has a number of fundamental advantages, especially for frontend integration.

The example was created as a prototype **showing how a web application can**

The example was created as a prototype showing how a web application can be implemented with ROCA and what advantages it offers.

The INNOQ employees Lucas Dohmen and Marc Jansing implemented the example.

Let's look at each web application the system is divided into.

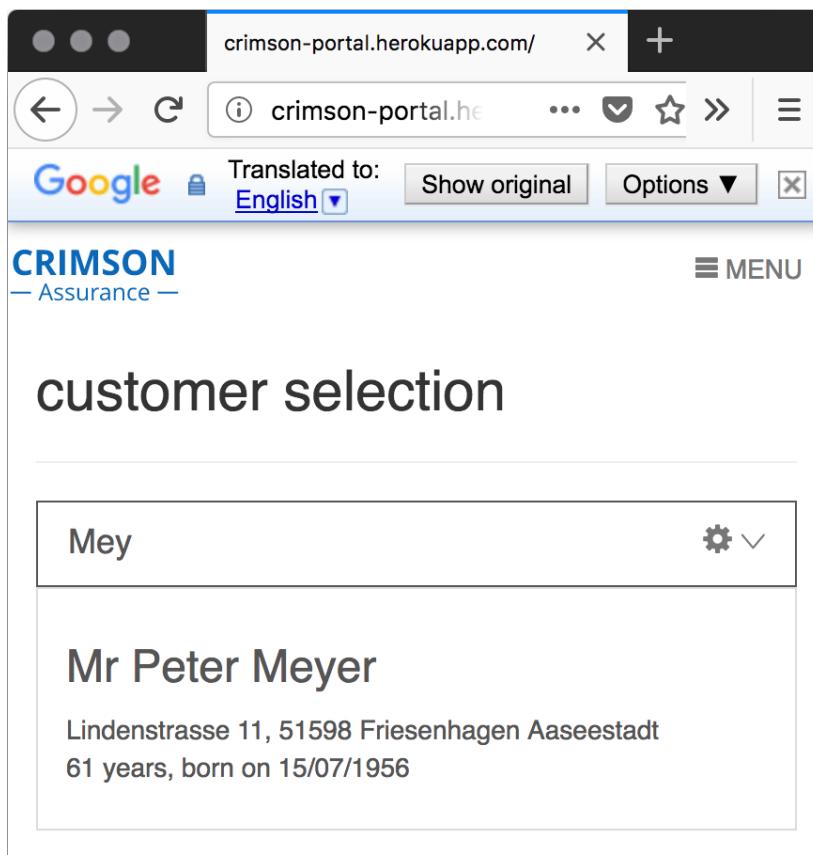
Search

The insurance application is available at <https://crimson-portal.herokuapp.com/>. It can also be run on a local computer as a Docker container.

The application is in German. However, nowadays browsers can translate web pages into other languages. The screenshots were taken with the Firefox browser and Google Translator's translation from German to English.

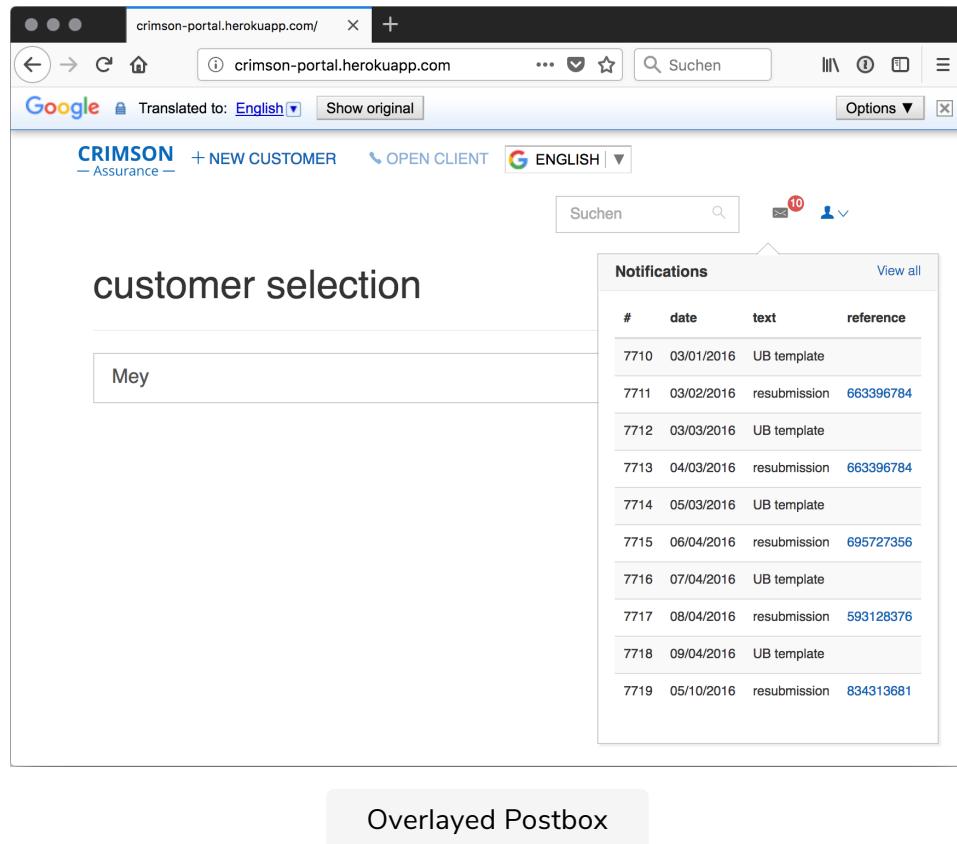
An input line appears on the main page to search for customers. Have a look at the screenshot below.

While the user enters the customer's name, matching names are suggested. For this, the frontend uses jQuery.



Postbox

Another functionality is the *postbox*. With a click on the postbox icon on the main page, the user gets an overview of the current news. The overview is displayed in the current main page with JavaScript. Have a look at the figure below.



The entire application **uses a uniform frontend**.

However, if you look at the address line, you will notice that several web applications are used.

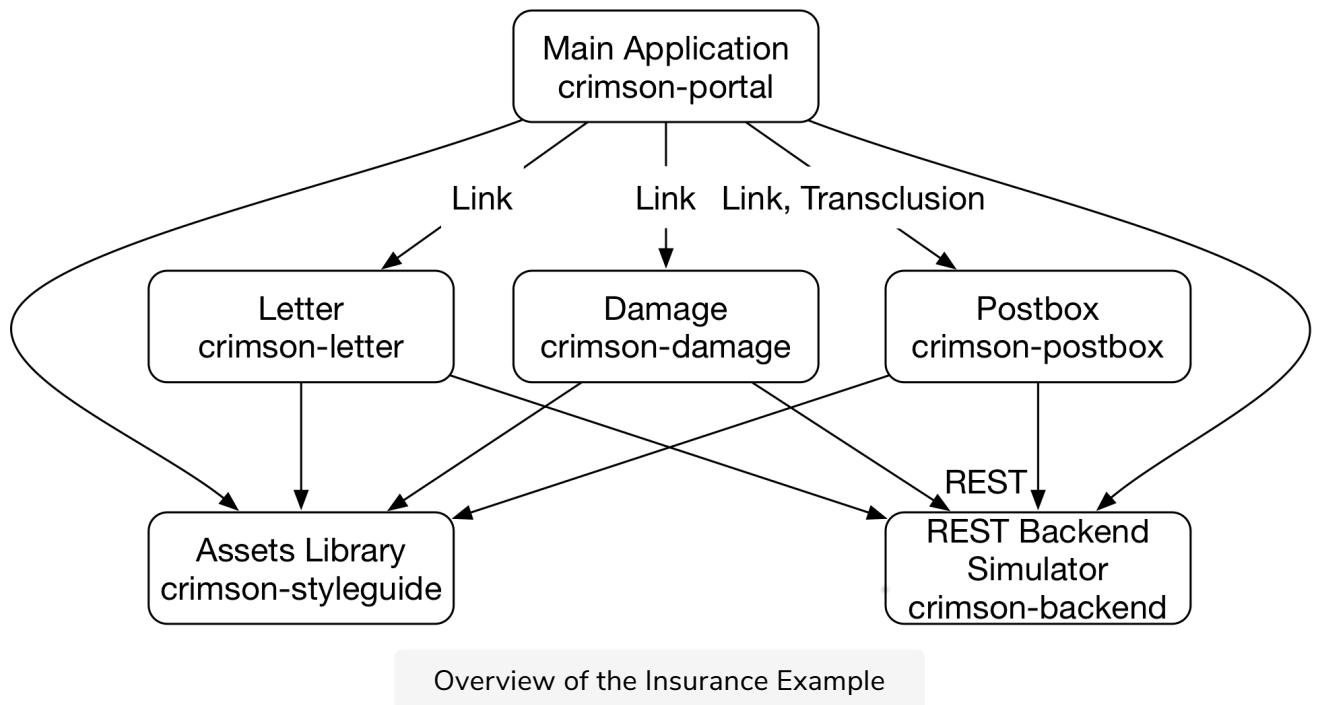
There is **one web application each**.

- for the main application (<https://crimson-portal.herokuapp.com/>)
- for reporting damages (**damage** application) (<https://crimson-damage.herokuapp.com/>)
- for writing letters (**letter** application) (<https://crimson-letter.herokuapp.com/>)
- for the postbox (**postbox** application) (<https://crimson-postbox.herokuapp.com/>).

Nevertheless, the frontend has the same look and feel for all these

applications.

Structure of the application



The drawing above shows how the different applications are integrated.

Main application

The *main* application is used to search for customers and to display the basic data of a customer. The code is available at <https://github.com/ewolff/crimson-portal>. There you will also find instructions on how to compile and start the application. The application is written in Node.js.

Assets

The *assets* at <https://github.com/ewolff/crimson-styleguide> contain artifacts used by all applications to achieve a consistent look and feel. The other projects refer to this project in the `package.json`. This allows the npm build to use the artifacts from this project. npm is a build tool specialized in JavaScript. The asset project includes CSS, fonts, images, and JavaScript code. Before the assets are used in the other projects, the build optimizes them in the asset project. For example, the JavaScript code is minified.

Damage

The *damage* application for reporting a damage is also written in Node.js. The code can be found at <https://github.com/ewolff/crimson-damage>.

Letter

The *letter* application is written in Node.js as well. The code is available at <https://github.com/ewolff/crimson-letter>.

Postbox

The code for the *postbox* can be accessed at <https://github.com/ewolff/crimson-postbox>. The postbox is implemented with Java and Spring Boot. To be able to use the shared asset project, the build is divided into two parts. The Maven build compiles the Java code, whereas npm is responsible for integrating the assets. npm then copies the assets into the Maven build.

Backend simulator

Finally, the *backend simulator* can be found at <https://github.com/ewolff/crimson-backend>. It receives REST calls and returns data regarding customers, contracts, and so on. This simulator is also written in Node.js.

In the next lesson, we'll discuss how this app was integrated.

Integration in Assurance App

In this lesson, we'll discuss the details of how the application we saw in the last chapter was integrated.

WE'LL COVER THE FOLLOWING ^

- Why monolithic backend?
- Integration with redirects
- Integration with links

Why monolithic backend?

This application uses a monolithic backend and frontend microservices because the **microservices** lack logic and they **are not self-contained systems**.

However, this architecture can still make sense. The frontend, at least, consists of microservices, and so **independent development of the microservices is possible**.

Also, it is possible to **use different technologies in each frontend microservice**. With a large number of available UI frameworks and the high speed of innovation, that is a clear advantage.

Also, it is probably not possible to migrate the backend into microservices. Another team might be responsible for it. Therefore, if the scope of the project is just to improve the frontend, there is no way to change the architecture of the backend.

While SCSs are generally a great idea, this example shows one exception to the rule.

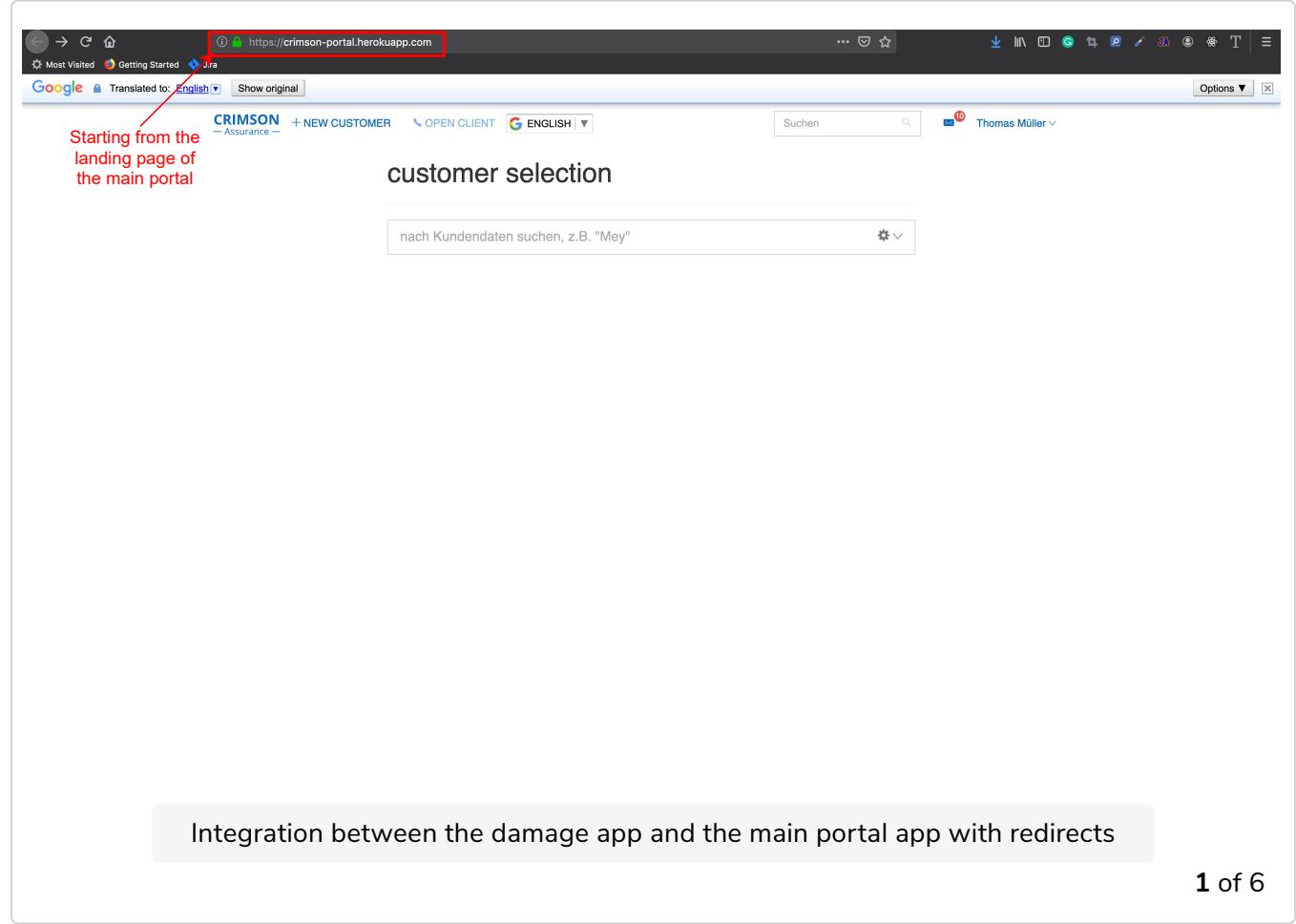
Integration with redirects

If a user in the *damage* application enters a claim for a car, the user is sent

If a user in the *damage* application enters a claim for a car, the user is sent back to the overview of the car displayed on the portal. The transition from the *damage* application to the portal is implemented with a redirect. The *damage* application sends an HTTP redirect after reporting the claim, leading to the web page of the *main* application.

A redirect is a very simple integration. The *damage* application needs to know only the URL for the redirect. The portal could even pass this URL to the *damage* application to further decouple the two applications.

Such an integration is also used if a user registers with their Google account on a web page. After the user agrees to register on the Google web page, the Google page sends a redirect back to the initial web page.



customer selection

mey

Mr Peter Meyer
Lindenstrasse 11, 51598 Friesenhausen Aaseestadt
63 years born, on 07/15/1956

Search for customer

Integration between the damage app and the main portal app with redirects

2 of 6

Lindenstrasse 11
51598 Friesenhausen
0169 1234567

9 years born, on 07/15/1956
Married, 2
kindergarten teacher

FUND FINANCES LIFE SUFFER ACCIDENT LIABILITY PROPERTY LAW MOTOR

consultation

- PHONE 4 years ago Maik Thomson
- New contract documents for your accommodation
- PHONE 4 years ago Jutta Jansen
- documents
- E-MAIL 4 years ago Jutta Jansen
- Updating their insurance
- E-MAIL 4 years ago Maik Thomson
- consultation
- E-MAIL 4 years ago Jutta Jansen
- Updating their insurance
- SEE MORE

Duration

Click on one of their contracts

#	branch	contribution	action
454069088	MOTOR VEHICLES	€ 37.45	Open Open in new tab
106303668	MOTOR VEHICLES	27.42 €	Open Open in new tab

Integration between the damage app and the main portal app with redirects

3 of 6

Most Visited Getting Started Options ▾

Note that we are still in the main portal Lindenstrasse 11
51598 Friesenhausen
0169 1234567 59 years born, on 07/15/1956
Married, 2 kindergarten teacher

FUND FINANCES LIFE SUFFER ACCIDENT LIABILITY PROPERTY LAW MOTOR

Year: 20.05.2015
Acquisition UN / FH:
Chassis NR.: dj3rij35j42

season: [] season to: []
License plate Type: schwarzes Kennzeichen Interchangeable license plates: Nein
Power kW: [] Horsepower: 340
replacement engine: Nein

[Write a letter](#) [Report damage](#)

Clicking on this link will lead to the damage application

car	RegioKl.	TypKl.	payable amount	Insurance tax included
	liability	Comprehensive		
	Teilkasko			

Integration between the damage app and the main portal app with redirects

4 of 6

Most Visited Getting Started Options ▾

In the damage application now CRIMSON Assurance + NEW CUSTOMER OPEN CLIENT ENGLISH Thomas Müller

Damage report: motor

insurance number: 454069088
Street: Lindenstrasse 11
Postal: 51598
City: Friesenhausen
district: Aaseestadt
damage: []

Clicking this button will submit a damage report and redirect back to the main portal

[Report damage](#)

Integration between the damage app and the main portal app with redirects

5 of 6

The screenshot shows the CRIMSON Assurance main portal application. At the top, there's a navigation bar with links like 'Most Visited', 'Getting Started', 'Google Translate' (set to English), and 'Show original'. Below the navigation is the CRIMSON Assurance logo and a search bar. The main content area displays a customer profile for 'Mr. Peter Meyer' with details such as address (Lindenstrasse 11, 51598 Friesenhausen, 0169 1234567), birth date (07/15/1956), and marital status (Married, 2 children). Below the profile are several icons representing different service categories: FUND, FINANCES, LIFE, SUFFER, ACCIDENT, LIABILITY, PROPERTY, LAW, and MOTOR. A green banner at the bottom states 'The damage has been successfully reported'. On the right side, there are dropdown menus for 'Duration' (00 Thomas Gosen) and 'graduation' (00 Thomas Gosen). The URL in the browser bar is https://crimson-portal.herokuapp.com/partners/4711/contracts/454069088?success=true.

Integration between the damage app and the main portal app with redirects

6 of 6

Integration with links

The integration of the applications is mainly done via links such as <https://crimson-letter.herokuapp.com/template?contractId=996315077&partnerId=4711> to display a web page for writing a letter.

They contain all the essential information necessary for the web page to write the letter: **the contract number and the partner number**.

In this way, the *letter* application can retrieve the data from the backend simulator and render it in the web page. The coupling between the main application and the letter application is very loose; it's just a link with two parameters. The main application does not need to know what is behind the link. As a result, the *letter* application can change its UI at any time without any impact on the *portal* application.

However, all applications use a common database from the backend simulator

and are consequently tightly coupled, because a change to the data would

affect the backend and the respective microservice

A screenshot of a web browser displaying the CRIMSON Assurance main portal at <https://crimson-portal.herokuapp.com>. The page title is "customer selection". At the top left, there are links for "Getting Started" and "Jira". A red arrow points from the text "Starting from the landing page of the main portal" to the "Getting Started" link. The search bar contains the placeholder text "nach Kundendaten suchen, z.B. "Mey"".

Integration between the letter app and the main portal app with links
1 of 8

A screenshot of the same web browser displaying the CRIMSON Assurance main portal. The search bar now contains "mey". A red box highlights the search result for "Mr Peter Meyer" with the address "Lindenstrasse 11, 51598 Friesenhausen Aaseestadt" and the note "63 years born, on 07/15/1956". A red arrow points from the text "Search for customer" to the search bar.

Integration between the letter app and the main portal app with links
2 of 8

Google Translated to: English Show original Options ▾

Lindenstrasse 11
51598 Friesenhausen
0169 1234567

59 years born, on 07/15/1956
Married, 2
kindergarten teacher

FUND FINANCES LIFE SUFFER ACCIDENT LIABILITY PROPERTY LAW MOTOR

consultation

PHONE 4 years ago Maik Thomson
New contract documents for your accommodation

PHONE 4 years ago Jutta Jansen
documents

E-MAIL 4 years ago Jutta Jansen
Updating their insurance

E-MAIL 4 years ago Maik Thomson
consultation

E-MAIL 4 years ago Jutta Jansen
Updating their insurance

SEE MORE

Duration

Click on one of their contracts

Contracts (3)	Applications (2)	Offers (5)
# 454069088	branch MOTOR VEHICLES	contribution € 37.45 action Open Open in new tab
106303668	MOTOR VEHICLES	27.42 € Open Open in new tab

Integration between the letter app and the main portal app with links

3 of 8

Note that we are still in the main portal

https://crimson-portal.herokuapp.com/partners/4711/contracts/211970442

Lindenstrasse 11
51598 Friesenhausen
0169 1234567

59 years born, on 07/15/1956
Married, 2
kindergarten teacher

FUND FINANCES LIFE SUFFER ACCIDENT LIABILITY PROPERTY LAW MOTOR

Year 20.05.2015

Acquisition UN / FH

Chassis NR. dj3rij35j42

season

License plate Type schwarzes Kennzeichen

Power kW

replacement engine Nein

season to

interchangeable license plates Nein

Horsepower) 340

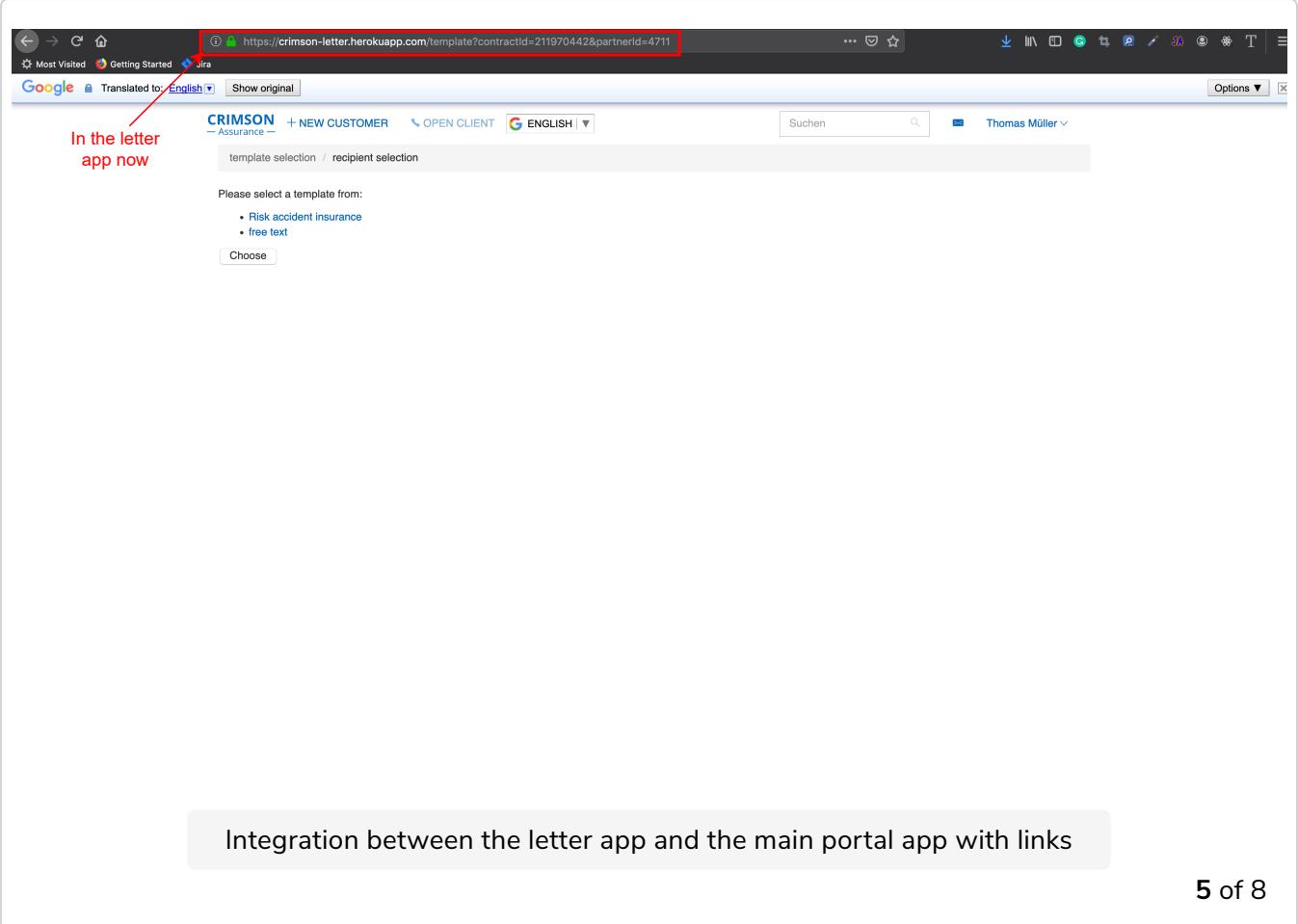
Write a letter Report damage

Clicking on this link will lead to the letter application

car	RegioL. TypL.	payable amount	Insurance tax included
liability			
Comprehensive			
Telikasko			

Integration between the letter app and the main portal app with links

4 of 8



In the letter app now

https://crimson-letter.herokuapp.com/template?contractId=211970442&partnerId=4711

CRIMSON Assurance + NEW CUSTOMER OPEN CLIENT ENGLISH | Suchen Thomas Müller | Options

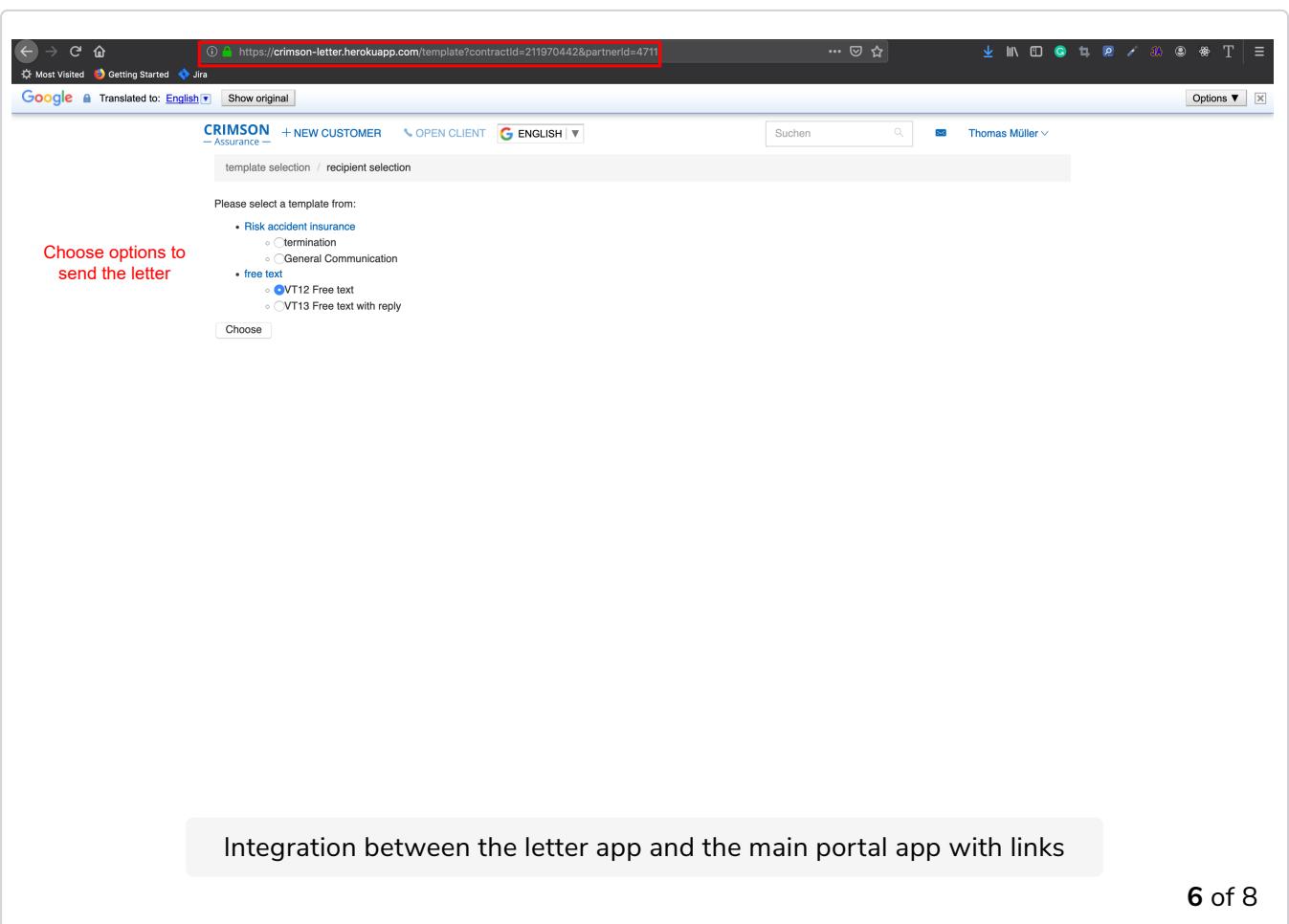
Please select a template from:

- Risk accident insurance
- free text

Choose

Integration between the letter app and the main portal app with links

5 of 8



Choose options to send the letter

https://crimson-letter.herokuapp.com/template?contractId=211970442&partnerId=4711

CRIMSON Assurance + NEW CUSTOMER OPEN CLIENT ENGLISH | Suchen Thomas Müller | Options

Please select a template from:

- Risk accident insurance
 - termination
 - General Communication
- free text
 - VT12 Free text
 - VT13 Free text with reply

Choose

Integration between the letter app and the main portal app with links

6 of 8

Integration between the letter app and the main portal app with links

7 of 8

The screenshot shows the CRIMSON Assurance application interface. At the top, there are navigation links for 'Most Visited', 'Getting Started', and 'Jira'. The main header includes the CRIMSON logo, '+ NEW CUSTOMER', 'OPEN CLIENT', 'ENGLISH' language switch, and a search bar. A user 'Thomas Müller' is logged in. The main content area is titled 'template selection / recipient selection'. It displays a table of recipients:

receiver	role	original	Copy	delivery
Peter Meyer	insured	<input checked="" type="radio"/> original	<input type="checkbox"/> Copy	Printed road
Officer Stephan Hillmann	clerk	<input type="radio"/> original	<input type="checkbox"/> Copy	Printed road

A red box highlights the URL in the browser's address bar: https://crimson-letter.herokuapp.com/recipients?contract=211970442&partner=4711&template=20.

Integration between the letter app and the main portal app with links

8 of 8

The screenshot shows the CRIMSON Assurance application interface. At the top, there are navigation links for 'Most Visited', 'Getting Started', and 'Jira'. The main header includes the CRIMSON logo, '+ NEW CUSTOMER', 'OPEN CLIENT', 'ENGLISH' language switch, and a search bar. A user 'Thomas Müller' is logged in. The main content area is titled 'The letter will be sent.' It lists the template details and recipient information:

- Template ID: 20
- Policy Number: 211970442
- Partner ID: 4711
- letters:
 - Peter Meyer
 - Goal: remote
 - Original: true
 - Copy: false
 - Officer Stephan Hillmann
 - Goal: remote
 - Original: false
 - Copy: false

A red box highlights the URL in the browser's address bar: https://crimson-letter.herokuapp.com/send.

This is a link back to the main portal → [Back to the Contract](#)

In the next lesson, we'll look at other ways that apps can be integrated on the client-side integration.

Other Integration Methods

In this lesson, we'll continue looking at integration methods

WE'LL COVER THE FOLLOWING

- Integration with JavaScript
- Presentation logic in the postbox
- Assets with integrated HTML
- Resilience
- With and without JavaScript



Integration with JavaScript

In the example, the integration of the frontend is practically always done via links. However, the *postbox* displays an overview of the current messages in the *main* application.

#	Datum	Text	Bezug
7710	01.03.2016	UB-Vorlage	
7711	02.03.2016	Wiedervorlage	519885820
7712	03.03.2016	UB-Vorlage	
7713	04.03.2016	Wiedervorlage	373652662
7714	05.03.2016	UB-Vorlage	
7715	06.04.2016	Wiedervorlage	436886298
7716	07.04.2016	UB-Vorlage	
7717	08.04.2016	Wiedervorlage	366455126
7718	09.04.2016	UB-Vorlage	
7719	10.05.2016	Wiedervorlage	524857809

The postbox application uses transclusion for integration with the main application

For this, a simple link is not enough. Still, this integration is also a link. A look into the HTML code shows:

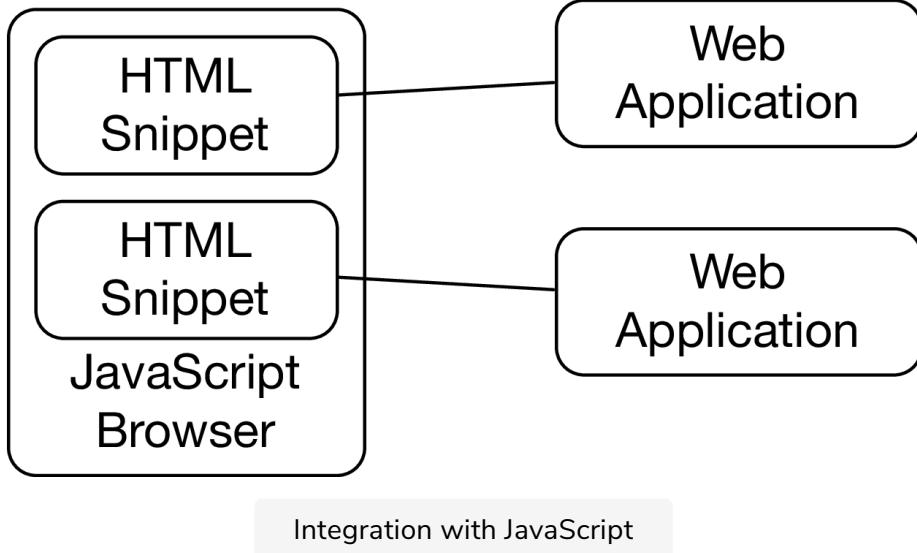
```
<a href="https://crimson-postbox.herokuapp.com/m50000/messages"
    class="preview" data-preview="enabled"
    data-preview-title="Notifications"
    data-preview-selector="table.messages-overview"
    data-preview-error-msg="Postbox unreachable!"
    data-preview-count="tbody>tr" data-preview-window>
```

Actually, when running the application, the HTML code contains German expressions. They can be translated by Google Translate. However, the HTML code remains unchanged and still includes German expressions. For convenience, the English expressions are shown in the HTML code in the listing here.

The link contains **additional attributes**. They ensure that the information of the *postbox* is displayed in the current web page and define how exactly this happens. This information is interpreted by less than 60 lines of JavaScript code from the asset project (see <https://github.com/ewolff/crimson-styleguide/tree/master/components/preview>). The code uses jQuery, so every application in the system now needs to use a compatible version of jQuery when using this code from the asset project.

However, this is not mandatory. Alternatively, any microservice that integrates the *postbox* with such a link can write its own code to interpret the link. After all, every microservice writes its own code to read data from JSON that other microservices provide. Therefore, code that integrates other projects' HTML should be fine, too.

This type of integration is called *transclusion* because it includes HTML from more than one microservice in a web page. In this example, the transclusion is implemented with JavaScript code that integrates the different backends (see the drawing below). The JavaScript code runs in the browser, loads HTML fragments of the other web applications, and displays them in the current web page.



Presentation logic in the postbox

With transclusion, *the postbox* retains control over how messages are displayed, even if the messages are shown as previews in another service. This leads to a clean architecture.

The code for displaying the postbox is located in the postbox service, even if the postbox is displayed in another service. This shows how frontend integration can contribute to an elegant solution and architecture.

The approach of dynamically including content from other URLs is not only used for the *postbox*. For each customer, there is also an overview of the offers, applications, claims, and contracts. The links for this are located below the postbox icon and are repeated further down in the inventory.

Just as with the postbox, this information is also referenced with links whose contents display on the web page. In this case, the URLs are in the same microservice but still provide a better modularization. It also shows that the code solves a general technical problem and is reusable beyond the integration between microservices.

Assets with integrated HTML

Transclusion embeds HTML fragments into other web pages. The HTML can require assets such as CSS or JavaScript. There are different approaches for ensuring that these assets are present.

- The HTML can be designed to not require any assets at all. Thus, no assets have to be shared between the microservices.
- The HTML uses only the assets from the shared asset library – in the example, `crimson-styleguide`. No special measures are necessary because the assets are present in all microservices.
- The HTML can also bring along or link to its own assets. However, in this case you have to be careful not to load the assets more than once if several contents are transcluded in one web page.

Till Schulte-Coerne has written a [blog post](#) about this topic.

Resilience

The application achieves a high degree of resilience and reliability through the consistent use of links. If one microservice fails, the other microservices continue to work; they can still display the links.

The JavaScript code contacts the *postbox* to transclude an overview of the messages on the web page. If this doesn't work, the code displays an exclamation point, but the application still works.

Thanks to JavaScript, loading the transcluded HTML is carried out in the background, so that the failure of one system does not affect the transclusion of the other content and high performance is achieved.

With and without JavaScript

The user can also use the applications if JavaScript is disabled. In this case, for example, the automatic completion of customer names does not work, nor does displaying the overview of messages in the *postbox*.

Nevertheless, the application remains usable. The start page is rendered completely as HTML on the server and does not require client-side templates. The *postbox* simply offers a link instead of the displayed overview. If the user clicks on this link, they get to the *postbox*.

1

Which of the following counts as transclusion?

COMPLETED 0%

1 of 3



In the next lesson, we'll look at an app.

Example

In this lesson, we'll look at a practical example of an app to demonstrate what we've learned in this chapter.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Network ports

Introduction

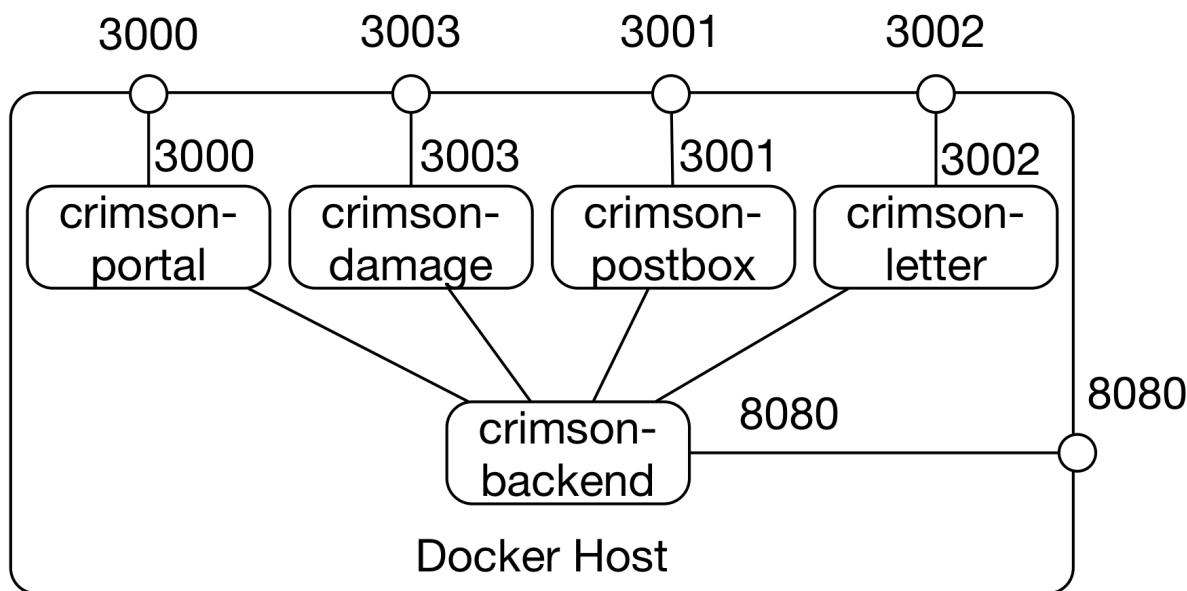
This example is not only provided in the Heroku cloud but also as a coding environment on Educative below.

To start the environment, press `run`. This takes quite a while because all the Docker containers are built, and also all the dependencies are downloaded from the Internet. The website will be available on a link generated below the environment such as: <https://x6jr4kg.educative.run>.

A detailed description of how the example can be started locally is available at <https://github.com/ewolff/crimson-assurance-demo/blob/master/HOW-TO-RUN.md>.

Network ports

The application is available at port 3000 on the Docker host. *Postbox* has port 3001, *letter* port 3002, and *damage* port 3003 (see the drawing below). The frontend services communicate with the backend, which runs in a separate Docker container.



Docker Containers in the Example

Of course, the ports of the Docker containers could be redirected to any other ports of the Docker host. Likewise, all applications in the Docker containers can use the same ports in each Docker container.

However, to save confusion, in this example, the port numbers of the containers are identical to those of the hosts.

You can also try the system directly on the web at [Heroku](#). The links then point to the separate applications for *postbox*, *letter*, and *damage* also deployed at Heroku. Heroku is a PaaS (Platform as a Service, see [chapter 14](#)) available in the public cloud.

In the next lesson, we'll look at a few variations to the approaches we've already discussed.

Variations

In this lesson, we'll look at some variations to the approaches we've discussed already in this chapter.

WE'LL COVER THE FOLLOWING ^

- Asset server
- Simpler JavaScript code
- Integration using other frontend integrations
- Other integrations

Asset server

The example uses a Node project for the shared assets. An alternative option is an asset server that stores the assets. Since the assets are static files loaded via HTTP, an asset server can just be a simple web server.

Over time, the assets will change. For the asset project from the example, a new version of the asset project would have to be created. The new version must be integrated in every microservice. This sounds like unnecessary work, but this way each application can be tested with a new version of the assets.

Therefore, even with an asset server, a new version of the assets should not simply be put into production, but the applications should be adjusted to the new version and also tested with the assets. The version of the assets can be included in the URL path. Thus, version 3.3.7 of Bootstrap might be found under `/css/bootstrap-3.3.7-dist/css/bootstrap.min.css`. A new version would be available under a different path – for example, `/css/bootstrap-4.0.0-dist/css/bootstrap.min.css`.

Simpler JavaScript code

The JavaScript code in the example is quite flexible and can also deal with the failure of a service. A primitive alternative is shown in the [SCS jQuery Project](#).

In essence, it uses the following JavaScript code:

```
$(document).ready(function() {
  $("a.embeddable").each(function(i, link) {
    $("<div />").load(link.href, function(data, status, xhr) {
      $(link).replaceWith(this);
    });
  });
});
```



The code uses jQuery to search for hyperlinks ([...](#)) with the CSS class `embeddable`, and then replaces the link with the content that link refers to.

This demonstrates how simple it is to implement a client integration with JavaScript. At minimum, it is just seven lines of jQuery code.

Of course, it is also possible that each microservice has its own code for transclusion. This seems like a redundant implementation at first, however, it is not uncommon for each microservice to have its own code to parse, e.g., a JSON data structure. So custom code per microservice for transcluding HTML can also be an option.

Integration using other frontend integrations

Of course, client-side integration and links with server-side integration (see [chapter 5](#)) can be combined. Both approaches have different benefits:

- Web pages with server-side integration originate entirely from the server. Thus, the integration makes sense when the web page can be correctly displayed only if all the content is integrated.
- With client-side integration, transclusion is not executed when the other server is not available. The web page is still displayed, just without transclusion. This can be the better option because it improves resilience. However, the webpage would need to be usable without the transcluded content.

However, a server-side integration requires additional infrastructure. For client-side integration, this is not necessary. Therefore, it makes sense to start with client-side integration and to supplement server-side integration when necessary.

Other integrations

Synchronous communication ([chapter 9](#)) or asynchronous communication ([chapter 6](#)) enable the communication of the backend system and therefore can be combined, of course, with client-side frontend integration.

QUIZ

1

In what case does server-side integration make sense?

COMPLETED 0%

1 of 2



In the next lesson, we'll look at some experiments you can conduct on links and client-side integration.

Experiments

In this lesson, we'll look at a few things you can do to experiment with the app.

WE'LL COVER THE FOLLOWING ^

- Try it yourself!
- Add your own microservice!

Here are a few experiments you can try on the app:

- Start the example and disable JavaScript in the browser. Is the example still usable? Specifically, does the *postbox* integration still work?
- Analyze the JavaScript code for transclusion at <https://github.com/ewolff/crimson-styleguide/tree/master/components/preview>. How difficult is it to replace this code with an implementation that uses a different JavaScript library? Try it in the live app below!

Try it yourself!

Note: it might take a while for the app to set up.

Add your own microservice!

Supplement the system with an additional microservice.

- A microservice that generates a note for a meeting with a client can serve as an example.
- Of course, to add the service you can copy and modify one of the existing

`Node.js` or the Spring Boot microservice.

- The microservice has to be accessible by the **portal** microservice. To achieve this, you have to integrate a link to the new microservice into the portal.
- The link can provide the partner ID to the new microservice. This ID identifies the customer and might be useful in figuring out which customer the note belongs to.
- After entering the note, the microservice can trigger a redirect back to the portal.
- For a uniform look and feel, you have to use the assets from the style guide project. The Spring Boot project for the **postbox** shows the integration for Spring/Java and the portal for `Node.js`. Of course, you can also use other technologies for the implementation of the new microservice.
- The microservice can store the data concerning the meeting in a separate database.

Try this in the live app above!

QUIZ

1

What is the purpose of the style guide project?

You may choose more than one answer.

COMPLETED 0%

1 of 2



In the next lesson, we'll look at a conclusion to this chapter.

Chapter Conclusion

That's it for this chapter! Here's a quick summary.

WE'LL COVER THE FOLLOWING ^

- ROCA
- Assets
- Self-contained systems
 - Benefits
 - Challenges

This example system is deliberately presented in the first chapter on frontend integration. It shows how much is already possible with a simple integration via links.

Only for the *postbox* do you need some JavaScript. Before using the advanced technologies for frontend integration, you should understand what is already possible with such a simple approach.

The example integrates different technologies. In addition to the `Node.js` systems, there is also a Java/Spring Boot application that seamlessly integrates into the system. This demonstrates that a **frontend integration results in only a few limitations regarding the technology choice**.

ROCA

ROCA helps especially with this type of integration. The **microservices can be accessed via links, making integration very easy**. At the same time, the applications are largely decoupled in terms of deployments and technology.

An application can easily be deployed in a new version and not affect the other applications. The applications can also be implemented in different technologies.

At the same time, the ROCA UI is comfortable and easy to use. Compared to a single page app (SPA), there are no compromises in user comfort.

Assets

Finally, the application shows how to handle assets, in this case by using a common `Node.js` project. As a result, **each application can decide for itself when to adopt a new version of the assets**. This is important because otherwise a change of assets is automatically rolled out to all applications and might cause problems in the applications.

However, several versions of the assets should be used only temporarily on the web page. After all, **the design, look, and feel should be uniform**. Dealing with the asset project in such a way that not all services always use the current version is only meant to minimize the risk of an update but **must not lead to long-term inconsistencies**.

However, the asset project also ensures that all web pages contain jQuery in the version that the asset project uses. Thus, the asset project limits the freedom of individual projects with regards to JavaScript libraries.

Self-contained systems

Unlike self-contained systems (see [chapter 2](#)), this solution uses a **common backend**. With an SCS, the logic should also be part of the respective SCS and not be implemented in another system.

However, it is still possible to use some SCS ideas even if all systems share a common backend. The systems do not have to deal with logic and storing data as much as an SCS, because they are in the backend. Thus, this system shows **how a well-modularized portal for a monolithic backend can be implemented**.

But this approach also presents challenges. Any changes to the system will probably affect one of the frontend applications and also the backend. Therefore, the development and deployment of the two components must be coordinated.

Benefits

- Loose coupling
- Resilience
- No additional server components
- Low technical complexity
- Links often enough

Challenges

- Uniform look and feel
-

That's it for this chapter! Let's discuss server-side integration using edge side includes (ESI) in the next chapter.

Introduction

In this lesson, we'll look at a quick introduction to ESI concepts and what this chapter holds for us.

WE'LL COVER THE FOLLOWING ^

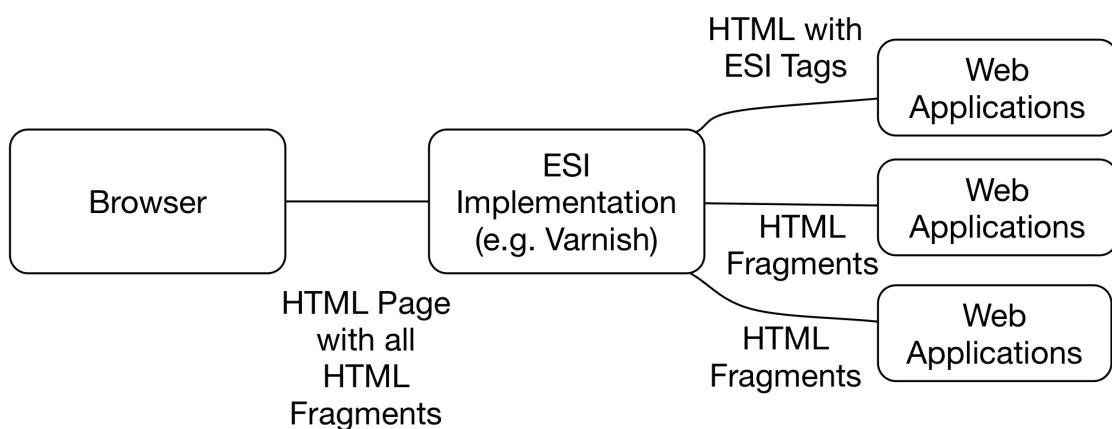
- ESI: concepts
 - Caches implement ESI
 - CDNs implement ESI
- Chapter walkthrough

ESI: concepts

Servers can also integrate multiple frontends. This chapter focuses on **ESI (Edge Side Includes)**.

ESI (Edge Side Includes) enables web applications to integrate HTML fragments of another web application (See the drawing below). To do so, the web application sends HTML containing ESI tags.

The ESI implementation analyzes the ESI tags and integrates HTML fragments of other web applications in the right positions.



Caches implement ESI

In the example, the web cache [Varnish](#) serves as an ESI implementation.

Other caches, such as [Squid](#), also support ESI. Websites use these caches to deliver web pages out of the cache upon incoming requests.

The servers handle requests only for cache-misses. This speeds up the website and decreases the load of the web servers.

CDNs implement ESI

Content Delivery Networks (CDNs) such as [Akamai](#) also implement the ESI standard. In principle, CDNs serve to deliver static HTML pages and images.

To do so, CDNs run servers at several Internet nodes so that every user can load web pages and images from a nearby server, thus **reducing loading times**.

Via the support of ESI, the assembling of HTML fragments can be done on a server that is close to the user. CDNs and caches implement ESI to be able to assemble web pages from different fragments.

Static parts can be cached, even if other parts have to be dynamically generated. This makes it possible to at least partially cache dynamic web pages which otherwise would have to be excluded from caches completely. This **improves performance**.

Therefore, **ESI** not only offers features for frontend integration but also **features especially useful for caching**.

Chapter walkthrough

Readers discover the following:

- How the web cache **Varnish implements ESI**.
- How applications can implement an **integration using ESI**.
- What benefits and disadvantages ESI has and what **alternatives to ESI** exist for implementing server-side frontend integration.

QUIZ

1

What is Edge Side Includes?

COMPLETED 0%

1 of 3



In the next lesson, we'll look at an example of how Edge Side Includes (ESI) can be used to assemble HTML fragments from different sources and how the entire HTML can be sent to the browser.

Example

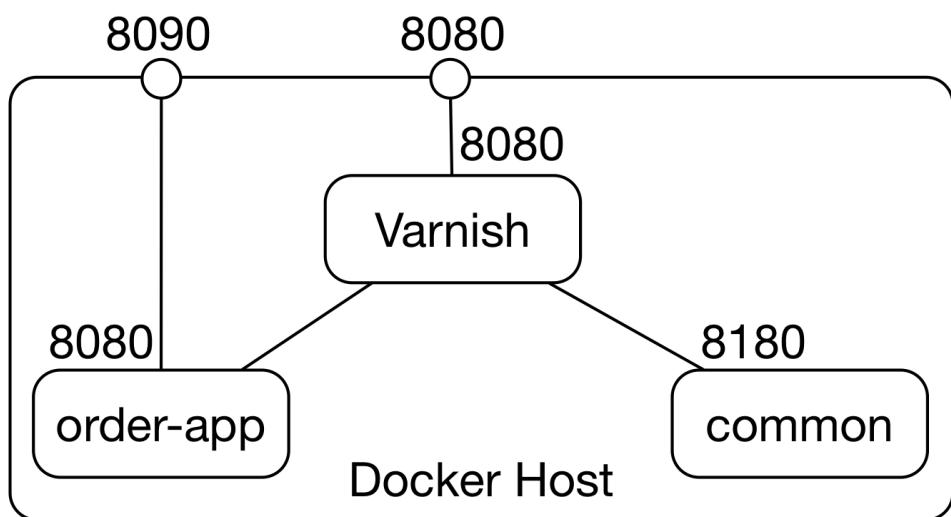
In this lesson, we'll look at ESI in action with an example!

WE'LL COVER THE FOLLOWING ^

- Introduction
- Running the example

Introduction

The [example](#) shows how Edge Side Includes (ESI) can be used to assemble HTML fragments from different sources and how the entire HTML can be sent to the browser. For this, the HTML contains special ESI tags that are replaced by HTML fragments.



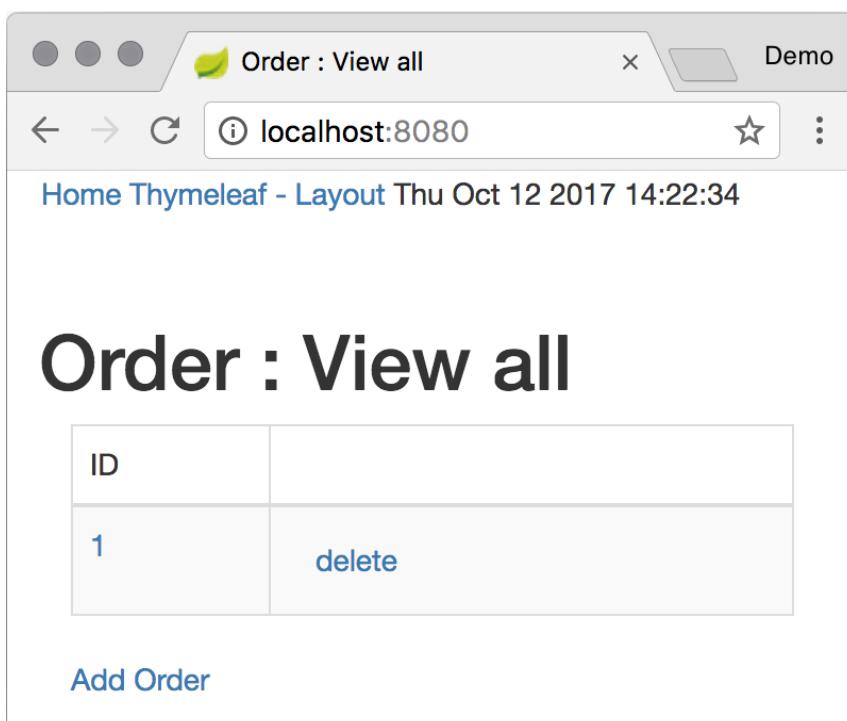
Overview of the ESI Example

The drawing above shows an overview of the structure of the example.

- The **Varnish cache directs the HTTP request** to the *order* microservice or the *common* service.
- The **order microservice contains the logic** for processing orders.

- The *common* service offers CSS assets and HTML fragments which microservices have to integrate into their HTML pages.

The example, therefore, shows a typical scenario. The applications like *order* deliver content that is displayed in a frame. The frame is provided by *common* so that all applications can uniformly integrate it.



Screenshot of the ESI Example

The screenshot above shows one page of the ESI example. The links to the home page, Thymeleaf, and the date are provided by the *common* service. The CSS and therefore the layout originate from the *common* service. The order service only provides a list of orders.

Thus, when additional microservices have to be integrated into the system, they only have to return the respective information in the middle. The frame and the layout are added by the common service.

During a reload, the time is updated but only every 30 seconds because the data is cached for that long. The cache works only if no cookies were sent in the request.

Running the example

To start the example, hit **run** below. You'll see a few commands running. You can access the app at the link generated below such as <https://x6jr4kg.educative.run>.

To run the code locally, follow these instructions:

<https://github.com/ewolff/SCS-ESI/blob/master/HOW-TO-RUN.md>

Varnish, which is available in the Docker host at port 8080, receives the HTTP requests and processes the ESI tags. If the Docker containers are running on the local computer, you can reach Varnish at port **8080**. For example, at <https://localhost:8080>.

The web pages of the order microservice can be accessed at

<https://localhost:8090>. These web pages contain the ESI tags and therefore appear broken if displayed in a web browser.

In the next lesson, we'll study Varnish, a web cache.

Varnish

In this lesson, we'll discuss Varnish.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Licence and support
- Caching with HTTP and HTTP headers
- Varnish Docker containers
- Varnish configuration
- Load balancing
- Evaluation of VCL

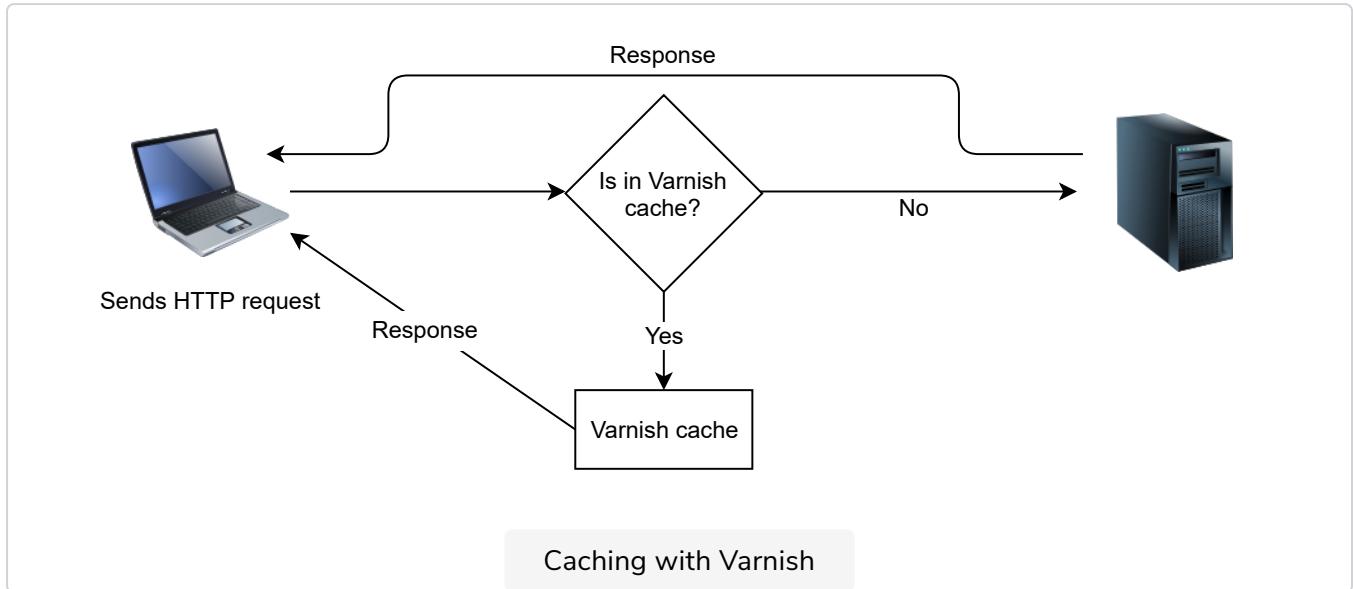
Introduction

Varnish is a web cache and is used as an ESI implementation in the example.

Varnish is mainly used for optimizing web servers:

1. It intercepts the HTTP requests to web servers.
2. It then caches the responses.
3. It forwards only those requests to the web servers that are **not** in the cache.

This improves performance.



Licence and support

Varnish is licensed under a [BSD license](#). The cache is mainly developed by [Varnish Software](#), which also provides commercial support.

Caching with HTTP and HTTP headers

Caching data correctly is not a trivial matter. Above all, the questions are **when can content be retrieved from the cache and when does the data have to be retrieved from the web server** because the data in the cache no longer represents the current state? Varnish uses its own HTTP headers for this.

The HTTP protocol has very good support for caching through its HTTP headers. The control over whether data is cached rests with the web server, which informs the cache of the settings via HTTP headers. Only the web server can decide whether a page can be cached because that depends on the domain logic.

Varnish Docker containers

In the example, Varnish runs in a Docker container which contains an Ubuntu 14.04 LTS. On the Ubuntu image, the Varnish version from the official Varnish repository will be installed.

Varnish configuration

Varnish offers a powerful configuration language. For this example, Varnish is installed in its own Docker container. The configuration can be found in the file `default.vcl` in the directory `docker/varnish/`. Here is an extract with the essential settings.

```
vcl 4.0;

backend default {
    .host = "order";
    .port = "8080";
}

backend common {
    .host = "common";
    .port = "8180";
}

sub vcl_recv {
    if (req.url ~ "^/common") {
        set req.backend_hint = common;
    }
}

sub vcl_backend_response{
    set beresp.do_esi = true;
    set beresp.ttl = 30s;
    set beresp.grace = 15m;
}
```

- `vcl 4.0;` chooses version 4 of the Varnish configuration language.
- The first backend has the name `default`. Each request arriving at the Varnish cache is passed on to the web server if there is no other configuration. The hostname `order` is resolved by Docker Compose. The `default` backend is the *order* microservice, which implements all functions to accept and display orders.
- The second backend has the name `common` and is provided by the host of the same name. In this case, Docker compose resolves the name to a Docker container. The `common` service provides headers and footers for the HTML pages of the microservices and `Bootstrap` as a shared library for the UI of the microservices.
- When an HTTP request arrives for a URL where the path starts with `/common`, the HTTP request is redirected to the `common` backend. The code

of the subroutine `vc1_recv` is responsible for this. Varnish automatically calls this routine to determine the routes for the requests.

- The subroutine `vc1_backend_response` configures Varnish. The present configuration not only enables ESIs but also implements the functionality of a reverse proxy and redirects requests to specific microservices.
 - `beresp.do_esi` configured Varnish so it interprets ESI tags.
 - `beresp.ttl` turns on the caching. Each page is cached for 30 seconds. However, this caching is very simple: if a new order has been created, it is not displayed until the cache has been invalidated. This can take up to 30 seconds. It would be better if a new order leads to the invalidation of the cache. This is still relatively easy to implement in the example, but in a complex application, it can be difficult to invalidate the correct pages. For example, goods are displayed on product pages and order pages, so several pages need to be invalidated if the data for the goods is changed. Thus, simple time-based caching can be the better solution, being easy to implement and possibly doing a good enough job. There is a [chapter](#) about cache invalidation in the Varnish book.
 - Finally, `beresp.grace` ensures that if a backend fails, the web pages are cached for 15 minutes. This can temporarily compensate for a backend failure. Of course, this works only if the web page is already in the cache because it has been accessed before the failure. Therefore, if the page is not in the cache or not cacheable at all, this feature does not help.

Load balancing

You can add [load balancing](#) to the Varnish configuration, thereby splitting the requests to the microservices across multiple microservices by defining multiple `backends` in the configuration. However, you can of course also rely on an external load balancer. In that case, Varnish would do only ESI and caching.

Evaluation of VCL

As you can see, VCL is a very powerful language that has many possibilities

for manipulating HTTP requests. That is essential, for example, in a case where you can only be sure a request can be cached if it does not contain cookies as cookies could change the response; therefore, VCL must be able to remove cookies.

A comprehensive documentation of Varnish and VCL can be found in the free [Varnish book](#).

QUIZ

1

What would be a side effect of setting the ttl to 15m, as per the following code snippet, in a highly interactive E-commerce app?

```
sub vcl_backend_response{
    set beresp.do_esi = true;
    set beresp.ttl = 5m;
    set beresp.grace = 15m;
}
```

COMPLETED 0%

1 of 3



The next lesson is about ESI.

ESI in Order & Common

In this lesson, we'll look at how ESI is used in the common and order applications.

WE'LL COVER THE FOLLOWING ^

- Order microservice
- HTML with ESI tags in the example
- ESI tags in the HTML head
- ESI Tags in the remaining HTML
- Result: HTML at the browser
- No tests without ESI infrastructure
- Effects on the application
- Common microservice
- Asset server

Order microservice

The *order* microservice offers a normal web interface, which has been supplemented with ESI tags in some places.

A typical HTML page of the *order* microservice looks like this:

```
<html>
<head>
  ...
  <esi:include src="/common/header"></esi:include>
</head>

<body>
  <div class="container">
    <esi:include src="/common/navbar"></esi:include>
    ...
  </div>
  <esi:include src="/common/footer"></esi:include>
</body>
</html>
```

HTML with ESI tags in the example

The *order* microservice is available at port 8090 of the Docker host. The output goes past the Varnish and still contains the ESI tags. At <http://localhost:8090/> the HTML with the ESI tags can be viewed.

The ESI tags look like normal HTML tags. They only have an `esi` prefix. Of course, a web browser cannot interpret them.

ESI tags in the HTML head

- **line 4:** In the `head` the ESI tags are used to integrate common assets like Bootstrap into all pages. Changing the header under `"/common/header"` causes all pages to get new versions of Bootstrap or other libraries. If the pages with a new version are no longer displayed correctly, such a change will cause problems. Therefore, the pages themselves should be responsible for using new versions. For example, a version number can be encoded in the URL in the ESI tag.

ESI Tags in the remaining HTML

- **line 9:** The ESI Include for `"/common/navbar"` ensures that each web page has the same navigation bar.
- **line 12:** Finally, `"/common/footer"` can contain scripts or a footer for the web page.

Result: HTML at the browser

Varnish collects these HTML snippets from the common service so that the browser receives the following HTML:

```
<html>
<head>
...
<link rel="stylesheet"
  href="/common/css/bootstrap-3.3.7-dist/css/bootstrap.min.css" />
<link rel="stylesheet"
  href="/common/css/bootstrap-3.3.7-dist/css/bootstrap-theme.min.css" />
</head>

<body>
  <div class="container">
    <a class="brand">
```

```
href="https://github.com/ultraq/thymeleaf-layout-dialect">
Thymeleaf - Layout </a>
Mon Sep 18 2017 17:52:01 </div></div>

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

The ESI tags have thus been **replaced** by suitable HTML snippets.

ESI offers many other features for securing a system against the failure of a web server or for integrating HTML fragments only under certain conditions.

No tests without ESI infrastructure

A problem with the ESI approach is that individual services cannot be tested without an ESI infrastructure. At the very least, they do not display any pages with meaningful content, because the ESI tags would have to be interpreted for that. This works only if the HTTP requests are routed through Varnish. Therefore, suitable environments containing a Varnish must be provided for the development.

Effects on the application

The application is a normal Spring Boot web application without any dependencies on Spring Cloud or ESI. This shows that pure frontend integration leads to a very loose coupling and has little impact on the applications.

Common microservice

In the example provided, the *common* service is a very simple [Go](#) application. It handles the three URLs `"/common/header"`, `"/common/navbar"`, and `"/common/footer"`. For these URLs, the Go code generates suitable HTML fragments.

Asset server

The Go code also contains a web server that provides static resources under `"/common/css/"` – the Bootstrap framework. In this way, the common microservice assumes the function of an asset server. Such a server offers CSS, images, or JavaScript code to applications. The ESI example shows an

images, or JavaScript code to applications. The ESI example shows an alternative for the integration of shared assets. In [chapter 4](#), a common asset

project has ensured that all applications can use the same assets. In the example in this chapter, an asset server is used for this purpose.

The application displays the current time in the navigation bar. This shows that dynamic content can also be displayed with ESI includes.

Q U I Z

1

What is the purpose of the common microservice?

COMPLETED 0%

1 of 2



In the next lesson, we'll look at some variations to this.

Variations

In this lesson, we'll look at variations on the server-side integration with ESI approaches we've already discussed.

WE'LL COVER THE FOLLOWING

- Different ESI implementations
- SSI
- Tailor
- Client-side integration
- Shared library
- Additional integration

Different ESI implementations

Of course, instead of Varnish, a different ESI implementation could be used by [Squid](#) or by a CDN like [Akamai](#).

SSI

Another option for server-side frontend integration is **SSI (Server-side includes)**. This is a feature that most web servers offer. <https://scs-commerce.github.io/> is an example of a system that uses SSI with the nginx webserver to integrate the frontends.

SSI and ESI have different benefits and disadvantages.

- Web servers are often already available in the infrastructure for SSL/TLS termination or for other reasons. Because web servers can implement SSI, no additional infrastructure is necessary.
- Caches not only speed up applications, but also compensate for web server failures for some time, thus improving resilience. This speaks for ESI and a cache like Varnish. ESI also has more features for further

optimizing caching.

- Correct caching can also be difficult to implement. For example, changing the data of a single data record can trigger a cascade of invalidations. Finally, every page and every HTML fragment containing information about the goods must be regenerated.

Tailor

[Tailor](#) is a system for server-side frontend integration that Zalando implemented as part of [Mosaic](#). It is optimized for showing the user the first parts of the HTML page as quickly as possible. For e-commerce, the rapid display of a web page is very important to keep users and can increase sales.



To achieve this, Tailor implements a [BigPipe](#). First, very simple HTML code is transferred to the user in order to be able to display a simple page very quickly. JavaScript is used to load more details step by step. Tailor implements this with asynchronous I/O using Node.js streams.

Client-side integration

Client-side integration does not use any additional infrastructure and can be the simpler option for frontend integration. Therefore, it makes sense to find out how far you can go with client-side integration before using server-side frontend integration.

For the integration of a header and footer, like in the example for this chapter, **server-side integration is the better choice** as a page cannot be displayed without these elements. The pages should be delivered to the user in such a way that they can be displayed to the user **without loading any additional content**.

Therefore, **client-side integration** for **optional** elements makes sense. Dealing with failed services is then a task for the client code. That might simplify the server implementation and the server setup.

Shared library

The example from [chapter 4](#) uses a library to deliver assets.

Theoretically, the HTML fragments that are integrated with ESI in this example could also be delivered as a library. But then all systems would have to be rebuilt and deployed for an additional link in the navigation. With the ESI solution, all you have to do is change the HTML fragment on the server.

Additional integration

Pure frontend integration is rarely enough. Therefore, a system will combine backend integration with synchronous (see [chapter 9](#)) or asynchronous (see [chapter 6](#)) communication mechanisms with frontend integration. An exception is a scenario like in [chapter 4](#), where a complex portal application is implemented. The parts of the portal can communicate through frontend integration. Backend integration is not necessary because the services do not implement a lot of logic and have no database; they only create a web interface.

Q U I Z

1

Which of the following is an advantage of SSI?

COMPLETED 0%

1 of 2



In the next lesson, we'll look at some experiments to further deepen your SSI understanding with ESI concepts.

Experiments

In this lesson, we'll look at some experiments that you can try out with SSI using ESI.

WE'LL COVER THE FOLLOWING ^

- Experiments
- Try it yourself!

Experiments

Here are some experiments you can do with all that you've learned in this chapter.

Supplement the system with an additional microservice.

Try it yourself!

Try all the experiments in the environment below.

- A microservice that simply displays a static HTML page can serve as an example.
- The new microservice should integrate header, navigation bar, and footer of the common microservice via ESI tags.
- Package the microservice as a Docker image and reference it in `docker-compose.yml`. There you can also determine the name of the Docker container.
- The microservice has to be accessible via Varnish. To achieve that, you have to integrate a new backend with the name of the Docker container in `default.vcl` and adapt the routing in `vcl_recv()`.

- Now you should be able to access the new microservice at <http://localhost:8080/name> if the Docker containers run on the local computer and the new service is configured in Varnish's routing with `name`. The ESI tags should have been replaced by HTML code.
- The Go application returns only a dynamic HTML fragment – the navigation bar with the current time. Instead of the Go application, a web server can also deliver static pages. For example, replace the Go application with an Apache httpd server that delivers the HTML fragments and the Bootstrap library. The time does not necessarily have to be displayed in the navigation bar, so a static HTML fragment is all that is needed.
- Change the caching so that the pages are immediately invalidated when new orders are received. You can, for example, use matching HTTP headers as explained in a [chapter](#) of the Varnish book. An alternative is to remove objects directly from the cache. The Varnish book also contains a [chapter](#) on this option.
- Replace the Varnish cache in the example with [Squid](#), which also implements ESI.
- Replace ESI with SSI and replace the Varnish cache with an Apache httpd or nginx.
- What happens if the web servers fail? Simulate the failure with `docker-compose up --scale common=0` or `docker-compose up --scale order=0`. Which parts of the web page still work? Is it still possible to place orders, for example?

In the next lesson, we'll look at a quick chapter conclusion.

Chapter Conclusion

In this lesson, we'll look at a quick summary of what we've learned in this chapter.

WE'LL COVER THE FOLLOWING ^

- Summary
- Benefits
- Challenges

Summary

- ESIs are a possible implementation of frontend integration and lead to **loose coupling**. The applications are simple web applications that, apart from the ESI tag, have no dependencies on the infrastructure.
- The integration with ESIs has the advantage that the **web pages are completely assembled by the cache and can be displayed directly in the browser**. Therefore, the page isn't delivered unusable in any way due to fragments that still have to be loaded.
- Using a cache together with ESI has the advantage that **fragments can be cached**.
 - This means that static pages and static parts of dynamic pages can be cached, which improves performance.
 - The pages can even be cached and assembled from a CDN that supports ESI, further improving performance.
- The cache can also be used to achieve a certain degree of **resilience**.
 - If a web server fails, the cache can return the old data. In this way, the web page remains available but could potentially return invalid information.

- However, the cache must hold the pages a long time in order for this to happen.
- In addition, the cache must also check the availability of the services.

Benefits

- Web page is always delivered in entirety
- Resilience via cache
- Higher performance via cache
- No code in the browser

Challenges

- Uniform look and feel
- Additional server infrastructure necessary

That's it for this chapter! In the next one, we'll discuss asynchronous microservices!

Introduction

This lesson contains a quick walkthrough of what future lessons hold for us.

WE'LL COVER THE FOLLOWING ^

- Chapter walkthrough

Asynchronous microservices have advantages over synchronous microservices.

Chapter walkthrough

This chapter discusses:

- How microservices can communicate asynchronously.
- Which protocols can be used for asynchronous communication.
- How events and asynchronous communication are linked.
- The advantages and disadvantages of asynchronous communication.

Let's start with a quick definition of asynchronous programming and how microservices can communicate asynchronously in the next lesson!

Definition

In this lesson, we'll introduce asynchronous microservices.

WE'LL COVER THE FOLLOWING ^

- No communication
- Does not wait for a response
- Asynchronous communication with no response

Asynchronous microservices are different from synchronous microservices, which are covered in depth in [Chapter 9](#).

A microservice is **synchronous** if it makes a request to other microservices while processing requests and waits for the result.

The logic to handle a request in the microservice might therefore not depend on the result of a request to a different microservice.

So, a definition of **asynchronous microservices** would be:

A microservice is **asynchronous** if:

- (a) It does not make a request to other microservices while processing requests. OR
- (b) It makes a request to other microservices while processing requests and does not wait for the result.

There are **two cases here**, let's discuss each.

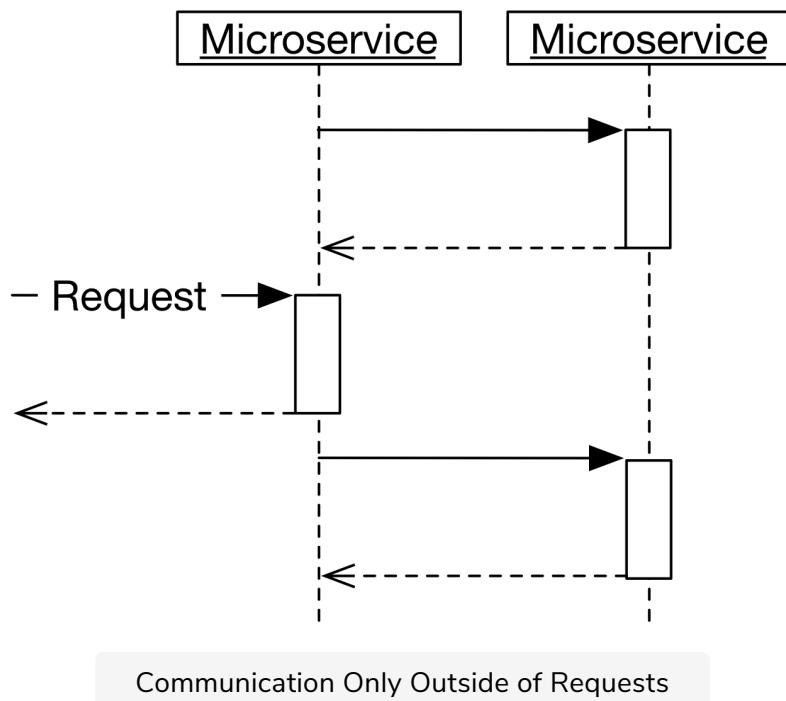
No communication

The microservice **does not communicate at all with other systems** while

processing a request. In that case, the microservice will typically

communicate with the other systems at a different time, see the drawing below.

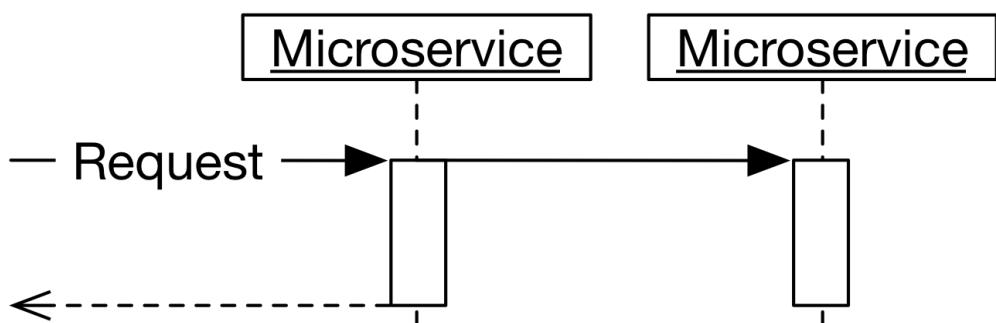
For example, the microservice can replicate data that is used when processing a request. In this way, customer data can be replicated so that when processing an order, the microservice can access the locally available customer data instead of having to load the necessary customer data for each request via a request to another system.



Does not wait for a response

The microservice sends a request to another microservice but **does not wait for a response**, see the drawing below.

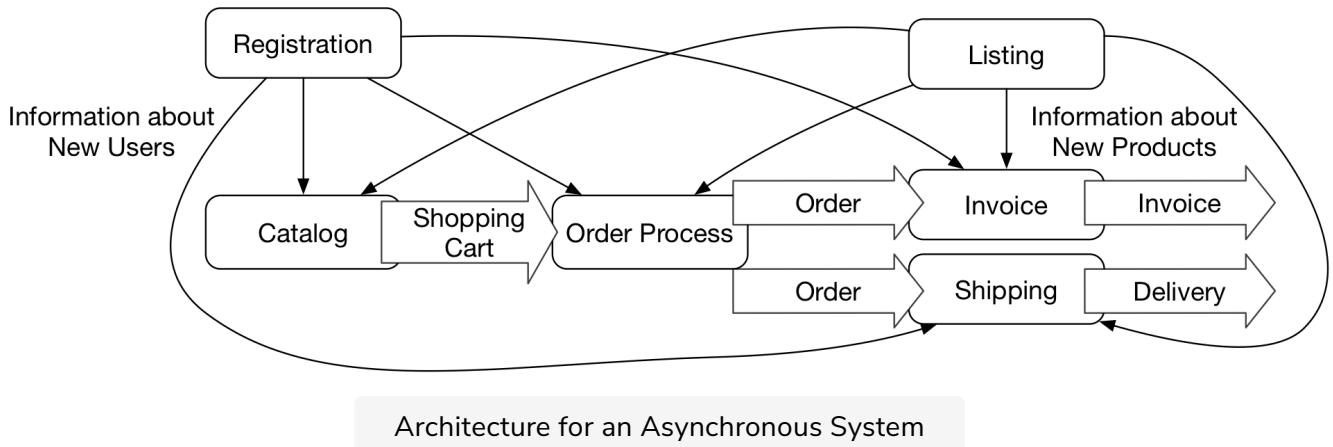
A microservice responsible for processing an order can send a request to another microservice which generates the invoice. A response to this request is not necessary for processing the order so there is no need to wait for it.



The drawing below shows an example of a more complex asynchronous architecture.

In this e-commerce system, orders are processed in the following way. It starts when customers can choose goods for an order through the catalog.

- The *order process* generates the orders.
- An *invoice* and a *shipping data record* is produced for the order.
- The *registration* microservice adds new customers to the system.
- The *listing* microservice is responsible for new goods.



Asynchronous communication with no response

#

The four systems, *catalog*, *order process*, *invoice*, and *shipping*, send asynchronous notifications for processing the orders.

- The **catalog** collects goods in the shopping cart. If the user orders the shopping cart, the *catalog* transfers the cart to the *order process*.
- The **order process** turns the shopping cart into an order.
- The order then becomes an invoice and a delivery.

Such requests **can be executed asynchronously**. No data has to flow back. The responsibility for the order is transferred to the next step in the process.

1

Suppose a microservice sends off a request for a resource but resumes processing. What kind of microservice is this?

COMPLETED 0%

1 of 2



In the next lesson, we'll learn about data replication, bounded contexts, and communication protocols.

Data Replication, Bounded Contexts, & Protocols

In this lesson, we'll study data replication, bounded contexts, and protocols.

WE'LL COVER THE FOLLOWING



- Data replication and bounded context
- Synchronous communication protocols
- Asynchronous communication protocols

Data replication and bounded context

Asynchronous communication becomes more complicated **if data is required** to execute a request.

For example, in the *catalog*, the *order process*, and the *invoice* data about products and customers has to be available.

Each of the systems stores a part of the information about these business objects.

- The *catalog* must display the products, so it has pictures and descriptions of the products.
- For *invoices*, prices and tax rates are important.

This corresponds to the definition of bounded contexts.

Each **bounded context** has its own domain model. That means that all data for the bounded context is represented in its domain model.

Therefore, the data specific for the bounded context should be stored in the bounded context in its own database schema.

Other bounded contexts should not access that data directly, which would

compromise encapsulation. Instead, the data should be accessed only by the logic in the bounded context and its interface.

Although it would be possible to have a system that contains all information about, for example, a product, this would not make a lot of sense. The model of the system would be very complicated. Also, it means that the domain model would be split across one system for an order process and another system for the product data. That **would lead to a very tight coupling**.

A third system, such as *registration* for customer data or *listing* for product data, must accept all the data and transfer the needed parts of the data to the respective systems. This can also be done asynchronously.

The other bounded contexts then store the information about products and customers in their local databases, making replication a result of events being processed. An event such as “new product added” makes each bounded context add some data to its domain model.

Registration or *listing* do not need to store the data. After they have sent the data to the other microservices, their job is done.

It is also possible to do an extract-transform-load approach. In that case, a batch would *extract* the data from one bounded context, transform it into a different format, and load it into the other bounded context. This is useful if a bounded context should be loaded with an initial set of data, or if inconsistencies in the data require a fresh start.

Synchronous communication protocols

Asynchronous communication as previously defined does not make any assumptions about the communication protocol used.

For **synchronous** communication, the server **must respond to each request**.

Examples are REST and HTTP. A request leads to a response that contains a status code and optional additional data. It is possible to implement asynchronous communication with a synchronous communication protocol. [Chapter 8](#) explains this in more detail.

Asynchronous communication protocols

It is more natural to implement asynchronous communication with an

asynchronous communication protocol. An asynchronous communication

protocol **sends messages and does not expect responses**. Messaging systems like Kafka (see [chapter 7](#)) implement this approach.

There is also a [presentation](#) which explains the difference between REST and messaging, and highlights that both technologies can be used to implement synchronous communication like **request/reply**, but also asynchronous communication like **fire & forget** or **events**.

Q U I Z

1

If a system is a bounded context it _____.

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss events in asynchronous communication.

Events

In this lesson, we'll learn all about events.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Events and DDD
- Patterns from strategic design
 - Specific events
 - Published language
- Sending minimal data in an event

Introduction

With asynchronous communication, the coupling of systems can be driven to different lengths. As already mentioned:

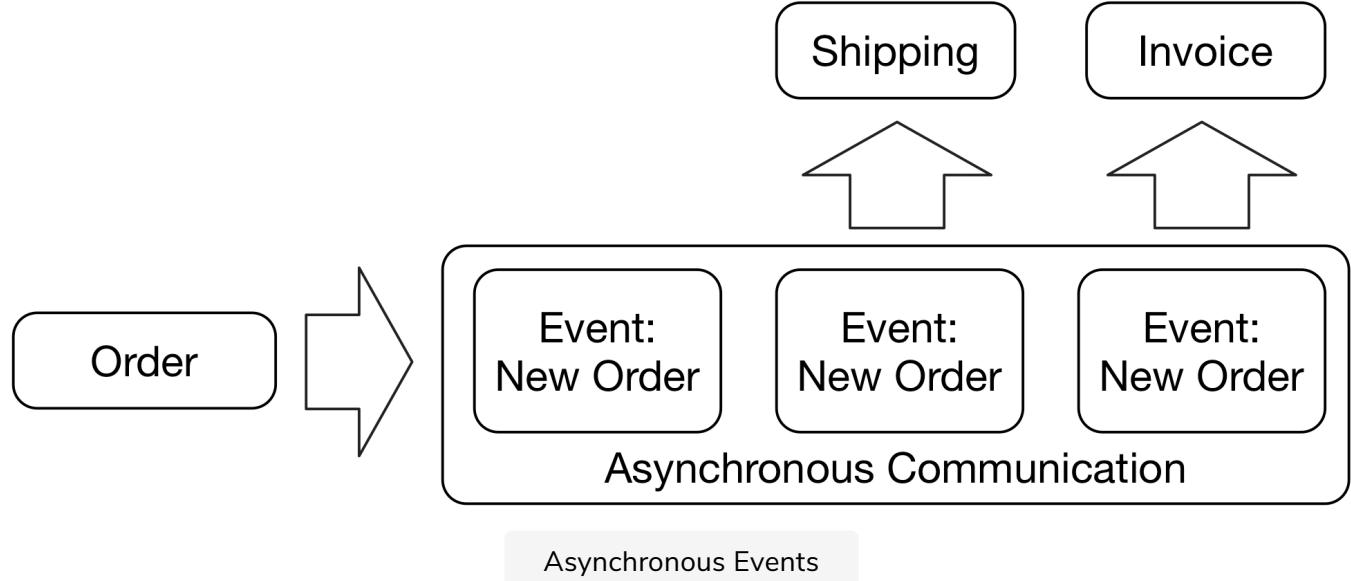
- The system for **order processing** could inform the **invoice** system asynchronously that an invoice has to be written.
- The ordering system thus determines exactly what the invoicing system has to do: generate an invoice.
- It also sends a message to the bounded context **shipping** to trigger the delivery.

The system can also be set up differently. The focus will then be on an event like, “There is a new order.” Every microservice can react appropriately to this:

- The **invoicing** system can write an invoice.
- The **shipping** system can prepare the goods for delivery.

Each microservice decides for itself how it reacts to the events, see the diagram below.

drawing below.



This leads to **better decoupling**. If a microservice has to react differently to a new order, the microservice can implement this change on its own.

It is also possible to **add a new microservice** that generates statistics when a new order is placed.

Events and DDD

However, this approach is not quite as easy to implement. The crucial question is **what data is transferred with the event?**

If the data is to be used for such different purposes as the writing of an invoice, statistics, or recommendations, then a large number of different attributes must be stored in the event.

This is problematic because domain-driven design shows that **each domain model is valid only in a bounded context**:

- For invoicing, prices and tax rates have to be known.
- For shipping, size and weight of the goods are needed to organize a suitable transport.

Patterns from strategic design

The views of *invoice* and *shipping* on the data of an order represent two bounded contexts, which can receive data from a third bounded context, the *order process*.

Domain-driven design defines patterns for this. For example, with customer/supplier, the team for invoicing and shipping can define what data it needs to receive. The team that provides the data for the order must meet these requirements.

This pattern defines the interaction of the teams that develop the bounded contexts that participate in the communication relationship. Such coordination is necessary regardless of whether events are sent, or whether communication between components takes place by a synchronous call.

In other words: Events may seem to decouple the system, but coordination regarding the necessary data must still take place.

This means that events do not necessarily lead to a truly decoupled system. In extreme cases, events can even lead to hidden dependencies. Who reacts to an event? If this question can no longer be answered, the system is hardly changeable anymore because the changes to the events have had unforeseeable consequences.

Specific events

A solution might be to provide specific types of events for each receiver. Each type of event would contain the information that this receiver needs.

If a new order appears in the system, an event is sent to the invoicing system with the data it needs. Another event is sent to shipping with the data for that system.

The two systems are truly decoupled: a change in the interface to one of the systems does not influence the other system. This can be the result of a customer/supplier relationship.

Published language

Another solution would be to use a published language. In that case, a common data structure exists that contains all information for all receivers. This might make it hard to understand which receivers use what as changes to the data structure might affect many receivers. However, the initial

the data structure might lead to unforeseen problems. However, there is just one data structure, so it is somewhat easier to implement the system.

A very important matter is the difference in information that each receiver needs. If it is mostly the same, a published language might be better. If it is very different, it might make more sense to have separate data structures. For the case of invoicing and shipping, there is probably not too much overlap, so two separate data structures might be the better alternative.

Sending minimal data in an event

There is yet another way to deal with this problem. In this event, only the number of a new order is sent along.

Afterwards every bounded context can decide for itself how to get the necessary data. There can be a special interface for each bounded context that provides the appropriate data for that specific bounded context.

QUIZ

1

Suppose there is no overlap between the data that two microservices require in order to respond to an event. Which of the following patterns does NOT suit this scenario well?

COMPLETED 0%

1 of 2



In the next lesson, we'll study event sourcing.

Event Sourcing

In this lesson, we'll study event sourcing.

WE'LL COVER THE FOLLOWING

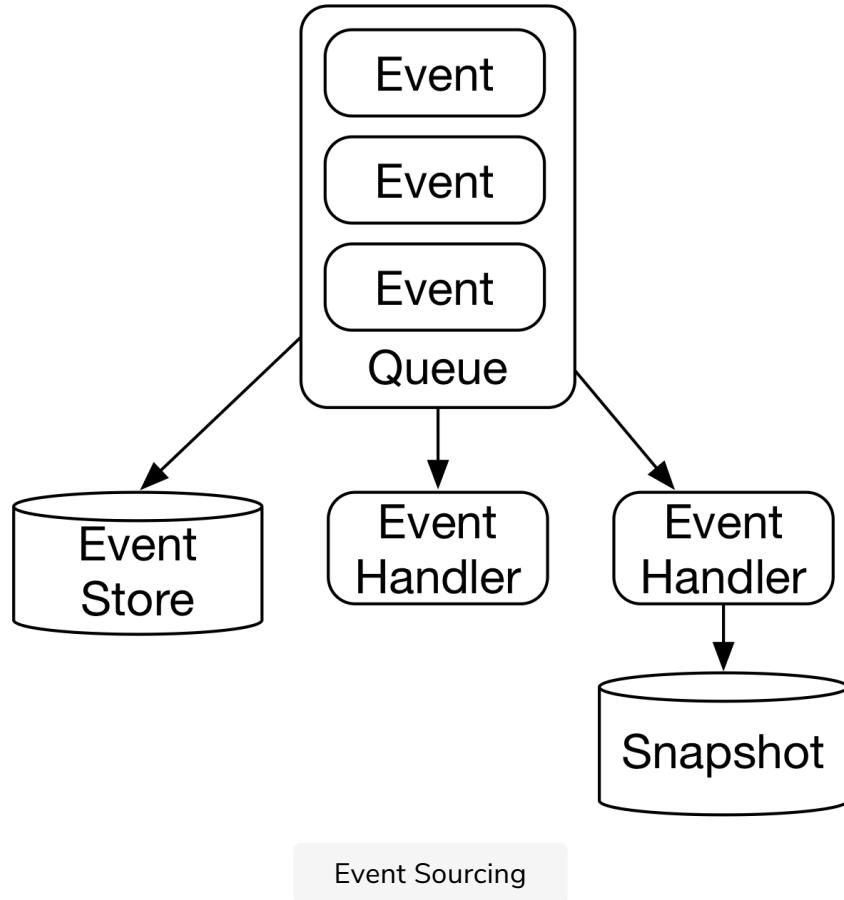


- Individual or shared event store?

An architecture's focus on events can also entail **other advantages**.

- The state each microservice has in its database is the result of the events it has received. The **state of a microservice can be restored** by resending all events it has received so far.
- The microservice can even change its internal domain model and then process the events again to rebuild its database with the **new version of the domain model**. This facilitates database schema migration.
- Thus, each microservice can have its own domain model according to the bounded context pattern, but **all microservices are still connected** by the events they send to each other.
- An overall state of the system no longer exists, but when all events are saved and can be retrieved, **the state of each microservice can be reconstructed**.

These ideas form the basis for event sourcing.



The elements of an event sourcing implementation are shown in the drawing above:

- The **event queue** sends the events to the recipients.
- The **event store** saves the events.
- **Event handlers** process the events. They can save their state as a *snapshot* in a database.

The event handler can read the current state from the snapshot, the snapshot can be deleted, and the snapshot can be restored on the basis of the events, which can be retrieved from the event store.

As an optimization, an event handler can also reconstruct its state from an older version of the snapshot.

There is a difference between the events for event sourcing and domain events; see [Christian Stettler's blog post](#).

Individual or shared event store?

The event store can be part of the microservice that receives events and stores them in its own event store. Alternatively, the infrastructure not only sends the events but also stores them.

At first glance, it seems better if the infrastructure stores the events because it simplifies the implementation of the microservices. In such a case, the event store would be implemented in the event queue.

If each microservice stores the events in its own event store, the microservice can store all relevant data in the event, which the microservice may have collected from different sources.

When storing events in the infrastructure, it is necessary to find a model of the event that satisfies all microservices.

Such a model for the events can be a challenge because of the concept of bounded context. After all, every microservice is a separate bounded context with its own domain model, so finding a common model is difficult.

Q U I Z

1

What is the difference between an event store and a snapshot?

In the next lesson, we'll look at some challenges associated with asynchronous microservices.

Challenges: Inconsistencies & CAP Theorem

In this lesson, we'll discuss some challenges that may arise when using asynchronous microservices.

WE'LL COVER THE FOLLOWING



- Old events
- Inconsistency
- CAP theorem
 - Reasons for the CAP Theorem
 - Compromises with CAP
 - CAP, events, and data replication

Old events

If the communication infrastructure for event sourcing has to store old events, it has to handle considerable amounts of data. Consequently, if old events are missing, the state of a microservice can no longer be reconstructed from the events.

As an optimization, it would be possible to **delete events that are no longer relevant**. If a customer has moved to a different address several times, the last address is probably the only relevant one. The others can then be deleted.

In addition, it must also be possible to **continue processing old events**. If the schema of the events changes in the meantime, **old events have to be migrated**. Otherwise, every microservice has to be able to handle events in all old data formats.

This is particularly difficult if new data has to be contained in events that have not yet been saved in old events.

Inconsistency

Due to asynchronous communication, the system is not consistent. Some microservices already have certain information while others do not.

For example, *order process* might already have information about an order, but *invoicing* or *shipping* does not know about the order yet.

This problem cannot be solved. It takes time for asynchronous communication to reach all systems.

CAP theorem

These inconsistencies are not only practical problems but **cannot even be solved in theory**.

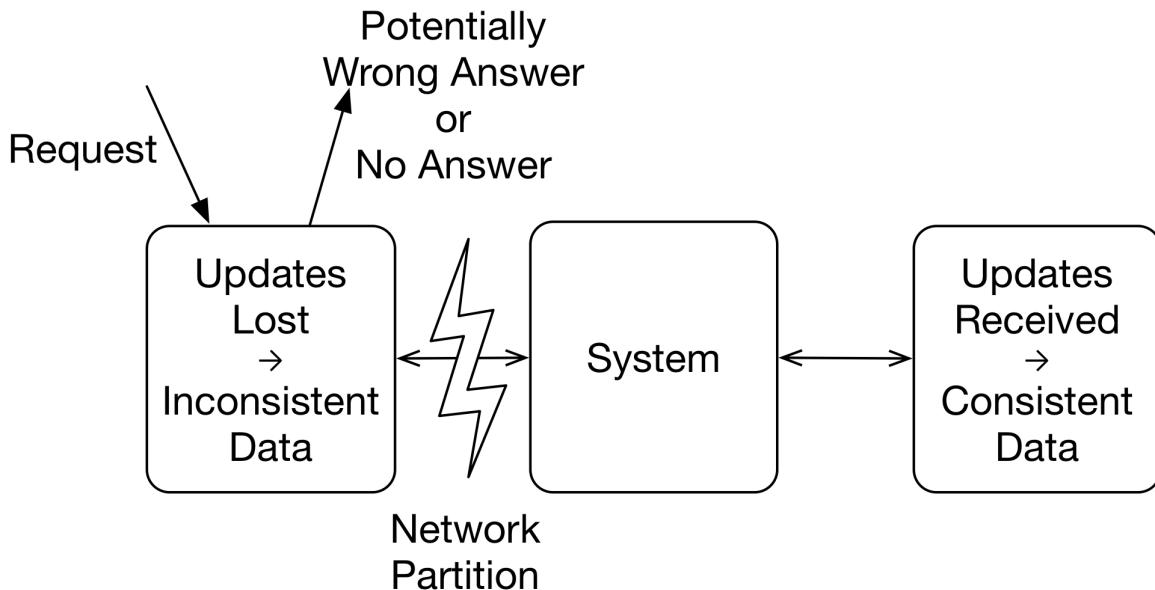
According to the [CAP theorem](#), three characteristics exist in a distributed system:

- **Consistency (C)** means that all components of the system have the same information.
- **Availability (A)** means that no system stops working because another system failed.
- **Partition tolerance (P)** means that a system will continue to work in case of arbitrary package loss in the network.

The **CAP theorem** states that a system can have a maximum of two features out of these three.

Partition tolerance is a special case. A system must react if the network fails. In fact, not even a complete failure is necessary; the package loss just needs to be high or the response time is very long. A system that responds very slowly is indistinguishable from a system that has failed completely.

Reasons for the CAP Theorem



If Communication Fails, a System Can Either Return a Potentially Wrong Response Upon a Request (AP) or None at All (CP)

There are **only two options** as to how a system can react to a request when the network is partitioned, see the drawing above.

1. **The system provides a response.** In this instance, the response can be wrong because changes have not reached the system; this is the *AP* case. The system is available, however, it might return a different response from systems that have obtained newer information. Thus, inconsistencies exist.
2. Alternatively, **the system returns no response;** this is the *CP* case. On the one hand, the system is not available when there is a problem. On the other hand, all systems always return the same responses and therefore are consistent as long as there is no network partitioning.

Compromises with CAP

Of course, you can make compromises. Let's take a **system with five replicas**:

- When writing, each replica confirms that the data has actually been written.
- When reading, several systems can be called to find out the latest state of the data.

Such a system with five replicas, in which one replica is read and only the

confirmation from one replica is waited for, **focuses on availability**. Up to four nodes can fail without the system failing.

However, it does not guarantee high consistency:

- The data could possibly be written to one node and – due to the time for replication to other nodes to occur – an old value is read from another node.

If the system with five replicas always waits for five nodes to be confirmed and always reads from five nodes, the data is always consistent.

However, the failure of a single node causes the system to become unavailable. A compromise can be to wait for confirmation of writes from three nodes and for reads from three nodes. In this way, inconsistencies can still be ruled out and the failure of up to two nodes can be compensated for.

CAP, events, and data replication

The CAP theorem actually considers data storage like NoSQL databases, which achieve performance and reliability via replication. But similar effects also occur when systems use events or data replication.

Ultimately, an event can be seen as a kind of data replication across multiple microservices. However, unlike the full replication of data between nodes of NoSQL databases, **each microservice can react differently to the event and may use only parts of the data**.

A microservice that relies on asynchronous communication, events, and data replication corresponds to an **AP system**.

Microservices may not have received some events yet, so the data may be inconsistent. Nevertheless, the system can process requests using local data and is therefore available even if other systems fail.

The CAP theorem says that the only alternative is a **CP system**. This would be consistent but not available.

For example, it could store the data in a central microservice which is

accessible by all. As a result, **all microservices would receive the latest**

data. However, **if the central microservice fails, all other microservices would no longer be available.**

QUIZ

1

Which of the following is NOT a part of the CAP theorem?

COMPLETED 0%

1 of 2



In the next lesson, we'll continue discussing the challenges that involve inconsistencies.

More on Inconsistencies

In this lesson, we'll further explore the CAP theorem.

WE'LL COVER THE FOLLOWING ^

- Are inconsistencies acceptable?
- Repairing inconsistencies
 - Guaranteed order of events
 - Event sourcing
 - Extending domain logic

Are inconsistencies acceptable?

As per the last lesson, **the inconsistency of an asynchronous system is inevitable unless you want to give up availability.**

It is therefore important to know the requirements for consistency, which requires some skill. Customers want a reliable system, data inconsistency seems to contradict this. That's why it is important to know what happens when the data is temporarily inconsistent and whether this really causes problems.

After all, the inconsistencies should usually disappear after a few seconds. Besides, **certain inconsistencies can even be tolerable** from a domain perspective.

- For example, if goods are listed days before the first sale, inconsistencies are initially acceptable and must only be corrected when the goods are finally being sold.

If **inconsistencies are not acceptable at all, asynchronous communication is not an option.**

This means that synchronous communication must be used with all its disadvantages. If the tolerance for temporarily inconsistent data is not known, this can lead to a wrong decision regarding the communication variant.

Repairing inconsistencies

Guaranteed order of events

In the simplest case, the inconsistencies disappear as soon as all events have reached all systems. However, there may be exceptions. An example:

- After registration, a customer receives an initial credit balance.
- A microservice receives the event for the initial balance, but it has not yet received the registration of the user.
- The microservice cannot credit the initial balance because the customer has not been created in the system.
- If the registration of the customer arrives, later on, the initial balance would have to be executed once again.

This problem can be solved when **the order of events can be guaranteed**. In this case, the problem will not arise in the first place. Unfortunately, many solutions cannot guarantee the order of events.

Event sourcing

Event sourcing can also help. This allows the microservice to always **reconstruct its state from the events**. Therefore, the state could be discarded and recreated from the events as long as those are available without any gaps.

Extending domain logic

It is also possible to extend the domain logic. In that case, if the event for the initial balance is received before the registration, a new customer object is created, but the object is marked as invalid as long as the data from the registration is missing.

The rest of the logic would need to handle these invalid customers. That **might make the business logic quite complex**.

An error might be hard to spot because now there are so many states that an

object might have. Therefore, **this solution should probably be avoided.**

QUIZ

1

A system cannot be asynchronous if _____ is completely unacceptable.

COMPLETED 0%

1 of 2



In the next lesson, we'll look at some other challenges.

Other Challenges

In this lesson, we'll continue to discuss the challenges that an asynchronous approach poses.

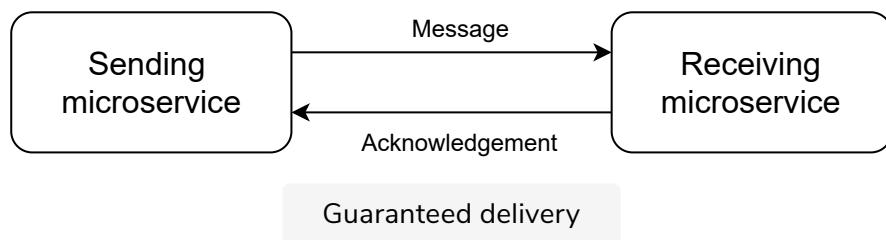
WE'LL COVER THE FOLLOWING ^

- Guaranteed delivery
- Idempotency
- One recipient
- Test

Guaranteed delivery

In an asynchronous system, the delivery of messages can be guaranteed if the system is appropriately implemented.

The sender has the messaging system confirm that it received the message. Afterwards, the messaging system has the recipient of the message acknowledge the receipt. However, if the recipient never picks up the data and thus prevents delivery, the sender has an acknowledgement, but the message still does not arrive at the recipient.



It is difficult to guarantee delivery when the recipient is anonymous. In this case, it is unclear who is supposed to receive the message and whether there are any recipients at all who should get the message. Therefore, it is also unclear who has to issue receipts.

Idempotency

If the messages are not acknowledged by the recipient, they are sent again. When the receiving microservice processes the message but is unable to acknowledge the message due to a problem or a failure, **the recipient receives the message a second time** although the recipient processed the message already.

This is an **at least once strategy**: the messages are sent at least once, and, in the described failure scenario, more often.

Therefore, one tries to design distributed systems in such a way that the microservices are **idempotent**. This means that a message can be processed more than once, but the state of the service no longer changes.

For example, when creating an invoice:

- The *invoice* microservice can first check in its own database whether an invoice has already been created.
- In this manner, an invoice is created only the first time the message is received.
- If the message is transferred again, it will be ignored.

One recipient

In addition, it can be necessary that only **one instance of a microservice processes a message**.

- For example, it would be incorrect from a domain perspective when multiple instances of the *invoice* microservice receive the order and all of them generate an invoice.
- This would generate multiple invoices.

For this, messaging systems normally have an option to send messages only to a single recipient. This recipient then has to confirm the message and process it. Such a communication type is called **point to point communication**.

Unfortunately, the rules for processing can be complex. When changes are made to customer data, parallel processing should be carried out as far as possible to ensure high performance.

However, changes to the data of a specific customer probably have to follow a sequence. For example, it would not be good if changes to the billing address are processed after the invoice has been written; the invoice would still contain the wrong address.

For this reason, it may be important to guarantee the order of messages.

Test

With asynchronous microservices, the **continuous delivery pipelines must be independent** to enable independent deployment.

To do this, **the testing of the microservices must be independent** of other microservices.

With asynchronous communication, a test can send a message to the microservice and check whether the system behaves as expected. **Timing can be difficult** because it is not clear when the microservice has processed the message and how long the test should wait for processing. The test can then check whether the microservice sends the correct messages in response.

This **allows very simple black box tests**, a test based on the interface without knowing about the internal structure of the microservice.

These tests do not place particularly high demands on the test environment. The environment just needs to be able to transmit messages. It is not necessary to install a large number of other microservices in the test environment. Instead, the messages that other microservices send or expect from the tested microservice can be the basis for the tests.

QUIZ

1

Consider the following scenario: Sending the same order details (identified by a unique order ID) multiple times to the shipping microservice results in **only one** instance of the order being shipped.

What concept is being demonstrated here?

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss some advantages and variations of asynchronous communication.

Advantages & Variations

In this lesson, we'll discuss advantages and variations to the approaches we've already discussed in this chapter.

WE'LL COVER THE FOLLOWING ^

- Advantages
- Variations

Advantages

Decoupling via events was presented in the lesson on [Events](#). Such an architecture achieves a high degree of decoupling.

Especially for distributed systems, asynchronous communication has a number of decisive advantages:

- When a communication partner fails, the message is sent later when the communication partner is available again. In this manner, asynchronous communication offers **resilience**, that is, protection against the failure of parts of the system.
- The processing and delivery of a message can nearly always be **guaranteed**. The messages are stored for a long time. Processing is assured, for example, by the recipients acknowledging the message.

In this manner, **asynchronous communication solves challenges caused by distributed systems**.

Variations

Let's discuss some variations you could apply to the techniques already discussed in this chapter.

The following two chapters introduce concrete technologies for implementing

- [Chapter 7](#) shows **Apache Kafka as an example for a message-oriented middleware (MOM)**. Kafka offers the option to store messages for a very long time. This can be helpful for event sourcing. This feature distinguishes Kafka from other MOMs which are also good options for microservices.
- [Chapter 8](#) demonstrates the implementation of asynchronous communication with **REST and the Atom data format**. This can be helpful when MOMs are too much of an effort as additional infrastructure.
- Asynchronous communication is easy to combine with **frontend integration** (see [chapter 3](#)) because these integrations focus on different levels: **frontend and logic**.
 - However, **inconsistencies easily occur during UI integration** when two microservices simultaneously present their state on one web page. When the microservices implement things of different domains, they use different data and therefore are rarely inconsistent.
- However, **a combination of asynchronous and synchronous communication (see section 9) should be avoided because** synchronous and asynchronous communication both start at the logic level.
 - However, even this combination might be sensible in special scenarios. For example, synchronous communication can be necessary if a response of a microservice is required immediately.

Q U I Z

1

Which of the following is NOT an advantage of asynchronous communication?

COMPLETED 0%

1 of 2



We'll look at a conclusion to this chapter in the next lesson.

Chapter Conclusion

In this lesson, we'll conclude this chapter with a quick summary of what we have learned.

WE'LL COVER THE FOLLOWING ^

- Summary

Summary

- **Asynchronous communication should be preferred over synchronous communication** between microservices due to the advantages concerning resilience and decoupling.
- **The only reason against this is inconsistency.** Therefore, it is important to know exactly what the requirements are, especially concerning consistency, in order to make the technically correct decision.
- Choosing asynchronous communication has **the potential to solve the essential challenges** of the microservices architecture and should, therefore, be considered in any case.

In the next chapter, we'll discuss Messaging and Kafka.

Introduction

In this lesson, we'll get a quick introduction to message-oriented middleware and a walkthrough of what the chapter holds for us.

WE'LL COVER THE FOLLOWING ^

- Message-oriented middleware (MOM)
- Chapter walkthrough

Message-oriented middleware (MOM)

This chapter shows the integration of microservices using a **message-oriented middleware (MOM)**. A MOM sends messages and ensures that they reach the recipient. MOMs are asynchronous, meaning that they do not implement request/reply as is done with synchronous communication protocols, they only send messages.

MOMs have different characteristics such as:

- **high reliability**
- **low latency**
- **high throughput**

MOMs also have a long history; they form the basis of numerous business-critical systems.

Chapter walkthrough

This chapter covers the following points:

- First, it gives an overview of the various MOMs and their differences. This allows readers to form an opinion on which MOM is most suitable for supporting their application.

- The introduction into Kafka shows why Kafka is especially well suited for a microservices system and how event sourcing (see [Events](#)) can be implemented with Kafka.
- Finally, the example in this chapter illustrates at the code level **how an event sourcing system with Kafka can be built in practice.**

Q U I Z

1

MOM stands for _____.

COMPLETED 0%

1 of 2



In the next lesson, we'll discuss message-oriented middleware in more detail.

Message-oriented Middleware (MOM)

In this lesson, we'll discuss message-oriented middleware in a bit more depth.

WE'LL COVER THE FOLLOWING



- Microservices & MOMs
- Variants of MOMs
 - Java messaging service
 - Advanced message queuing protocol
 - ZeroMQ
 - MQTT

Microservices & MOMs

Microservices are decoupled by a MOME. A microservice sends a message to or receives it from the MOME. This means that **the sender and the recipient do not know each other**, only the communication channel. Service discovery is therefore not necessary. Sender and recipient find each other via the topic or queue through which they exchange messages.

Load balancing is also easy. If several recipients have registered for the same communication channel, a message can be processed by one of the recipients and the load can be distributed, thereby eliminating the need for a specific infrastructure for load balancing.

However, a MOME is **a complex software that handles all communication**. Therefore, the MOME must be highly available and has to offer a high throughput. MOMEs are generally very mature products, but ensuring adequate performance under all conditions requires a lot of know-how, for example, concerning the configuration.

Variants of MOMEs

In the area of MOMs, the following products are popular.

Java messaging service

[JMS](#) (Java Messaging Service) is a standardized API for the programming language Java and part of the Java EE standard.

Well known implementations are [Apache ActiveMQ](#) or [IBM MQ](#), which was previously known as IBM MQSeries. However, many more [JMS products](#) are available. Java application servers that support the entire Java EE profile – not just the web profile – have to contain a JMS implementation, so that JMS is often anyway available.

Advanced message queuing protocol

[AMQP](#) (Advanced Message Queuing Protocol) does not standardize an API, but a network protocol at the level of TCP/IP. This allows for a simpler exchange of the implementation.



[RabbitMQ](#), [Apache ActiveMQ](#), and [Apache Qpid](#) are the best known implementations of the AMQP standard. There are also a lot more [implementations](#).

ZeroMQ

In addition, there is [ZeroMQ](#), which does not comply with any of the standards.

MQTT

Lastly, [MQTT](#) is a messaging protocol that plays a prominent role for the Internet of Things (IoT).

All of these MOM technologies can be used to build a microservices system. If a certain technology is already in use and knowledge about its use is readily available, a decision to use an already known technology can make a lot of sense. It takes a lot of effort to run a microservices system.

The **use of a well-known technology reduces risk and effort**. The requirements for availability and scalability of MOMs are high. A well-known MOM can help to meet these requirements in a simple way.

Q U I Z

1

When a MOM is used with a microservices system, the microservices communicate with _____.

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss the architecture of Kafka.

The Architecture of Kafka

In this lesson, we'll discuss the architecture of Kafka.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Kafka stores the message history
- Kafka: license and committers
- APIs
- Records
- Topics
- Partitions
- Example

Introduction

In the area of microservices, [Kafka](#) is an interesting option. In addition to typical features such as **high throughput and low latency**, Kafka can compensate for the failure of individual servers via **replication and can scale** with a larger number of servers.

Kafka stores the message history

Above all, Kafka is able to store an extensive message history. Usually, MOMs aim only to deliver messages to recipients. The MOM then deletes the message because it has left the MOM's area of responsibility, thus saving resources.

However, it also means that approaches such as event sourcing (see [Events](#)) are possible only if every microservice stores the event history itself. Kafka, on the other hand, can save records permanently. Kafka can also handle large amounts of data and can be distributed across multiple servers.

Kafka also has stream-processing capabilities. For this, applications receive the data records from Kafka, transform them, and send them back to Kafka.

Kafka: license and committers

Kafka is licensed under **Apache 2.0**. This license grants users extensive freedom.

The project is run by the **Apache Software Foundation**, which manages several open-source projects.

Many committers work for the company **Confluent**, which also offers commercial support, a Kafka Enterprise solution, and a solution in the cloud.

APIs

Kafka offers a separate API for each of the three different tasks of a MOM:

- The **producer API** serves to send data.
- The **consumer API** serves to receive data.
- Finally, the **streams API** serves to transform the data.

Kafka is written in Java. The APIs can be used with a language-neutral protocol. [Clients](#) for many programming languages are available.

Records

Kafka organizes data in **records**. This is what other MOMs call “messages”.

Records contain the transported data as a **value**. Kafka treats the value as a black box and does not interpret the data. In addition, records have a **key** and a **timestamp**.

A record could contain information about a new order or an update to an order. The key can then be composed of the identity of the record and information about whether the record is an update or a new order for example `update42` or `neworder42`.

Topics

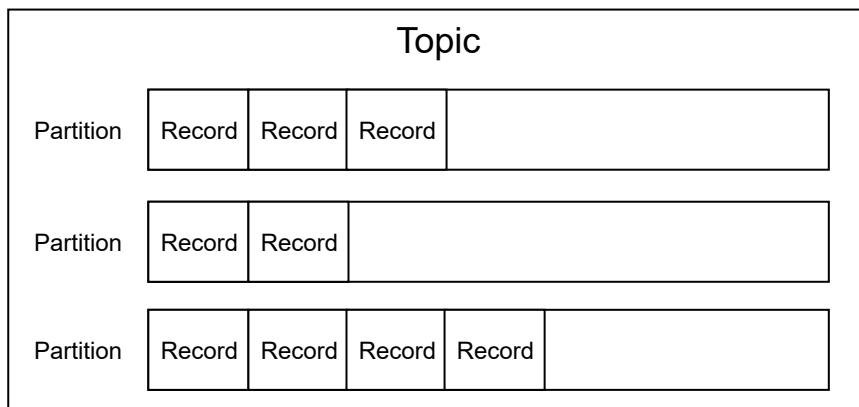
A **topic** is a named set of records. Producers send records to a topic and

consumers receive them from a topic.

If microservices in an e-commerce system are interested in new orders or want to inform other microservices about new orders, they could use a topic called “order.” New customers would be another topic which could be called “customer.”

Partitions

Topics are divided into **partitions**. Partitions allow strong guarantees concerning the order of records, but also parallel processing.



When a producer creates a new record, it is appended to a partition. Therefore, each record is stored in only one single partition.

Records are usually assigned to partitions by calculating the hash of the key of the record. However, a producer can also implement its own algorithm to assign records to a partition.

For each partition, **the order of the records is preserved**. That means the order in which the records are written to the partition is also the order in which consumers read the records. There is **no guarantee of order across partitions**. Therefore, partitions are also a concept for parallel processing: reading in a partition is linear. A consumer has to process each record in order. **Across partitions, processing can be parallel**.

More partitions have **different effects**. They allow more parallelism, but at a cost of higher overhead and resource consumption. It makes sense to have a considerable number of partitions, but not too many. Hundreds of partitions are typical.

Basically, a partition is just a file to which records are appended. Appending

Basically, a partition is just a file to which records are appended. Appending data is one of the most efficient operations on a mass storage device.

Moreover, such operations are very reliable and easy to implement. This makes the implementation of Kafka not too complex.

Example

To continue the example with the “order” topic: there might be a record with the key `neworder42` that contains an event about the order 42 that was just created and `updated42` which contains an update to the order 42.

With the default key algorithm, the keys would be hashed. The two records might, therefore, end up in different partitions and no order would be preserved. This is not ideal because the two events obviously need to be processed in the correct order. It makes no sense to process `updated42` before `neworder42`.

However, it is perfectly fine to process `updated42` and `updated21` because the orders probably do not depend on each other. The producer would need to implement an algorithm that sends the records with the keys `updated42` and `neworder42` to the same partition.

Q U I Z

1

What is the difference between a partition and a topic?

In the next lesson, we'll continue discussing the architecture of Kafka.

More on The Architecture of Kafka

In this lesson, we'll continue learning about the architecture of Kafka.

WE'LL COVER THE FOLLOWING



- Commit
- Polling
- Records, topics, partitions, and commits
- Replication
 - Example
- Leader and follower
- Retry sending

Commit

For each consumer, **Kafka stores the offset for each partition**. This offset indicates which record in the partition the consumer read and processed last. It helps Kafka to ensure that each record is eventually handled.

When consumers have processed a record, they commit a new offset. In this way, Kafka knows at all times which records have been processed by which consumer and which records still have to be processed. Of course, consumers can commit records before they are actually processed. As a result, **records that never get processed is a possibility.**

The commit is on an offset, for example, “all records up to record 10 in this partition have been processed.” **A consumer can commit a batch of records**, which results in better performance because fewer commits are required.

But then **duplicates can occur**. This happens when the consumer fails after processing a part of a batch and has not yet committed the entire batch. At restart, the application would read the complete batch again, because Kafka

restarts at the last committed record and thus at the beginning of the batch.

Kafka also supports **exactly once semantics** that is, a guaranteed one-time delivery.

Polling

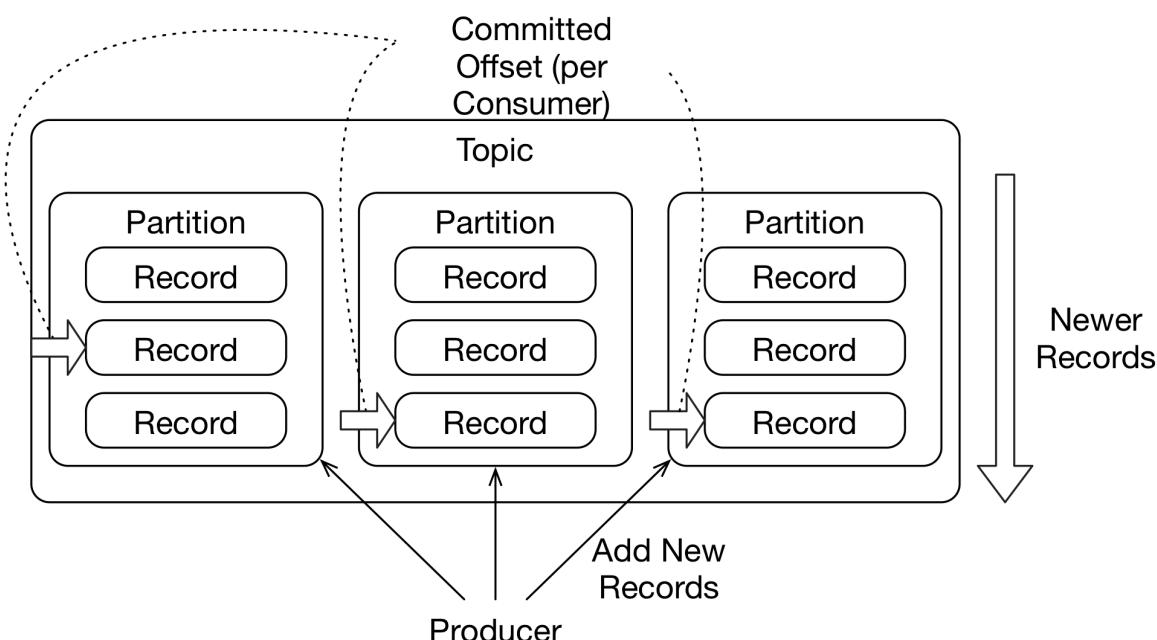
The consumers **poll** the data, meaning they fetch new data and process it.

With the push model, on the other hand, the producer would send the data to the consumer.

Polling doesn't seem to be very elegant. However, in the absence of a push, the consumers are protected from too much load when a large number of records are being sent and have to be processed. **Consumers can decide for themselves when they process the records.**

Libraries like Spring Kafka for Java, which is used in the example, poll new records in the background. The developer implements methods to handle new records. Spring Kafka then calls them. The polling is hidden from the developer.

Records, topics, partitions, and commits



The drawing above shows an example. The topic is divided into **three partitions**, each containing three records.

In the lower part of the figure are the newer records. The producer creates new records at the bottom. The consumer has not yet committed the latest record for the first partition but has for all other partitions.

Replication

Partitions store the data. Because data in one partition is independent of data in the other partitions, **partitions can be distributed over servers**:

- Each server then processes some partitions. This allows load balancing.
- To handle a larger load, new servers need to be added and some partitions need to be moved to the new server.
- The partitions can also be **replicated** so that the data is stored on several servers, meaning Kafka can be made fail-safe. If one server crashes or loses its data, other replicas still exist.

Example

- The number N of replicas can be configured. When writing, you can determine how many in-sync replicas must commit changes.
- With $N = 3$ replicas and two in-sync replicas, the cluster remains available even if one of the three replicas fails.
- Even if one server fails, new records can still be written to two replicas. If a replica fails, no data is lost because every write operation must have been successful on at least two replicas.
- Even if a replica is lost, the data must still be stored on at least one additional replica.

Kafka thus **supports some fine tuning** regarding the CAP theorem (see [Events](#)) by changing the number of replicas and in-sync replicas.

Leader and follower

The replication is implemented in such a way that one leader writes and the remaining replicas write as followers. The producer writes directly to the

remaining replicas write as follows. The producer writes directly to the leader. Several write operations can be combined in a batch.

On the one hand, it takes longer before a batch is complete and for the changes to be actually saved. On the other hand, throughput increases because it is more efficient to store multiple records at once.

The overhead of coordinating the writes happens just once for the full batch and not for each record.

Retry sending

If the transfer to the consumer was not successful, the producer can use the API to specify that the transfer is attempted again. The default setting is that sending a record is not repeated, thus causing **records to be lost**. If the transfer is configured to occur more than once, the record may already have been successfully transferred despite the error. In this case, **there would be a duplicate**, which the consumer would have to deal with.

One possibility is to develop the consumer in such a way that it offers **idempotent processing**. This means that the consumer is in the same state, no matter how often the consumer processes a record (see [Challenges](#)).

For example, if a duplicate is received, the consumer can determine that it has already modified the record accordingly and ignore it.

QUIZ

1

Suppose a Kafka consumer has committed that it has processed 20 records. However, it fails mid-processing. Which of the following statements best describes what happens next?

COMPLETED 0%

1 of 3



In the next lesson, we'll continue our discussion on the architecture of Kafka.

Even More on The Architecture of Kafka

In this lesson, we'll continue learning about the architecture of Kafka.

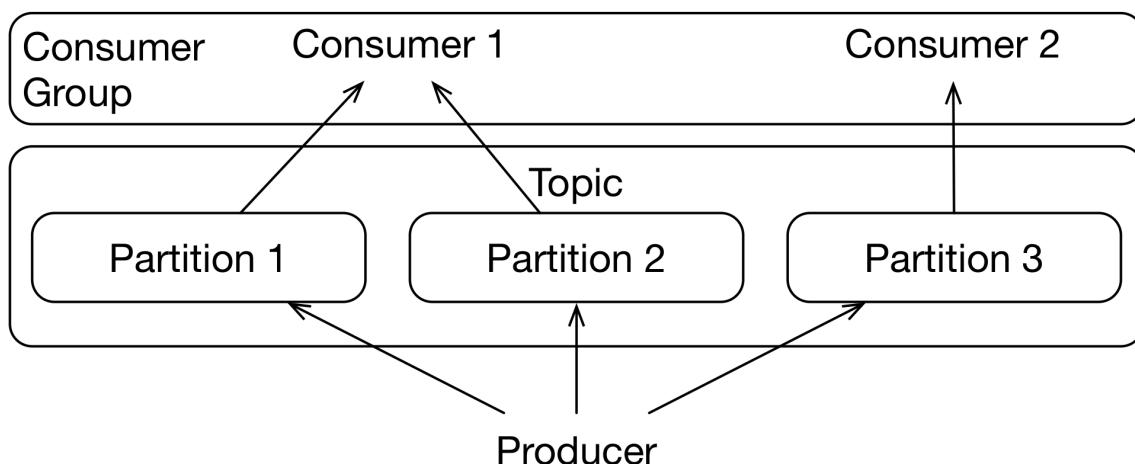
WE'LL COVER THE FOLLOWING ^

- Consumer groups
- Example
- Persistence
- Log compaction

Consumer groups

An event like `neworder42` should probably be processed only once by one instance of the invoicing microservice. Only one instance of a microservice should receive it, ensuring that only one invoice is written for this order. However, another instance of a microservice might work on `neworder21` in parallel.

Consumers are organized in consumer groups where each partition sends records to exactly one consumer in the consumer group. One consumer can be responsible for several partitions.



Thus, a consumer receives the messages of one or multiple partitions.

Example

The drawing above shows an example. Consumer 1 receives the messages of partitions 1 and 2 and consumer 2 receives the messages of partition 3.

The invoicing microservice instances could be organized in a consumer group, ensuring that only one instance of the invoicing microservice processes each record.

When a consumer receives a message from a partition, it will also later receive all messages from the same partition. The order of messages per partition is also preserved meaning that records in different partitions can be handled in parallel, and at the same time the sequence of records in a single partition is guaranteed.

Therefore, the instance of the invoicing microservice that receives `neworder42` would also receive `updated42` if those records are sent to the same partition. So, the instance would be responsible for all events about the order 42.

Of course, this applies only if the mapping of consumers to partitions remains stable. For example, if new consumers are added to the consumer group for scaling, the mapping can change.

The new consumer would need to handle at least one partition that was previously handled by a different consumer.

The maximum number of consumers in a consumer group is equal to the number of partitions, because each consumer must be responsible for at least one partition. Ideally, there are more partitions than consumers so that we can add more consumers when scaling.

Consumers are always members of a consumer group. Therefore, they receive records sent only to their partitions. If each consumer is to receive all records from all partitions, then there must be a separate consumer group for each consumer with only one member.

Persistence

Kafka is a mixture of a messaging system and data storage solution. The

records in the partitions can be read by consumers and written by producers.

The default retention for records is seven days, but it can be changed. The records can also be saved permanently where the consumers merely store their offset.

A new consumer can therefore process all records that have ever been written by a producer in order to update its own state.

If a consumer is too slow to handle all records in a timely manner, Kafka stores them for quite a long time allowing the consumer to process the records later to keep up.

Log compaction

However, this means that Kafka has to store more and more data over time. Some records, however, eventually become irrelevant. If a customer has moved several times, you may only want to keep the last information about the last move as a record in Kafka. **Log compaction is used for this purpose.**

All records with the same key are removed, except for the last one.

Therefore, the **choice of the key is very important** and must be considered from a domain logic point of view in order to have all the relevant records still available after log compaction.

Log compaction for the order topic would remove all events with the key **updated42** but the very last one. As a result, only the very last update to the order remains available in Kafka.

QUIZ

1

In a Kafka-based system, N consumers exist and M partitions exist.

Which of the following is NOT possible?

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss events with Kafka.

Events with Kafka

In this lesson, we'll discuss events with Kafka.

WE'LL COVER THE FOLLOWING



- Introduction
- Sending events
 - After action has taken place
 - Before data is changed
 - Send in a batch

Introduction

Systems that communicate via Kafka can easily exchange events (see also [Events](#)).

- Records can be saved permanently, and a consumer can read out the history and rebuild its state. The consumer does not have to store the data locally but can rely on Kafka.
- However, this means that all relevant information must be stored in the record. [Events](#) discussed the benefits and disadvantages of this approach.
- If an event becomes irrelevant due to a new event, the data can be deleted by Kafka's log compaction.
- Via consumer groups, Kafka can ensure that a single consumer handles each record. This simplifies matters, for example, when an invoice is to be written. In this case, only one consumer should write an invoice. It would be an error if several consumers were to create several invoices at the same time.

Sending events

The producer can send the events at different times.

After action has taken place

The simplest option is to send the event **after the actual action has taken place**.

The producer **first processes an order** before informing the other microservices about the order with an event. In this case, though, the producer could possibly change the data in the database and not send the event if, for example, it fails prior to sending the event.

Before data is changed

However, the producer can also send the events **before the data is actually changed**. When a new order arrives, the producer sends the event before modifying the data in the local database.

This doesn't make much sense though as events are information about an event that has already happened.

Finally, **an error may occur during the action**. If this happens, the event has already been sent, although the action never actually took place.

Sending events before the actual action also has the disadvantage that the **actual action is delayed**. First, an event is sent, which takes some time, and only after the event has been sent can the action be performed. The action is delayed by the time it takes to send the event. This can lead to a performance problem.

Send in a batch

It is also possible to collect the events in a local database and to **send them in a batch**.

In this case, writing the changed data and generating the data for the event in the database can take place in one transaction. The transaction can ensure that either the data is changed, and an event is created in the database, or neither takes place.

This solution also achieves **higher throughput** because batches can be used in Kafka to send several events to the database table at once.

However, the **latency is higher**; a change can be found in Kafka only after the next batch has been written.

Q U I Z

1

Suppose you are working on an app that involves extremely time sensitive events. Which of the discussed approaches would you use in that case?

COMPLETED 0%

1 of 2



In the next lesson, we'll look at a live example of Kafka!

Example: Introduction

In this lesson, we'll introduce a Kafka based coding example.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Data model for the communication
- Domain-driven design and strategic design
- Implementation of the communication

Introduction

The example in this section is based on the example for events from [Events](#) (see the drawing below).

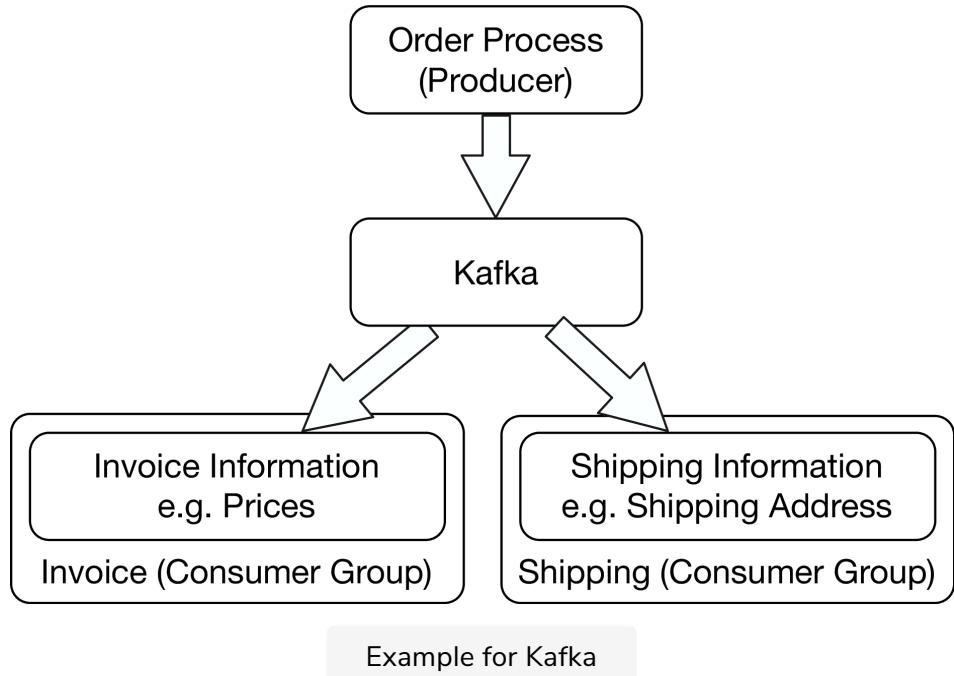
The microservice **microservice-kafka-order** is responsible for creating the order as it sends the orders to a Kafka topic. Therefore, the **microservice-kafka-order** is the **producer**.

Two microservices read the orders; the microservice **microservice-kafka-invoicing** issues an invoice for an order, and the microservice **microservice-kafka-shipping** delivers the ordered goods.

The two microservices are organized in **two consumer groups**. Each record is just consumed and processed by:

- one instance of the microservice **microservice-kafka-invoicing**
- one instance of **microservice-kafka-shipping**.

Data model for the communication



The two microservices **microservice-kafka-invoicing** and **microservice-kafka-shipping** require different information:

- The invoicing microservice requires the billing address and information about the prices of the ordered goods.
- The shipping microservice needs the delivery address but does not require prices.

Both microservices read the necessary information from the same Kafka topic and records. The only difference is what data they read from the records. Technically, this is easily done because the data about the orders is delivered as JSON. Thus, the two microservices can just ignore unneeded fields.

Domain-driven design and strategic design

In the example, the communication and conversion of the data are deliberately kept simple.

They implement the DDD pattern **published language**. There is a standardized data format from which all systems read the necessary data. With a large number of communication partners, the data model can become confusingly large.

In such a case, **customer/supplier** could be used. The teams responsible for shipping and invoicing dictate to the order team what data an order must

shipping and invoicing relate to the order team which takes care of order intact contain to allow shipping and invoicing. The order team then provides the

necessary data. The interfaces can even be separated, but this seems to be a step backwards.

After all, **published language** offers a common data structure that all microservices can use. In reality, however, it is a mixture of the two data sets that are needed by shipping, invoicing, and ordering.

Separating this one model into two models for the communication between invoicing and order or delivery and order makes it obvious which data is relevant for which microservice and makes it easier to assess the impact of changes. The two data models can be further developed independently of each other. This serves the goal of microservices to **make software easier to modify and to limit the effects of a change**.

The patterns **customer/supplier** and **published language** originate from the strategic design part of the domain-driven design (DDD). The lesson, [Events](#), also discusses what data should be contained in an event.

Implementation of the communication

Technically, communication is implemented as follows. The Java class `Order` from the project *microservice-kafka-order* is serialized in JSON. The classes `Invoice` from the project *microservice-kafka-invoicing* and `Shipping` from the project *microservice-kafka-shipping* get their data from this JSON. They ignore fields unrequired in the systems. The only exceptions are the `orderLines` from `Order`, which in `shipping` are called `shippingLines` and in `Invoice` are called `invoiceLine`. For the conversion, there is a `setOrderLine()` method in the two classes to deserialize the data from JSON.

QUIZ

1

What pattern for communication is used in the given example?

COMPLETED 0%

1 of 3



In the next lesson, we'll look at the data aspects of this example.

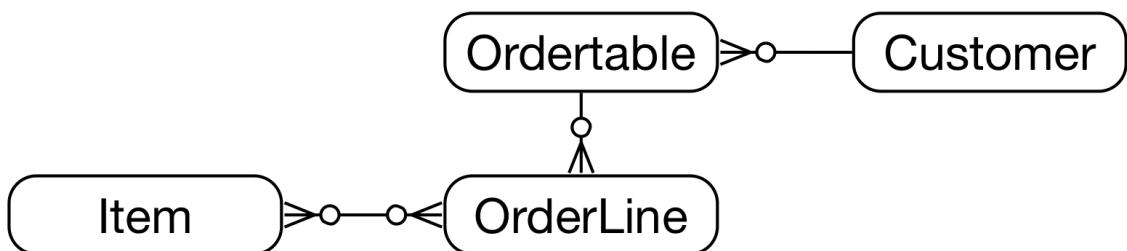
Example: Data Model

In this lesson, we'll look at the upcoming coding example from the perspective of its data model.

WE'LL COVER THE FOLLOWING ^

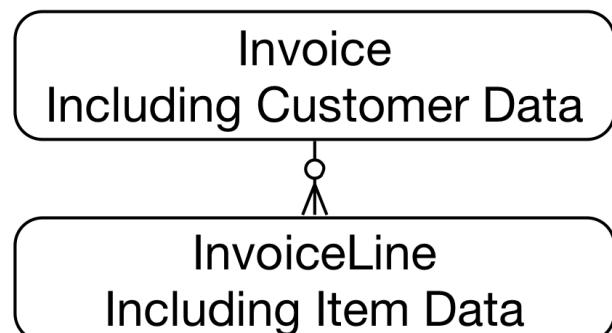
- Data model for the database
- Inconsistencies

Data model for the database



Data Model in the System `microservice-kafka-order`

The database of the order microservice (see the drawing above) contains a table for the orders (**Ordertable**) and the individual items in the orders (**OrderLine**). Goods (**Item**) and customers (**Customer**) also have their own tables.



Data Model in the System `microservice-kafka-order`

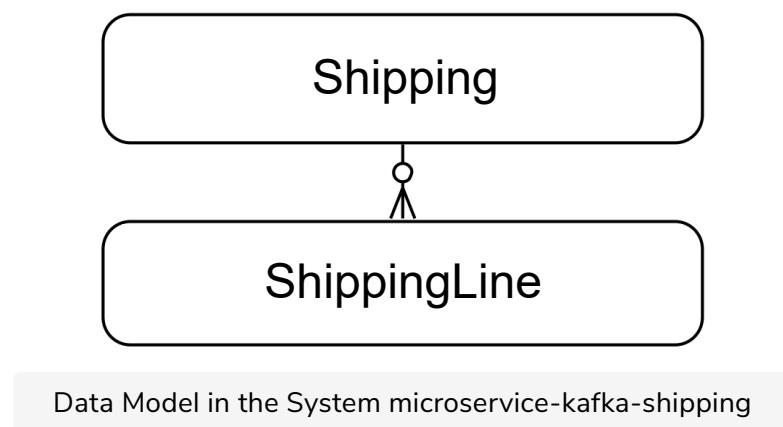
In the microservice **microservice-kafka-invoice**, the tables for customers and items are missing. The customer data is stored as part of **invoice**, and the

Items are missing. The customer data is stored as part of `Invoice`, and the item data as part of `invoiceLine` (see the drawing above). The data in the

tables are copies of the customers' and items' data at the time when the order was transferred to the system.

This means that **if a customer changes their data or a product changes its price, it does not affect the previous invoices**. That is correct from a domain logic perspective. After all, a price change should not affect invoices that have already been written.

Otherwise, getting the correct price and customer information at the time of the invoice can be implemented only with a complete history of the data, which is quite complex. With the model used here, it is **also very easy to transfer discounts or special offers to the invoice**. It is necessary to send a different price for a product.



Data Model in the System `microservice-kafka-shipping`

For the same reason, the microservice **microservice-kafka-shipping** has only the database tables `Shipping` and `ShippingLine`. Data for customers and items is copied to these tables so that the data is stored there as it was when the delivery was triggered.

This example illustrates how *bounded context* simplifies the domain models.

Inconsistencies

The example also shows another effect: the information in the system can be inconsistent. Orders without invoices or orders without deliveries can occur, but such conditions are not permanent. At some point the Kafka topic will be read out with the new orders, and the new orders will then generate an invoice and a delivery.

QUIZ

1

How many tables does the order microservice's database schema have?

COMPLETED 0%

1 of 3



In the next lesson, we'll look at this coding example in detail.

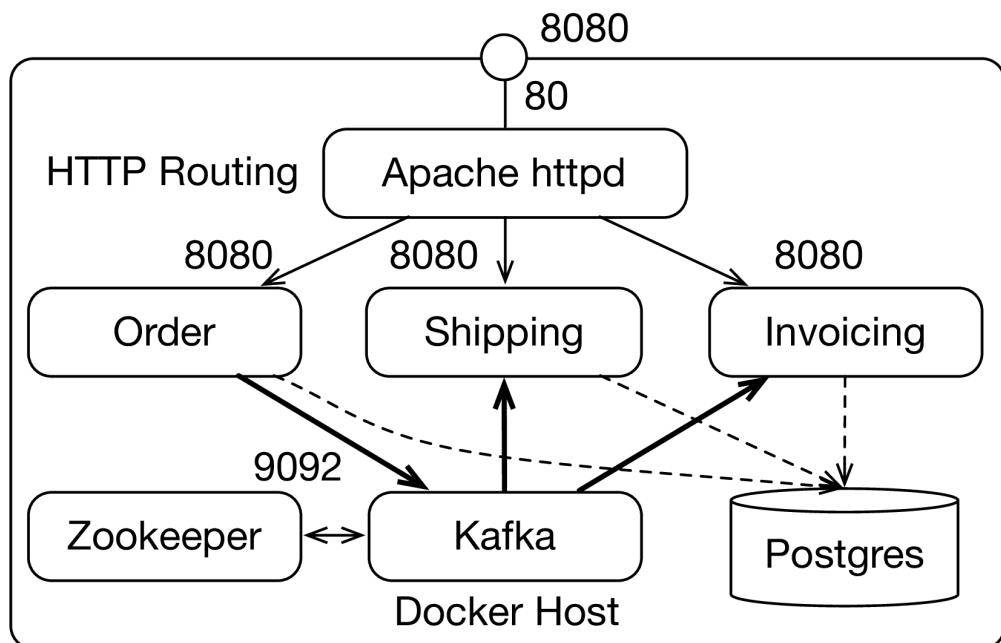
Example: Technical Structure & Live App

In this lesson, we'll look at a real live example of Kafka.

WE'LL COVER THE FOLLOWING

- Technical structure
- Apache httpd
- Zookeeper
- Kafka instance
- Postgres database
- Running the example locally
- Key for the records
- Implementing a custom partitioner
- Sending all information about the order in the record

Technical structure



The drawing above shows how the example is structured technically.

Apache httpd

The **Apache httpd** distributes incoming HTTP requests. Thus, there can be multiple instances of each microservice. This is useful for showing the distribution of records to multiple consumers.

In addition, only the Apache httpd server is accessible from the outside. The other microservices can be contacted only from inside the Docker network.

Zookeeper

Zookeeper serves to coordinate the Kafka instances and stores information about the distribution of topics and partitions. The example uses the image at <https://hub.docker.com/r/wurstmeister/zookeeper/>.

Kafka instance

The **Kafka instance** ensures the communication between the microservices. The order microservice sends the orders to the shipping and invoicing microservices. The example uses the Kafka image at <https://hub.docker.com/r/wurstmeister/kafka/>.

Postgres database

Finally, the order, shipping, and invoicing microservices use the same *Postgres database*. Within the database instance, each microservice has its own separate database schema.

Thus, the microservices are **completely independent** in regard to their database schemas. At the same time, one database instance can be enough to run all the microservices.

The alternative would be to **give each microservice its own database instance**. However, this would increase the number of Docker containers and would make the demo more complex.

Running the example locally

The example can be found at <https://github.com/ewolff/microservice-kafka>. To

start the example, you have to first download the code with `git clone`

<https://github.com/ewolff/microservice-kafka.git>. Afterwards, the command

`./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) has to be executed in the directory `microservice-kafka` to compile the code. See the [appendix](#) for more details on Maven and how to troubleshoot the build. Then `docker-compose build` has to be executed in the directory `docker` to generate the Docker images and `docker-compose up -d` for starting the environment. See the [appendix](#) for more details on Docker, Docker Compose and how to troubleshoot them. The Apache httpd load balancer is available at port 8080. If Docker runs locally, it can be found at <http://localhost:8080>. From there, you can use the order microservice to create an order. The microservices shipping and invoicing should display the order data after some time.

At <https://github.com/ewolff/microservice-kafka/blob/master/HOW-TO-RUN.md> extensive documentation can be found that explains installation and instructions for starting the example step by step.

Key for the records

Kafka transfers the data in records where each record contains an order. The key of the record is the order ID with the extension `created`, for example, `1created`. Just the order ID would not be enough. In case of a log compaction, all records with an identical key are deleted except for the last record. There can be different records for one order.

One record can be a result of the generation of a new order, and other records might be results of the different updates. Thus, the key should contain more than the order ID to keep all records belonging to an order during log compaction. When the key corresponds to the order ID, only the last record would be left after a log compaction.

However, this approach has the **disadvantage that records belonging to one order can end up in different partitions** and with different consumers because they have different keys. This means that, for example, records for the same order can be processed in parallel, which can cause errors.

Implementing a custom partitioner

To solve this problem, a function has to be implemented which assigns all

To solve this problem, a function has to be implemented which assigns **all records for one order to one partition**. A partition is processed by a single consumer, and the sequence of the messages within a partition is guaranteed. Thus, it is ensured that all messages located in the same partition for one order are processed by the same consumer in the correct sequence.

This function is called a [partitioner](#). Therefore, it is possible to write custom code for the distribution of records onto the partitions. This allows a producer to write all records that belong together from a domain perspective into the same partition and to have them processed by the same consumer although they have different keys.

Sending all information about the order in the record

A possible alternative would be to **use only the order ID as the key**. To avoid the problem with log compaction, it is possible to send the complete state of the order along with each record so that a consumer can reconstruct its state from the data in Kafka, although only the last record for an order remains after log compaction.

However, this requires a data model that contains all the data all consumers need. It takes a lot of effort to design such a data model, besides being complicated and difficult to maintain. It also contradicts the bounded context pattern somewhat, even though it can be considered a published language.

QUIZ

1

Why did we not create separate database instances for each microservice in the example above?

COMPLETED 0%

1 of 4



In the next lesson, we'll look at topics and partitions.

Example: Topics & Partitions

In this lesson, we'll study how our example is divided into topics and partitions.

WE'LL COVER THE FOLLOWING



- Technical parameters of the partitions and topics
- No replication in the example
- Producers
- Consumers
- Consumer groups

Technical parameters of the partitions and topics

#

The topic `order` contains the order records. Docker Compose configures the Kafka Docker container based on the environment variable `KAFKA_CREATE_TOPICS` in the file `docker-compose.yml` in such a way as to create the topic `order`.

The topic `order` is divided into **five partitions**, as a greater number of partitions allows for more concurrency. In the example scenario, it is not important to have a high degree of concurrency. More partitions require more file handles on the server and more memory on the client. When a Kafka node fails, it might be necessary to choose a new leader for each partition. This also takes longer when more partitions exist.

This **argues for a lower number of partitions** as used in the example in order to save resources. The number of partitions in a topic can still be changed after creating a topic.

However, in that case, the mapping of records to partitions will change. This can cause problems because then the assignment of records to consumers is changed. Therefore, when changing the number of partitions, care should be taken to ensure that the new mapping does not cause issues with consumer assignments.

not unambiguous anymore. Therefore, the number of partitions should be chosen sufficiently high from the start.

No replication in the example

For a production environment, a replication across multiple servers is necessary to compensate for the failure of individual servers. For a demo, the level of complexity required is not needed, so that only one Kafka node is running.

Producers

The order microservice has to send the information about the order to the other microservices. To do so, the microservice uses the `KafkaTemplate`. This class from the Spring Kafka framework encapsulates the producer API and facilitates the sending of records. Only the method `send()` has to be called. This is shown in the code piece from the class `OrderService` in the listing.

```
public Order order(Order order) {  
    if (order.getNumberofLines() == 0) {  
        throw new IllegalArgumentException("No order lines!");  
    }  
    order.setUpdated(new Date());  
    Order result = orderRepository.save(order);  
    fireOrderCreatedEvent(order);  
    return result;  
}  
  
private void fireOrderCreatedEvent(Order order) {  
    kafkaTemplate.send("order", order.getId() + "created", order);  
}
```

Behind the scenes, Spring Kafka converts the Java objects to JSON data with the help of the Jackson library. Additional configurations such as the configuration of the JSON serialization can be found in the file `application.properties` in the Java project. In `docker-compose.yml`, environment variables for Docker compose are defined, which are evaluated by Spring Kafka; these are the Kafka host and the port. Thus, with a change to `docker-compose.yml`, the configuration of the Docker container with the Kafka server can be changed and the producers can be adapted in such a way that they use the new Kafka host.

Consumers

The consumers are also configured in `docker-compose.yml` and with the

The consumers are also configured in `docker-compose.yml` and with the `application.properties` in the Java project. Spring Boot and Spring Kafka automatically build an infrastructure with multiple threads that read and process records. In the code, only a method is annotated with `@KafkaListener(topics = "order")` in the class `OrderKafkaListener`.

```
@KafkaListener(topics = "order")
public void order(Invoice invoice, Acknowledgment acknowledgment) {
    log.info("Received invoice " + invoice.getId());
    invoiceService.generateInvoice(invoice);
    acknowledgment.acknowledge();
}
```



One parameter of the method is a Java object that contains the data from the JSON in the Kafka record. During deserialization the data conversion takes place.

Invoicing and shipping read only the data they need; the remaining information is ignored. Of course, in a real system, it is possible to implement more complex logic rather than just filtering the relevant fields.

The other parameter of the method is of the type `Acknowledgement`. This allows the consumer to commit the record. When an error occurs, the code can prevent the acknowledgement. In this case, the record would be processed again.

The **data processing in the Kafka example is idempotent**. When a record is supposed to be processed, first the database is queried for data stemming from a previous processing of the same record. If the microservice finds such data, the record is obviously a duplicate and is not processed a second time.

Consumer groups

The setting `spring.kafka.consumer.group-id` in the file `application.properties` in the projects `microservice-kafka-invoicing` and `microservice-kafka-shipping` defines the consumer group to which the microservices belong. All instances of shipping or invoicing each form a consumer group. Exactly one instance of the shipping or invoicing microservice receives a record. This ensures that an order is not processed in parallel by multiple instances.

Using `docker-compose up --scale shipping=2`, more instances of the shipping microservice can be started. If you look at the log output of an instance with `docker logs -f mskafka_shipping_1`, you will see which partitions are assigned to this instance and that the assignment changes when additional instances are started. Similarly, you can see which instance processes a record when a new order is generated.

It is also possible to have a look at the content of the topic. To do so, you have to start a shell on the Kafka container with `docker exec -it mskafka_kafka_1 /bin/sh`. The command `kafka-console-consumer.sh --bootstrap-server kafka:9092 --topic order --from-beginning` shows the complete content of the topic. Because all the microservices belong to a consumer group and commit the processed records, they receive only the new records. However, a new consumer group would indeed process all records again.

Try these in the Kafka coding environment above!

QUIZ

1

What are the arguments for a **lower** number of partitions?

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss Kafka-based testing and other data formats.

Example: Testing & Other Data Formats

In this lesson, we'll discuss the testing with Kafka and the data format Avro.

WE'LL COVER THE FOLLOWING ^

- Tests with embedded Kafka
- Avro as data format

Tests with embedded Kafka

In a JUnit test, an *embedded Kafka server* can be used to analyze the functionality of the microservices. In this case, a Kafka server runs in the same Java Virtual Machine (JVM) as the test. Thus, it is not necessary to build up an infrastructure for the test, and consequently, there is no need to tear down the infrastructure again after the test.

This requires two things essentially:

- An embedded Kafka server has to be started. With a class rule, JUnit can be triggered to start a Kafka server prior to the tests and to shut it down again after the tests. Therefore, a variable with `@ClassRule` must be added to the code.

```
@ClassRule  
public static KafkaEmbedded embeddedKafka =  
    new KafkaEmbedded(1, true, "order");
```



- The Spring Boot configuration must be adapted in such a manner that Spring Boot uses the Kafka server. This code is found in a method annotated with `@BeforeClass`, so that it executes before the tests.

```
@BeforeClass  
public static void setUpBeforeClass() {  
    System.setProperty("spring.kafka.bootstrap-servers",
```



```
    embeddedKafka.getBrokersAsString());  
}
```

Avro as data format

Avro is a data format quite frequently used with Kafka and Big Data solutions from the Hadoop area. Avro is a binary protocol, but also offers a JSON-based representation. There are Avro libraries for Python, Java, C#, C++, and C.

Avro supports schemas, meaning that each dataset is saved or sent together with its schema. For optimization, a reference to a schema from the schema repository can be used rather than a copy of the complete schema. Thereby, it is clear which format the data has. The schema contains a documentation of the fields. This ensures the long-term interpretation of the data, and that the semantics of the data are clear.



In addition, the data can be converted to another format upon reading. This facilitates the [schema evolution](#). New fields can be added when default values are defined so that the old data can be converted into the new schema by using the default value. When fields are deleted, a default value can be given so that new data can be converted into the old schema. In addition, the order of the fields can be changed because the field names are stored.

An advantage of the flexibility associated with schema migration is that old records can be processed with current software and the current schema. Also, software based on an old schema can process new data. Message-oriented middleware (MOM) typically does not have such requirements because messages are not stored for very long. Only upon long-term record storage does schema evolution turn into a challenge.

QUIZ

Q

Lots of complex additional infrastructure is required to run tests with Kafka.

COMPLETED 0%

1 of 1



In the next lesson, we'll discuss some variations of the messaging and Kafka recipe.

Variations & Experiments

In this lesson, we'll look at some variations to the approach that we've learned in this chapter and some experiments you can do based on them.

WE'LL COVER THE FOLLOWING ^

- Recipe variations
- Other MOMs
- Atom
- Frontend integration
- Synchronous mechanisms
- Experiments

Recipe variations

The example sends the data for the event along in the records. There are alternatives to this (see [Example](#)):

- The **entire dataset is always sent along**, in this example that means the complete order.
- The records could contain only an ID of the dataset for the order. As a result, the **recipient can retrieve just the information about the dataset it really needs**.
- An individual topic exists for each client. All the **records have their own data structure** adapted to the client.

Other MOMs

An alternative to Kafka would be **another MOM**. This might be a good idea if the team has experience with a different MOM. Kafka differs from other MOMs in the long-term storing of records.

However, this is relevant only for event sourcing. And even then, every microservice can save the events itself. Therefore, storage in the MOM is not absolutely necessary. It can even be difficult because the question of the data model arises.

Atom

Likewise, asynchronous communication with Atom (see [chapter 8](#)) can be implemented.

In a microservices system, however, there should only be one solution for asynchronous communication so that the effort for building and maintaining the system does not become too great.

Therefore, using Atom and Kafka or any other MOM at the same time should be avoided.

Frontend integration

Kafka can be combined with frontend integration (see [chapter 3](#)).

Synchronous mechanisms

These approaches act at different levels so that a combined use does not represent a problem. A combination with synchronous mechanisms (see [chapter 9](#)) makes less sense because the microservices should communicate either synchronously or asynchronously.

Still, such a combination might be sensible in situations where synchronous communication is necessary.

Experiments

Try the following experiments in the coding environment below!

Supplement the system with an **additional microservice**.

- As an example, a microservice can be used that credits the customer with

- As an example, a microservice can be used that credits the customer with a bonus depending on the value of the order or counts the orders.
- Of course, you can copy and modify one of the existing microservices.
- Implement a Kafka consumer for the topic `order` of the Kafka server `kafka`. The consumer should credit the customer with a bonus when ordering or count the messages.
- In addition, the microservice should also present an HTML page with the information (customer bonuses or number of messages).
- Place the microservice into a Docker image and reference it in the `docker-compose.yml`. There you can also specify the name of the Docker container.
- In `docker-compose.yml`, create a link from the container `apache` to the container with the new service, and from the container with the new service to the container `kafka`.
- The microservice must be accessible from the homepage. To do this, you have to create a load balancer for the new Docker container in the file `000-default.conf` in the Docker container `apache`. Use the name of the Docker container, and then add a link to the new load balancer to the file `index.html`.
- It is possible to start additional instances of the shipping or invoicing microservice. This can be done with `docker-compose up -d --scale shipping=2` or `docker-compose up -d --scale invoicing=2`. `docker logs mskafka_invoicing_2` can be used to look at the logs. In the logs the microservice also indicates which Kafka partitions it processes.
- Kafka can also transform data with Kafka streams. Explore this technology by searching for information about it on the web!
- Currently, the example application uses JSON. Implement a serialization with `Avro`. A possible starting point for this can be <https://www.codenotfound.com/2017/03/spring-kafka-apache-avro-example.html>.
- Log compaction is a possibility to delete superfluous records from a topic. The [Kafka documentation](#) explains this feature. To activate log

compaction, it has to be switched on when the topic is generated. See also

<https://hub.docker.com/r/wurstmeister/kafka/>. Change the example in such a way that log compaction is activated.

QUIZ

1

What are the reasons for using a MOM *other* than Kafka?

COMPLETED 0%

1 of 2



We'll look at a quick chapter conclusion in the next lesson!

Chapter Conclusion

We'll conclude this chapter with a quick summary of what we've learned.

WE'LL COVER THE FOLLOWING ^

- Benefits
- Challenges

Kafka offers an interesting **approach for the asynchronous communication between microservices**.

Benefits

- Kafka can **store records permanently**, which facilitates the use of event sourcing in some scenarios. In addition, approaches like Avro are available for solving problems like schema evolution.
- The **overhead for the consumers is low**. They have to remember only the position in each partition.
- With partitions, Kafka has a concept for parallel processing and, with consumer groups, it has a concept to guarantee the order of records for consumers. In this way, Kafka can **guarantee delivery** to a consumer and at the same time **distribute the work among several consumers**.
- Kafka can **guarantee the delivery of messages**. The consumer commits the records it has successfully processed. If an error occurs, it does not commit the record and tries to process it again.

Therefore, it is worth taking a look at this technology, especially for microservices, even if other asynchronous communication mechanisms are also useful.

Challenges

Challenges

However, Kafka also provides some challenges.

- Kafka is an additional infrastructure component. It must be operated. Especially with messaging solutions, **the configuration is often not easy**.
 - If Kafka is used as event storage, the records must contain all the data that all clients need. This is often **not easy to implement** because of the bounded context.
-

In the next chapter, we'll discuss asynchronous communication with Atom and REST.

Introduction

In this lesson, we'll look at a quick walkthrough of what this chapter holds for us.

WE'LL COVER THE FOLLOWING ^

- Chapter walkthrough

This chapter deals with the integration of microservices based on the **Atom data format**. An [example](#) is provided which uses a simple scenario to illustrate what integration with the help of Atom and REST can look like.

Chapter walkthrough

The readers will learn:

- **What** the Atom format is, **how** it functions, and **how it can be exploited** for the asynchronous communication of microservices.
- **What alternatives to Atom** exist for the asynchronous communication via REST/HTTP and what **advantages and disadvantages** the different formats have.
- How **HTTP and REST can be efficiently used**, not only for asynchronous communication.
- What an implementation of an **asynchronous system based on HTTP, REST, and Atom can look like**.

In the next lesson, we'll discuss the Atom format.

The Atom Format

In this lesson, we'll look at the Atom format.

WE'LL COVER THE FOLLOWING ^

- Introduction
- MIME Type
- Feed
 - Meta data
 - Optional meta data
- Entry

Introduction

Atom is a data format originally developed to **make blogs accessible to readers**. An Atom document contains blog posts that subscribers can read in chronological order. Besides blogs, Atom can also be used for **podcasts**.

Because the protocol is so flexible, it is **also suitable for other types of data**.

- For example, a microservice can offer information about **new orders** to other microservices as an **Atom feed**. This corresponds to the REST approach, which uses established web protocols such as HTTP for the integration of applications.

Atom is not a protocol, but a **data format**.

A GET request to a new URL such as <http://innoq.com/order/feed> can return a document with orders in the Atom format. This document can contain links to the details of the orders.

MIME T

MIME Type

HTTP-based communication indicates the type of content with the help of **MIME types (Multipurpose Internet Mail Extensions)**.

The MIME type for Atom is `application/atom+xml` .

Thanks to content negotiation, other data representation can be offered in addition to Atom for the same URL.

Content negotiation is built into HTTP. The HTTP client signals via an accept header what data formats it can process. The server then sends a response in a suitable format. Thus, accept headers enable a client to request all orders as JSON or the last changes as an Atom feed under the same URL.

Feed

```
<?xml version="1.0" encoding="UTF 8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Order</title>
  <link rel="self" href="http://localhost:8080/feed" />
  <author>
    <name>Big Money Online Commerce Inc.</name>
  </author>
  <subtitle>List of all orders</subtitle>
  <id>tag:ewolff.com/microservice-atom/order</id>
  <updated>2017 04 20T15:28:50Z</updated>
  <entry>
    ...
  </entry>
</feed>
```



A document in Atom format is called an **Atom feed**.

An Atom feed contains different components.

Meta data

First, the feed defines **meta data**. The Atom standard defines that this meta data has to comprise a number of elements.

- `id` is a globally unambiguous and a permanent URI (Uniform Resource Identifier) to identify the feed. In the list, this is the first item.

Identifier). It has to identify the feed. In the listing, this is done via a [tag URI](#).

- `title` is the title of a feed in a human-readable form.
- `updated` contains the point in time when a feed has been last changed. This information is important for finding out whether new data exists.
- There also has to be an `author` including `name`, `email`, and `uri`. This is very helpful for blogs.
 - However, it **does not seem to make sense for Atom feeds**, which only transport data. However, it can be useful for indicating a contact person in case of problems.
- Multiple `link` elements are recommended.
 - Each element has an attribute called `rel` for indicating the relationship between the feed and the link.
 - The attribute `href` contains the actual link.
 - A `link` can be used to provide a link to a HTML representation of the data.
 - In addition, the feed can use a `self` link to indicate at which URL it is available.

Optional meta data

Additional optional meta data includes:

- `category` narrows down the topic area of the feed, for example, to a field such as sports.
- Analogous to `author`, `contributor` contains information about people who contribute to the feed.
- `generator` indicates the software that produced the feed.
- `icon` is a small icon, whereas `logo` represents a larger icon. This makes it possible to represent a blog or podcast on the desktop. It is not that useful for a data feed.
- `rights` define things, for example, the copyright. This element is likewise the interpretation of the `rights` element.

- Finally, `subtitle` is a human-readable description of the feed.

As the listing illustrates, the fields `id`, `title`, and `updated` are enough for an Atom feed with data. Having a `subtitle` for documentation is helpful. There should also be a `link` for documenting the URL of the feed. All other elements are not really needed.

Entry

In the previous example, the most important thing is missing, the feed entries. Such an entry adheres to the following format:

```
<entry>
  <title>Order 1</title>
  <id>tag:ewolff.com/microservice-atom/order/1</id>
  <updated>2017 04 20T15:27:58Z</updated>
  <content type="application/json"
    src="http://localhost:8080/order/1" />
  <summary>This is the order 1</summary>
</entry>
```



An entry contained in the feed must consist of the following data according to the Atom standard:

- `id` is the globally unambiguous ID as URI. Thus, there cannot be a second entry with the id `tag:ewolff.com/microservice-atom/order/1` in this feed. This URI is a `tag URI` meant for globally unambiguous identifiers.
- `title` is a human-readable title for the entry.
- `updated` is the timepoint when the entry has last been changed. It has to be set so that the client can decide whether it already knows the last state of a certain entry.

In addition, the following elements are recommended:

- As described for the feed, `author` names the author of the entry.
- `link` can contain links, for example, `alternate` as a link to the actual entry.
- `summary` is a summary of the content. It has to be provided if the content

- `summary` is a summary of the content. It has to be provided if the content is only accessible via a link or if it does not have an XML media type. This is the case in the example.
- `content` can be the complete content of the entries or a link to the actual content. To enable access to the data of the entry, either `content` has to be offered, or a `link` with `alternate`.
- In addition, `category` and `contributor` are optional, analogous to the respective elements of the feed. `published` can indicate the first date of publication, the element rights can state the rights, and source can name the original source if the entry was a copy from another feed.

In the example, the elements `id`, `title`, `summary`, and `updated` are filled. Access to the data is possible via a link in `content`. The data could also directly be contained in the `content` element in the document. However, in that case, the document would be very large.

Thanks to the links the document remains small. Each client has to access the additional data via links, and can limit itself to only the relevant data for a respective client.

Q U I Z

1

Atom is _____.

In the next lesson, we'll look at Atom caching.

Atom Caching

In this lesson, we'll study Atom caching.

WE'LL COVER THE FOLLOWING ^

- Tools
- Efficient polling of Atom feeds
- HTTP caching
 - Data unchanged
 - Data has changed
 - Static Atom feed
- ETags
- Pagination and filtering
- Push vs. pull
- Guaranteed delivery
- Old events

Tools

At https://validator.w3.org/feed/#validate_by_input, a validator is provided that can check whether an Atom feed is valid, checking to see if all necessary elements are present.

There are systems such as [Atom Hopper](#) that offer a server with the Atom format. In this way, an application does not have to generate Atom data, but can post new data to the server. The server then converts the information into Atom. Clients can fetch the Atom data from the server.

Efficient polling of Atom feeds

Asynchronous communication with Atom requires that the client regularly requests the data from the server and processes new data. This is called **polling**.

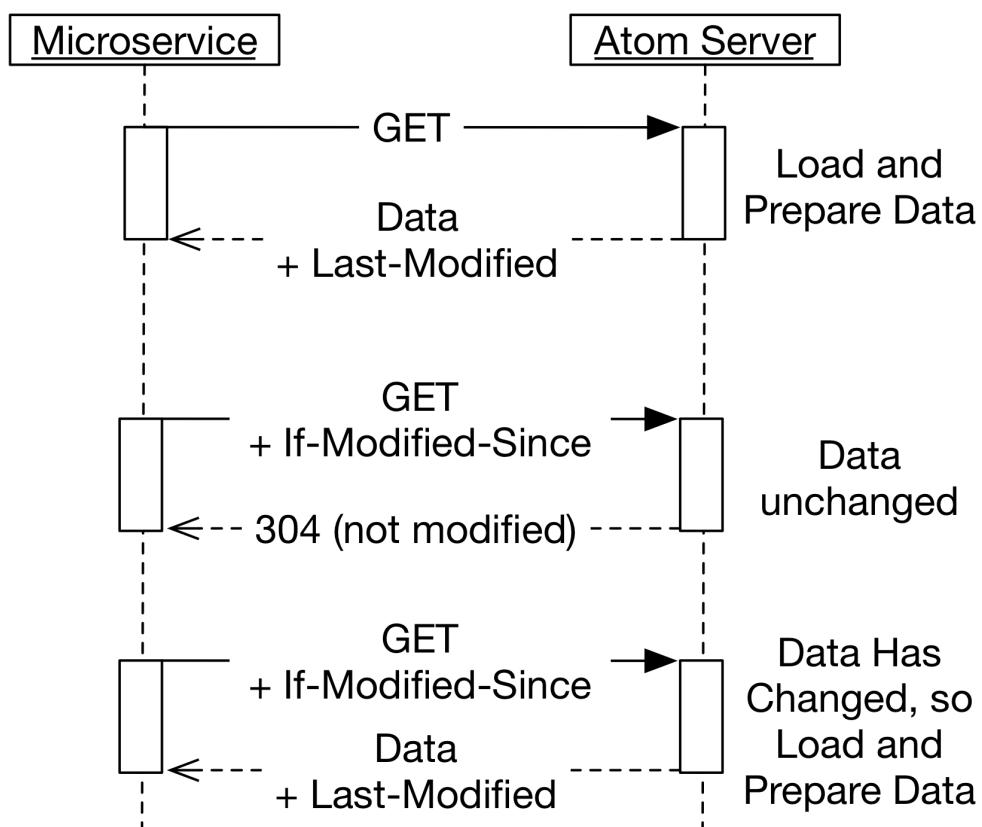
The client can periodically read out the feed and check the `updated` field in the feed to see if the data has changed. If this is the case, the client can use the `updated` elements of the entries to find out which entries are new and process them.

This is very time consuming because the entire feed has to be generated and transmitted. Additionally, only a few entries are read from the feed, most requests will show that there are actually no new entries. For this purpose, it does not make sense to create and transfer the complete feed.

HTTP caching

A very simple way to solve this problem is **HTTP caching**, see the drawing below.

HTTP provides a header with the name `Last-Modified` in the HTTP response. This header indicates when the data was last changed. This header takes over the function of the `updated` field from the feed.



Data unchanged

- The client stores the value of this header. On the next HTTP GET request, the client sends the read value in the `If-Modified-Since` header with the GET request.
- The server can now respond with an HTTP status of `304 (not modified)` if the data has not changed. Then, no data is transferred except for the status code.

Whether data has changed can often be determined very easily. For example, you can implement code that determines the time of the last change in the database. This is much **more efficient than converting all data into an Atom representation**.

Data has changed

- If the data has actually changed, the server responds normally with a status of `200 (OK)`.
- Also, a new value is sent in the `Last-Modified` header so that the client can use HTTP caching again.

The demo implements such an approach:

1. For a request with a set `If-Modified-Since` header, the database is used to determine the time of the last change.
2. This is compared with the value from the header.
3. An HTTP status `304` is returned if no data has changed.
4. In this case, only one database query is required to respond to the HTTP request.

Static Atom feed

An alternative would be to **create the Atom feed once** and store it on a web server as a **static resource**. In this case, dynamic generation is performed once. This approach also works efficiently for very large feeds.

In that case, it would be up to the web server to implement the HTTP caching for the static resource.

ETags

Another approach would be HTTP caching with **ETags**.

- Here, the **server returns an ETag** together with the data.
- The **ETag can be compared to a version number or checksum**.
- For any further requests, the **client sends the ETag along**.
- The **server uses the ETag to determine whether the data has changed and provides data only if new data is available**.

In the example, however, it is much easier to find out whether new orders have been accepted since a certain point in time. It therefore does not make sense to use ETags in the example.

Pagination and filtering

If a client is not interested in all events, it can signal this by setting parameters in the URL.

- This makes it possible to **paginate**. For example, with a URL like <http://ewolff.com/order?from=23&to=42>.
- The events can be filtered as well: <http://ewolff.com/order?name=Wolff>.

Of course, **pagination and filtering can be combined with caching**.

However, if each client uses its own pagination and filters, caching on the server-side can become inefficient, because too much different data has to be stored for the different clients.

Therefore, it may be necessary to **restrict the pagination and filtering options** in order to increase the efficiency of the cache.

Push vs. pull

HTTP optimizations such as conditional GETs can significantly speed up communication.

- But they remain **pull mechanisms**, the client queries the server.
- In case of a **push mechanism** the client is actively notified by the server about changes.

The push model seems to be more efficient but pulls have the advantage that the client requests new data when it can actually process it. This prevents the

the client requests new data when it can actually process it. This prevents the client from handling requests if it is under too much load.

Guaranteed delivery

Atom via HTTP cannot guarantee the delivery of the data. The server only provides the data. Whether it will be read at all is an open question.

Monitoring can help to identify problems and remedy them if necessary, so it would be very unusual if the Atom resources are never needed.

However, the HTTP protocol does not have any measures for issuing receipt acknowledgements.

Old events

In principle, an **Atom feed can contain all events that have ever occurred.** As mentioned in [Events](#), this might be interesting for **event sourcing**, in which case a microservice can reconstruct its state by processing all old events again.

If the data on old events is already stored in a microservice, the microservice needs only to make it available.

- For example, if a service already contains all orders, it can offer this information additionally as an Atom feed if necessary.
- In this case, no additional storage of old events is necessary. Thus, the effort to make the old events accessible is very low.

QUIZ

1

What is an advantage of using a pull model?

COMPLETED 0%

1 of 2



In the next lesson, we'll look at Atom in action!

Example

In this lesson, we'll look at an example of asynchronous communication with Atom.

WE'LL COVER THE FOLLOWING



- Introduction
- Running the example
- Implementation of the Atom view
- Implementation of the controller
- Implementation of HTTP caching on the server
- Implementation of HTTP Caching on the client
- Data processing and scaling
- Atom cannot send data to a single recipient

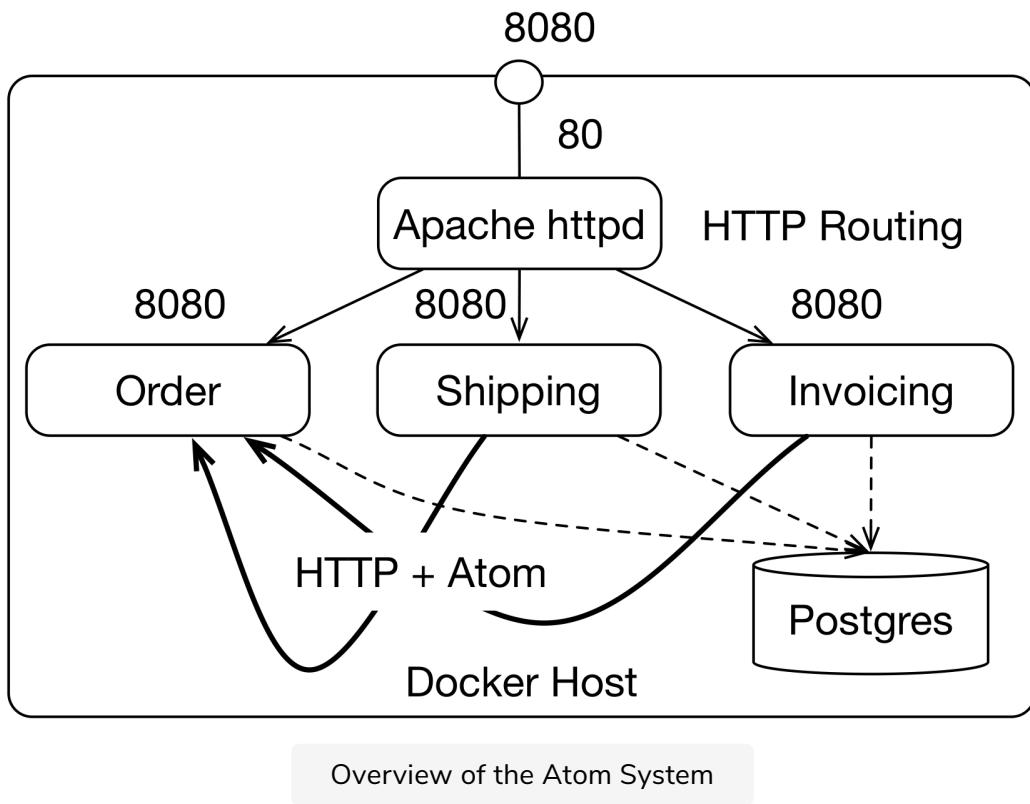
Introduction

The example can be found at <https://github.com/ewolff/microservice-atom>.

The example for Atom is analogous to the [example](#) in the Kafka chapter and is based on the example for [events](#) from chapter 6.

- The **ordering** system creates orders.
- Based on the data in the order, the **invoicing** microservice creates invoices.
- The **shipping** microservice creates deliveries.

The data models and database schemas are identical to the Kafka example. Only the **communication is now done via Atom**.



Overview of the Atom System

The drawing above shows how the example is structured:

- The **Apache httpd** distributes calls to the microservices.
 - For this purpose, the Apache httpd uses **Docker Compose** service links.
 - Docker compose offers simple **load balancing**. The Apache httpd uses the load balancing of Docker compose to forward external calls to one of the microservice instances.
- The **order microservice** offers an Atom feed from which the invoicing and shipping microservice can read the information about new orders.
- All microservices use the same **Postgres database**.
 - Within the database, each microservice has its own separate database schema. Thus, the microservices are completely independent regarding the database schema.
 - At the same time, **one database instance is enough to run all microservices**.

Running the example

To start the environment, press `run`.

The Apache httpd load balancer will be available at port 8080, at the link generated below such as: <https://x6jr4kg.educative.run/>. From there you can use the other microservices to add new orders that will eventually also appear in the invoicing and shipping microservice.

You can explore the code running Linux commands such as `ls` (list files in the directory), `cd` (change directory), `cat` (print contents of file), and `pwd` (print working directory) in the given terminal. You can even edit and create more files with `nano` and `vim`.

<https://github.com/ewolff/microservice-atom/blob/master/HOW-TO-RUN.md> describes the necessary steps in detail for running the example locally.

Implementation of the Atom view

The class `OrderAtomFeedView` in the project `microservice-atom-order` implements the Atom feed as a view with the framework Spring MVC.

Spring MVC splits the system into MVC (**model, view, controller**).

- The **controller** contains the **logic**.
- The **model** contains the **data**.
- The **view displays** the data from the model.
 - Spring MVC offers support for a plethora of view technologies for HTML.
 - For Atom, Spring uses the [Rome Library](#). It offers different Java objects to display data as feeds and entries in the feeds.

Implementation of the controller

```
public ModelAndView orderFeed(WebRequest webRequest) {  
    if ((orderRepository.lastUpdate() != null)  
        && (webRequest.checkNotModified(orderRepository.lastUpdate().getTime()))) {  
        return null;  
    }  
    return new ModelAndView(new OrderAtomFeedView(orderRepository),  
        "orders", orderRepository.findAll());  
}
```

- **(Line 1):** The method `orderFeed()` in the class `OrderController` is responsible for displaying the Atom feed with the help of `OrderAtomFeedView`.
- **(Line 6):** As shown in the listing, `OrderAtomFeedView` is returned as the view and a list of the orders as the model. The view then displays the orders from the model in the feed.

Implementation of HTTP caching on the server

- Spring provides the `checkNotModified()` method in the `WebRequest` class to simplify the handling of HTTP caching. The time of the last update is passed to the method.
- The `lastUpdate()` method of the class `OrderRepository` determines this time point with a database query. Each order contains the time at which it was placed and `lastUpdate()` returns the maximum value.
- `checkNotModified()` compares this passed value with the value from the `If-Modified-Since` header in the request.
 - If no more recent data needs to be returned, the method returns `true`.
- Then, `orderFeed()` returns `null`, so that Spring MVC returns an HTTP status code 304 (Not Modified).

In this case, the server makes a simple query to the database and returns an HTTP response with a status code; it does not provide any data.

Implementation of HTTP Caching on the client

On the client-side, HTTP caching must also be implemented.

- **(line 1):** The microservices `microservice-order-invoicing` and `microservice-order-shipping` implement the polling of the Atom feed in the method `pollInternal()` of the class `OrderPoller`.
- **(lines 4/5):** They set the `If-Modified-Since` header in the request. The value is determined from the variable `lastModified`.

- **(lines 15-17)**: It contains the value of the `Last-Modified` header of the last HTTP response. If no data has been changed in the meantime, the server responds to the GET request directly with an HTTP status `304` and it is clear that no new data exists.
- **(line 11)** Accordingly, data is processed only if the status is not `NOT_MODIFIED`.

```
public void pollInternal() {
    HttpHeaders requestHeaders = new HttpHeaders();
    if (lastModified != null) {
        requestHeaders.set(HttpHeaders.IF_MODIFIED_SINCE,
            DateUtils.formatDate(lastModified));
    }
    HttpEntity<?> requestEntity = new HttpEntity(requestHeaders);
    ResponseEntity<Feed> response =
        restTemplate.exchange(url, HttpMethod.GET, requestEntity, Feed.class);

    if (response.getStatusCode() != HttpStatus.NOT_MODIFIED) {
        Feed feed = response.getBody();
        ... // evaluate feed data
        if (response.getHeaders().getFirst(HttpHeaders.LAST_MODIFIED) != null) {
            lastModified =
                DateUtils.parseDate(
                    response.getHeaders().getFirst(HttpHeaders.LAST_MODIFIED)));
        }
    }
}
```



- `pollInternal()` is called by the method `poll()` in the class `OrderPoller`. The user can call this method with a button in the web UI.
- In addition, the microservice calls the method every 30 seconds because of the `@Scheduled` annotation.

Data processing and scaling

If there are **multiple instances** of invoicing and shipping microservices, **each instance polls the Atom feed and processes the data**.

Of course, it is not correct that several instances write an invoice for an order or initiate a delivery because then an order would create multiple invoices or deliveries.

Therefore, **each instance must determine what orders are already processed** and what data is in the database. If another instance has already

processed and what data is in the database. If another instance has already

created the data record for the invoice or delivery of the order, then the entry from the Atom feed for the order must be ignored.

To do this, `ShippingService` and `InvoicingService` use a transaction in which a data record is first searched for in the database. **A new data record is written only if none yet exists.** Therefore, only one instance of the microservices can write the data record.

All others read the data and find out that another instance has already written a data record. With a very large number of instances, this can cause a considerable load on the database. In return, the services are **idempotent**. No matter how often they are called, the state in the database, in the end, is always the same.

Atom cannot send data to a single recipient

This is a disadvantage of Atom. It is not easy to send a message to exactly one instance of a microservice. Instead, the **application has to deal with multiple instances reading the message from the Atom feed.**

Thus, especially when a lot of point-to-point communication is necessary, the Atom approach can be disadvantageous.

The **application must also be able to deal with messages not being processed:**

- If a message has been read, the process can fail before the message has been processed.
- As a result, no data might be written for some messages.
- In this case, however, another instance of the microservice would eventually read the message and process it, so **retries are actually quite easy to implement with Atom.**

In the next lesson, we'll discuss some recipe variations and experiments you can do based on them.

Variations

In this lesson, we'll look at some variations and experiments that can be done on the approaches discussed in the previous lesson.

WE'LL COVER THE FOLLOWING ^

- Rich site summary
- JSON feed
- Custom data format
 - Disadvantages
- Alternatives to HTTP
- Including event data

Instead of Atom, a different format can be used for the communication of changes via HTTP.

Rich site summary

Atom is just one format for feeds. An alternative is [RSS](#).

- There are different versions of RSS.
- RSS is older than Atom.
- Atom has learned from RSS and represents the more modern alternative.
- Blogs and podcasts should offer feeds as RSS and Atom to reach as many clients as possible.

Because server and client are under the control of the same developer in case of microservices, it is **not necessary to support multiple protocols**.

Therefore, **Atom is a better and more modern alternative**.

JSON feed

Another alternative is **JSON Feed** as it defines a data format for a feed.

However, it **uses JSON rather than XML**, which is used by Atom and RSS.

Custom data format

Atom and RSS are only formats for communicating changes. Some of the elements are not useful for data, but only for blogs or podcasts. The useful part of Atom and RSS is the list of changes and the links to the actual data. Atom and RSS, therefore, use hypermedia to communicate the changes without delivering all data.

Of course, it's feasible to **define your own data format**, which contains the changes and links to the data. In addition to the links, the data can be embedded directly into the document.

Disadvantages

Compared to Atom and RSS, a custom data format has the disadvantage in that it is **not standardized**.

- For a standardized data model, libraries are available.
- Learning about the format is easier.
- There are tools such as validators in which a user can read and display Atom data with an Atom reader, which is useful for troubleshooting.

However, the data to be transported is not very complex, and so a custom data format has **no major disadvantages**.

In essence, even with Atom, the approach simply uses hypermedia as an essential component of REST. It provides a list of links that clients can use to get more information about the changes to the data. This procedure can also be easily implemented with a custom data format.

Alternatives to HTTP

HTTP supports features such as scaling or reliability very well. Most applications already use HTTP without Atom to deliver web pages or provide REST services.

The alternative to HTTP would be a messaging system such as **Kafka** (see [chapter 7](#)), which can also be used to implement asynchronous communication.

- These messaging systems must be **scalable** and have to provide **high availability**.
- The messaging systems offer these features in principle but must be tuned and configured accordingly.
- This is especially challenging if you have never operated such a messaging system before.
- In particular, the advantages of HTTP concerning operation argue for using this protocol also for asynchronous communication.

Including event data

Of course, the feed can also **include the event data rather than just links**.

- In this way, a **client can work with the data without further requests** to load the linked data; but the feed gets bigger.
- The question also arises as to **what data should be included in the feed**. Each client may need different data, which makes it difficult to model the data.
- Sending links has the advantage that the **client can select the appropriate representation** of the data with content negotiation.

QUIZ

COMPLETED 0%

1 of 2



We'll look at a few experiments you could try out in the next lesson.

Experiments

In this lesson, we'll look at a few experiments that can be done with the example we previously looked at.

WE'LL COVER THE FOLLOWING ^

- Examine logs
- Create new microservice
- Other experiments

Examine logs

Try the following experiments in the coding environment given below!

- Start the system and examine the logs of *microservice-order-invoicing* and *microservice-order-shipping* with `docker logs -f msatom_invoicing_1` respectively `docker logs -f msatom_shipping_1`.
- The **microservices log messages when they poll data** from the Atom feed, because there are new orders.
- If you start additional instances of a microservice with `docker compose up --scale`, these **new instances will collect orders via the Atom feed and log information about them**. In doing so, only one instance writes at a time; the other ones ignore the data.
- **Create orders and notice this behavior based on the log messages.**
- Explore the code to find out what the log messages mean and where they are put out.

Create new microservice

Supplement the system with an additional microservice.

As an example, a microservice can be used that credits the customer with

- As an example, a microservice can be used that credits the customer with a bonus depending on the value of the order or that counts the orders.
- Of course, you can copy and modify one of the existing microservices.
- Implement a microservice which polls the URL `http://order:8080/feed`.
- In addition, the microservice should display an HTML page with some information (customer bonus or number of calls).
- Package the microservice in a Docker image and reference it in `docker-compose.yml`. There you can also determine the name of the Docker container.
- Create a link from the container `apache` to the container with the new service in `docker-compose.yml` and from the container with the new service to the container `order`.
- The microservice has to be accessible via the homepage. For this purpose, you have to create a load balancer for the new Docker container in the file `000-default.conf` in the Docker container `apache`. Use the name of the Docker container for this. Then, add a link to the new load balancer in `index.html`.
- Optional: Add HTTP caching format.

Other experiments

- Currently, it is only possible to request all orders at once in the Atom feed. **You can implement paging so that only a subset of the orders is returned.**
- At the moment, the system runs with Docker compose. However, it could also run on a different infrastructure. **Port the system to one of these platforms:**
 - On a microservices platform ([chapter 12](#)).
 - On Kubernetes. [Chapter 13](#) discusses Kubernetes in more detail.
 - On Cloud Foundry. [chapter 14](#) deals with Cloud Foundry.
- Instead of using the Atom format, you could also deliver **your own representation of a feed**.
 - For example, as a JSON document. Change the implementation in the example so that it uses its own custom data.

We'll conclude this chapter in the next lesson.

Chapter Conclusion

We'll conclude this chapter with a quick summary of what we've learned.

WE'LL COVER THE FOLLOWING



- Summary
- HTTP
- Old events
- No guaranteed delivery
- All microservices can receive messages
- Guaranteed order of messages
- Benefits
- Challenges

Summary

HTTP

REST and the Atom format offer an easy way to implement asynchronous communication. **HTTP is used as the communication protocol** which has several advantages:

- HTTP is well understood.
- HTTP has already proven its scalability many times.
- Most of the time HTTP is used in projects to transfer other content like JSON via REST or to deliver HTML pages.
- **Most teams have the necessary experience** to implement scalable systems with HTTP.
- HTTP caching is supported hence, **polling of Atom resources can be implemented very efficiently**.

Old events

Old events

Having old events still available helps because another microservice with event sourcing can rebuild its state by processing all events again.

With an Atom-based system, a microservice must also provide old events. This can be done easily in some cases.

If the microservice has stored the old information anyway, it only needs to provide it as an Atom feed. In this case, access to old events is easy to implement.

The data in the Atom Feed comes from the same source as all other representations. Therefore, the data is consistent with the data that can be queried via other means. All these services show different representations of the same data.

No guaranteed delivery

Atom cannot guarantee the delivery of the message. Therefore, the microservices in which the messages are processed should be implemented idempotently and try to read the message several times to ensure that they are processed.

All microservices can receive messages

Atom has no way to limit the reception of a message to just one microservice. Therefore, the microservice must select an instance that then processes the message, or else you have to rely on the idempotency.

Guaranteed order of messages

Atom can guarantee the order of the messages because the feed is a linear document so the order is fixed. However, this requires that the order of the entries in the feed does not change.

Atom is a very simple alternative for asynchronous communication.

Benefits

- Atom **does not require additional infrastructure**, just HTTP.
- **Old events are easily accessible if necessary.** This can be advantageous

for event sourcing.

- The **sequence can be guaranteed**.
- The **Atom feed is consistent**. It is just another representation of the data and contains the same information as all other representations of the data.

Challenges

- **Guaranteed delivery is difficult**. The recipient can attempt to read the data several times to guarantee all data is processed. However, the responsibility for this lies with the recipient, not with the infrastructure.
 - **Messages for only one recipient are difficult**. All recipients poll the messages. They must then decide which recipient actually processes the message.
 - In part, Atom is **not well suited for the representation of events** because it was originally designed for blogs.
-

In the next chapter, we'll study synchronous microservices concepts.

Introduction

In this lesson, we'll get a quick walkthrough of what the rest of the chapter holds for us.

WE'LL COVER THE FOLLOWING ^

- Chapter walkthrough

Synchronous microservices are one way in which microservices can communicate.

Chapter walkthrough

This chapter shows:

- What is meant by synchronous communication between microservices.
- What the architecture of a synchronous microservices system can look like.
- Which advantages and disadvantages are associated with synchronous communication between microservices.

In the next lesson, we'll look at the definition of synchronous microservices.

Definition

In this lesson, we'll look at the definition of synchronous microservices and get a quick introduction to an example.

WE'LL COVER THE FOLLOWING ^

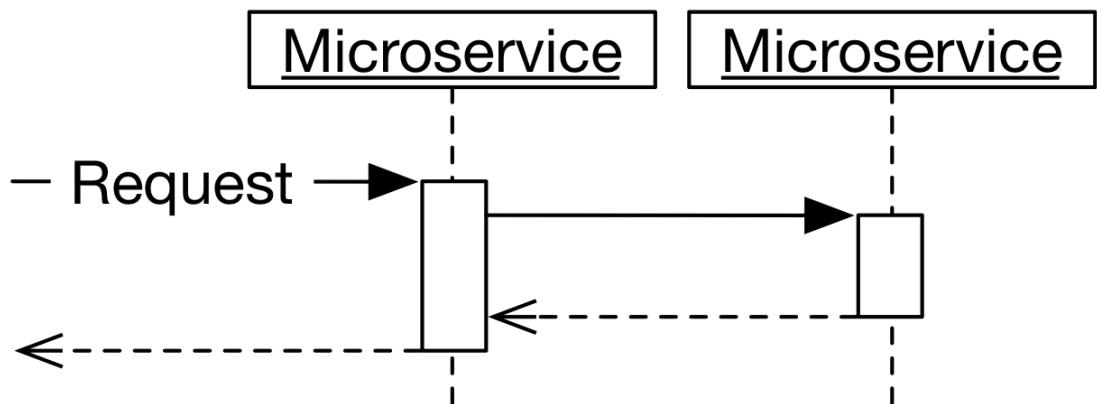
- Introduction
- Synchronous protocols
- Asynchronous protocols
- An example

Introduction

This chapter deals with the technical options for implementing synchronous microservices. [Chapter 6](#) already introduced the term “synchronous microservices”.

A microservice is **synchronous** if it makes a request to other microservices while processing requests and waits for the result.

The logic to handle a request in the microservice might not depend on the result of a request to a different microservice.



Synchronous Communication

The drawing above illustrates this kind of communication. While the left microservice processes a request, it calls the right microservice and waits for the result of this call.

Synchronous and asynchronous communication according to this definition are **independent of the communication protocol**.

Synchronous protocols

A synchronous communication protocol means that **a request returns a result**.

For example, a REST or HTTP GET returns a result in an HTTP status, a JSON document, or an HTML page. If a system processes a REST request, makes a REST request itself to another system, and waits for the response, it is synchronous. Asynchronous REST systems were discussed in [chapter 8](#).

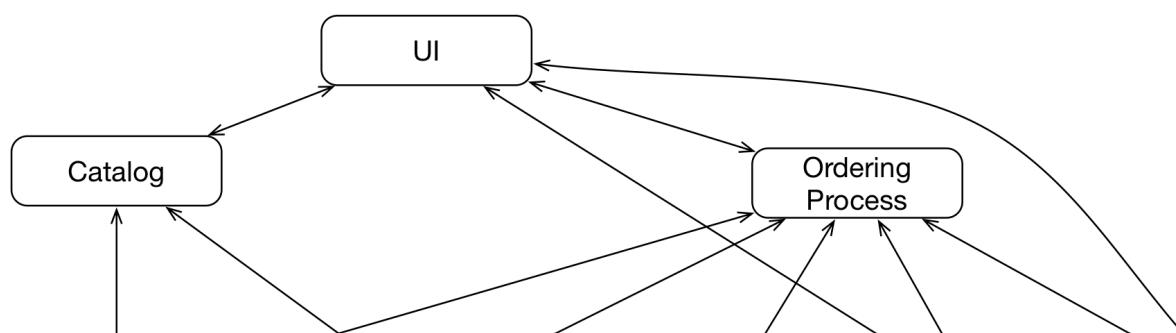
Asynchronous protocols

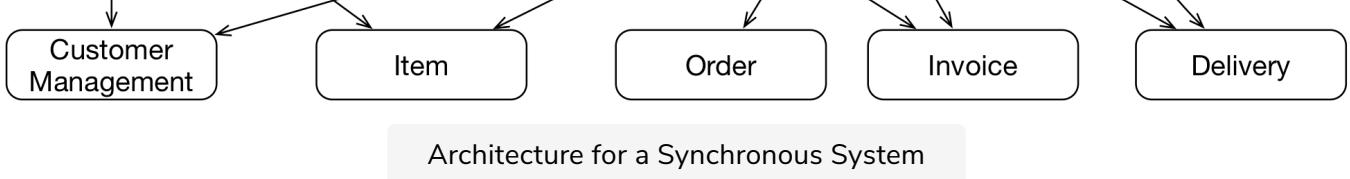
Asynchronous communication protocols, on the other hand, send messages to which the recipients react. There is no direct response.

Synchronous communication with an asynchronous protocol occurs when one system sends a message with an asynchronous communication protocol to another system and then waits to receive a response with an asynchronous communication protocol.

An example

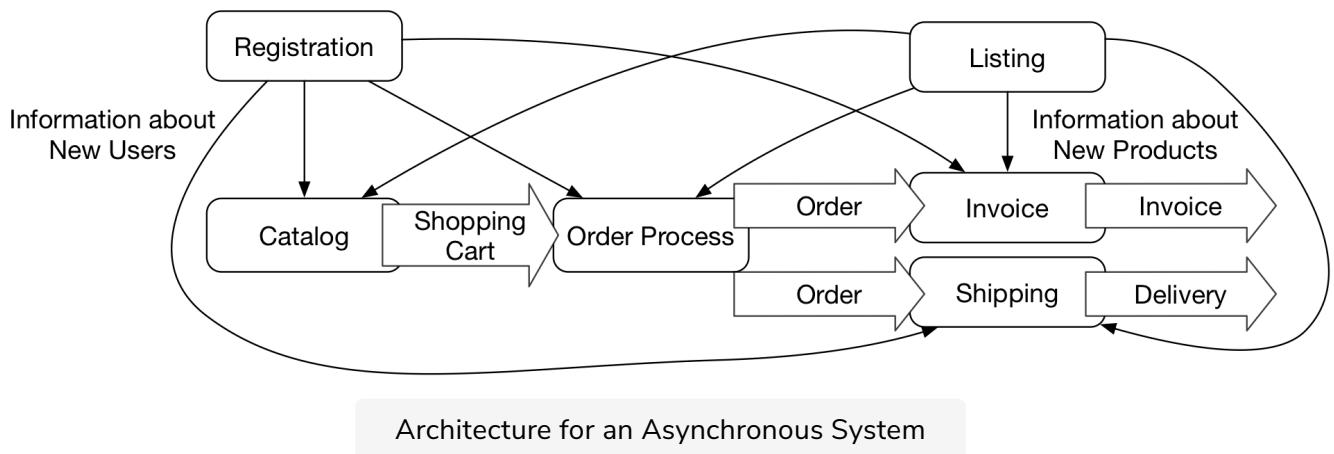
For example, a microservice for orders is synchronous if it calls a microservice for customer data while processing an order and waits for the customer data.





The drawing above shows an exemplary synchronous architecture, which corresponds to the asynchronous architecture displayed in the third drawing within [this lesson](#) from chapter 6.

Here it is again for a refresher.



It describes an excerpt from an e-commerce system.

- The microservices **customer management, items, order, invoice, and delivery** manage the respective data.
- The **catalog** displays all information about the goods and considers the customer's preferences.
- Finally, the **order process** serves to order goods, issue the invoice, and deliver the goods.

The UI accesses **catalog** and **order process** and thus makes the processes implemented in these microservices available to the user. The UI can also display invoice and delivery data.

QUIZ

1

A microservice is synchronous if it _____.

COMPLETED 0%

1 of 3



In the next lesson, we'll continue studying details of the example introduced here.

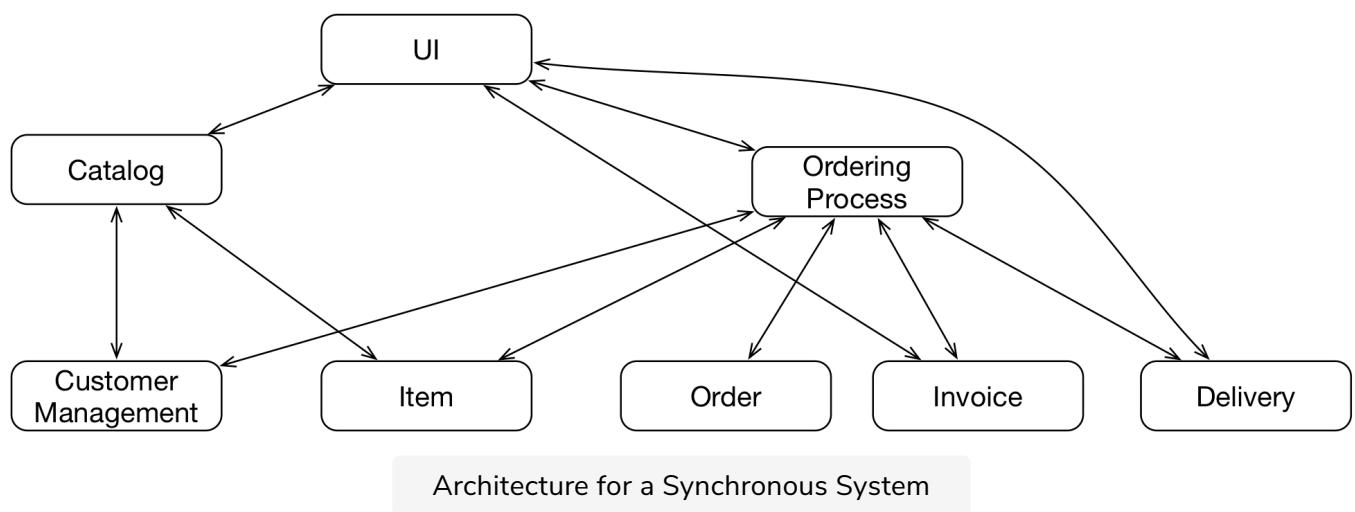
Example

In this lesson, we'll continue to study the details of the example we just saw.

WE'LL COVER THE FOLLOWING

- Consistency
- Disadvantages
- Bounded context
- Tests
- Stubs
- Consumer-driven contract tests
- The pact test framework

Here's the illustration of the example again:



Consistency

The architecture of this system ensures that **all microservices display the same information about a product or order at all times** because they use synchronous calls to access the respective microservices in which the data is stored.

There is **only one place where each piece of the data is stored**. Every microservice uses that data and therefore shows the latest data.

Disadvantages

However, this architecture also has disadvantages:

- Centralized data storage can lead to problems because the data for displaying the catalog is completely different from that needed for the order process.
- The customer's purchasing behavior is important for the catalog in order to display the correct products.
- For orders, the delivery address or the preferred delivery service are relevant.
- Storing this data in a microservice can make the data model complex.
- *Domain-driven design* states that a domain model is only valid in a particular bounded context, making a centralized model problematic.
- The illustration also reveals another problem; most of the functionalities use many microservices, complicating the system.
- In addition, the failure of a microservice means that many functionalities are not available if no special precautions are taken.
- Performance may also suffer because the microservices have to wait for results from many other microservices.

Bounded context

Such an architecture is often found in synchronous architectures. However, it is not mandatory. Theoretically, it is conceivable that **bounded contexts could communicate synchronously with each other**.

Bounded contexts typically exchange **events**. This is particularly easy with asynchronous communication and **not a great fit for synchronous communication**.

Tests

To enable independent deployment, the tests of each microservice must be **as independent as possible** and integration tests should be kept to a minimum.

For tests in synchronous systems, the communication partners must be available. These can be the microservices used in production.

In this case, setting up an environment is hard because many microservices have to be available, and dependencies arise between the microservices because a new version of a microservice must be made available to all clients in order to enable tests.

Stubs

An alternative are stubs that simulate microservices and simplify the setup of the test environment. The **tests no longer depend on other microservices** because the tests use stubs instead of microservices.

Consumer-driven contract tests

Finally, [consumer-driven contract tests](#) can be a solution. With this pattern, the **client writes a test for the server** and can test whether it meets the client's expectations.

This makes it easier to change the server because changes to the interface can be tested without a client. In addition, the **tests no longer depend on the client microservice**.

Consumer-driven contract tests can be written with a **testing framework like JUnit**, for example. The team that implements the called microservice must then execute the tests. If the tests fail, they have made an incompatible change to the microservice.

Either the microservice must be changed to be compatible or the team that wrote the consumer-driven contract test must be informed so that they can change their microservice in such a way that the interface is used differently according to the change. Consumer-driven contract tests therefore formalize the definition of the interface.

As part of the macro architecture, **all teams must agree on a test framework** in which the consumer-driven contract tests are written.

This framework **does not necessarily have to support the programming language in which the microservices are written** when the consumer-driven contract tests use the microservices as black boxes and only access them through the REST interface.

The pact test framework

One option is the framework [Pact](#). It enables writing tests of a REST interface in a programming language. This results in a JSON file containing the REST requests and the expected responses.

With Pact implementations, the JSON file can be interpreted in different programming languages. This increases the technology freedom. An example for Pact with Java can be found at <https://github.com/mvitz/pact-example>.

QUIZ

1

Suppose you are designing a banking app that requires an incredible degree of consistency. How would you design this app?

COMPLETED 0%

1 of 3



In the next lesson, we'll look at the benefits and challenges of synchronous microservices.

Benefits & Challenges

In this lesson, we'll discuss the benefits and challenges of the synchronous microservices approach.

WE'LL COVER THE FOLLOWING ^

- Benefits
- Challenges
 - Low performance
 - Failure propagation
 - Higher level of dependency
- Technical solutions
 - Service discovery
 - Resilience
 - Load balancing
 - Routing
- API gateways

Benefits

In practice, synchronous microservices are a relatively frequently used approach. Many well-known examples of microservice architectures use such a concept.

This has the following advantages:

- All services can access the same dataset, so there are **fewer consistency problems**.
- Synchronous communication is a **natural approach if the system is to offer an API**. Any microservice can implement part of the API. The API can be the product or the API can be used by mobile applications.

- After all, it can be **easier to migrate into such an architecture**. For example, the current architecture can already have a division into different synchronous communication endpoints or teams can exist for each of the functionalities.
- Calling methods, procedures, or functions in a program is usually synchronous. **Developers are familiar with this model** so they can understand it more easily.

Challenges

Synchronous microservices create a number of challenges. Let's discuss a few:

Low performance

The communication with other microservices during request processing causes the latencies for responses of other microservices and the communication times via the network to add up.

Therefore, synchronous communication **can lead to a performance problem**. When a microservice reacts slowly, this can have **consequences for a plethora of other microservices**.

- In the example in the [last lesson](#), the *catalog* uses two other microservices (*item* and *customer management*). Thus, the latencies of these three systems can add up.
- A request to the *catalog microservice* creates requests to the *customer management microservice* and to the *item microservice*.
- Only when all microservices have responded to the requests, the user gets to see the result.

Failure propagation

When a synchronous microservice calls a failed microservice, the **calling microservice may crash and the failure propagates**. This makes the system very vulnerable.

The vulnerability of the microservices and the additional waiting times can **prevent the reliable operation of microservices systems** with synchronous communication. These problems have to be solved when a microservices

system uses synchronous communication.

Higher level of dependency

In addition, synchronous communication can create a higher level of dependency in the domain logic. Asynchronous communication often focuses on events (see [Events](#) from chapter 6).

In this case, a microservice can decide how to react to events. In contrast, synchronous communication typically defines what a microservice is supposed to do.

- In the example, the **order process** would make the **invoice microservice** generate an invoice.
- With events, the **invoice microservice** would decide how to react to the event.
- This facilitates the extendibility of the system.

If the customer is supposed to be credited with bonus points for an order, there has to be an additional microservice reacting to the already existing event.

That event is probably already processed by more than one microservice, so one more receiver needs to be added. In a **synchronous architecture, an additional system has to be called**.

Technical solutions

For implementing a system of synchronous microservices some **technical solutions** are required. Let's discuss them.

Service discovery

The microservices have to know how they can communicate with other microservices. Usually, this requires an **IP address and a port**.

Service discovery serves to find the port and IP address of a service.

Service discovery should be **dynamic**:

- Microservices can be **scaled**. Then there are **new IP addresses** at which additional instances of a microservice are available.
- In addition, a microservice can **fail**. Then it is **not available anymore** at the known IP address.

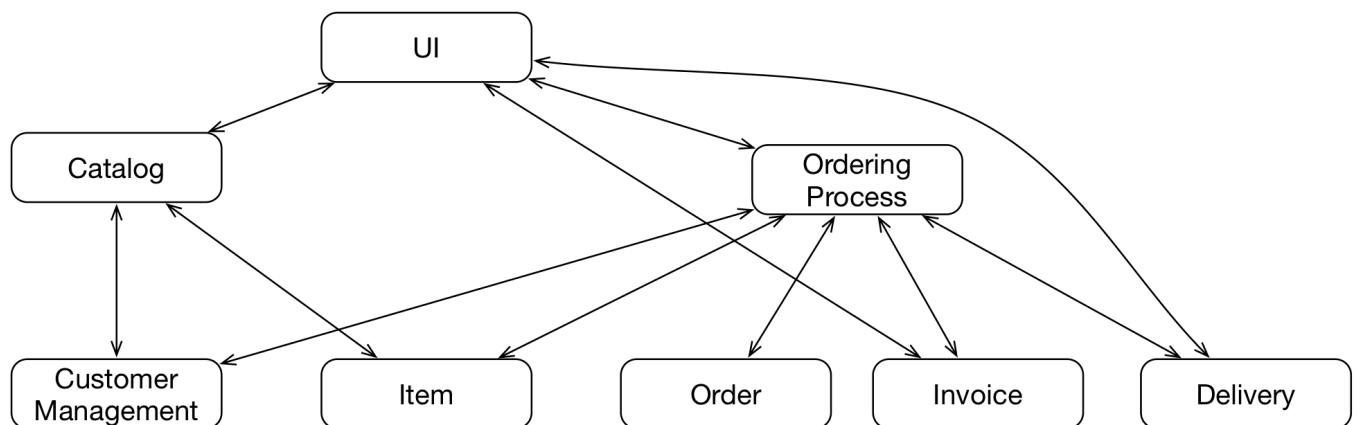
Service discovery can be very simple. **DNS (Domain Name System)** provides IP addresses for servers on the Internet. This technology already provides a simple service discovery.

Resilience

When communication is synchronous, microservices have to be **prepared for the failure of other microservices**. It has to be prevented that the calling microservice fails as well.

Otherwise, there will be **failure cascade**:

- First one microservice fails.
- Then other microservices call this microservice and fail.
- In the end, the entire system will be down.



When the service for the items fails in the architecture above, the catalog and order process could fail in turn.

This would render a large part of the system unusable. Therefore, there has to be a technical solution to achieve **resilience**.

Load balancing

Each microservice should be **scalable independently** of the other

microservices. Load has to be distributed between microservices.

This does not only pertain to access from the outside, but also to internal communication. Therefore, there has to be a *load balancing* for each microservice.

Routing

Finally, every access from the outside should be forwarded to the responsible microservices. This requires **routing**.

- A user might want to use the catalog and the order process.
- While they are separate microservices, they should appear as parts of the **same system** to the outside.

Consequently, the technologies for synchronous microservices which are discussed in the following chapters have to offer solutions for **service discovery, resilience, load balancing, and routing**.

API gateways

For complex APIs, complex routing of requests to the microservices might be needed. **API Gateways** offer additional features:

- Most of the API gateways can perform user authentication.
- They can throttle the network traffic for individual users to support a high number of users at the same time. This can be supplemented by centralized logging of all requests or caching.
- API gateways can also solve aspects like monitoring, documentation, or mocking.

The implementations of [Apigee](#), [3scale by Red Hat](#), [apiman](#), or cloud products like the [Amazon API Gateway](#) and the [Microsoft Azure API Gateway](#) are examples of API gateways.

The examples in this course do not offer public REST interfaces, only REST interfaces for internal use. The public interfaces are just web pages and the examples do not use API gateways.

1

What would happen if a system had hardcoded service discovery so that an IP address was written against a microservice in a file that all microservices had access to?

Choose two options from the answers below.

COMPLETED 0%

1 of 2



In the next lesson, we'll look at some variations of the synchronous approach we've discussed already.

Variations

We'll look at a few quick variations of what we've learned in this chapter now.

WE'LL COVER THE FOLLOWING



- Frontend integration
- Asynchronous communication
- Combination
- Future chapters

Frontend integration

Frontend integration ([chapter 3](#)) can be a good addition to synchronous communication.

Asynchronous communication

Asynchronous communication (see [chapter 6](#)) is another alternative.

Synchronous and asynchronous communication are possibilities for microservices to communicate with each other on the logic level.

One of these options should be enough to build a microservices system.

Combination

Of course, a **combination** of synchronous and asynchronous is also possible.

The asynchronous communication with Atom and the synchronous communication with REST use the same infrastructure so that these two communication mechanisms can be used together very easily.

Future chapters

The following chapters show concrete implementations for synchronous

The following chapters show concrete implementations for synchronous communication.

The examples all use REST for communication. Today REST is the preferred architecture for synchronous communication.

In principle, other approaches such as [SOAP](#) or [Thrift](#) are also conceivable.

Q U I Z

Q

In what situation would a combination of asynchronous and synchronous communication be a good idea?

COMPLETED 0%

1 of 1



We'll conclude this chapter with the next lesson.

Chapter Conclusion

That's it for this chapter. Here's a high-level summary.

WE'LL COVER THE FOLLOWING ^

- Summary
 - Technically complex
- Few advantages

Summary

Technically complex

At first glance, synchronous microservices are **simple** because they correspond to the **classic programming model**.

Classic Programming
=
Synchronous Programming

But synchronous microservices lead to a **high degree of technical complexity** because microservices have to deal with the **failure** of other microservices and **latencies** add up.

The resulting systems are distributed systems and thus technically complex.

Few advantages

This has few advantages. It is advisable to choose this approach only if there are good reasons for doing so.

In general, asynchronous communication makes it easier to deal with the challenges of distributed systems.

In the next chapter, we'll try a new recipe using the Netflix Stack!

Introduction

Let's kick off this chapter with a quick introduction to the Netflix stack and a chapter walkthrough!

WE'LL COVER THE FOLLOWING ^

- Where does the Netflix stack come from?
- License & technology
- Chapter walkthrough

Where does the Netflix stack come from?

Netflix has developed a new platform for online video streaming to meet the high performance and scalability requirements in this area. The result was **one of the first microservices architectures**.

Later, Netflix released its technologies as open-source projects making the Netflix stack one of the first stacks to implement microservices.

License & technology

The components of the Netflix stack are open source and are under the very liberal **Apache license**.

The Netflix projects are all based on **Java** and are integrated into Spring Cloud, making it easy to use them with Spring Boot.

Chapter walkthrough

This chapter provides the following content:

- Overview of the Netflix microservices stack
- Details about service discovery with Eureka, routing with Zuul, load balancing with Ribbon, and resilience with Hystrix

- The advantages and disadvantages of the Netflix stack

In this way, the reader can assess the suitability of these technologies for a concrete project and a microservices system with these technologies.

Q U I Z

1

Netflix projects are primarily based on ____.

COMPLETED 0%

1 of 3



In the next lesson, we'll look at an example of the Netflix stack in practice.

Example

In this lesson, we'll introduce a Netflix stack coding example.

WE'LL COVER THE FOLLOWING



- Introduction
- Architecture of the example
- Running the example
- Docker containers and ports
 - Routing via Zuul
 - Service discovery via Eureka
 - Hystrix dashboard

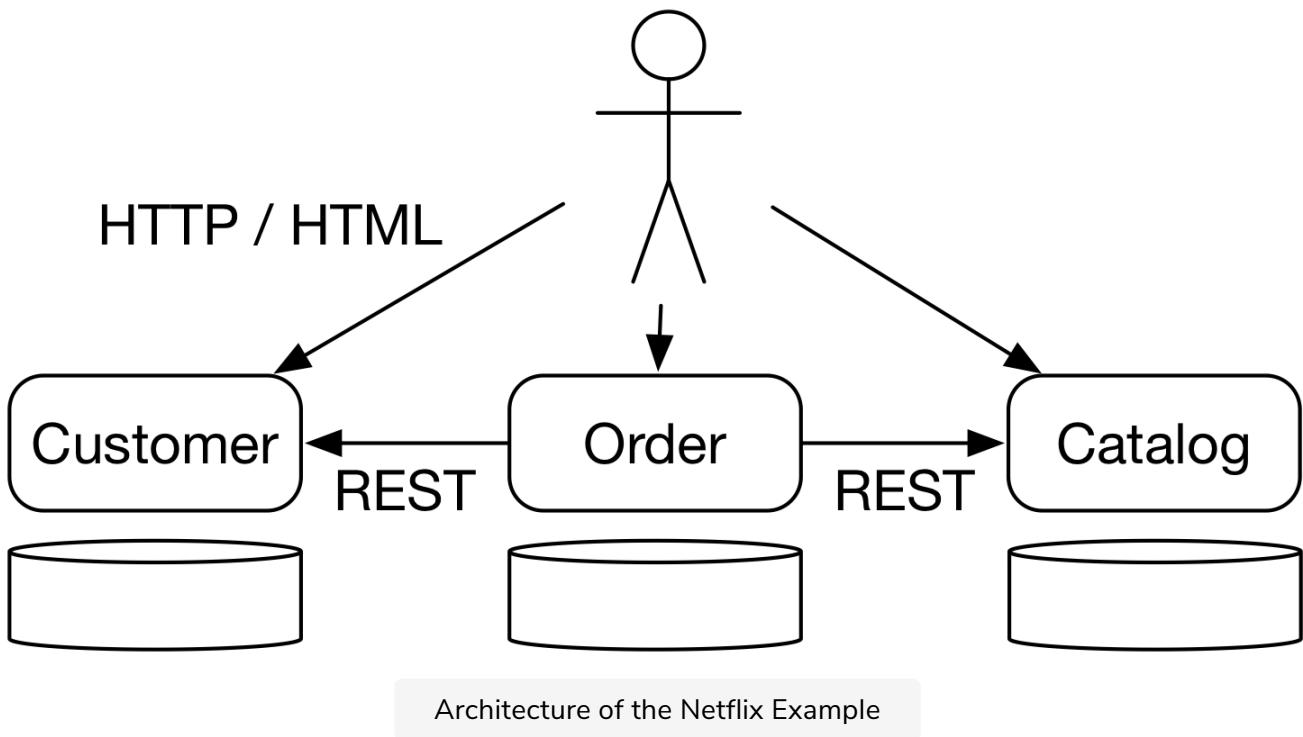
Introduction

The example for this chapter can be found at

<https://github.com/ewolff/microservice>. It consists of **three** microservices:

- The **catalog** microservice that manages the information about the items.
- The **customer** microservice that stores the data of the customers.
- The **order** microservice that can accept new orders by using the catalog and the customer microservice.

Architecture of the example



- Each of the microservices has its own web interface with which users can interact.
- Among each other, the microservices communicate via REST.
- The order microservice requires information about customers and items from the other two microservices.

In addition to the microservices, there is a **Java application** that displays the **Hystrix dashboard** where monitoring the Hystrix circuit breakers is visualized.

The drawing in the section [Docker containers and ports](#) shows the entire example at the level of the Docker containers.

Running the example

First, the code has to be downloaded with `git clone`

`https://github.com/ewolff/microservice.git`. Then the code has to be compiled with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in the directory `microservice-demo`. See [this lesson](#) in the appendix for more details on Maven and how to troubleshoot the build. Afterwards, the Docker containers can be built with `docker-compose build` in the directory `docker` and started with `docker-compose up -d`. See [this lesson](#) and the one often in the appendix for more details on Docker, `docker-compose`

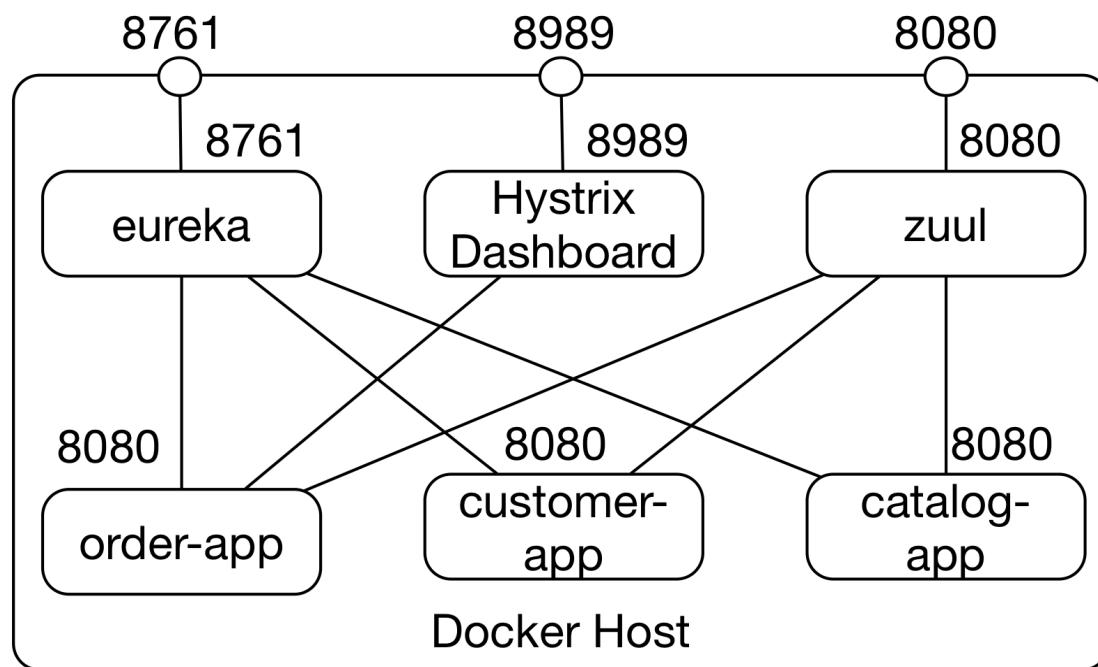
and the one after in the appendix for more details on Docker, docker-compose

and how to troubleshoot them. Subsequently, the Docker containers are available on the Docker host.

<https://github.com/ewolff/microservice/blob/master/HOW-TO-RUN.md> in detail explains the steps that need to be performed to build and run the example.



Docker containers and ports



Docker Containers in the Netflix Example

The Docker containers communicate via an **internal network**. Some Docker containers can also be used via a port on the Docker host. The Docker host is the computer on which the Docker containers run.

The three microservices **order**, **customer**, and **catalog** each run in their own Docker containers. Access to the Docker containers is only possible *within* the Docker network.

Routing via Zuul

In order to be able to use the services from the outside, **Zuul** provides routing.

- The Zuul container can be accessed from outside under port **8080** and

forwards requests to the microservices.

- If the Docker containers are running **locally**, the URL is <http://localhost:8080>.
- At this URL, there is also a web page available which includes links to all microservices, Eureka, and the Hystrix dashboard.

Service discovery via Eureka

Eureka serves as a **service discovery** solution.

- The dashboard is available at port [8761](#).
- This port is also accessible at the Docker host.
- For a **local Docker installation**, the URL is <http://localhost:8761>.

Hystrix dashboard

Finally, the **Hystrix dashboard** runs in its own Docker container that can also be accessed under port [8989](#) on the Docker host, for example at <http://localhost:8989>.

1

The three microservices run ____.

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss service discovery with Eureka in more detail.

Eureka: Service Discovery

In this lesson, we'll discuss service discovery with Eureka.

WE'LL COVER THE FOLLOWING



- Introduction
- Features
 - REST interface
 - Replication
 - Cache improved performance
 - Amazon Web Services support
 - Crash detection
- Servers
- Client
- Registration
- Other programming languages
 - Sidecars
- Access to other services

Introduction

Eureka implements service discovery as we learned in the [introduction](#).

For synchronous communication, microservices have to find out at **which port and IP address other microservices can be accessed**.

Features

Let's discuss some essential characteristics of Eureka.

REST interface

Eureka has a **REST interface**.

Microservices can use this interface to **register or request information** about other microservices.

Replication

Eureka supports **replication**.

The information from the Eureka servers is distributed to other servers enabling the system to **compensate for the failure** of Eureka servers.

In a distributed system, service discovery is essential for communication between microservices. Therefore, service discovery must be implemented in a way that a **failure of one server does not cause the entire service discovery to fail**.

Cache improved performance

Due to **caches on the client**:

- The performance of Eureka **very good**.
- **Availability is improved** because the information is stored on the client compensating for server failure.
- The server only sends information to the client about new or deleted microservices and not information about all registered services, **communication very efficient**.

Amazon Web Services support

Netflix supports **AWS (Amazon Web Services)**, i.e., the Amazon Cloud.

In AWS, servers run in **availability zones**. These are basically separate data centers. The failure of an availability zone does not affect other availability zones.

Several availability zones form a **region** which is located in a geographical zone. For example, the data centers for the region called EU-West-1 are located in Ireland.

Eureka can take regions and availability zones into account and offer a microservice instance from the same availability zone to a client as a result of

the service discovery in order to increase speed.

Crash detection

Eureka expects the microservices to regularly send **heartbeats**.

In this way, Eureka detects crashed instances and excludes them from the system. This increases the probability that Eureka will return service instances that are available.

However, it may happen that a microservice instance is returned even though it is no longer running.

But whether a microservice instance has crashed or not is obvious once it is used so that a different instance can be used if it has crashed.

Servers

The Netflix Eureka project is available for download at [GitHub](#). You can build the project and get both the server and the client.

The Spring Cloud project also supports Eureka, the Eureka server can even be started as a Spring Boot application. For this, the main class which also has the annotation `@SpringBootApplication` has to be annotated with

`@EnableEurekaServer`. In `pom.xml` in the `dependencyManagement` section, the Spring Cloud dependencies have to be imported, and a dependency on the library `spring-cloud-starter-eureka-server` has to be inserted. In addition, a configuration in the `application.properties` file is necessary. The project `microservice-demo-eureka-server` provides all of that and implements an Eureka server.

At first glance, it does not seem to make sense to build the Eureka server by yourself in this way, especially since the implementation essentially consists of an annotation. But the Eureka server can be treated like all the other microservices. The Spring Cloud Eureka server is a JAR file and, like all other microservices, can be stored and started in a Docker image. It is also possible to secure it like a Java web application with Spring security and configure logging and monitoring as with all other microservices.

The screenshot shows a web browser window titled "Eureka" with the URL "localhost:8761". The page has a dark header with the "spring Eureka" logo. Below the header, there's a section titled "System Status" containing various configuration parameters. Under "DS Replicas", there's a table listing microservices and their registration details. At the bottom, there's a "Eureka Dashboard" button.

Environment	test
Data center	default
Current time	2017-06-22T19:51:30 +0000
Uptime	01:14
Lease expiration enabled	true
Renews threshold	10
Renews (last min)	108

Application	AMIs	Availability Zones	Status
CATALOG	n/a (1)	(1)	UP (1) - 5d50e86620ae:catalog:8080
CUSTOMER	n/a (1)	(1)	UP (1) - e34c51981195:customer:8080
ORDER	n/a (1)	(1)	UP (1) - f513a9945277:order:8080
TURBINE	n/a (1)	(1)	UP (1) - e741763d63e1:turbine:8989
ZUUL	n/a (1)	(1)	UP (1) - 20dec567ae39:zuul:8080

Eureka Dashboard

Eureka provides a **dashboard**, see the screenshot above.

- It displays an overview of the microservices which are registered with Eureka.
 - This includes the names of the microservices and the URLs at which they can be accessed.
- However, the URLs only work in the Docker internal network meaning

- However, the URLs only work in the Docker internal network meaning the links in the dashboard do not work.
- The dashboard is accessible on the Docker host at port 8761 i.e. <http://localhost:8761> if the example runs locally.

Client

Each microservice is a Eureka client and must **register** with the Eureka server in order to inform the Eureka server about the name of the microservice, the IP-address, and port at which it can be reached.

Spring Cloud simplifies the configuration for clients.

Registration

The client has to have a dependency on `spring-cloud-starter-eureka` to add the necessary libraries. The main class which has the annotation `@SpringBootApplication` additionally has to be annotated with `@EnableEurekaClient`.

An alternative is `@EnableDiscoveryClient`. In contrast to `@EnableEurekaClient`, the annotation `@EnableDiscoveryClient` is generic. Thus it also works with Consul (see [chapter 11](#)).

```
spring.application.name=catalog
eureka.client.serviceUrl.defaultZone=http://eureka:8761/eureka/
eureka.instance.leaseRenewalIntervalInSeconds=5
eureka.instance.metadataMap.instanceId=${spring.application.name}:${random.value}
eureka.instance.preferIpAddress=true
```

In the file `application.properties` shown above, appropriate settings must be entered for the application to register.

- `spring.application.name` contains the name under which the application registers at the Eureka server.
- `eureka.client.serviceUrl.defaultZone` defines which Eureka server is to be used.
- The setting `eureka.instance.leaseRenewalIntervalInSeconds` ensures that the registration information is replicated every **five seconds** and is

the registration information is replicated every **five seconds** and is replicated faster than the default setting.

- This makes new microservice instances **usable more quickly**. In production, this value should **not** be set so low that there is no network traffic.
- `eureka.instance.metadataMap.instanceId` provides each microservice instance with a random ID, for instance, to be able to discriminate between two instances for load balancing.
- Due to `eureka.instance.preferIpAddress`, the services register with their IP address and not with their host name. This avoids problems that arise because hostnames cannot be resolved in the Docker environment.

During registration, the name of the microservice is automatically converted to uppercase letters. Thus, `order` is turned into `ORDER`.

Other programming languages

- For programming languages other than Java, a library must be used to access Eureka.
- There are some libraries that implement Eureka clients for certain programming languages.
- Of course, Eureka provides a REST interface which can also be used.

Sidecars

To use the Netflix infrastructure with other programming languages, it is also possible to use a **sidecar**.

A **sidecar** is an application written in Java that uses the Java libraries to talk to the Netflix infrastructure.

The application uses the sidecar to communicate with the Netflix infrastructure, so the sidecar is a **helper** for the real application. That way the application can be written in **any programming language**. Essentially the sidecar is the **interface** to the Netflix infrastructure.

In this way, Eureka can support other programming languages; but this

requires an additional process that **consumes additional resources**.

Netflix itself offers [Prana](#) as a sidecar. Spring Cloud also provides an implementation of such a [sidecar](#).

Access to other services

In the example in this chapter, **Ribbon** implements the access to other services in order to implement load balancing across multiple instances.

Thus, the Eureka API is only used via Ribbon to find information about other microservices.

Q U I Z

1

In what case could Eureka return an instance of a microservice that has crashed **despite** the implementation of heartbeats?

COMPLETED 0%

1 of 4



In the next lesson, we'll study routing with Zuul in more detail.

Router: Zuul

In this lesson, we'll discuss routing with Zuul.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Zuul vs. reverse proxy
- Zuul in the example

Introduction

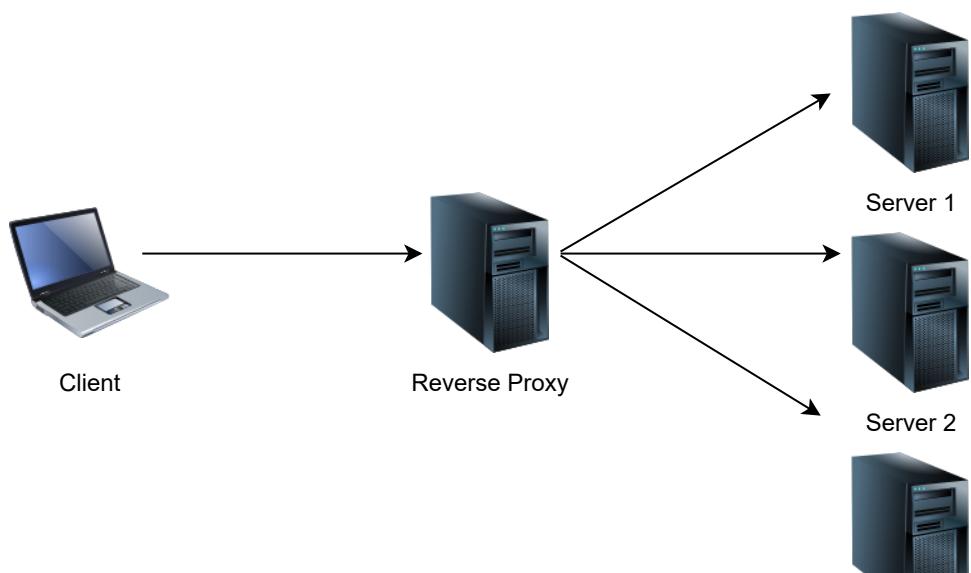
Zuul is the **routing solution** that is part of the Netflix stack.

Zuul is responsible for forwarding external calls to the correct microservice.

Zuul vs. reverse proxy

The routing could also be provided by a Reverse Proxy.

A **Reverse Proxy** is a web server that is configured to forward incoming calls to other servers.



Reverse Proxy

Zuul has a feature that a reverse proxy is lacking, **dynamic filters**:

- Depending on the properties of the HTTP request or external configurations, Zuul can **forward certain calls to specific servers** or execute logic for logging, for example.
- A developer can write **custom code** for the routing decision.
- The code can even be dynamically loaded as Groovy code at runtime.

In this way, Zuul ensures **maximum flexibility**.

- Zuul filters can also be used to implement central functionalities such as **logging of all requests**.
- A Zuul filter can implement the login, send information about the current user with the HTTP requests, and thereby implement **authentication**.

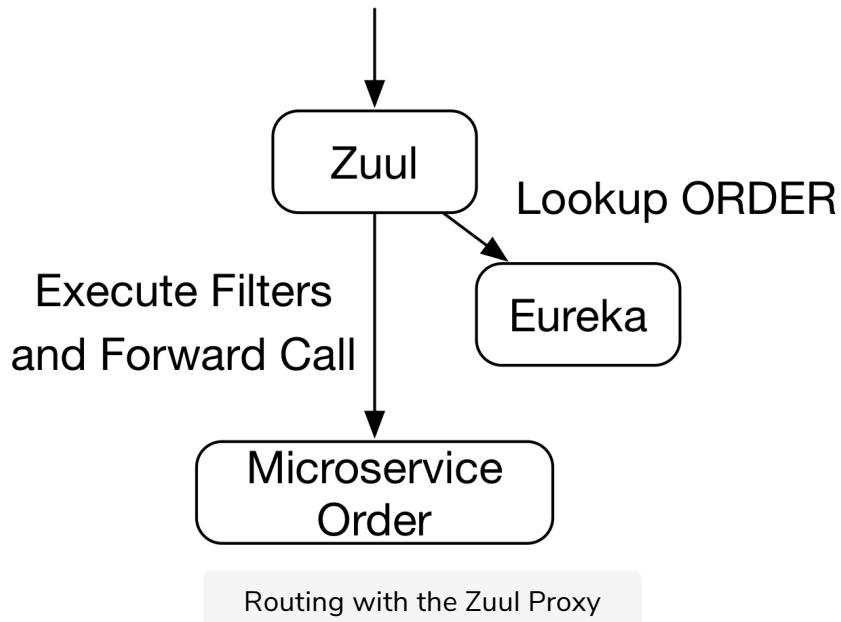
Zuul can **take over typical functionalities of an API gateway** (see [Benefits & Challenges](#)).

Zuul in the example

In the example, Zuul is configured as a proxy and does not contain any special code. Zuul forwards access to a URL such as <http://localhost:8080/order> to the microservice called ORDER.

Forwarding works for all microservices registered in Eureka. Zuul reads the names of all microservices from Eureka and forwards the requests.

`http://localhost:8080/order`



Of course, Zuul “reveals” which microservices the microservices system is made up of. However, Zuul can be reconfigured by routes and filters in a way where completely different microservices can be accessed under the same URL.

Zuul can also deliver static content. In the example, Zuul provides the web page from which the various microservices can be accessed.

Q U I Z

1

What is the difference between Zuul and a regular reverse proxy?

In the next lesson, we'll discuss load balancing with Ribbon.

Load Balancing: Ribbon

In this lesson, we'll check out load balancing with Ribbon.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Central load balancer
- Client-side load balancing
- Ribbon API
 - Ribbon with Consul
 - RestTemplate

Introduction

Microservices have the advantage that each microservice can be scaled independently of the other microservices.

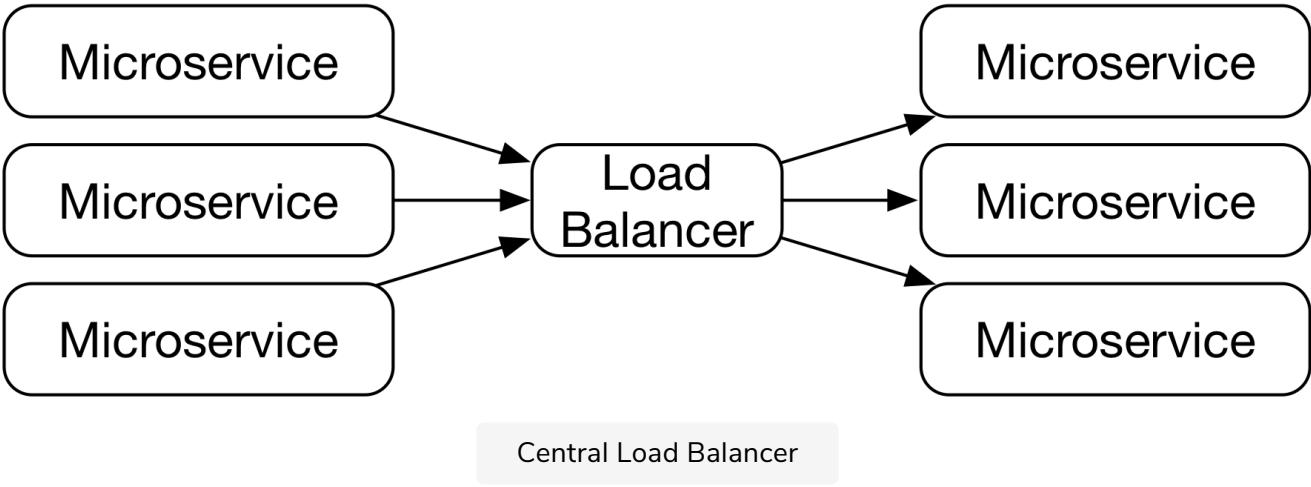
To do this, it is necessary that the call to a microservice can be **distributed to several instances by a load balancer**.

Central load balancer

Typically, a single load balancer is used for all calls. Therefore, a single load balancer, which processes all requests from all microservices, can also be used for an entire microservices system.

However, such an approach leads to a **bottleneck** since all network traffic must be routed through this single load balancer.

The load balancer is also a single point of failure. If the load balancer fails, all network traffic stops functioning and the **entire microservices system fails**.

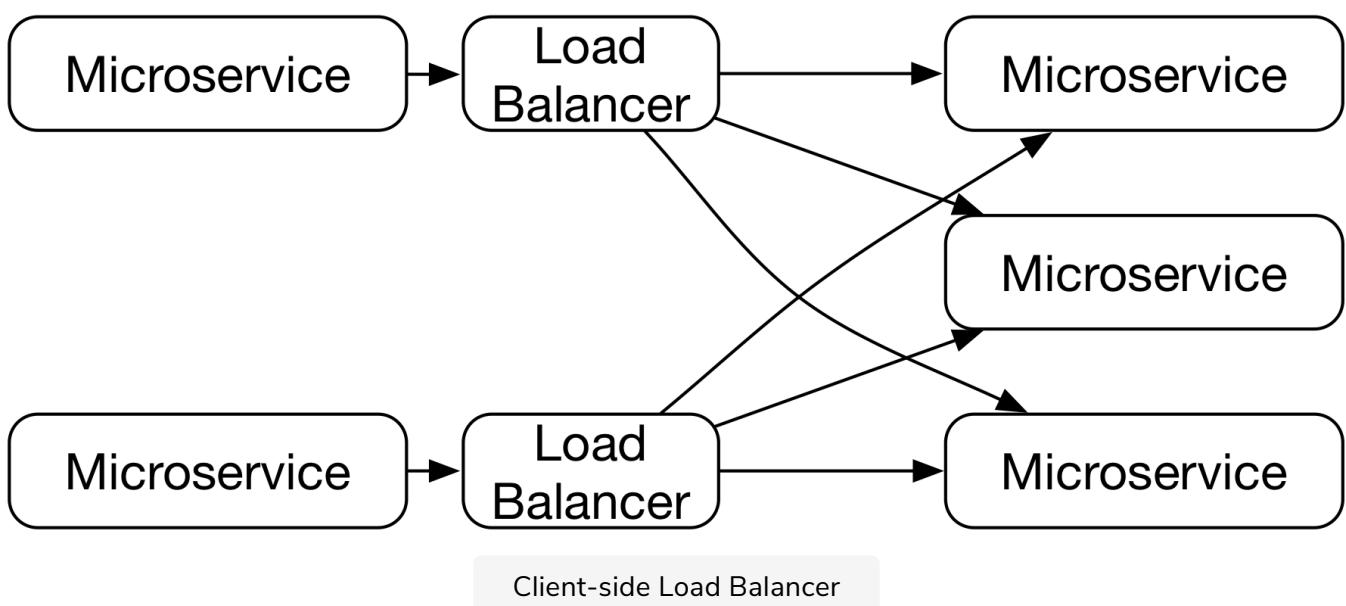


Decentralized load balancing would be better. For this, each microservice must have its own load balancer.

If such a load balancer fails, **only one** microservice will fail.

Client-side load balancing

The idea of client-side load balancing can be implemented by a “normal” load balancer such as Apache httpd or nginx. A load balancer is deployed for each microservice. The load balancer must obtain the information about the currently available microservices from the service discovery.



It is also possible to write a library that distributes requests to other microservices to different instances. This library must read the currently available microservice instances from the service discovery and then, for each request, select one of the instances. This is how Ribbon [Ribbon](#) works.

Ribbon API

Ribbon offers a relatively simple API for load balancing.

```
private LoadBalancerClient loadBalancer;  
    // Spring injects a LoadBalancerClient  
ServiceInstance instance = loadBalancer.choose("CUSTOMER");  
url = String.format("http://%s:%s/customer/",  
    instance.getHost(), instance.getPort());
```

Spring Cloud injects an implementation of the interface `LoadBalancerClient`. First, a call to the `LoadBalancerClient` selects an instance of a microservice. This information is then used to fill a URL to which the request can be sent.

Ribbon supports various strategies for selecting an instance. Thus, approaches other than a simple round robin are feasible.

Ribbon with Consul

As part of the Netflix stack, Ribbon supports Eureka as a service discovery tool, but it also supports Consul. In the Consul example (see [chapter 11](#)) the access to the microservices is implemented identical to the Netflix example.

RestTemplate

Spring contains the `RestTemplate` to easily implement REST calls. If a `RestTemplate` is created by Spring and annotated with `@LoadBalanced`, Spring makes sure that a URL like `http://order/` is forwarded to the order microservice. Internally, Ribbon is used for this.

<https://spring.io/guides/gs/client-side-load-balancing/> shows how this approach can be implemented with a `RestTemplate`.

QUIZ

1

In the following code,

```
private LoadBalancerClient loadBalancer;
```

```
// Spring injects a LoadBalancerClient
ServiceInstance instance = loadBalancer.choose("ORDER");

url = String.format("http://%s:%s/order/",
instance.getHost(), instance.getPort());
```

Load balancing is done for which microservice?

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss resilience with Hystrix.

Resilience: Hystrix

In this lesson, we'll look at resilience with Hystrix in detail.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Resilience patterns
 - Timeout
 - Fail fast
 - Bulkhead
 - Circuit breaker

Introduction

With synchronous communication between microservices, it is important that the failure of one microservice does not cause other microservices to fail as well.

Otherwise, the unavailability of a single microservice can cause further microservices to fail until the entire system is no longer available.

The microservices may return errors because they cannot deliver reasonable results due to a failed microservice.

However, it must not happen that a microservice waits for the result of another microservice for an infinite period of time and thereby becomes unavailable itself.

Resilience patterns

Michael T. Nygard's book, *Release It!* describes different patterns with which the resilience of a system can be increased. Hystrix implements some of these patterns. Let's discuss them.

Timeout

A **timeout** prevents a microservice from waiting too long for another microservice.

Without a timeout, a **thread can block for a very long time** because the thread does not get a response from another microservice. If all threads are blocked, the microservice will fail because there are no more threads available for new tasks.

Hystrix executes a request in a separate thread pool. Hystrix controls these threads and can terminate the request to implement the timeout.

Fail fast

Fail Fast describes a similar pattern. It is better to generate an error as quickly as possible.

The code can check at the beginning of an operation whether all necessary resources are available.

If this is not the case, the request can be terminated immediately with an error. This reduces the time that the caller has to block a thread or other resources.

Bulkhead

Hystrix can use its own thread pool for each type of request. For example, a separate thread pool can be set up for each called microservice.

If the call of a particular microservice takes too long, only the thread pool for that particular microservice is emptied, while the others still contain threads.

This will limit the impact of the problem and is called a **bulkhead**. This term was coined in analogy to a watertight bulkhead in a ship which divides the ship into different segments. If a leak occurs, only part of the ship is flooded with water so that the ship does not sink.

Circuit breaker

Finally, Hystrix implements a **circuit breaker**.

This is a fuse analogous to the ones used in the electrical system of a house.

This is a case analogous to the fuse used in the electrical system of a house.

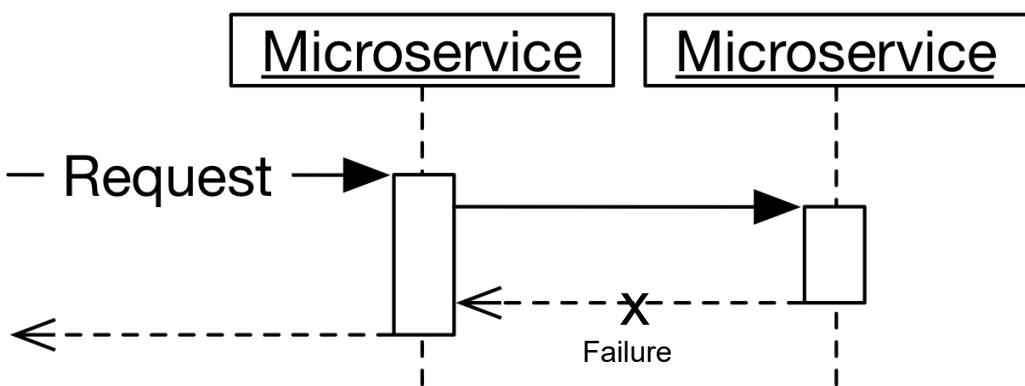
There, a circuit breaker is used to cut off the current flow if there is a short

circuit, preventing a fire from breaking out.

The Hystrix circuit breaker has a different approach. If a **system call results in an error, the circuit breaker is opened and does not allow any calls to pass through.**

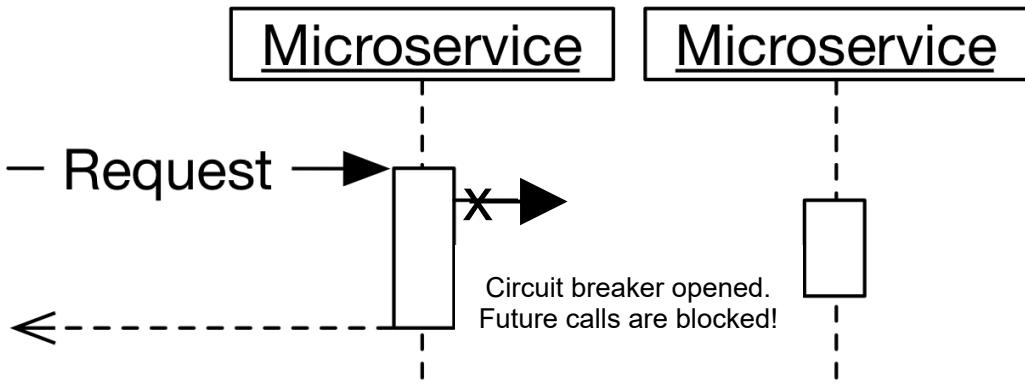
After some time, a call is allowed to pass through again. Only when this call is successful, is the circuit breaker closed again. This prevents a faulty microservice from being called. This **saves resources and avoids blocked threads.**

In addition, the circuit breakers of the different clients are closing one by one so that a failed and recovered microservice gradually handles the full load. This **reduces the probability that it will fail again immediately after starting up.**



Circuite breaker parttern in Hystrix

1 of 2



Circuite breaker parttern in Hystrix

2 of 2

-



QUIZ

1

The timeout pattern ____.

COMPLETED 0%

1 of 2



In the next lesson, we'll discuss some variations of the approaches that we've discussed in this lesson.

Resilience: Hystrix Implementation

In this lesson, we'll study how Hystrix is used in code.

WE'LL COVER THE FOLLOWING ^

- Implementation
- Monitoring
 - Hystrix dashboard
 - Other monitoring options
 - Turbine

Implementation

Hystrix offers an implementation of most resilience patterns as a Java library.

The Hystrix API requires **command objects instead of simple method calls**. These classes supplement the method call with the necessary Hystrix functionalities.

When using Hystrix with Spring Cloud, it is not necessary to implement commands. Instead, the methods are annotated with `@HystrixCommand`. It activates Hystrix for this method and the attributes of the annotation configure Hystrix.

```
@HystrixCommand(  
    fallbackMethod = "getItemsCache",  
    commandProperties = {  
        @HystrixProperty(  
            name = "circuitBreaker.requestVolumeThreshold",  
            value = "2") })  
public Collection<Item> findAll() {  
    ...  
    this.itemsCache = pagedResources.getContent();  
    ...  
    return itemsCache;  
}
```

The listing shows the access from the order microservice to the catalog microservice.

- **Lines 5-6:** `circuitBreaker.requestVolumeThreshold` specifies how many calls in a time window must cause errors for the circuit breaker to open.
- **Line 2:** The `fallbackMethod` attribute of the annotation configures the method `getItemsCache()` as a fallback method.
- **Line 7:** `findAll()` stores the data returned by the catalog microservice in the instance variable `itemsCache`.
- **Line 2:** The `getItemsCache()` method serves as a fallback and reads the result of the last call from the instance variable `itemsCache` and returns it. Have a look at the listing below for its implementation.

```
private Collection<Item> getItemsCache() {  
    return itemsCache;  
}
```

The reasoning behind this is that **it is better that the service continues to work with outdated data than a service that does not work at all..**

This can lead to orders being charged at an outdated price. However, this is **better than accepting no orders at all.**

In general, **if a service fails, a default value can be used or an error can be reported.** Reporting an error is the correct solution if incorrect data cannot be accepted under any circumstances.

Which approach is correct in the end is a decision that **depends on the domain logic.** It should be avoided that in case of an error, the REST call burdens the server or blocks the client for too long.

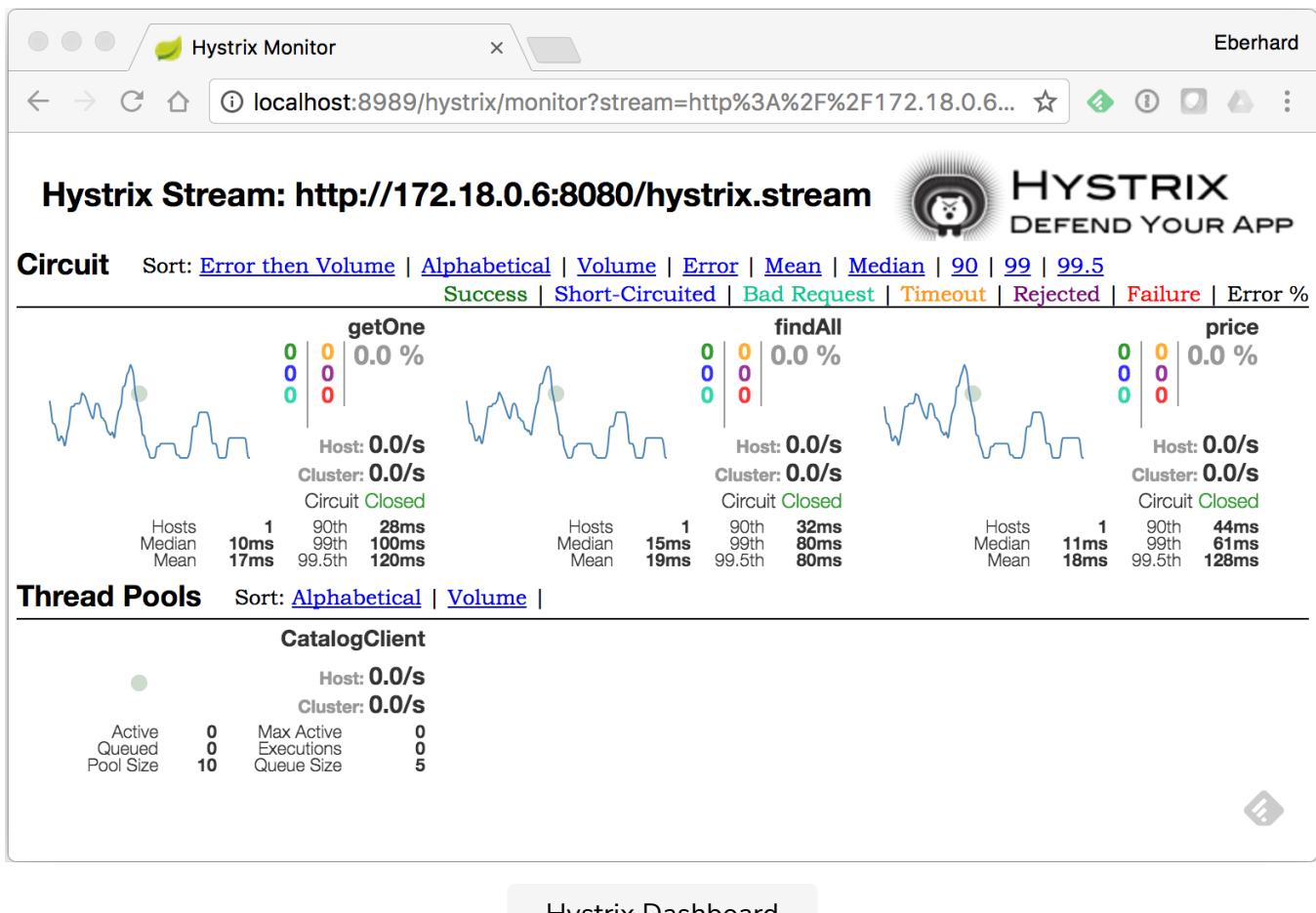
Monitoring

The state of the circuit breakers provides a good overview of the state of the system. An open circuit breaker is an indication of a problem.

Hystrix is a good source of metrics as it provides information about the circuit breaker state via HTTP as a stream of JSON data.

Hystrix dashboard

The Hystrix dashboard can display this data on a web page and thereby show what is happening in the system at the moment, see the screenshot below.



The upper area shows the state of the circuit breaker for the functions

`getOne()`, `findAll()`, and `price()`. The circuit breakers of all three functions are `closed` and there are no errors at the moment. The dashboard also shows information about the average latency of the requests and current throughput.

Hystrix executes the calls in a separate thread pool. The state of this thread pool is also shown on the dashboard. It contains ten threads and is currently processing no requests.

Other monitoring options

The Hystrix metrics are also available via the Spring Boot mechanisms and can be exported to other monitoring systems.

Thereby, Hystrix metrics can **seamlessly integrate into an existing monitoring infrastructure**.

Turbine

- The metrics of a single microservice instance are not particularly meaningful.
- Microservices can be scaled independently.
- This means that many instances can exist for each microservice.
- This means that the Hystrix metrics of all instances must be displayed together.

This can be done with [Turbine](#). This tool queries the HTTP data streams of the Hystrix servers and consolidates them into a **single stream of data displayed by the dashboard**.

Spring Cloud offers a simple way to implement a Turbine server, see for instance: <https://github.com/ewolff/microservice/tree/master/microservice-demo/microservice-demo-turbine-server>.

QUIZ

1

In the given code, what happens if the `findAll()` method fails?

```
@HystrixCommand(  
    fallbackMethod = "getItemsCache",  
    commandProperties = {  
        @HystrixProperty(  
            name = "circuitBreaker.requestVolumeThreshold",  
            value = "2") })  
public Collection<Item> findAll() {  
    ...  
    this.itemsCache = pagedResources.getContent();  
    ...  
    return itemsCache;  
}
```

COMPLETED 0%

1 of 3



In the next lesson, we'll look at some interesting variations.

Variations

In this lesson, we'll discuss some variations to the approaches we've already looked at.

WE'LL COVER THE FOLLOWING ^

- Zuul2
- resilience4j
- Consul
- Kubernetes
- HTTP proxy
- Atom
- Frontend integration

Netflix is only one technological option for implementing synchronous microservices.

There are **various alternatives** to the technologies of the Netflix stack.

Zuul2

The Zuul project is not maintained very well anymore. An alternative might be [Zuul2](#). It is based on [asynchronous I/O](#) so it consumes less resources and is more stable. However, Spring Cloud [won't support Zuul2](#). Another alternative might be [Spring Cloud Gateway](#). The approach with Apache for routing shown in [chapter 11](#) and Kubernetes in [chapter 13](#) is probably even better.

resilience4j

Netflix does not invest in Hystrix that much anymore. They suggest using [resilience4j](#) instead, which is a very similar Java library that supports typical resilience patterns.

Consul

The **Consul example** ([chapter 11](#)) uses:

- **Consul instead of Eureka for service discovery**
- **Apache httpd instead of Zuul for routing**
- However, this project also uses **Hystrix for resilience**
- and **Ribbon for load balancing.**

Consul supports DNS and can handle any programming language as implemented in the Consul DNS example. Consul Template offers the possibility to configure services with Consul by filling a configuration file template with the data from Consul. In the example, Apache httpd is configured this way.

Eureka has quite a few advantages over Consul. Apache httpd as a web server is familiar to many developers and might therefore be less risky compared to Zuul. On the other hand, Zuul provides dynamic filters that Apache httpd does not support.

Kubernetes

Kubernetes (see [chapter 13](#)) and a *PaaS*, such as *Cloud Foundry* (see [chapter 14](#)), offer **service discovery, routing, and load balancing**. At the same time, **the code remains independent** of the infrastructure.

Nevertheless, the examples in those chapters use **Hystrix** for resilience, too.

These solutions require the use of a Kubernetes or PaaS environment. Thus, it is no longer possible to just deploy some Docker containers on a Linux server.

HTTP proxy

Functionalities like load balancing and resilience can be implemented with an **HTTP proxy** instead of Ribbon.

This is a further development of the sidecar concept. An example is [Envoy](#). This proxy implements some resilience patterns. Envoy is also part of Istio and is used as a sidecar in Istio application.

Istio is a service mesh that supports many technologies for the operation of microservice systems.

With a proxy, the application itself can be kept free of these aspects. Apache httpd or nginx can at least implement load balancing. They could also provide basic features of a sidecar.

Atom

Asynchronous communication (see [chapter 6](#)) seems to be a contradiction to communication via a synchronous protocol like REST. But *Atom* (see [chapter 8](#)) can be combined with concepts from this chapter.

Atom uses REST, so the microservices only need to implement other types of REST resources.

A collaboration with messaging systems like **Kafka** (see [chapter 11](#)) is also conceivable.

However, in this case, the system not only has the complexity of the messaging system but must also offer a REST environment.

Frontend integration

Frontend integration ([chapter 3](#)) works on a different level than REST and can be combined with the Netflix stack.

In particular, integration with **links and JavaScript** ([Chapter 4](#)) is possible without any problems.

With *ESI* (see [chapter 5](#)) Varnish instead of Zuul implements routing. Varnish would have to extract the IP addresses of the microservices from Eureka.

However, this is not possible.

Zuul and Hystrix are not maintained that much anymore.

COMPLETED 0%

1 of 3



In the next lesson, we'll look at some experiments that can be conducted with the Netflix stack.

Experiments

In this lesson, we'll discuss some experiments you can do on the Netflix stack.

WE'LL COVER THE FOLLOWING



- Try the experiments in the following widget!
- Additional microservice
- Scaling and load balancing
- Simulate failure
- Extend access using Hystrix
- Extend Zuul setup
- Filter with Zuul
- Create your own microservice

Try the experiments in the following widget!

Additional microservice

Supplement the system with an additional microservice.

- A microservice that is used by a call center agent to create notes for a call can be used as an example. The call center agent should be able to select the customer.
- You can copy and modify one of the existing microservices.
- Register the microservice in Eureka.
- The customer microservice must be called via Ribbon. The microservice will be found automatically via Eureka, otherwise, the microservice must

will be found automatically via Eureka, otherwise, the microservice must be looked up explicitly in Eureka.

- Package the microservice in a Docker image and add the image to `docker-compose.yml`. There you can also determine the name of the Docker container.
- Create a link in `docker-compose.yml` from the container with the new service to the container `eureka`. That way the microservice can register at the Eureka server.
- The microservice must be accessible from the homepage. To do this, you have to create a link similar to the other links in the file `index.html` in the Zuul project. Zuul automatically sets up the routing for the microservice as soon as the microservice is registered in Eureka.

Scaling and load balancing

Try scaling and load balancing.

- Increase the number of instances of a service with `docker-compose up --scale customer=2`.
- Use the Eureka dashboard to determine whether two customer microservices are running. It is available at port 8761, at <http://localhost:8761/> when Docker is running on the local computer.
- Observe the logs of the order microservice with `docker logs -f ms_order_1` and have a look at whether different instances of the customer microservice are called. This should be the case because Ribbon is used for load balancing. For this, you have to trigger requests to the order application, e.g., a simple reload of the starting page.

Simulate failure

Simulate the failure of a microservice.

- Watch the logs of the order microservice with `docker logs -f ms_order_1` and have a look at how the catalog microservice is called. For this, you have to trigger requests to the order application. For example, you can reload the starting page.

- Find the IP address of the order microservice with the help of the Eureka dashboard at port 8761, <http://localhost:8761> when Docker is running locally.
- Open the Hystrix dashboard at port 8989 on the Docker host, at <http://localhost:8989> when Docker is running on the local computer.
- Using this IP address enter the URL of the Hystrix JSON data stream in the Hystrix dashboard. This can be <http://172.18.0.6:8080/actuator/hystrix.stream>. The Hystrix dashboard should show closed circuit breakers.
- Shut down all catalog instances with `docker-compose up --scale catalog=0`.
- Watch the log of the order microservices during the next calls.
- Also observe the Hystrix dashboard. The circuit breaker will only open when multiple calls have failed.
- When the circuit breaker is open, the order microservice should work again since the fallback is activated, then a cached value is used.

Extend access using Hystrix

Only access to the catalog microservice is safeguarded with Hystrix.

In the order microservice the class `CustomerClient` in package `com.ewolff.microservice.order.clients` implements the access to the customer microservice.

Extend the access to the customer microservice using Hystrix. For this, use the class `CatalogClient` from the same package as an example.

Extend Zuul setup

Extend the Zuul setup by a fixed route. In `application.yml`, in directory `src/main/resource`, in project `microservice-demo-zuul-server` add for example the following to make the INNOQ homepage appear at

```
zuul:  
  routes:  
    innoq:  
      path: /innoq/**  
      url: http://innoq.com/
```

Filter with Zuul

Add a filter to the Zuul configuration.

A tutorial dealing with Zuul filters can be found at
<https://spring.io/guides/gs/routing-and-filtering/>.

Create your own microservice

- Create your own microservice that only returns simple HTML.
- Integrate it into Eureka and deploy it as part of the Docker compose environment.
- If it is registered in Eureka, it can be addressed immediately from the Zuul proxy at a URL like <http://localhost:8080/mymicroservice> .

We'll conclude this chapter with the next lesson.

Chapter Conclusion

We'll conclude this chapter with a quick summary.

WE'LL COVER THE FOLLOWING ^

- Summary
- Service discovery
- Resilience
- Load balancing
- Routing
- Advantages
- Challenges

Summary

The Netflix stack provides a variety of projects to build microservice architectures. The stack solves the typical challenges of synchronous microservices as follows:

Service discovery

Service discovery is offered by Eureka.

Eureka focuses on Java with the Java client, but also offers a REST API and libraries for other languages and can therefore be used with other languages.

Resilience

For **resilience**, Hystrix is the de facto standard for Java and covers this area very well.

- Non-Java applications can use Hystrix via a sidecar at most.
- There are ports of Hystrix for other languages like Go.

- Hystrix is independent of the other technologies and can be used on its own.

Load balancing

Ribbon implements client-side **load balancing**, which has many advantages.

However, since Ribbon is a Java library, other technologies are difficult to use with Ribbon, especially in the area of load balancing. There are numerous load balancers that provide alternatives.

Ribbon relies on **Eureka** for service discovery but **can also use Consul**.

Routing

Routing is solved by Zuul. Zuul's dynamic filters are very flexible, but many developers are familiar with reverse proxies based on web servers like Apache httpd or nginx.

In this case, a reverse proxy might be the safer option. Additional features like SSL termination or request throttling may also be required, which Zuul does not offer.

[Chapter 11](#) shows how Apache httpd can be used with Consul to provide routing. Zuul requires Eureka to find the microservices.

The servers in the Netflix stack are written in Java so that the servers from the Netflix stack and the microservices can be packed into JAR files.

Thanks to Spring Cloud, they are uniformly configurable.

The handling of metrics and logs is identical and this uniformity can be an advantage for operation.

Advantages

- Eureka and client-side caching is very fast and resilient.
- Zuul is very flexible because of its filters.
- Client-side load balancing avoids single points of failure or bottlenecks.

- Hystrix is very mature and the de facto standard for Java.

Challenges

- The Netflix stack implements many solved problems (e.g. reverse proxy, service discovery) again.
 - The technologies focus on Java and therefore limit technology freedom.
 - The code depends on the Netflix stack, Ribbon, Hystrix, and also Eureka due to `@EnableDiscoveryClient` and the client API.
-

In the next chapter, we'll discuss a new recipe: REST with Consul and Apache httpd.

Introduction

In this lesson, we'll look at a quick introduction to Consul and a chapter walkthrough.

WE'LL COVER THE FOLLOWING ^

- Where does Consul come from?
- License and technology
- Chapter walkthrough

Where does Consul come from?

Consul is a product of the company [Hashicorp](#), which offers various products in the field of microservices and infrastructure. Of course, Hashicorp also offers commercial support for Consul.

License and technology

[Consul](#) is an open source product. It is written in Go and is licensed under the [Mozilla Public License 2.0](#). The code is available at [GitHub](#).

Chapter walkthrough

This chapter shows an implementation for a synchronous microservices system with Consul and the Apache httpd server.

Essential contents of the chapter are:

- Consul is a very powerful service discovery technology.
- Apache httpd can be used as a load balancer and router for HTTP requests in a microservices system.
- Consul Template can create a configuration file for the Apache httpd server that includes information about all registered microservices.

Consul Template configures and restarts Apache httpd when new microservice instances are started.

QUIZ

1

What is consul?

COMPLETED 0%

1 of 2



In the next lesson, we'll look at an example of REST with Consul and Apache httpd.

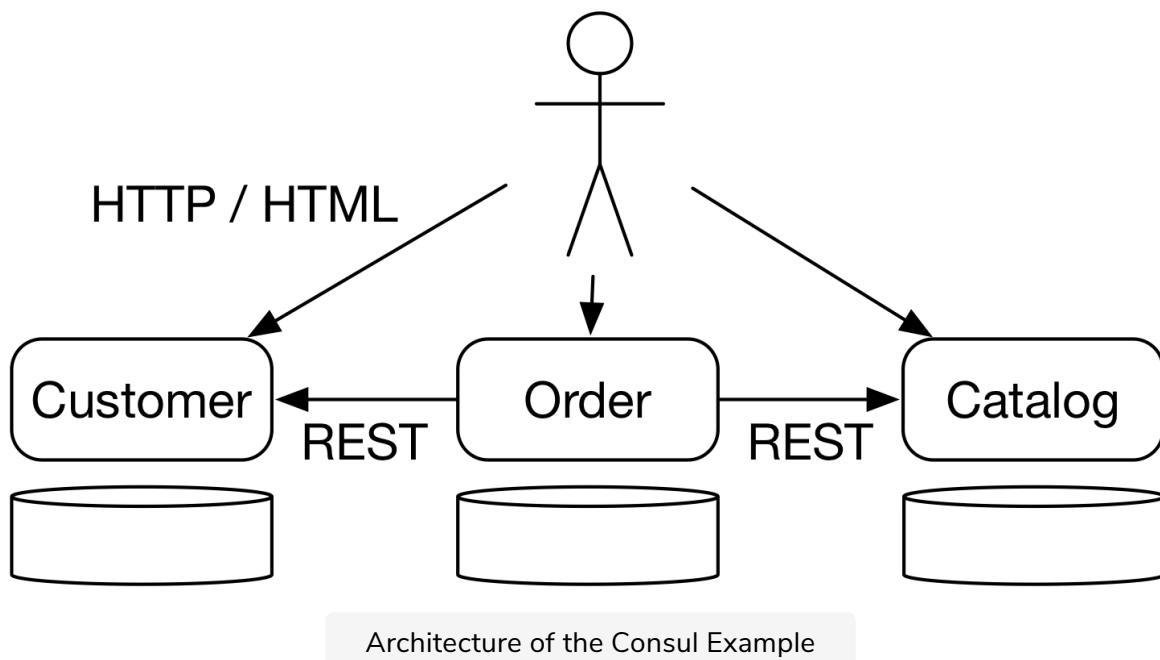
Example

In this lesson, we'll introduce an example.

WE'LL COVER THE FOLLOWING ^

- Architecture of the example
- Building the example

The domain structure is identical to the example in the Netflix chapter ([chapter 10](#)) (see the drawing below) and consists of **three microservices**.



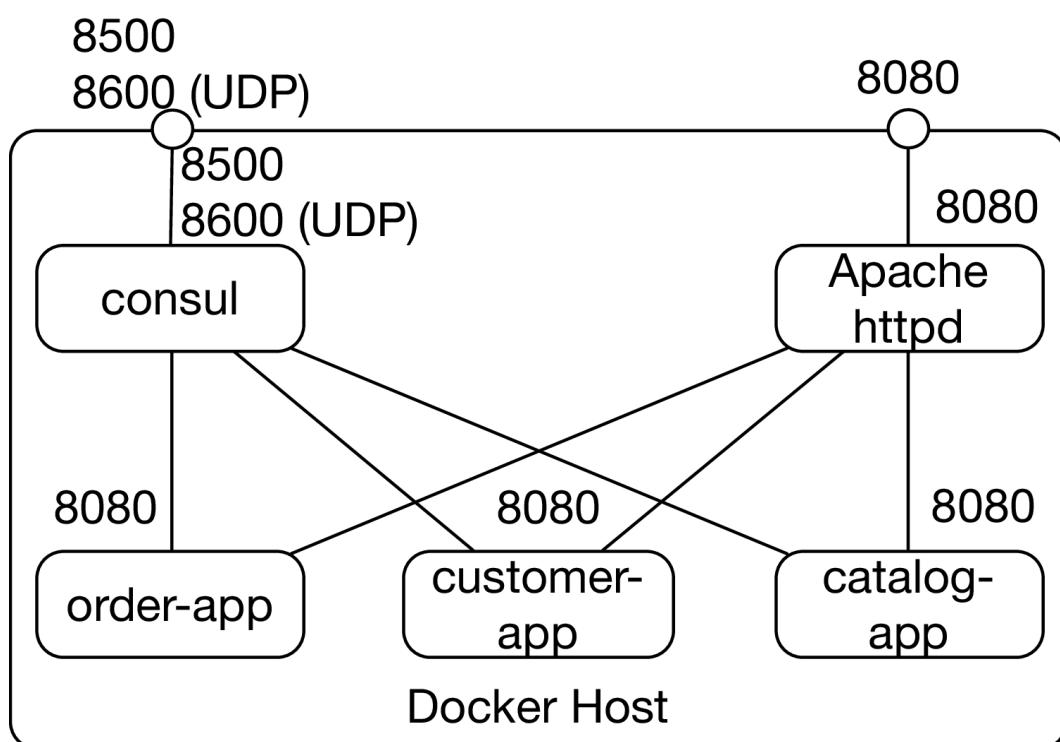
- The **catalog** microservice manages the information such as price and name for the items that can be ordered.
- The **customer** microservice stores customer data.
- The **order** microservice can accept new orders. It uses the catalog and customer microservice.

Architecture of the example

The example in this chapter uses [Consul](#) for service discovery and [Apache httpd server](#) for routing the HTTP requests.

An overview of the Docker containers is shown in the drawing below. The three microservices provide their UI and REST interfaces at the port 8080. They are only accessible within the network between the Docker containers. Consul offers port 8500 for the REST interface and the HTML UI as well as UDP port 8600 for DNS requests.

These two ports are bound to the Docker host and are accessible from other computers. The Docker host also provides the Apache httpd at port 8080. Apache httpd forwards calls to the microservices in the Docker network so that the microservices can also be accessed from outside.



Overview of the Consul Example

Building the example

First download the code with `git clone`

<https://github.com/ewolff/microservice-consul.git>. Then the code has to be translated with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in directory `microservice-consul-demo`. See [this lesson](#) for more details on Maven and how to troubleshoot the build. Afterwards the Docker containers can be built by running `./mvnw dockerfile:build`.

Docker containers can be built in the directory `docker` with `docker-compose build` and started with `docker-compose up -d`. See [this lesson](#) for more details on Docker, docker-compose and how to troubleshoot them. Subsequently, the Docker containers are available on the Docker host.

When the Docker containers are running on the local computer, the following URLs are available:

- <http://localhost:8500> is the link to the Consul dashboard.
- <http://localhost:8080> is the URL of the Apache httpd server. It can display the web UI of all microservices.

<https://github.com/ewolff/microservice-consul/blob/master/HOW-TO-RUN.md> describes the necessary steps for building and running the example in detail.

In the next lesson, we'll look at service discovery with Consul.

Service Discovery

In this lesson, we'll study service discovery with Consul.

WE'LL COVER THE FOLLOWING ^

- Distinguishing features
 - HTTP REST API & DNS support
 - Configuration file generation
 - Health checks
 - Replication
 - Multiple data centers
 - Service configuration
- Consul dashboard
- Reading data with DNS
- Consul Docker image

Consul is a service discovery technology that ensures microservices can communicate with each other.

Distinguishing features

Consul has some features that set it apart from other service discovery solutions.

HTTP REST API & DNS support

Consul has an *HTTP REST API* and supports **DNS**.

- **DNS** (Domain Name System) is the system that maps host names such as www.innoq.com to IP addresses on the Internet.
- In addition to returning IP addresses, it can return ports at which a service is available.

- This is a feature of the SRV DNS records.

Configuration file generation

With [Consul Template](#), Consul can generate configuration files.

- The files may contain IP addresses and ports of services registered in Consul.
- *Consul Template* also provides Consul's service discovery to systems that cannot access Consul via the API.
- The systems have to use some kind of configuration file, which they often already do anyway.

Health checks

Consul can perform **health checks** and exclude services from service discovery when the health check fails.

- For example, a health check can be a request to a specific HTTP resource to determine whether the service can still process requests.
- A service may still be able to accept HTTP requests, but it may not be able to process them properly due to a database failure.
- The service can signal this through the health check.

Replication

Consul supports **replication** and can ensure high availability.

If a Consul server fails, other servers with replicated data take over and compensate for the failed server.

Multiple data centers

Consul also supports **multiple data centers**.

- Data can be replicated between data centers to further increase availability and protect Consul against the failure of a data center.
- The search for services can be limited to the same data center.
- Services in the same data center usually deliver higher performance.

Service configuration

Finally, Consul can be used not only for service discovery, but for the **configuration of services**.

- Configuration has different requirements.
- Availability is important for service discovery.
- Faulty information can be tolerated.
- If the service is accessed and is not available, it doesn't matter. You can simply use another instance of the service.
- However, if services are configured incorrectly, this can lead to errors.
- **Correct information is more important** for configuration than for service discovery.

Consul dashboard

Access to the information about registered microservices is provided by Consul in its **dashboard**, see the screenshot below.

It shows **all services** that have registered at the Consul server and the result of their **health checks**.

In the example, all microservices are healthy and can accept requests.

In addition to the registered services, it displays the **servers on which Consul is running** and the **contents of the configuration database**. Access to the dashboard is possible at port **8500** on the Docker host.

The screenshot shows the Consul UI interface. At the top, there are tabs for SERVICES, NODES, KEY/VALUE, and ACL. The SERVICES tab is selected, indicated by a purple border. Below the tabs, there is a 'DC1' dropdown menu and a gear icon. A search bar labeled 'Filter by name' and a dropdown menu for 'any status' are also present. The main area lists four services: 'catalog' (2 passing), 'consul' (1 passing), 'customer' (2 passing), and 'order' (2 passing). Each service entry has a green vertical bar to its left.

Consul Dashboard

Reading data with DNS

It is also possible to access the data from the Consul server via DNS. This can be done, for example, with the tool `dig`. `dig @localhost -p 8600 order.service.consul`. returns the IP address of the order microservice if the Docker containers run on the local computer `localhost`.

Communication to the Consul DNS server takes place via the UDP port 8600. `dig @localhost -p 8600 order.service.consul. SRV` returns the IP address and the port at which the service is available. DNS SRV records are used for this purpose; they are part of the DNS standard and allow you to specify a port for services in addition to the IP address.

Consul Docker image

The example uses a Consul Docker image directly from the manufacturer Hashicorp. It is configured so that Consul only stores the data in the main memory and runs on a single node. This simplifies the setup of the system and reduces resource consumption.

Reduces resource consumption.

This configuration is **unsuitable for a production environment** because data can be lost, and a failure of the Consul Docker container would bring the entire system to a standstill. In production, a cluster of Consul servers should be used, and the data should be stored persistently.

QUIZ

1

How can Consul implement a health check on a microservice that lists customers?

COMPLETED 0%

1 of 2



In the next lesson, we'll discuss Apache httpd.

Routing: Apache httpd

In this lesson, we'll discuss routing with Apache httpd.

WE'LL COVER THE FOLLOWING ^

- Introduction
- Reverse proxy
- Load balancer

Introduction

The [Apache httpd server](#) is one of the most widely used web servers. There are modules that adapt the server to different usage scenarios. In the example, modules are configured that turn Apache httpd into a reverse proxy.

Reverse proxy

While a **conventional proxy** can be used to process traffic from a **network to the outside**, a **reverse proxy** is a solution for **inbound network** connections.

It can forward external requests to specific services. This means that the entire microservices system can be accessible under one URL but can use different microservices internally.

The concept of a reverse proxy has already been explained in the lesson, [Router: Zuul](#).

Load balancer

In addition, Apache httpd serves as a **load balancer** by distributing network traffic to make the application scalable.

In the example, there is only one Apache httpd, that functions simultaneously as a reverse proxy and a load balancer for requests from the outside.

The requests the microservices send to each other are not handled by this load balancer. For the communication between the microservices, the library **Ribbon** is used, as we already saw in the Netflix example (see [Load Balancing: Ribbon](#)).

One of the strengths of this solution is that it uses a **well-proven software** that many teams have already gained experience with.

As microservices place high demands on the operation and infrastructure, such a conservative choice is advantageous to avoid the effort that goes into learning another technology. Instead of Apache httpd you can also use example [nginx](#).

There are also approaches like [Fabio](#) which are written specifically for the load balancing of microservices and are easier to use and configure.

Q U I Z

1

Can Apache httpd work as a load balancer and reverse proxy at the same time?

COMPLETED 0%

1 of 3



In the next lesson, we'll discuss the Consul template.

Consul Template

In this lesson, we'll discuss Consul Templates.

WE'LL COVER THE FOLLOWING

- Introduction
- The template
- Starting the Consul Template
- Conclusion

Introduction

For each microservice, Apache httpd must have an entry in its configuration file. For this, [Consul Template](#) can be used.

In the example, the `00-default.ctmpl` file is used as a template to create the Apache httpd configuration. It is written in the [Consul Templating Language](#). For each microservice, it creates an entry that distributes the load between the instances and redirects external requests to these instances.

The template

The essential part is:

```
 {{range services}}  
  
 <Proxy balancer://{{.Name}}>  
 {{range service .Name}}  BalancerMember http://{{.Address}}:{{.Port}}  
 {{end}}  
 </Proxy>  
 ProxyPass      /{{.Name}} balancer://{{.Name}}  
 ProxyPassReverse /{{.Name}} balancer://{{.Name}}  
  
 {{end}}
```

- The Consul Template API functions are enclosed in `{{` and `}}`.
- For each service a configuration is generated with `{{range service}}`.
- It contains a reverse proxy that is configured with the `<Proxy>` element and the `Name` of the microservice.
- This element contains the microservice instances as `BalancerMember` with the `Address` and `Port` of each instance for distributing the load between the microservice instances.
- The end is formed by `ProxyPass` and `ProxyPassReverse` which, likewise, belong to the reverse proxy.

Starting the Consul Template

The Consul Template is started with the following section from the `Dockerfile` in the directory `docker/apache`:

```
CMD /usr/bin/consul-template -log-level info -consul consul:8500 \\
    -template "/etc/apache2/sites-enabled/000-default.ctmpl:/etc/apache2/sites-enabled/000-default.conf:apache2ctl -k graceful"
```

Consul Template executes the command `apache2ctl -k graceful` if there are new services or if services have been removed and the configuration has therefore been changed. This causes Apache httpd to read the updated configuration and restart. However, open connections are not closed, they remain open until communication is terminated. If no Apache httpd is running yet, one will be started. Thus, Consul Template takes control of Apache httpd and ensures that one instance of Apache httpd is always running in the Docker container.

To do this, Consul Template must run in the same Docker container as Apache httpd. This contradicts the Docker philosophy that only one process should run in one container. However, this cannot be avoided in the concrete example because these two processes are so closely related.

Conclusion

Consul Template can ensure that a microservice can be reached from outside as soon as it has registered with Consul. To do this, the Apache httpd server does not need to know anything about the service discovery or Consul. It

does not need to know anything about the service discovery or Consul. It receives the information in its configuration and restarts.

In the next lesson, we'll study Consul and Spring Boot.

Consul and Spring Boot

In this lesson, we'll discuss Consul and Spring boot.

WE'LL COVER THE FOLLOWING



- Introduction
- Code dependencies
- Health check with Spring Boot Actuator
- Consul and Ribbon

Introduction

The Consul integration in Spring Boot is comparable to the integration of Eureka (see [Eureka: Service Discovery](#)).

There is a configuration file `application.properties`. Here is the relevant section:

```
spring.application.name=catalog
spring.cloud.consul.host=consul
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.preferIpAddress=true
spring.cloud.consul.discovery.instanceId=\${spring.application.name}:
\${spring.application.instance_id:\${random.value}}
```

The section configures the following values:

- `spring.application.name` defines the name under which the application is registered in Consul.
- `spring.cloud.consul.host` and `spring.cloud.consul.port` determine at which port and on which host Consul can be accessed.
- Via `spring.cloud.consul.discovery.preferIpAddress` the services register

with their IP address and not with the host name. This circumvents

problems that arise because host names cannot be resolved in the Docker environment.

- `spring.cloud.consul.discovery.instanceId` assigns an unambiguous ID to each microservice instance for discriminating between instances for load balancing.

Code dependencies

In addition, a dependency to `spring-cloud-starter-consul-discovery` has to be inserted in `pom.xml` for the build. Moreover, the main class of the application that has the annotation `@SpringBootApplication` has to be annotated with `@EnableDiscoveryClient`.

Health check with Spring Boot Actuator

Finally, the microservice must provide a health check. Spring Boot contains the Actuator module for this, which offers a health check as well as metrics. The health check is available at the URL `/health`. This is exactly the URL Consul requests. It is enough to add a dependency to `spring-boot-starter-actuator` in the `pom.xml`. Specific health checks may need to be developed if the application depends on additional resources.

Consul and Ribbon

Of course, a microservice must also use Consul to communicate with other microservices. For this, the Consul example uses the Ribbon library analogous to the Netflix example (see [Load Balancing: Ribbon](#)).

Ribbon has been modified in the Spring Cloud project so that it can also deal with Consul. Because the microservices use Ribbon, the rest of the code is unchanged compared to the Netflix example.

Q U I Z

1

Which host can Consul be accessed on in the following configuration file?

```
spring.application.name=catalog
spring.cloud.consul.host=consul_host
spring.cloud.consul.port=9500
spring.cloud.consul.discovery.preferIpAddress=true
spring.cloud.consul.discovery.instanceId=\${spring.application.name}:
\${spring.application.instance_id:\${random.value}}
```

COMPLETED 0%

1 of 2



In the next lesson, we'll discuss DNS and registrar.

DNS and Registrar

WE'LL COVER THE FOLLOWING



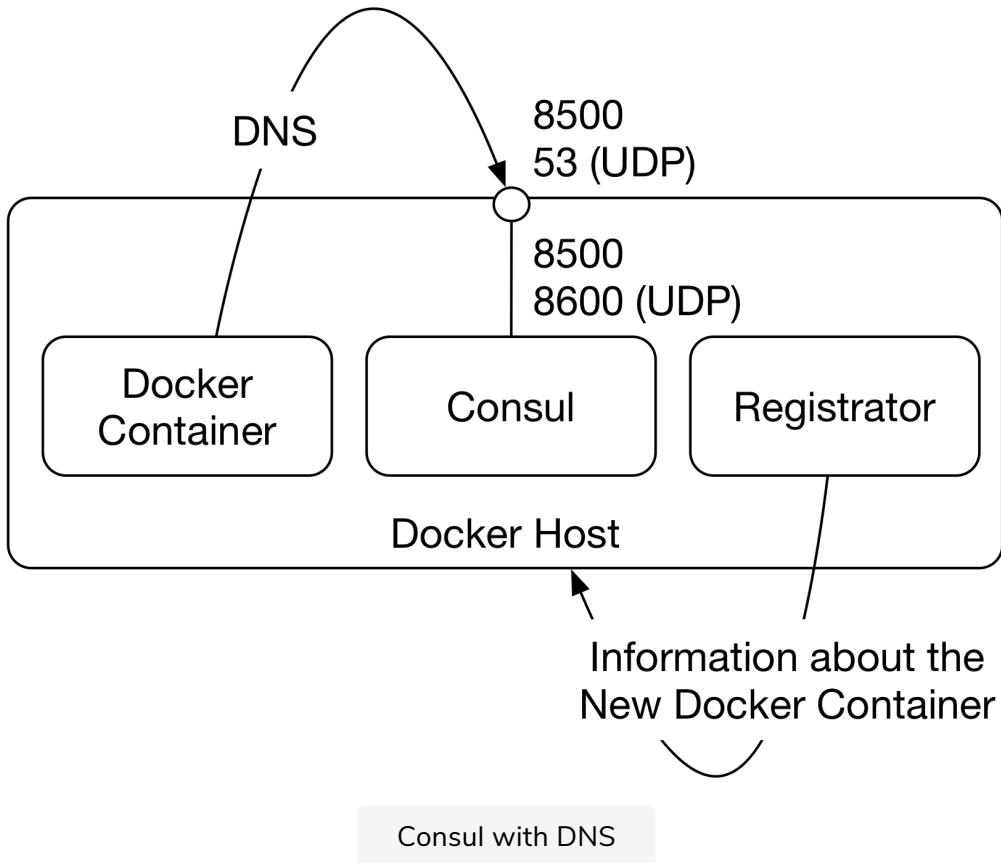
- Introduction
- Structure of the example
- Configuring DNS access
- Consul
- Configuration is also possible in a transparent manner

Introduction

The microservices have code dependencies to the Consul API for **registering**. This is not necessary, a [Registrar](#) can register Docker containers with Consul without the need for code.

When the Docker containers are configured in such a way that they use Consul as a DNS server, the lookup of other microservices can also occur without code dependencies. This eliminates any dependencies on Consul in the project.

Structure of the example



The drawing above shows an overview of the approach.

- **Registrator** runs in a Docker container. Via a socket, Registrator collects information from the Docker daemon about all newly launched Docker containers. The Docker daemon runs on the Docker host and manages all Docker containers.
- The **DNS interface of Consul** is bound to the UDP port **53** of the Docker host. This is the default port for DNS.
- The **Docker containers** use the Docker host as a DNS server.

Configuring DNS access

The `dns` setting in the `docker-compose.yml` is configured to use the IP address in the environment variable `CONSUL_HOST` as the IP address of the DNS server. Therefore, the IP address of the Docker host has to be assigned to `CONSUL_HOST` before starting `docker-compose`. Unfortunately, it is not possible to configure DNS access from the Docker containers in a way that does not use this environment variable.

Consul

Registrar registers every Docker container started with Consul; not only the microservices, but the Apache httpd server or Consul itself can be found among the services in Consul.

Consul registers the Docker containers with `.service.consul` added to the name. In `docker-compose.yml`, `dns_search` is set to `.service.consul` so that this domain is always searched. In the end, the order microservice uses the URLs `http://msconsuldns_customer:8080/` and `http://msconsuldns_catalog:8080/` to access the customer and catalog microservices.

Docker compose creates the prefix `msconsuldns` for the name of the Docker containers to separate this project from other projects. This name is also used by Consul Template to configure the Apache httpd for routing.

In this setup, Consul is responsible for load balancing. If there are several instances of a microservice, Registrar registers them all under the same name. Consul then returns one of the instances for each DNS request.

The executable example can be found at

<https://github.com/ewolff/microservice-consul-dns> and the instructions for starting it, at <https://github.com/ewolff/microservice-consul-dns/blob/master/HOW-TO-RUN.md>.

As a result of this approach, the microservices **no longer contain any code dependencies on Consul**. Therefore, with these technologies, it is no problem to implement microservices with a programming language other than Java.

Configuration is also possible in a transparent manner

[Envconsul](#) also enables configuration data to be read from Consul and made available to the applications as environment variables. In this way, Consul can also configure the microservices. Without this they have to include Consul-specific code.

In the next lesson, we'll look at some variations of the approaches we've already discussed.

Variations

In this lesson, we'll look at some variations in the approach we've discussed so far.

WE'LL COVER THE FOLLOWING ^

- Combination with frontend integration
- Combination with asynchronous communication
- Other load balancers
- Service meshes

Consul is very flexible and can be used in many different ways.

Combination with frontend integration

Like other approaches Consul can be combined with frontend integration (see [chapter 3](#)).

SSI (Server-side Includes) with Apache httpd is simple to combine since an Apache httpd is already present in the system.

Combination with asynchronous communication

Synchronous communication can be combined with asynchronous communication (see [chapter 6](#)). However, one type of communication should suffice normally.

Atom or other asynchronous approaches via HTTP (see [chapter 8](#)) are easy to integrate into an HTTP-based system like Consul.

Other load balancers

Instead of Apache httpd, a server like **nginx** or a load balancer like **HAProxy** can also be used for routing of requests from the outside.

Ribbon can be replaced by such a load balancer so that the internal load balancing is then configured with Consul Template.

In this case, only one type of load balancer is used for load balancing and routing. Unlike the Java library Ribbon, Apache httpd or nginx can be used with any programming language.

Each microservice then has its own httpd or nginx instance so that no bottleneck or single point of failure is created.

Service meshes

Service meshes provide a lot of useful features for resilience, monitoring, tracing, and logging. Istio is an example of a service mesh.

A service mesh injects proxies into the communication between the microservices. Istio supports Consul to achieve that.

With Istio, Consul can be extended to become a **complete platform for the operation** of microservices.

Q U I Z

1

Why is it easy to combine Consul with frontend?

In the next lesson, we'll look at some experiments that can be conducted with the approach we've learned in this chapter.

Experiments

In this lesson, we'll look at some experiments that can be conducted.

WE'LL COVER THE FOLLOWING ^

- Supplement with additional microservice & without DNS
- Supplement with additional microservice & with DNS
- Run on a cluster
- Configure Spring Boot
- Replace Apache httpd with another server
- Scaling & load balancing

Supplement with additional microservice & without DNS

Supplement the Consul system without DNS with an additional microservice.

- A microservice that is used by a call center agent to create notes for a call can be used as an example. The call center agent should be able to select the customer.
- You can copy and modify one of the existing microservices.
- Register the microservice in Consul.
- Ribbon has to be used to call the customer microservice and does the lookup of the microservice in Consul. Otherwise, the microservice must be searched explicitly in Consul.
- Package the microservice in a Docker image and reference the image in `docker-compose.yml`. You can also specify the name of the Docker container.

- Create a link in `docker-compose.yml` from the container with the new service to the container `consul`.
- The microservice must be accessible from the homepage. To do this, you have to create a link in the file `index.html` in the Docker container `apache`. Consul Template automatically sets up the routing for the microservice in Apache as soon as the microservice is registered in Consul.

Supplement with additional microservice & with DNS

Supplement the DNS Consul system (see [DNS and Registrar](#)) with an additional microservice.

- A microservice that is used by a call center agent to create notes for a call can be used as an example. The call center agent should be able to select the customer.
- The call to the customer microservice has to use the host name `msconsuldns_customer`.
- You can copy and modify one of the existing microservices.
- A registration in Consul is not necessary since Registrar automatically registers each Docker container.
- Package the microservice in a Docker image and reference the image in `docker-compose.yml`. There you can also specify the name of the Docker container.
- Configure the DNS server for the new microservice in `docker-compose.yml` similar to the other microservices.
- The microservice must be accessible from the homepage. To do this you have to create a link in the file `index.html` in the Docker container `apache`. Consul Template automatically sets up the routing for the microservice in Apache as soon as the microservice is registered in Consul.

Run on a cluster

Currently, the Consul installation is not a cluster and is therefore unsuitable for a production environment. Change the Consul installation so that Consul runs in the cluster and the data from the service registry is saved to a hard disk. To do this, the configuration has to be changed in the [Dockerfile](#) and several instances of Consul have to be started. For more information, see <https://www.consul.io/docs/guides/bootstrapping.html>.

Configure Spring Boot

Consul can also be used for saving the configuration of a Spring Boot application, see <https://cloud.spring.io/spring-cloud-consul/#spring-cloud-consul-config>. Use Consul to configure the example application with Consul.

Replace Apache httpd with another server

Replace Apache httpd with nginx, another web server, or HAProxy. To do this, you need to create an appropriate Docker image or search for a matching Docker image in the [Docker hub](#). In addition, the web server must be provided with reverse proxy extensions and configured with Consul Template. Additional documentation about Consul Template can be found on the [Github web page](#). For many systems there are also [Consul Template examples](#).

Scaling & load balancing

Try scaling and load balancing.

- Increase the number of instances of a service, for example with `docker-compose up --scale customer=2`.
- Use the Consul dashboard to check if two customer microservices are running. It is available under port 8500, at <http://localhost:8500> when Docker is running on the local computer.
- Observe the logs of the order microservice with `docker logs -f msconsul_order_1` and see if different instances of the customer microservice are called. This should be the case because Ribbon is used for load balancing. To do this, you must trigger requests to the order application by reloading the homepage.

application by reloading the homepage.

- Add your own health check for one of the microservices. Check whether load balancing actually excludes a service when the health check is no longer successful.
-

We'll conclude this chapter with a quick summary in the next lesson.

Chapter Conclusion

In this lesson, we'll look at a quick conclusion to the chapter.

WE'LL COVER THE FOLLOWING ^

- Summary
 - Service discovery
 - Increased transparency
 - Configuration
 - Resilience
 - Routing
 - Load balancing
- Comparison to Netflix
- Advantages
- Challenges

Setting up a microservices system with Consul is another option for a synchronous system.

Summary

This infrastructure meets the typical challenges of synchronous microservices as follows.

Service discovery

Service discovery is covered by Consul. Consul is very flexible; due to the DNS interface and Consul Template it can be used with many technologies. This is particularly important in the context of microservices. While a system might not need to use a variety of technologies from the start, in the long term it is advantageous to be able to integrate new technologies.

Increased transparency

Consul is **more transparent** to use than Eureka. The Spring Cloud applications still need special Consul configurations, but Consul offers a configuration in Apache format for Apache httpd so at least in this case Consul is transparent.

If Registrator is used to register the microservices, and Consul is used as a DNS server, Consul is **fully transparent** and can be used without any code dependencies. With Envconsul, Consul can even configure the microservices without code dependencies.

Configuration

Consul can be used to **configure** the microservices. In this way, with only one technological approach, both service discovery and configuration can be implemented.

Resilience

Resilience is not implemented in this example.

Routing

Routing with Apache httpd is a relatively common approach. This reduces the technological complexity, which is quite high in a microservices system anyway. With the large number of new technologies and a new architectural approach, it is helpful to cover some areas with established approaches.

Load balancing

Load balancing is implemented with Ribbon, like in the Netflix example (see [Load Balancing: Ribbon](#)). However, it is not a problem to provide each microservice instance with an Apache httpd which is configured by Consul Template in such a manner that it provides load balancing for outbound calls. For Consul DNS, Consul even implements load balancing transparently with the DNS server. In that case, no additional technology for load balancing is needed.

Comparison to Netflix

- The technology stack from this example has the **advantage** that it also

- The technology stack from this example has the **advantage** that it also supports heterogeneous microservices systems because there are no more code dependencies on Consul if using DNS.
- Consul as a service discovery technology is more powerful than Eureka with the DNS interface it provides and Consul Template.
- Apache is a standard reverse proxy that is widely used. It is therefore more mature than Zuul.
- Zuul is not supported any more while Apache is still one of the most broadly used web servers.
- For resilience, the stack does not offer a good solution. However, it can still be combined with libraries such as Hystrix.
- The main benefit of the Consul technology stack is its **independence from a concrete language** and environment.
- The Netflix stack is based on Java and it is hard to integrate other languages.
- Netflix has discontinued several of the projects, i.e., Hystrix and Zuul.
- The Consul stack is usually preferable over the Netflix stack.

Advantages

- Consul does not have a Java focus but supports many different technologies.
- Consul supports DNS.
- Consul Template can configure many services (Apache httpd) transparently via configuration files.
- Entirely transparent registration and service discovery with Registrar and DNS are possible.
- The use of well-established technologies such as Apache httpd reduces the risk.

Challenges

- Consul is written in Go. Therefore, monitoring and deployment differ from Java microservices.
-

That's it for this chapter! From the next chapter onwards, we'll discuss microservices platforms.

Introduction

We'll kick off this chapter with a quick walkthrough of what is to come.

WE'LL COVER THE FOLLOWING ^

- Chapter walkthrough

Chapter walkthrough

The following chapters describe microservices platforms. Readers learn that:

- Microservices platforms **provide support** for the operation and also for the communication of microservices.
- **PaaS (Platform as a Service)** cloud offerings and Docker scheduler are examples of microservices platforms.
- Microservices platforms **have their advantages and disadvantages**, which makes them superior to other approaches in some scenarios.

Q U I Z

1

What do microservices platforms do?

COMPLETED 0%

1 of 2



In the next lesson, we'll discuss what microservices platforms are.

Definition

In this lesson, we'll look at the definition of microservices platforms.

WE'LL COVER THE FOLLOWING



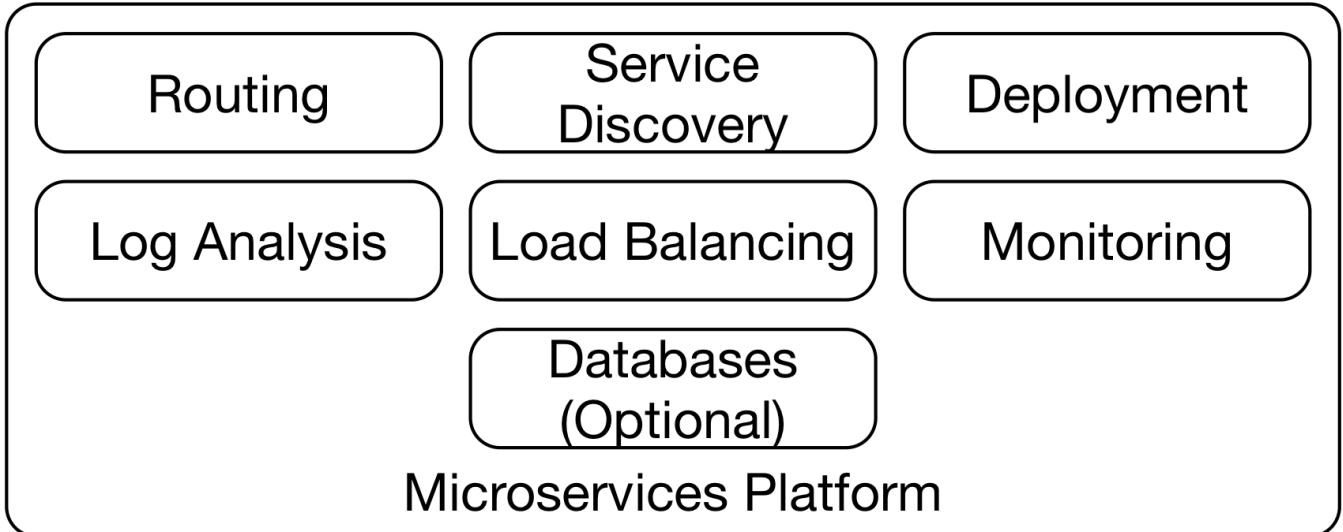
- Support for HTTP and REST
- Expenditure for installation and operation
- Migration to a microservices platform
- Influence on the macro architecture
- Specific platforms

The platforms in the next chapters differ from all other technologies presented so far in that they enable the communication of microservices and support aspects of operation such as deployment, monitoring, or log analysis.

Support for HTTP and REST

The platforms support HTTP and REST with load balancing, routing, and service discovery, but can be supplemented with other communication mechanisms. This makes it possible to use the platforms for setting up asynchronous systems.

However, this chapter considers synchronous communication. Microservices on platforms that use this type of communication differ most from microservices on different infrastructures concerning their mode of operation.



Features of Microservices Platforms

Expenditure for installation and operation

Microservices platforms are very powerful, but consequently very complex. They make it easier to work with microservices but installing and operating the platform itself can really be a challenge.

If the platform is running in the public cloud, the complex installation and operation does not play a role, because the operator of the public cloud has to take care of it. However, if an installation is carried out in the company's own data center, the operations team has to make the effort.

However, the effort for installing the platform has to be made only once. After that, deploying the microservices is much easier. This means that the cost of installing the platform will be amortized quickly.

Only limited operations support is necessary when installing the microservices. In this way, microservices can be deployed quickly and easily even if operations cannot support each deployment of each microservice.

Migration to a microservices platform

In contrast to the solutions shown so far, microservices platforms require a **fundamental change in the operation and installation of the applications**.

The other examples can be run on virtual machines or even physical servers and can be combined with existing deployment tools.

Since the microservices platforms also cover the deployment and operation of the microservices, they include features for which operations have already established tools in most cases. Therefore, the use of a microservices platform is **a bigger step than the use of other technologies**. Such a move can be a deterrent for conservative operations teams.

In addition, the introduction of microservices often leads to **organizational changes** and new technologies in addition to a new architecture.

It can be helpful if a complex microservices platform does not have to be introduced in this situation.

Influence on the macro architecture

Of course, a platform only supports certain technologies. In addition to the programming language, these are the technologies for deployment, monitoring, logging, and so on. The platform therefore defines large parts of the technical infrastructure.

Therefore, microservices platforms have a relationship to the macro architecture.

- The macro architecture should state the **requirements of the platform** ensuring that the platform can run the microservices. The platform specifies deployment and logging in the macro architecture.
- The microservices platform **restricts the options** the team can choose in the macro architecture. After all, it is impossible to operate microservices that cannot be run on the platform. For example, the platform can define restrictions concerning the programming language.
- The microservices platform can **enforce** compliance with macro architecture rules. If developers disregard the rules, the microservices cannot run on the platform, ensuring that the rules are actually observed.

Specific platforms

The following two chapters show two different approaches for microservices platforms:

platforms.

- **Kubernetes** (see [chapter 13](#)) can run Docker containers and solves challenges such as load balancing, routing, and service discovery at the network level. It is quite flexible since it is able to run arbitrary Docker containers. With [Operators](#) or [Helm](#), other services can be integrated into Kubernetes for monitoring.
- **Cloud Foundry** (see [chapter 14](#)) supports applications. All you have to do is provide Cloud Foundry with a Java application. Cloud Foundry creates a Docker container from it, which can then be executed. Cloud Foundry also solves load balancing, routing, and service discovery. In addition, Cloud Foundry includes additional infrastructure such as databases.

QUIZ

1

Suppose at a company, the decision on whether to run your platform on a public cloud or to run it in your own data center lies with you. What is a factor that you will consider when making your decision?

COMPLETED 0%

1 of 3



In the next lesson, we'll look at some variations in the microservices platform

approach that we have looked at so far.

Variations

In this lesson, we'll discuss some variations in the microservices platform approach that we have looked at so far.

WE'LL COVER THE FOLLOWING ^

- REST
- Frontend integration
- Operation
- Physical hardware
- Virtual hardware

REST

Microservices platforms appear to be particularly suitable for synchronous microservices and REST communication, as they offer particularly good support for this.

Frontend integration

The platforms can be extended to allow other communication mechanisms and frontend integration can be implemented with these platforms.

Client frontend integration is completely independent of the platform used. Only with server-side frontend integration would the server have to be installed and operated on the platform.

Operation

The platform can also cover the operational aspect for asynchronous microservices. Better support for the operation of the microservices alone is a good reason to use the platforms. Operation is one of the most important challenges with microservices. This aspect is independent of the communication mechanism used.

Physical hardware

The question arises whether there are other environments for the operation of microservices. A theoretical alternative to the platforms is physical hardware, however, physical hardware is hardly used any more for cost reasons.

Virtual hardware

Virtual hardware is inflexible and heavyweight, so the only alternative to a platform is “docker without scheduler”. In this scenario, Docker containers are installed on classic servers.

In this case, the macro architecture standardizes the operation to use **only one technology for log analysis or monitoring** and thus to work efficiently.

Microservices platforms **already have such features**. You do not have to build support for **logging or monitoring** yourself, but you can use these parts of the platform. This can be a simpler solution because implementing log analysis for many microservices can be very costly.

Resilience and other features such as **load balancing have to be implemented** at the virtual machine level since the Docker infrastructure does not offer these.

At the end of the day, it can happen that the **features of a microservices platform** like log analysis, monitoring, reliability, and load balancing are **recreated step by step** instead of installing and using a prepackaged microservices platform.

QUIZ

1

_____ frontend integration will have to be installed and operated on the platform.

COMPLETED 0%

1 of 2



In the next lesson, we'll conclude this chapter.

Chapter Conclusion

We'll conclude this chapter with a quick summary.

WE'LL COVER THE FOLLOWING ^

- Summary

Summary

Microservices platforms appear to be very helpful due to the large number of features, even though the operation of these platforms can be complex.

In practice, **Kubernetes** is a very important platform for the operation of microservices, while **PaaS such as Cloud Foundry** are not as useful, despite their seemingly useful features.

Microservices platforms should be considered for microservices because they represent **a significant simplification and complete solution** to typical microservices challenges.

The only reason **against** a microservices platform is the **high cost of installation**. In the case of a small number of microservices or at the beginning of a project, the installation of a microservices platform can be bypassed. In addition, this effort is eliminated if a public cloud offer is used.

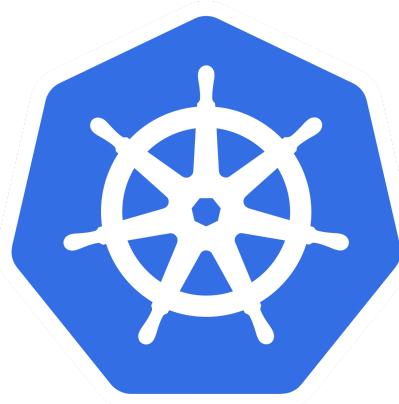
In the next chapter, we'll study Kubernetes!

Introduction

In this lesson, we'll look at a quick introduction to Kubernetes.

WE'LL COVER THE FOLLOWING ^

- Definition
- Licence and community
- Kubernetes versions
- Kubernetes concepts
 - Nodes
 - Pod
 - Replica set
 - Deployment
 - Services
 - Declarative
- Features
- Chapter walkthrough



kubernetes

Definition #

[Kubernetes](#) continuously gaining importance as a runtime environment for the development and operation of microservices.

Licence and community

Kubernetes is an open source project and under the Apache license. It is managed by the Linux foundation and was originally created at Google. An extensive ecosystem has emerged around Kubernetes, offering various extensions.

Kubernetes versions

There are different Kubernetes variants.

[Minikube](#) is a version of Kubernetes for installing a test and development system on a laptop.

There are more [versions](#) that can be installed on servers or used as cloud offerings:

- [kops](#) is a tool which enables the installation of a Kubernetes cluster in different types of environments like AWS (Amazon Web Services).
- [Amazon Elastic Container Service for Kubernetes \(Amazon EKS\)](#) provides Kubernetes clusters on AWS.
- Google Cloud also supports Kubernetes with the [Google Container Engine](#).
- Microsoft Azure provides the [Azure Container Service](#)
- IBM Bluemix provides Kubernetes with the [IBM Bluemix Container Service](#).

Kubernetes concepts

In addition to Docker concepts, Kubernetes introduces several additional concepts.

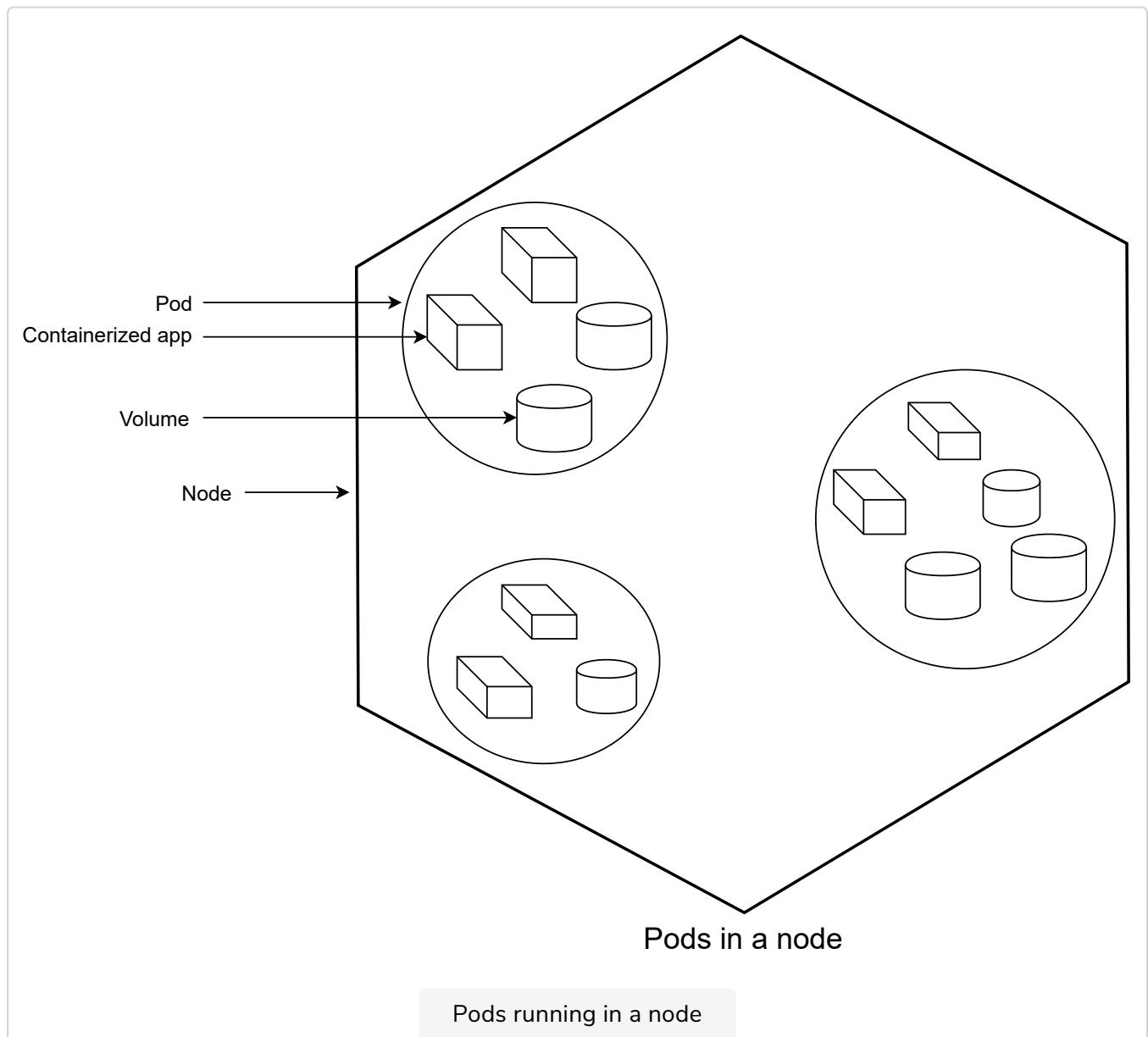
Nodes

Nodes are the servers on which Kubernetes runs. They are organized in a cluster.

Pod

A **Pod** is multiple Docker containers that together provide a service. For example, this can be a container with a microservice and a container for log processing. The Docker containers in a pod can share Docker volumes and efficiently exchange data.

Containers that belong to one pod run on one node. To scale the system to more nodes, more instances of the pod need to be started and distributed on the nodes.



Replica set

A **replica set** ensures that a certain number of instances of a pod runs. This allows the load to be distributed to the pods. In addition, the system is fail-safe. If a pod fails, a new pod is automatically started.

Deployment

A **deployment** generates a replica set and provides the required Docker images.

Services

Services make pods accessible. The services are registered under one name in the DNS and have a fixed IP address under which they can be contacted throughout the cluster. In addition, services enable routing for requests from the outside.

Declarative

Kubernetes is **declarative** meaning the configuration defines a desired state. Kubernetes makes sure that the system fits the desired state.

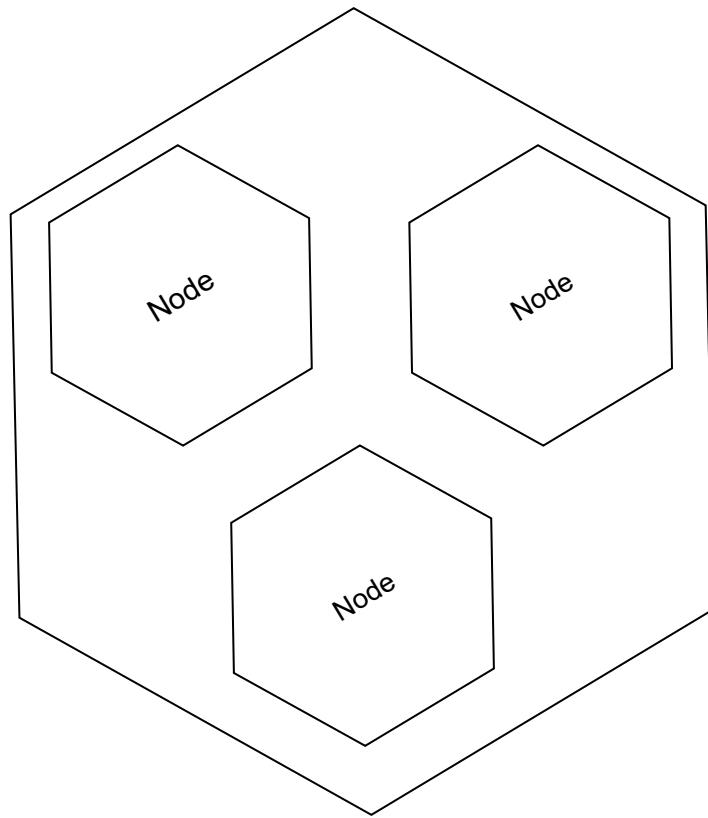
A replica set actually defines the number of pods that should be running, and it is up to Kubernetes to make the actual number of running pods match that number.

If the configuration of the replica set changes, the number of running pods is adjusted accordingly. And if some of the pods crash, enough pods are started to match the declaration in the replica set.

Features

As mentioned above, Kubernetes offers a platform based on Docker that supports important features:

- Kubernetes runs Docker containers in a **cluster** of nodes. Thus, Docker containers can use all resources the in the cluster.



Cluster

- In the event of a failure, Docker containers can be restarted. This is possible even if the original node on which the container was running is no longer available. In this way, the system achieves **fail-safety**.
- Kubernetes also supports **load balancing** and can distribute the load between multiple nodes.
- Finally, Kubernetes supports **service discovery**. Microservices that run in Docker containers can easily find each other and communicate with each other via Kubernetes.
- Since Kubernetes works on the level of Docker containers, the microservices have **no code dependencies** to Kubernetes. This is not only elegant, but means that a Kubernetes system supports virtually all programming languages and frameworks for the implementation of microservices.

This chapter describes Kubernetes, a runtime environment for Docker containers. The reader gets to know the following points in this chapter:

- Kubernetes can run Docker containers in a cluster and comprises a complete infrastructure for microservices.
- Kubernetes does not introduce code dependencies into the example.
- MOMs or other tools can be run on Kubernetes.

QUIZ

1

What is a pod in and what is its function?

COMPLETED 0%

1 of 4



In the next lesson, we'll look at a Kubernetes example.

The Example with Kubernetes

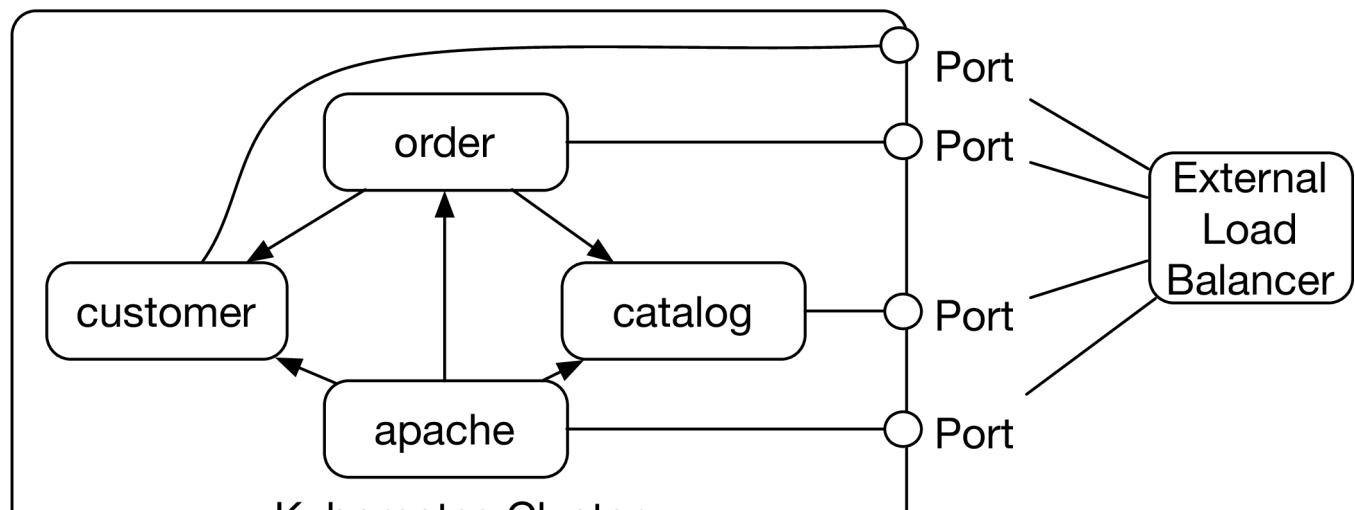
In this lesson, we'll introduce an example with Kubernetes.

WE'LL COVER THE FOLLOWING

- Introduction
- Implementing microservices with Kubernetes
- Service discovery
- Fail-safety
- Load balancing
- Service discovery, fail-safety, and load balancing without code dependencies
- Routing with Apache httpd
- Routing with node ports
- Routing with load balancers
- Routing with Ingress

Introduction

The services in this example are identical with the examples presented in the two preceding chapters (see [Example](#)).



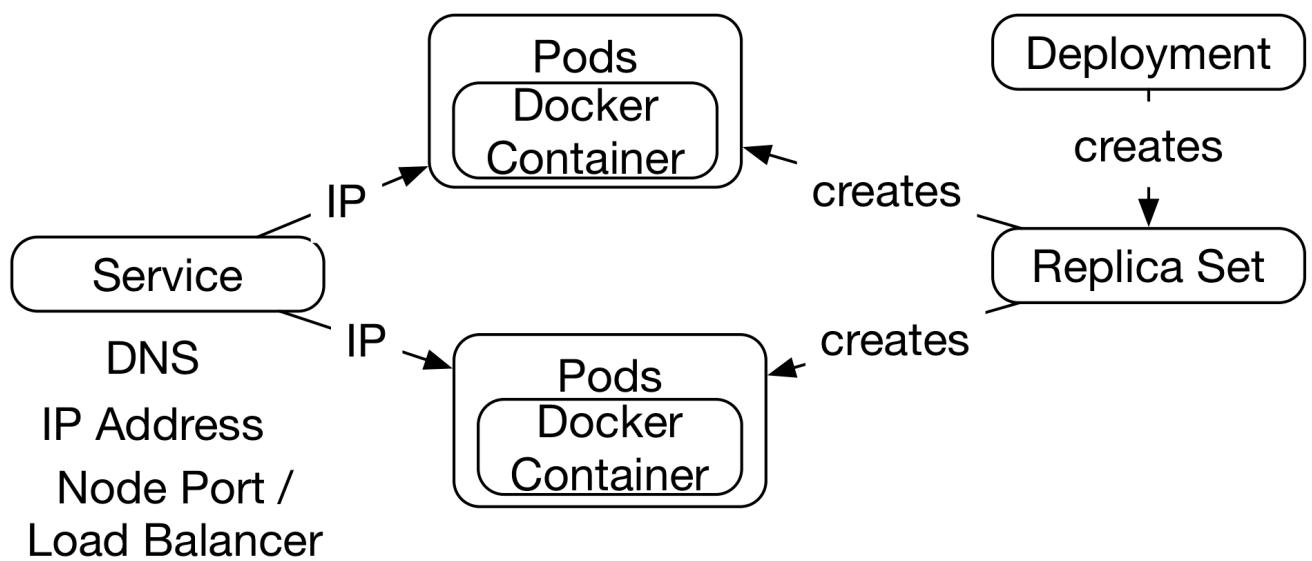
- The **catalog** microservice manages the information about the items. It provides an HTML UI and a REST interface.
- The **customer** microservice stores the customer data and also provides an HTML UI and a REST interface.
- The **order** microservice can receive new orders. It provides an HTML UI and uses the REST interfaces of the catalog and customer microservice.
- An **Apache web server** facilitates access to the individual microservices. It forwards the calls to the respective services.

The drawing above shows how the microservices interact. The order microservice communicates with the catalog and the customer microservice while the Apache httpd server communicates with all other microservices to display the HTML UIs.

The microservices are accessible from the outside via node ports. On each node in the cluster, a request to a specific port will be forwarded to the service. However, the port numbers are assigned by Kubernetes, so there is no port number in the figure.

A load balancer is set up by the Kubernetes service to distribute the load across Kubernetes nodes.

Implementing microservices with Kubernetes



The drawing above shows the interaction of the Kubernetes components for a microservice.

- A **deployment** creates a **replica set** with the help of Docker images.
- The **replica set** starts one or multiple pods.
- The **pods** in the example only comprise a single Docker container in which the microservice is running.

Service discovery

A **service** makes the replica set **accessible**:

1. The service provides the pods with an IP address and a DNS record.
2. Other pods communicate with the service by reading the IP address from the DNS record.
3. Thereby, Kubernetes implements **service discovery with DNS**.

In addition, microservices receive the IP addresses of other microservices via **environment variables**. Thus, they could also use this information to access the service.

Fail-safety

The microservices are so fail-safe because the replica set ensures that a certain number of pods is always running. **If a pod fails, a new one is started..**

Load balancing

Load balancing is also covered. The number of pods is determined by the replica set meaning that the service implements load balancing. All pods can be accessed with the same IP address which the service defines. Requests go to this IP address but **are distributed to all instances**.

The service implements this functionality by interfering with the IP network between the Docker containers. Since the IP address is cluster-wide unique, this mechanism works even if the pod is moved from one node to another.

Kubernetes **does not implement load balancing at the DNS level**. If it did, different IP addresses would be returned for the same service name for each DNS lookup so that the load would be distributed during DNS access.

However, this kind of approach presents a number of challenges. DNS supports caching and if a different IP address is to be returned each time a DNS access occurs, the caching must be configured accordingly. However, problems still occur because caches are not invalidated in time.

Service discovery, fail-safety, and load balancing without code dependencies

For load balancing and service discovery, no special code is necessary.

A URL like <http://order:8080/> suffices. Accordingly, the microservices do not use any special Kubernetes APIs. There are no code dependencies to Kubernetes and no specific Kubernetes libraries are used.

Routing with Apache httpd

In the example, Apache httpd is configured as a reverse proxy. It therefore routes access from the outside to the correct microservice. Apache httpd leaves load balancing, service discovery, and fail-safety to the Kubernetes infrastructure.

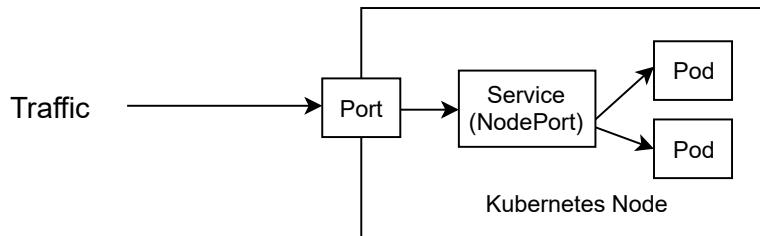
Routing with node ports

Services also offer a solution for routing, i.e., external access to microservices.

1. The service generates a node port.
2. Under this port, the services are available on every Kubernetes node.
3. If the pod that implements the service is not available on the called Kubernetes node, Kubernetes forwards a request to the node port to another Kubernetes node where the pod is running.

In this way, an external load balancer can distribute the load to the nodes in the Kubernetes cluster.

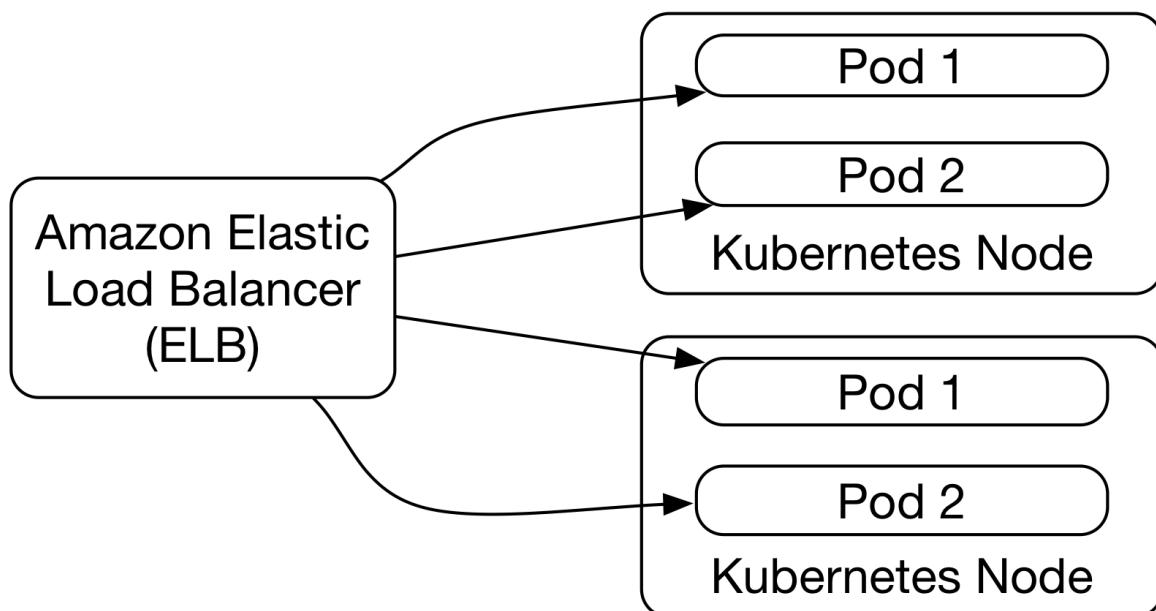
The requests are simply distributed to the node port of the service on all nodes in the cluster. The service type **NodePort** is used for this purpose, which must be specified when creating the service.



Routing with node ports

Routing with load balancers

Kubernetes can create load balancers in a Kubernetes production environment. In an Amazon environment, for example, Kubernetes configures an ELB (Elastic Load Balancer) to access the node ports in the cluster. The service type **LoadBalancer** is used for this purpose.



Kubernetes Service Type LoadBalancer in the Amazon Cloud

The services in the example are of the type **LoadBalancer**. However, if they run on Minikube, they are treated as services of type **NodePort** because Minikube cannot configure and provide load balancers.

Routing with Ingress

Kubernetes offers an extension called **Ingress**, which can **configure and alter**

the access of services from the Internet.

Ingress can also implement:

- load balancing
- terminate SSL
- implement virtual hosts

This behavior is implemented by an **Ingress controller**.

In the example, the Apache web server forwards requests to the individual microservices. This could also be done with a **Kubernetes Ingress**. Then the routing would be done by a part of the Kubernetes infrastructure which might work better.

The **configuration is done in Kubernetes YAML files** and supports other Kubernetes parts like services. However, the example contains a few static web pages that the Apache web server provides. So, the Apache web server has to be configured anyways.

Using an Ingress is therefore not a huge advantage.

Q U I Z

1

How are the microservices in the example accessible from outside?

COMPLETED 0%

1 of 2



In the next lesson, we'll discuss this same example in much more detail.

The Example in Detail

Let's look at an example of how Kubernetes can be used.

WE'LL COVER THE FOLLOWING ^

- Setup
 - Kubernetes YAML
- Some Minikube commands

Setup

The example is accessible at <https://github.com/ewolff/microservice-kubernetes>. <https://github.com/ewolff/microservice-kubernetes/blob/master/HOW-TO-RUN.md> explains in detail how to install the required software for running the example.

The following steps are necessary for running the example:

- [Minikube](#) as minimal Kubernetes installation has to be installed.
Instructions for this can be found at
<https://github.com/kubernetes/minikube#installation>.
- [kubectl](#) is a command line tool for handling Kubernetes and also has to be installed. Its installation is described at
<https://kubernetes.io/docs/tasks/tools/install-kubectl/>.
- The script `docker-build.sh` generates the Docker images for the microservices and uploads them into the public Docker hub. This step is optional since the images are already available on the Docker hub. It only has to be performed when changes were introduced to the code or to the configuration of the microservices. Before starting the script, the Java code has to be compiled with `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in directory `microservice-kubernetes-`

`demo`. Then the script `docker-build.sh` creates the images with `docker build` and with `docker tag` they receive a globally unique name and `docker push` uploads them into the Docker hub. Using the public Docker hubs spares the installation of a Docker repository and thereby facilitates the handling of the example.

- The script `kubernets-deploy.sh` deploys the images from the public Docker hub in the Kubernetes cluster and thereby generates the pods, the deployments, the replica sets, and the services. For this, the script uses the tool `kubectl`. `kubectl run` serves to start the image which is downloaded at the indicated URL in the Docker hub. In addition, it is defined which port the Docker container should provide. So, `kubectl run` generates the deployment, which creates the replica set and thereby the pods. `kubectl expose` generates the service which accesses the replica set and thus creates the IP address, node port resp. load balancer and DNS entry.

This excerpt from `kubernetes-deploy.sh` shows the use of the tools using the catalog microservice as an example.

```
#!/bin/sh
if [ -z "$DOCKER_ACCOUNT" ]; then
    DOCKER_ACCOUNT=ewolff
fi;
...
kubectl run catalog \
--image=docker.io/$DOCKER_ACCOUNT/microservice-kubernetes-demo-catalog:la
test
  \
--port=80
kubectl expose deployment/catalog --type="LoadBalancer" --port 80
...
```

Kubernetes YAML

An alternative is to use Kubernetes YAML files. They describe the desired state of deployments and services. For example, here is the part of `microservices.yaml`, for the catalog microservices.

```
apiVersion: apps/v1beta1
kind: Deployment

metadata:
  creationTimestamp: null
  labels:
    run: catalog
    name: catalog
spec:
  replicas: 1
  selector:
    matchLabels:
      run: catalog
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: catalog
    spec:
      containers:
        - image: docker.io/ewolff/microservice-kubernetes-demo-catalog:lates
t
        name: catalog
        ports:
          - containerPort: 8080
        resources: {}
status: {}
```

...

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    run: catalog
    name: catalog
spec:
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    run: catalog
```

```
type: LoadBalancer  
status:  
loadBalancer: {}  
...  
...
```

The information in the YAML file is very similar to the parameters of the commands above. Using `kubectl apply -f microservices.yaml` all the services and deployment would be created in the Kubernetes cluster. The same command would be used to update the services and deployments after any changes.

Some Minikube commands

`minikube dashboard` displays the dashboard in the web browser, which displays the deployments and additional elements of Kubernetes. This makes it easy to understand the state of the services and deployments. See the screenshot below.

The screenshot shows the Kubernetes Dashboard interface. On the left, there's a sidebar with 'Cluster' and 'Namespace' sections. Under 'Cluster', options like Namespaces, Nodes, Persistent Volumes, Roles, and Storage Classes are listed. Under 'Namespace', 'default' is selected. Below the sidebar, tabs for 'Workloads', 'Daemon Sets', 'Deployments', and 'Jobs' are visible, with 'Workloads' currently active. The main content area has two tables: 'Deployments' and 'Pods'. The 'Deployments' table lists five entries: 'hystrix-das...', 'customer', 'order', 'apache', and 'catalog'. Each entry includes a green checkmark icon, a label (e.g., 'run: hyst...', 'run: cus...'), the number of pods ('1 / 1'), their age ('11 days'), and the image source ('docker.io/...'). The 'Pods' table is partially visible at the bottom. At the bottom center of the dashboard is a button labeled 'Kubernetes Dashboard'.

Name	Labels	Pods	Age	Images
hystrix-das...	run: hyst...	1 / 1	11 days	docker.io/...
customer	run: cus...	1 / 1	11 days	docker.io/...
order	run: order	1 / 1	11 days	docker.io/...
apache	run: apa...	1 / 1	11 days	docker.io/...
catalog	run: cat...	1 / 1	11 days	docker.io/...

`minikube service apache` opens the Apache service in the web browser and thereby offers access to the microservices in the Kubernetes environment.

The script `kubernetes-remove.sh` can be used to delete the example. It uses `kubectl delete service` for deleting the services, and `kubectl delete deployments` for deleting the deployments.

In the next lesson, we'll look at some additional Kubernetes features.

Additional Kubernetes Features

In this lesson, we'll discuss some additional Kubernetes features.

WE'LL COVER THE FOLLOWING



- Monitoring with liveness and readiness probes
- Configuration
- Separating Kubernetes environments with namespaces
- Applications with state
 - Persistent volumes & stateful sets
 - Operators
- Extensions with Helm

Kubernetes is a powerful technology with many features. Here are some examples of additional Kubernetes features:

Monitoring with liveness and readiness probes

Kubernetes recognizes the failure of a pod via [Liveness Probes](#). A custom Liveness Probe can be used to determine **when a container is started anew** depending on the needs of the application.

A [Readiness Probe](#), however, **indicates whether a container can process requests or not**.

- For example, if the application is blocked by processing a large amount of data or has not yet started completely, the Readiness Probe can report this state to Kubernetes.
- In contrast to a Liveness Probe, the container is not restarted as a result of a failed Readiness Probe.
- Kubernetes assumes that after some time the pod will signal via the

Readiness Probe that it can handle requests.

Configuration

The configuration of applications is possible with [ConfigMaps](#). The configuration data is provided to the applications as values in environment variables.

Separating Kubernetes environments with namespaces

Kubernetes environments can be separated with [Namespaces](#).

Namespaces are virtual clusters so that services and deployments are completely separated.

Separation with namespaces allows **different environments to coexist**; it is possible for multiple teams to share a cluster and use namespaces to separate their environments.

Separation with namespaces also makes it possible to **separate the microservices from infrastructure** like databases or monitoring infrastructure. That way users only see the services and deployments they are interested in.

Applications with state

Persistent volumes & stateful sets

Kubernetes **can also handle applications that have state**. Applications without state can simply be restarted on another node in a cluster. This facilitates fail-safety and load balancing.

If the application has state and therefore requires certain data in a Docker volume, the required Docker volumes must be available on each node the application runs on. This makes the **handling of such applications more complex**.

Kubernetes offers [persistent volumes](#) and [stateful sets](#) for dealing with this

Operators

Another option for dealing with applications that have state are [operators](#). They allow the automated installation of applications with state.

For example, there is the [Prometheus operator](#) which installs the monitoring system Prometheus in a Kubernetes cluster. It introduces Kubernetes resources for Prometheus components such as [Prometheus](#), [ServiceMonitor](#), and [Altermanager](#).

With the Prometheus operator, these keywords are used by the Kubernetes configuration instead of pods, services, or deployments. The operator also determines how Prometheus saves the monitoring data, and thus solves a key challenge.

Extensions with Helm

Kubernetes offers a complex ecosystem with numerous extensions. [Helm](#) offers the possibility to install extensions as [Charts](#) and thereby assumes the functionality of a package manager for Kubernetes. This extensibility is an important advantage of Kubernetes.

Using a Helm chart, just the name of the microservice has to be defined. The Kubernetes configuration is generated with a template and the provided name. This makes the installation of the microservices much easier and more uniform.

QUIZ

1

What is the difference between a readiness probe and a liveness probe?

COMPLETED 0%

1 of 3



In the next lesson, we'll look at some variations that can be conducted.

Variations

In this lesson, we'll look at some variations in Kubernetes.

WE'LL COVER THE FOLLOWING ^

- MOMs in Kubernetes
- Frontend integration with Kubernetes
- Docker Swarm and Docker Compose
- Docker vs. virtualization

Kubernetes offers a runtime environment for Docker containers and is very flexible.

MOMs in Kubernetes

The example in this chapter uses communication with **REST**. Of course, it is also possible to operate a MOM like Kafka ([chapter 7](#)) in Kubernetes.

However, MOMs store transmitted messages to guarantee delivery. Kafka even saves the complete history. Reliable storage of data in a Kubernetes cluster is feasible, but not easy. Using a MOM other than Kafka does not solve the problem.

All MOMs store messages permanently to guarantee delivery. **For reliable communication with a MOM, Kubernetes has to store the data reliably and scalably.**

Frontend integration with Kubernetes

Kubernetes can be quite easily combined with frontend integration ([chapter 3](#)), since Kubernetes does not make any assumptions about the UI of the applications.

Client-side frontend integration does not place any demands on the backend.

Client-side frontend integration does not place any demands on the backend. For server-side integration, a cache or web server must be hosted in a Docker container.

However, these servers do not store any data permanently, so they can easily be operated in Kubernetes.

Docker Swarm and Docker Compose

Kubernetes offers a very powerful solution and is further developed by many companies in the container area.

However, Kubernetes is **also very complex** due to its many features.

A cluster with Docker Compose and Docker Swarm can be a **simpler** but **less powerful** alternative. However, Docker Swarm and Compose also offer basic features like **service discovery** and **load balancing**.

Docker vs. virtualization

As Kubernetes takes over cluster management, it includes features that virtualization solutions also offer.

This can also lead to **operational concerns**, as reliable cluster operation is a challenge. Another technology in this area is often viewed critically. When deciding against Kubernetes, Docker can still be used without a scheduler.

But then the Kubernetes features for service discovery, load balancing, and routing are missing. They probably have to be implemented by different means.

QUIZ

1

What is one way that Kubernetes can store messages for reliable communication with MOMs?

COMPLETED 0%

1 of 2



Coming up next, we'll look at some experiments that can be tried with Kubernetes.

Experiments

In this lesson, we'll look at some fun experiments you can try.

WE'LL COVER THE FOLLOWING



- Additional microservice
- Interactive tutorial
- Run a rolling update
- Try a hosted solution
- Test load balancing
- Deploy Docker registry examples
- Kubernetes workshop
- Port asynchronous examples to Kubernetes
- Familiarize yourself with the log
- Checkout logs of some pods

Additional microservice

Supplement the Kubernetes system with an additional microservice.

- A microservice that is used by a call center agent to create notes for a call can be used as an example. The call center agent should be able to select the customer.
- For calling the customer microservice the hostname `customer` has to be used.
- Of course, you can copy and modify one of the existing microservices.
- Package the microservice in a Docker image and upload it into the Docker repository. This can be done by adapting the script `docker-build.sh`.
- Adapt `kubernetes-deploy.sh` in such a manner that the microservice is deployed.

- Adapt `kubernetes-remove.sh` in such a manner that the microservice is deleted.

Interactive tutorial

<https://kubernetes.io/docs/getting-started-guides/> is an interactive tutorial that shows how to use Kubernetes. It complements this chapter well. Work through the tutorial to get an impression of the Kubernetes features.

Run a rolling update

Kubernetes supports rolling updates. A new version of a pod is rolled out in such a way that there are **no interruptions to the service**. See <https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/>.

Run a rolling update! To do this you need to create a new Docker image. The scripts for compiling and delivering to the Docker hub are included in the example.

Try a hosted solution

Cloud providers such as Google or Microsoft offer Kubernetes infrastructures, see <https://kubernetes.io/docs/getting-started-guides/#hosted-solutions>.

Make the example work in such an environment! The scripts can be used without changes because `kubectl` also supports these technologies.

Test load balancing

Test the load balancing in the example.

- `kubectl scale` alters the number of pods in a replica set. `kubectl scale -h` indicates which options there are. For example, scale the replica set `catalog`.
- `kubectl get deployments` shows how many pods are running in the respective deployment.
- Use the service. For example, `minikube service apache` opens the web page with links to all microservices. Select the order microservice and get the orders displayed.
- `kubectl describe pods -l run=catalog` displays the running pods. There

you can also find the IP address of the pods in a line which starts with `IP`.

- Log into the Kubernetes node with `minikube ssh`. To read out the metrics, you can use a command like `curl 172.17.0.8:8080/metrics`. You have to adapt the IP address. This way you can display the metrics of the catalog pods which Spring Boot creates. For example, the metrics contain the number of requests that have been responded with an HTTP 200 status code (OK). If you use the catalog microservice via the web page, each pod should process some of the requests and the metrics of all pods should go up.
- Use `minikube dashboard` for observing the information in the dashboard.

Deploy Docker registry examples

The example currently uses the public Docker hub. Install your own [Docker registry](#). Save the Docker images of the example in the registry and deploy the example from this registry.

Kubernetes workshop

At <https://github.com/GoogleCloudPlatform/kubernetes-workshops> you can find material for a Kubernetes workshop to further familiarize yourself with this system.

Port asynchronous examples to Kubernetes

Port the Kafka example (see [chapter 7](#)) or the Atom example (see [chapter 8](#)) to Kubernetes.

This shows how asynchronous microservices can also run in a Kubernetes cluster. Kafka stores data, which can be difficult in a Kubernetes system. Explore how to run a Kafka cluster in a Kubernetes system in production.

Familiarize yourself with the log

Use `kubectl logs -help` for familiarizing yourself with the log administration in Kubernetes.

Take a look at the logs of at least two microservices.

Checkout logs of some pods

Check out logs of some pods

Use [kail](#) for displaying the logs of some pods.

QUIZ

1

What does the command `kubectl scale --replicas=5 catalog` do?

COMPLETED 0%

1 of 2



We'll conclude this chapter with a quick summary in the next lesson.

Chapter Conclusion

In this lesson, we'll conclude this chapter with a quick summary of what we have learned.

WE'LL COVER THE FOLLOWING ^

- Summary
- Advantages
- Challenges

Summary

Kubernetes solves the challenges of synchronous microservices as follows:

- DNS offers **service discovery**. Thanks to DNS, microservices can be used transparently in any programming language. However, DNS only provides the IP address, so the port must be known. No code is required to register the services. When you start the service, a DNS record is created automatically.
- **Load balancing** is ensured by Kubernetes by distributing the traffic for the IP address of the Kubernetes service to the individual pods on the IP level. This is transparent for callers and for the called microservice.
- **Routing** is covered by Kubernetes via the load balancer or node ports of the services. This is also transparent for the microservices.
- **Resilience** is offered by Kubernetes via the restarting of containers and load balancing.
 - In addition, a library like Hystrix can be useful for implementing timeouts or circuit breakers.
 - A proxy like [Envoy](#) can be an alternative to Hystrix. Envoy is also part of Istio and implements resilience for Istio.

Within one package, Kubernetes offers complete support for microservices in the cluster including:

- service discovery
- load balancing
- resilience
- scalability

In this way, Kubernetes **solves many challenges** that arise during the operation of a microservices environment. The code of the microservices remains free of these concerns. **No dependencies** on Kubernetes are introduced into the code.

This is attractive, but also represents a **fundamental change**. While Consul or the Netflix stack run on virtual machines or even bare metal, **Kubernetes requires everything to be packed in Docker containers**. This can be a fundamental change compared to an existing mode of operation and can make the migration to this environment harder.

Advantages

- Kubernetes solves most typical challenges of microservices (load balancing, routing, service discovery).
- The code has no dependencies on Kubernetes.
- Kubernetes covers operation and deployment.
- The Kubernetes platform enforces standards and is thereby the definition of a macro architecture.

Challenges

- A complete change of operation is required to use Kubernetes instead of other log or deployment technologies.
- Kubernetes is very powerful, but also very complex.

That's it for this chapter! We'll look at a new recipe with the next chapter.

Introduction

In this lesson, we'll look at a quick chapter walkthrough.

WE'LL COVER THE FOLLOWING ^

- Chapter walkthrough

There is a plethora of services available on the cloud and PaaS is one of them.

Chapter walkthrough

This chapter introduces PaaS (Platform as a Service) as a runtime environment for microservices.

The text answers the following questions:

- What is a PaaS and how does it differ from other runtime environments?
- Why is a PaaS suited for microservices?

As a concrete example for a PaaS, the chapter introduces the use of Cloud Foundry.

In the next lesson, we'll look at a definition of PaaS.

PaaS: Definition

In this lesson, we'll look at a definition of PaaS.

WE'LL COVER THE FOLLOWING



- IaaS
- SaaS
- PaaS
- PaaS restricts flexibility and control
- Routing and scaling
- Additional services
- Public cloud
- PaaS in your own data center
- Macro architecture

In the cloud, there are three fundamentally different services offered.

IaaS

An **IaaS (Infrastructure as a Service)** offers virtual computers on which software has to be installed.

IaaS is a simple solution that corresponds to **classical virtualization**.

The decisive difference is the billing model, which for **IaaS bills only the actually used resource per hour or per minute**.

SaaS

SaaS (Software as a Service) denotes a cloud offer where **software can be rented** for word processing or financial accounting.

For software development, version controls or continuous integration servers

can be purchased as SaaS.

PaaS

PaaS stands for **Platform as a Service**. PaaS offers a platform on which custom software can be installed and run. The developer only provides the application to the PaaS, the PaaS makes the application executable.

Unlike Docker and Kubernetes (see [chapter 13](#)), **the operating system and the software installed on it is not under the developer's control**.

For the microservices examples, the `Dockerfile` specifies that an Alpine Linux distribution and a specific version of the Java Virtual Machine (JVM) should be used. This is no longer necessary with a PaaS. The JAR file contains the executable Java application and everything the PaaS needs.

To start the application in the runtime environment, the PaaS can create a Docker container, but the decision which JVM and Linux distribution to use is up to the PaaS.

The PaaS must be prepared to run different types of applications. .NET applications and Java applications require their own virtual machine, while Go applications do not.

The **appropriate environment must be created by the PaaS**. Nowadays PaaS are flexible enough to support different environments, even your own environment. Therefore, you can usually define which JDK should be used.

	Your own data center	IaaS	PaaS	SaaS
Data	✓	✓	✓	✓
Application	✓	✓	✓	
Databases	✓	✓		
Operating				

System	✓	✓		
Virtualization	✓			
Physical Servers	✓			
Network & Storage	✓			
Data Center	✓			

PaaS restricts flexibility and control

Developers have less control over the Docker images.

However, the question is whether a developer should spend time with the selection of the JVM and the Linux distribution in the first place. Often, these issues lie with operations anyway.

Some PaaS offer the possibility to configure the Linux distribution or JVM. Often, even complete runtime environments can be defined by the user. Nevertheless, **flexibility is limited**.

To run existing applications, the PaaS might not be flexible enough. For example, an application might need a specific JVM version that the PaaS does not support.

Microservices are usually newly developed, so **this isn't a big disadvantage**.

Routing and scaling

The PaaS must forward requests from the user to the application, so **routing is a feature of a PaaS**.

Similarly, PaaS can usually scale applications individually, ensuring scalability.

Additional services

Many PaaS can provide the application with additional services such as databases.

Typical features for operation, such as support for analyzing log data or monitoring, are often part of a PaaS.

Public cloud

PaaS are offered in the public cloud where developers only have to deploy their application into the PaaS and then the application runs on the Internet.

This is a simple way to provide Internet applications. In the public cloud, there are further advantages; if the application is under high load, it can scale automatically.

Scalability is virtually unlimited as the application's public cloud environment can provide lots of resources.

PaaS in your own data center

The situation is somewhat different when the applications run in your own data center.

The use of PaaS is very easy for the developers, but the PaaS must first be installed. This can be a complicated process, which outweighs the advantages somewhat.

However, the PaaS **only needs to be installed once**. After that, developers can use the PaaS to install a variety of applications and bring them into production.

Operations only need to ensure that the PaaS functions reliably.

Particularly in the case of operations departments that still have manual processes and where the provision of resources takes a long time, **PaaS can considerably accelerate the rollout of applications** without the need for major changes to the organization or processes.

Macro architecture

As already shown in [chapter 12](#), microservices platforms have an impact on the macro architecture.

While a system like Kubernetes (see [chapter 13](#)) can run any kind of Docker container, a PaaS works at the level of applications. A PaaS is, therefore, **more restrictive**. For example:

- All Java applications will be standardized to one or a few Java versions and Linux distributions.
- Programming languages that are not supported by the PaaS cannot be used to implement microservices.
- Monitoring and deployment are determined by the selection of the PaaS.

Therefore, a PaaS creates an **even higher standardization of the macro architecture** than is the case in a Kubernetes environment.

Modern **PaaS can be customized** and are quite flexible. In that case, the way the PaaS is customized, and which technology options are supported, can serve as a way to define the macro architecture.

Then the macro architecture is not defined by the PaaS supplier but by whoever customizes the PaaS.

Q U I Z

1

Suppose your company is looking for an IaaS. Which profile best matches that of an IaaS?

In the next lesson, we'll study the PaaS technology, Cloud Foundry.

Cloud Foundry

In this chapter, we'll gain a quick introduction to the PaaS technology, Cloud Foundry.

WE'LL COVER THE FOLLOWING ^

- Why Cloud Foundry?
 - Open source
 - Easy to install
 - Common
 - Can be installed on your own data center
- Flexibility

Why Cloud Foundry?

Cloud Foundry serves as PaaS technology for the example in this course. The following are the reasons for this:

Open source

Cloud Foundry is an **open source project** involving a number of companies.

Cloud Foundry is managed by a **foundation**, in which Cloud Foundry providers such as **Pivotal, SAP, IBM, and Swisscom** are organized.

This ensures broad support in addition to the already many PaaS based on the Cloud Foundry.

Easy to install

Cloud Foundry can be easily installed as a Pivotal Cloud Foundry for Local Development on a *laptop* to set up a local PaaS for developers to test microservices systems.

Common

There are many public cloud providers who have an offering based on Cloud Foundry. An overview can be found at: <https://www.cloudfoundry.org/how-to-try-cloud-foundry/>.

Can be installed on your own data center

Finally, Cloud Foundry can be installed in your own data center. [Pivotal Cloud Foundry](#) is, for example, an option for this.

Flexibility

Cloud Foundry is a **very flexible PaaS**.

- Cloud Foundry supports applications in **different programming languages**. A buildpack must be available for the chosen programming language. The buildpack creates the Docker image from the application, which is then executed by Cloud Foundry. The [list of buildpacks](#) shows which buildpacks can be downloaded from the Internet.
- In a Cloud Foundry system, **modified or self-written buildpacks** can be installed. This enables support for additional programming languages or the adaptation of existing support to fit your needs.
- The **configuration of the buildpacks** can change memory settings or make other adjustments. Thus, an existing or self-written buildpack can be adapted to the needs of the microservice.
- It is also possible to deploy **Docker containers in a Cloud Foundry** environment. However, in this case, it is important to pay attention to the special features of Docker under Cloud Foundry. Ultimately, this makes it possible to run virtually any software with Cloud Foundry.

QUIZ

1

Cloud Foundry is ____.

You can choose multiple answers

You can choose multiple answers.

COMPLETED 0%

1 of 4



In the next lesson, we'll look at a Cloud Foundry example.

The Example with Cloud Foundry

In this lesson, we'll look at an example with cloud foundry.

WE'LL COVER THE FOLLOWING

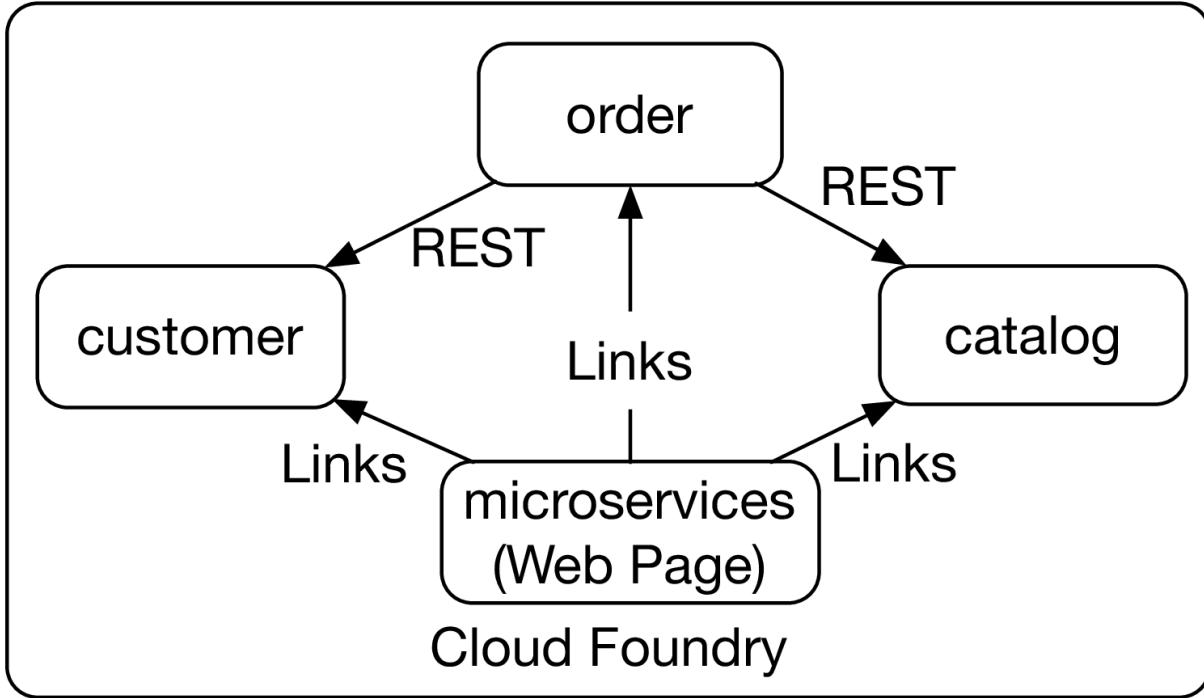


- Introduction
- Starting Cloud Foundry
- Deploying the microservices
- DNS for routing or service discovery
- Using databases and other services
- Example for a service from the marketplace
- Using services in applications
- Services for asynchronous communication

Introduction

The microservices in this example are identical to the examples from the previous chapters (see [Example](#)).

- The **catalog** microservice controls the information concerning the goods.
- The **customer** microservice stores the customer data.
- The **order** microservice can receive new orders. It uses the catalog and customer microservice via REST.
- In addition, there is the **Hystrix dashboard**, a Java application for visualizing the monitoring of the Hystrix circuit breaker.
- Finally, there is a web page **microservices** that contains links to the microservices and thus facilitates the entry into the system.



The Microservices System in Cloud Foundry

Starting Cloud Foundry

A detailed description of how the example can be built and started is provided at <https://github.com/ewolff/microservice-cloudfoundry/blob/master/HOW-TO-RUN.md>.

The Cloud Foundry example is available at

<https://github.com/ewolff/microservice-cloudfoundry>. Download the code with
`git clone https://github.com/ewolff/microservice-cloudfoundry.git`.

To start the system, the application is first compiled with Maven. To do so, you have to execute `./mvnw clean package` (macOS, Linux) or `mvnw.cmd clean package` (Windows) in the sub directory `microservice-cloudfoundry-demo`.

The example is supposed to be started on a local Cloud Foundry installation. The required installation is described at <https://pivotal.io/pcf-dev>. Upon the start of the Cloud Foundry environment, the PaaS should be assigned enough memory with `cf dev start -m 8086`. After the login with `cf login -a api.local.pcfdev.io --skip-ssl-validation` the environment should be usable.

Deploying the microservices

Now deploy the microservices. Execute `cf push` in the sub directory

microservice-cloudfoundry-demo This command has been updated to the file microservice-cloudfoundry-deploy/cf-push.sh

`manifest.yml`, with which the microservices for Cloud Foundry are configured. `cf push catalog` deploys a single application such as `catalog`.

```
---  
memory: 750M  
env:  
  JBP_CONFIG_OPEN_JDK_JRE: >  
    [memory_calculator:  
      {memory_heuristics:  
        {metaspace: 128}}]  
applications:  
  - name: catalog  
    path: .../microservice-cloudfoundry-demo-catalog-0.0.1-SNAPSHOT.jar  
  - name: customer  
    path: .../microservice-cloudfoundry-demo-customer-0.0.1-SNAPSHOT.jar  
  - name: hystrix-dashboard  
    path: .../microservice-cloudfoundry-demo-hystrix-dashboard-0.0.1-SNAPSHOT.jar  
  - name: order  
    path: .../microservice-cloudfoundry-demo-order-0.0.1-SNAPSHOT.jar  
  - name: microservices  
    memory: 128M  
    path: microservices
```

In detail, the following parts of the configuration can be distinguished.

- **Line 2** ensures that each application is provided with 750 MB RAM.
- **Lines 3 – 7** change the memory distribution so that enough memory for the Java bytecode is available in the meta space of the JVM.
- **Lines 8 – 16** configure individual microservices while simultaneously the JAR files to be deployed are specified. For each of the microservices, the common settings in lines 2-7 apply. The paths to the JARs are abbreviated to increase the clarity of the listing.
- Finally, **lines 17–19** deploy the application `microservices` that displays a static HTML page with links to the microservices. In the directory `microservices` an HTML file `index.html` is stored and an empty file `Staticfile` marks the content of the directory as static web application.

Cloud Foundry uses the Java buildpack to create Docker containers, which are then started. At <http://microservices.local.pcfdev.io/> the static web page is provided which allows the user to use the individual microservices.

As you can see, the configuration to run the microservices on Cloud Foundry

is very simple.

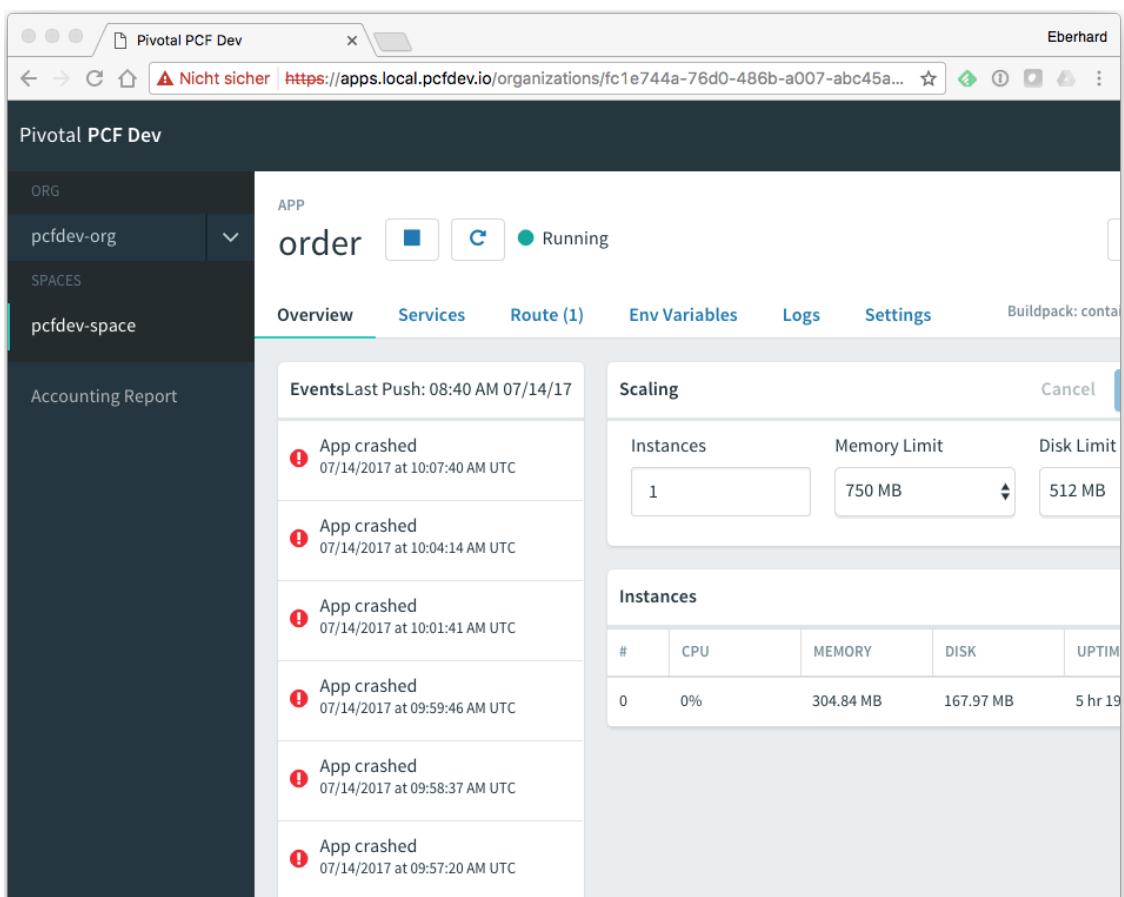
DNS for routing or service discovery

The microservices themselves have no code dependencies to Cloud Foundry. DNS is used for service discovery.

The order microservice calls the catalog and customer microservices. To do this, it uses the host names `catalog.local.pcfdev.io` and `customer.local.pcfdev.io`, which are derived from the names of the services. The `local.pcfdev.io` domain is the default but can be customized in the Cloud Foundry configuration.

`catalog.local.pcfdev.io`, `order.local.pcfdev.io`, and `customer.local.pcfdev.io` are also the hostnames used in the web browser for the links in the HTML UI of the microservices. Behind this is a routing concept in Cloud Foundry that makes the microservices accessible from outside and implements load balancing.

With `cf logs`, it is possible to have a look at the microservices logs. `cf events` returns the last entries. At <https://local.pcfdev.io/> a dashboard with basic information about the microservices and with an overview of the logs is available, see the screenshot below.



With `cf ssh catalog` the user can login to the Docker container in which the microservice `catalog` runs. This allows the user to examine the environment more closely.

Using databases and other services

It is possible to provide the microservices with additional **services** such as databases. `cf marketplace` shows the services in the marketplace. These are all services that are available in the Cloud Foundry installation.

From these services, instances can be created and made available to the applications.

Example for a service from the marketplace

`p-mysql` is the name of the MySQL service provided by the local Cloud Foundry installation that can be used to run the examples. With `cf marketplace -s p-mysql` you can obtain an overview of the different offerings for the service `p-mysql`.

`cf cs p-mysql 512mb my-mysql` generates a service instance named `my-mysql` with the configuration `512mb`. The command `cf bind-service` can make the service available to an application.

With `cf ds my-mysql` the service can be deleted again.

Using services in applications

The application must be configured with the information for accessing the service. For this, Cloud Foundry uses environment variables that contain server addresses, user accounts, and passwords.

The application must read this information. There are different possibilities for this in the respective programming languages. The [buildpack documentation](#) contains more information.

A configuration using environment variables can also be used for settings provided during the installation of the microservice. In the `manifest.yml`, variables can be set that can be read by deployed applications.

Services for asynchronous communication

Some services add asynchronous communication to Cloud Foundry. For example, the local Cloud Foundry installation offers **RabbitMQ** and **Redis** as services.

These are both technologies that can send messages asynchronously between microservices. Other Cloud Foundry offerings can provide additional MOMs as services.

In the next lesson, we'll look at some recipe variations.

Variations & Experiments

In this lesson, we'll look at some variations and experiments that can be conducted with Cloud Foundry.

WE'LL COVER THE FOLLOWING ^

- Variations
- Experiments

Variations

This chapter refers to the PaaS concept. Cloud Foundry is not the only available PaaS.

- [OpenShift](#) supplements Kubernetes with support for different programming languages to automate the generation of the Docker containers.
- [Amazon Elastic Beanstalk](#) is only available on the Amazon Cloud. It can install applications in virtual machines and scale these virtual machines. Elastic Beanstalk represents a simplification compared to the IaaS approach. Since Beanstalk is based on IaaS and some additional features, Beanstalk rests on a very stable foundation. In Beanstalk applications, additional services from the Amazon offer can be used. These include databases and MOMs. Elastic Beanstalk benefits from the numerous components available in the Amazon Cloud.
- [Heroku](#) is only available in the public cloud. Similar to Cloud Foundry it has buildpacks for supporting different programming languages and a marketplace for additional services.



Experiments

- Supplement the Kubernetes system with an additional microservice.
 - As an example, you can take a microservice used by a call center agent to make notes about a conversation. The call center agent should be able to select the customer.
 - Calling the customer microservice has to use the host name `customer.local.pcfdev.io`.
 - You can copy and modify one of the existing microservices.
 - Enter the microservice in the file `manifest.yml`.
 - The deployment of a Java application can be derived quite easily from the existing `manifest.mf`. For other languages the [documentation of the buildpacks](#) might be helpful.
- Get acquainted with the possibilities of running Cloud Foundry. Start the example and have a look at the logs with `cf logs`. Log into the Docker container of an application with `cf ssh` and see the latest events of a microservice with `cf events`.

It is also possible to use the Cloud Foundry infrastructure in other areas of the application. However, these experiments are harder to do.

- Replace the integrated database in the microservices with MySQL. For information on how to start MySQL with Cloud Foundry, see [The Example](#).

information on how to start MySQL with Cloud Foundry, see [The Example with Cloud Foundry](#). However, the code and configuration of the

microservices must be changed in order for the applications to use MySQL. There is a [Guide](#) for that.

- Cloud Foundry also provides [RabbitMQ](#) in the marketplace. This MOM can be used for asynchronous communication between microservices. A [guide](#) shows how to use RabbitMQ with Spring so you can port the example from [chapter 7](#) to RabbitMQ and then run it with a RabbitMQ instance created by Cloud Foundry.
- An alternative is [user-provided service instances](#). With this approach, a microservice can obtain information about the Kafka instance with Cloud Foundry mechanisms. The Kafka instance is not running under the control of Cloud Foundry. Change the Kafka example from [chapter 7](#) so that it uses a user-provided service Kafka instance. This concerns the configuration of the port and host for the Kafka server.
- Change one of the microservices and use [Blue/Green Deployment](#) to deploy the change so that the microservice does not fail during the deployment. The Blue/Green Deployment creates a new environment and then switches to the new version so that no downtime occurs.

We'll look at Serverless in the next lesson.

Serverless

In this lesson, we'll study serverless.

WE'LL COVER THE FOLLOWING



- Introduction
- REST with AWS Lambda and the API gateway
- Glue code

Introduction

PaaS deploy applications. **Serverless** goes even further and enables the deployment of **individual functions**.

Thereby, Serverless allows **even smaller deployments than with PaaS**. A REST service can be divided into a variety of functions, one per HTTP method and resource.

The advantages of Serverless are similar to those of PaaS:

- A high degree of abstraction and thus relatively **simple deployment**.
- Serverless functions are only activated when a request is made so that **no costs are incurred if no requests are processed**.
- They can also **scale very flexibly**.

Serverless technologies include:

- [AWS Lambda](#)
- [Google Cloud Functions](#)
- [Azure Functions](#)
- [Apache OpenWhisk](#). OpenWhisk allows you to install a Serverless environment in your own data center.

As with a PaaS, Serverless includes **support for operation**. Metrics and log management are provided by the cloud provider.

REST with AWS Lambda and the API gateway

With AWS Lambda, REST services can be implemented. The API gateway in the AWS cloud can call Lambda functions.

A separate function can be implemented for each HTTP operation. Instead of a single PaaS application, there are many Lambda functions.

A technology like [Amazon SAM](#) can make the large number of Lambda functions quite easy to use.

Even if there are a lot of functions to deploy, it is barely any more effort for the developers than if they implement the REST methods in a class. Often Lambda functions can significantly reduce the cost of running a solution.

Glue code

In another area, Lambda functions are helpful; in response to an event in the Amazon Cloud, a Lambda function can be called.

[S3 \(Simple Storage Service\)](#) offers storage for large files in the Amazon Cloud. When a new file is uploaded, a Lambda function can convert it to another format.

However, these are not real microservices, but rather glue code to complement functionalities in Amazon services.

Q U I Z

1

What is a lambda function?

COMPLETED 0%

1 of 4



We'll conclude this chapter in the next lesson.

Chapter Conclusion

We'll end this chapter with a quick summary of what we have learned.

WE'LL COVER THE FOLLOWING ^

- Summary
- Benefits
- Challenges

Summary

For synchronous microservices, Cloud Foundry's solutions are very similar to those of Kubernetes.

- **Service discovery** also works via DNS making it transparent for client and server. In addition, no code needs to be written for the registration.
- **Load balancing** is also transparently implemented by Cloud Foundry. If several instances of a microservice are deployed, the requests are automatically distributed to these instances.
- For the **routing** of external requests, Cloud Foundry relies on DNS and a distribution of the requests to the various microservice instances.
- For **resilience**, the Cloud Foundry example uses the Hystrix library. Cloud Foundry itself does not offer a solution in this area.

In addition, a PaaS provides a standardized runtime environment for microservices and can be an important antidote to the high level of operational complexity that microservices bring. Thus, a **PaaS enforces a standardization** that is often desirable in the context of macro architecture.

Compared to Docker or Kubernetes ([chapter 13](#)), a **PaaS provides less flexibility**. This can also be a strength due to the standardization that goes

hand in hand with reduced flexibility.

Modern PaaS also offers the possibility of **adapting the environment** with concepts like buildpacks or running Docker containers with arbitrary applications.

Benefits

- PaaS solves typical problems of microservices like load balancing, routing, and service discovery.
- Cloud Foundry introduces no code dependencies.
- PaaS cover operation and deployment.
- PaaS enforce standardization and are thereby the definition of a macro architecture.
- Developers only have to deliver applications. Docker is hidden.

Challenges

- Cloud Foundry requires a complete switch of the operation approach.
- Cloud Foundry is powerful, but also complex.
- Cloud Foundry provides a high degree of flexibility, however, compared to Docker containers this flexibility still has its limits.

The next chapter is an Appendix of installation guides that you might find helpful.

Installation of the Environment

This lesson gives you an introduction to what a local setup for these examples will look like!

WE'LL COVER THE FOLLOWING ^

- Introduction
- Installing necessary software

Introduction

This course focuses first on introducing technologies. An example system is presented for each technology in each chapter. In addition, the quick start allows the reader to rapidly gain practical experience with the different technologies and to understand how they work with the help of the examples.

- First, the necessary software has to be *installed* on the computer. The installation is described in this lesson.
- *Maven* handles the build of the examples. The [next lesson](#) explains how to use Maven.
- All the examples use *Docker* and *Docker Compose*. A couple of lessons in this appendix describe the most important commands for Docker and Docker Compose.

For the Maven-based build and also for Docker and Docker Compose, the chapters contain basic instructions and advice on troubleshooting.

Installing necessary software

The source code of the examples is available on Github. For access, version control *git* must be installed, see <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>. If the installation was successful, a call of `git` in the command prompt will work.

The examples are implemented in Java. Therefore, *Java* has to be installed. Instructions can be found at

https://www.java.com/en/download/help/download_options.xml. Since the examples have to be compiled, a JDK (Java Development Kit) has to be installed. The JRE (Java Runtime Environment) is not enough. When the installation is completed, it should be possible to start `java` and `javac` in the command prompt.

The examples run in Docker containers. This requires an installation of *Docker Community Edition*, see <https://www.docker.com/community-edition/>. Docker can be called with `docker`. This should work without errors after the installation.

The examples require a lot of memory in some cases. *Docker should have about 4 GB available*. Otherwise, Docker containers may be terminated due to lack of memory. Under Windows and macOS you can find the settings for this in the Docker application under Preferences/Advanced. If there is not enough memory, Docker containers are terminated. This is shown by the entry `killed` in the logs of the containers.

After installing Docker, you should be able to call `docker-compose`. If *Docker Compose* cannot be invoked, a separate installation is necessary, see <https://docs.docker.com/compose/install/>.

In the next example, we'll take a closer look at Maven!

Maven Commands

In this lesson, we'll look at Maven.

WE'LL COVER THE FOLLOWING ^

- Directories
- Maven wrapper
- Commands
- Troubleshooting

Maven is a build tool. The configuration for a project is stored in a `pom.xml` file. <http://start.spring.io/> offers a simple possibility for generating new Spring Boot projects with suitable `pom.xml` files. To do so, the user has to enter some settings on the web page. Then the web page creates the project with a `pom.xml`.

Maven can combine multiple projects to a [multi module project](#). In this case the definitions meant to apply to all modules are stored in a single `pom.xml`. All modules reference this `pom.xml`.

The `pom.xml` is stored in a directory, and the modules are saved in a subdirectory. They have their own `pom.xml` with the information specific to the respective module.

On the one hand, Maven can be started for the entire project in the directory containing the `pom.xml`. In this case Maven builds the entire project with all its modules. On the other hand, Maven can be started in the directory for a specific module. Then the Maven commands relate to this one module.

Directories

A Maven module has a fixed file structure.

- The directory `main` contains all files of the module.

- The directory `test` comprises files that are only needed for tests.

Beneath these directories there is a standardized directory structure.

- `java` contains the Java code.
- `resources` contains resources that are adopted into the application.

Maven wrapper

After the [installation](#), Maven can be used by starting `mvn`. The rest of this appendix assumes such a Maven installation.

Instead of installing Maven, the [Maven Wrapper](#) can be used. In that case, a script is created that downloads and installs Maven. Then `./mvnw` (Linux, macOS) or `./mvnw.cmd` (Windows) must be used to execute Maven. All examples for the book include a Maven wrapper, so this approach can be used, too.

Commands

The most important commands for Maven are:

- `mvn package` downloads all dependencies from the Internet, compiles the code, executes the tests, and creates an executable JAR file. The result is provided in the sub directory `target` of the respective module. `mvn package -Dmaven.test.skip=true` does not execute the tests. `mvn package -DdownloadSources=true -DdownloadJavadocs=true` downloads the source code and the JavaDoc of the dependent libraries from the Internet. The JavaDoc contains a description of the API. Development environments can display JavaDoc and the library source code for the user.
- `mvn test` compiles and tests the code but does not create a JAR.
- `mvn install` adds a step to `mvn package` by copying the JAR files into the local repository in the `.m2` directory in the home directory of the user. This allows other projects and modules to declare the module as a dependency in `pom.xml`. However, this is not necessary for the examples so `mvn package` is enough.

- `mvn clean` deletes all results of preceding builds. Maven commands can be combined. `mvn clean package` compiles everything after the results of the old builds have been deleted.

The result of the Maven build is a JAR (Java Archive). The JAR contains all components of the application including the libraries. Java directly supports this file format. Therefore, it is possible to start a microservice with `java -jar target/microservice-order-0.0.1-SNAPSHOT.jar`.

Troubleshooting

When `mvn package` does not work:

- Try out the `mvn clean package` to delete all old build results.
- Use the `mvn clean package package -Dmaven.test.skip=true` in order to skip the tests.
- The tests might fail because there is still a server running on your machine on port 8080. Make sure this is not the case.

In the next lesson, we'll look at how to install Docker and some basic Docker commands!

Docker Installation and Docker Commands

In this lesson, we'll look at how to install Docker and some basic commands.

WE'LL COVER THE FOLLOWING ^

- Starting off
- Overview
- Docker machine drivers

Starting off

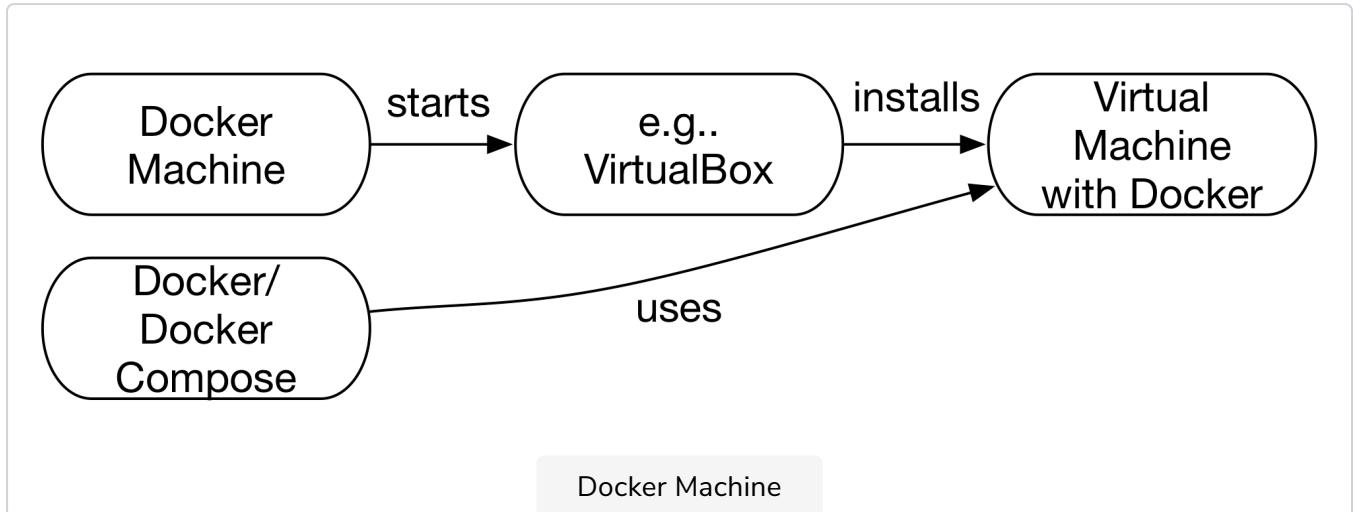
Docker Machine is a tool that can install Docker hosts. From a technical point of view, the installation is easy to do. Docker Machine loads an ISO CD image with boot2docker from the Internet.

boot2docker is a Linux distribution and provides an easy way to run Docker containers. After that, Docker Machine starts a virtual machine with this boot2docker image.

Particularly convenient in the Docker machine is the fact that **using Docker containers on external Docker hosts is just as easy as using local Docker containers**. The Docker command-line tools only need to be configured to use the external Docker host. Afterward, the use of the Docker host is transparent.

Overview

The figure below shows an overview of Docker Machine. Docker Machine installs a virtual machine on which Docker is installed. Docker and other tools, such as Docker Compose, can then use this virtual machine as if it were the local computer.



The command:

```
docker-machine create --driver virtualbox dev
```

creates a Docker host with the name `dev` with the virtualization software Virtualbox. This requires that Virtualbox be installed on the computer.

Afterwards,

```
eval "$(docker-machine env dev)"
```

on **Linux/macOS** configures Docker in such a way that the `docker` command line tools use the Docker host in the Virtualbox machine. If necessary, the shell used must be specified.

```
eval "$(docker-machine env --shell bash dev)"
```

For **Powershell on Windows**, the command is:

```
docker-machine.exe env --shell powershell dev
```

and for **cmd.exe on Windows**, it is:

```
docker-machine.exe env --shell cmd dev
```

`docker-machine rm dev` deletes the Docker host again.

Docker machine drivers

Virtualbox is only one option. There are many other [Docker Machine drivers](#) for cloud providers such as Amazon Web Services (AWS), Microsoft Azure, or Digital Ocean.

In addition, there are drivers for virtualization technologies such as VMware, vSphere, or Microsoft Hyper-V. Using any of these, Docker Machine can easily install Docker hosts on many different environments.

In the next lesson, we'll look at Docker-compose!

Docker and Docker Compose Commands

In this lesson, we'll take a look at Docker Compose!

WE'LL COVER THE FOLLOWING ^

- Docker Compose
- Docker
- State of a container
- Lifecycle of a container
- Docker images
- Cleaning up
- Troubleshooting

Docker Compose is used for the coordination of multiple Docker containers. Microservices systems usually consist of many Docker containers. Therefore, it makes sense to start and stop the containers with Docker Compose.

Docker Compose

Docker Compose uses the file `docker-compose.yml` to store information about the containers. The [Docker documentation](#) explains the structure of this file.

Upon starting, `docker-compose` outputs all possible commands. The most important commands for Docker Compose are:

- `docker-compose build` generates the Docker images for the containers with the help of the `Dockerfiles` referenced in `docker-compose.yml`.
- `docker-compose pull` downloads the Docker images referenced in `docker-compose.yml` from Docker hub.
- `docker-compose up -d` starts the Docker containers in the background. Without `-d` the containers will start in the foreground so that the output

of all Docker containers happens on the console. It is not particularly

clear which output originates from which Docker container. The option `-scale` can start multiple instances of a service, e.g., `docker-compose up -d --scale order=2` starts two instances of the order service. The default value is one instance.

- `docker-compose down` stops and deletes the containers. In addition, the network and the Docker file systems, are deleted.
- `docker-compose stop` stops the containers. Network, file systems and containers are not deleted.

Docker

At startup `docker`, outputs all valid commands without parameters.

Tip: Tab-pressing completes names and IDs of containers and images.

Here is an overview of the most important commands. The container `ms_catalog_1` is used as an example.

State of a container

- `docker ps` displays all running Docker containers. `docker ps -a` also shows stopped Docker containers. The containers, like the images, have a hexadecimal ID and a name. `docker ps` outputs this information. For other commands, containers can be identified by name or hexadecimal ID. For the example in this book, the containers have names like `ms_catalog_1`. This name consists of a prefix `ms` for the project, the name of the service `catalog`, and the sequence number `1`. The name of the container is often confused with the name of the image (e.g. `ms_catalog`).
- `docker logs ms_catalog_1` shows the previous output of the container `ms_catalog_1`. `docker logs -f ms_catalog_1` also displays all other outputs that the container still outputs.

Lifecycle of a container

- `docker run ms_catalog --name="container_name"` starts a new container with the image `ms_catalog` that gets the name `container_name`. The

parameter `--name` is optional. The container then executes the command that is stored in the `CMD` entry of the `Dockerfile`. You can execute a command in a container with `docker run <image> <command>`. `docker run ewolff/docker-java /bin/ls` executes the command `/bin/ls` in a container with the Docker image `ewolff/docker-java`. The command displays the files in the root directory of the container. If the image does not yet exist locally, it is automatically downloaded from the Docker hub on the Internet. When the command has been executed, the container shuts itself down.

- `docker exec ms_catalog_1 /bin/ls` executes `/bin/ls` in the running container `ms_catalog_1`. Thus, with these commands you can start tools in an already running container. `docker exec -it ms_catalog_1 /bin/sh` starts a shell and redirects input and output to the current terminal. This way, you have a shell in the Docker container and can interactively work with the container.
- `docker stop ms_catalog_1` stops the container. It first sends a SIGTERM so that the container can shut down cleanly, and then a SIGKILL.
- `docker kill ms_catalog_1` terminates execution of the container with a SIGKILL. But the container is still there.
- `docker rm ms_catalog_1` permanently deletes the container.
- `docker start ms_catalog_1` starts the container that was stopped. As the data is not deleted when the container was stopped, all data is still available.
- `docker restart ms_catalog_1` restarts the container.

Docker images

- `docker images` displays all Docker images. The images have a hexadecimal ID and a name. For other commands, images can be identified by both mechanisms.
- `docker build -t=<name> build <path>` creates an image with the name `name`. The `Dockerfile` has to be stored in the directory `path`. When no version is indicated, the image gets the version `latest`. As an alternative, the version can also be indicated in the format `-t=<name:version>`. The

[Dockerfiles lesson](#) describes the format of the [Dockerfiles](#).

- `docker history <image>` shows the layers of an image. For each layer, the ID, the executed command, and the size of the layer are displayed. The image to be displayed can be identified by its name if there is only one version of the image with that name. Otherwise, the name and version must be specified via `name:version`. Of course, it is also possible to use the hexadecimal ID of the image.
- `docker rmi <image>` deletes an image. As long as a container is still using the image, it cannot be deleted.
- `docker push` and `docker pull` store Docker images in a registry or load them from a registry. If no other registry is configured, the public Docker hub is used.

Cleaning up

There are several commands to clean up the Docker environment.

- `docker container prune` deletes all stopped containers.
- `docker image prune` deletes all images that do not have a name.
- `docker network prune` deletes all unused Docker networks.
- `docker volume prune` deletes all Docker volumes that are not used by a Docker container.
- `docker system prune -a` deletes all stopped containers, all unused networks, and all images that are not used by at least one container. All that remains is what the currently running containers need.

Troubleshooting

If an example does not work:

- Are all containers running? `docker ps` displays the running containers, `docker ps -a` shows the terminated ones.
- Logs can be displayed with `docker logs`. This works for terminated containers too. The term `Killed` in the logs denotes that too little memory

is available. Under Windows and macOS you can find the settings for this

in the Docker application under Preferences/ Advanced. Docker should have about 4 GB assigned.

- In the case of more complex problems, you can start a shell in the container with `docker exec -it ms_catalog_1 /bin/sh` and examine the container more closely.
-

Conclusion

Thank you for taking this course! That's a wrap.

Thanks for taking this course. We hope you learned a lot from it. Feel free to drop us a message on our discussion forum or a feedback email!

Stay tuned for more courses in this microservices series!

If you haven't already checked it out, do take a look at [An Introduction to Microservice Principles and Concepts](#).