Microservices Recipes

Eberhard Wolff

Microservices Recipes

Eberhard Wolff

Also By Eberhard Wolff

Microservices Primer

Microservices - Ein Überblick

Microservices Rezepte

Microservices - A Practical Guide

Domain-Driven Design Referenz

Contents

Introduction **1**
Acknowledgement 1

Basics: Microservices **3**
Independent Systems Architecture (ISA) Principles . . . 3
Terms 3
Principles 3
Reasons 5
Self-contained Systems 6
Conclusion & Outlook 7

Concept: Frontend Integration **9**
Why Frontend Integration? 9
Recipe: ESI (Edge Side Includes) 10
Alternative Recipes: Links and JavaScript 14
Conclusion 16
Experiments 16

Concept: Asynchronous Microservices **17**
Definition 17
Why Asynchronous Microservices? 18
Recipe: Messaging with Kafka 18
Alternative Recipe: REST with Atom 21
Conclusion 23
Experiments 23

Concept: Synchronous Microservices **25**
Definition 25
Why Synchronous Microservices? 25
Challenges 26
Recipe: Kubernetes 27
Alternative Recipes: Netflix, Consul, Cloud Foundry . . 31
Conclusion 34
Experiments 34

What next? **35**

Introduction

This brochure introduces the terms *microservice* and *self-contained system*. It continues with an overview of different concepts and recipes for the implementation of microservices. The recipe metaphor expresses that the text describes each approach *in practical terms*. For each recipe one *example implementation* is provided. Readers must combine *several* recipes for their projects, as they would need to do for the menu of a multi-course meal. And finally, there are *variations and alternatives* for every recipe. *Experiments* invite you to get hands-on experience with the examples.

The code for the examples can be found on Github. There is also an overview[1], which briefly explains all the demos in this brochure and a few additional ones.

A detailed presentation of the recipes and other concepts around microservices can be found in the book Microservices - A Practical Guide[2].

Acknowledgement

I would like to thank everyone with whom I discussed microservices, who asked me questions or worked with me. There are far too many to name them all. The discussions help a lot and are a lot of fun!

Many of the ideas and also the implementations are unthinkable without my colleagues at INNOQ.

[1] http://ewolff.com/microservices-demos.html
[2] http://practical-microservices.com/

Special thanks go to Jörn Hameister for the extensive feedback on the German edition of this brochure!

Basics: Microservices

Unfortunately, there is no generally agreed on definition of the term "microservice". So this brochure will first explain the basic ideas behind microservices.

Independent Systems Architecture (ISA) Principles

ISA[3] (Independent Systems Architecture) is a collection of basic principles for microservices. It is based on experiences collected with microservices in many different projects.

Terms

In the principles the term *must* is used for principles that have to be strictly adhered to. *Should* describes principles that have many advantages, but do not necessarily have to be complied with.

The ISA principles speak of a *system*. The IT landscape of a company consists of many systems. Every system can work with a different architecture and can therefore be implemented based on different principles.

Principles

1. The system must be divided into *modules* which offer *inter-*

[3] http://isa-principles.org

faces. Access to other modules is only possible through these interfaces. Modules must therefore not be directly dependent on the deployment details of another module, for example, the data models in the database.

2. Modules must be *separate processes, containers or virtual machines* to maximize independence.
3. The system must have two clearly separated levels of architectural decisions:
   - The *macro architecture* includes decisions that affect all modules.
   - The *micro architecture* comprises those decisions that can be made differently for each module.

   All other principles from point 4 onwards are part of the macro architecture.
4. The choice of *integration options* must be limited and standardized for the system. The integration can use synchronous or asynchronous communication and / or it can take place at the frontend level.
5. *Communication* must be limited to a set protocols like REST or messaging implementations like Kafka. Also metadata, e.g. for authentication, must be standardized.
6. Each module must have its *own independent continuous delivery pipeline*. Tests are part of the continuous delivery pipeline. Therefore, the tests of the modules must be independent.
7. *Operations* should be standardized. This includes configuration, deployment, log analysis, tracing, monitoring and alarms. There may be exceptions to the standard when a module has very specific requirements.
8. *Standards* for operation, integration or communication should be defined on the interface level. The protocol can be standardized as REST, and data structures for the communication can be standardized. But each module should be free to use a different REST library.

9. Modules must be *resilient*. They must not fail if other modules are not available or communication problems occur. It must be possible to stop a module and start it in a different environment (servers, networks, configuration, etc.) without losing data or other state.

Reasons

ISA shows that microservices are a way of modularization (principle 1). So ideas like information hiding[4] or high cohesion[5] / low coupling are also applicable to microservices. The difference to traditional modules is the implementation as a separate container (principle 2). This allows more freedom in the technical implementation of the modules. However, the modules are still part of a system, so in the end the freedom must be limited accordingly (principle 3). Consequently, integration (principle 4) and communication (principle 5) have to be standardized because otherwise there will be no system but rather separate isolated islands.

Independent deployment is an advantage of microservices and can only be ensured if each microservice has its own continuous delivery pipeline (principle 6).

Standardization (principle 7) facilitates operations, especially in the face of a large number of microservices. But the standards must not restrict the freedom of technology and therefore should only define the interface (principle 8). A microservices system will maybe not use many different technologies from the beginning, but you should keep the opportunity open to be well prepared for further development. In the long run, new technologies will appear on the market, and it is advantageous if the system can use them.

A microservices system is a distributed system. That increases the likelihood of failure of servers, the network or a microservice.

[4] https://en.wikipedia.org/wiki/Information_hiding
[5] https://en.wikipedia.org/wiki/Cohesion_%28computer_science%29

Therefore, resilience (principle 9) is essential. Also, resilience allows microservices to be moved to a different environment. That is an advantage if the system runs in a cluster.

Thus, the ISA principles provide a good basis for a microservices system.

Self-contained Systems

The ISA principles define important principles for microservices, but they do not cover all decisions and features. For example, organization, domain architecture and the question whether microservices should contain a UI or not are not discussed.

Self-contained systems (SCSs) are an approach to microservices that has already proven itself in many projects. All essential information on SCSs can be found on the website http://scs-architecture.org/. Here is an overview of the most important characteristics:

- Each SCS is an *autonomous web application*. The code for the web interface is included in the SCS. Thus, a user can use one SCS without relying on other SCSs.
- SCSs are not allowed to *share a UI*. After all, an SCS should be used via its own UI.
- An SCS can have an *optional API*. But the API is not strictly necessary, since the SCS already has a web interface for the user. For mobile clients or other SCSs, access via an API maybe useful.
- The SCS contains *data and logic*. A new feature typically requires changes to UI, logic, and data. All these changes can be done in a single SCS.
- For one SCS exactly *one team* is responsible. However, a team can be in charge of more than one SCS.

- To avoid tight coupling, SCSs should *not share business code*. Only the sharing of common technical code is allowed. As a rough rule: Only code that would be published as open source might be shared between SCS.
- To further decouple the SCS, the SCS *should not share infrastructure*, for example, no shared database should be used. For cost reasons, compromises can be made.
- The *communication* between SCSs is *prioritized*:
  - Frontend integration has the highest priority.
  - This is followed by asynchronous communication
  - and finally, synchronous communication is also possible.

    The focus is on decoupling and resilience. The higher-priority types of communication help to achieve these goals.

As mentioned, the SCS idea has already proven itself in many projects. The links on the website[6] give an impression of some of these projects. The approach is only usable for web applications, since every SCS has a web interface. However, the separation into systems that implement part of the business logic and are developed by one team, is also sensible for other types of systems.

Conclusion & Outlook

ISA defines the principles all microservices systems should comply with while SCS defines best practices that have been used successfully in many projects.

The following chapters describe technical recipes for the communication between microservices in the order of priorities suggested by SCS: frontend integration, asynchronous communication and finally, synchronous communication.

[6] http://scs-architecture.org/links.html

Concept: Frontend Integration

Microservices can include a web frontend. Self-contained systems (SCSs) even must have a web frontend. Therefore, microservices can be integrated at the frontend.

Why Frontend Integration?

Frontend integration creates *loose coupling*. If links are used for the integration, only the URL has to be known by the other system. What is behind the URL and how the information is presented can be changed without affecting the system that displays the URL in a link.

Another benefit of frontend integration is the *free choice of frontend technologies*. Especially with frontend technologies, there are lots of innovations. Constantly, there are new JavaScript frameworks and new ways to design attractive user interfaces (UIs). An important advantage of microservices is the freedom of technology. Every microservice can choose its own technologies. If technology freedom should also apply to the frontend, then every microservice must contain its own frontend, potentially using a different frontend technology. For this, the frontends of the microservices must be integrated so that the frontends appear to be part of a single system.

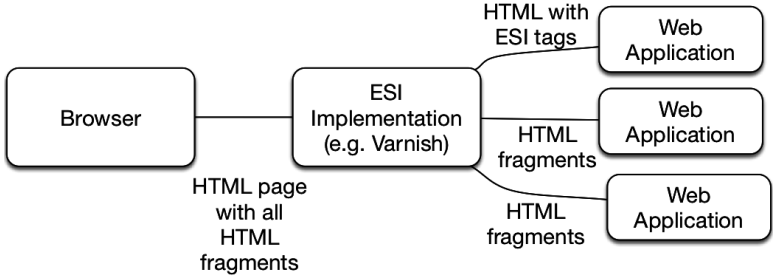
Thanks to frontend integration, *the entire functionality* for a domain is implemented *in a single microservice*. For example, a microservice might be responsible for displaying received messages even if the display of the messages is integrated in the UI of another microservice. If more information, such as a priority of the message,

should be displayed, the changes to the logic, the data management and the presentation can be implemented by changing only one microservice even if another microservice displays that information in its UI.

Recipe: ESI (Edge Side Includes)

ESI[7] (Edge Side Includes) integrates HTML fragments of a microservice with fragments of other microservices. For this, the microservice generates HTML, which contains ESI tags. The ESI implementation evaluates the ESI tags and includes HTML fragments of others microservices.

ESI is mainly implemented by caches. By splitting the HTML pages, static fragments can be cached even if they are integrated into a dynamic website. CDNs (Content Delivery Networks) can also implement ESI. CDNs are actually used to deliver static HTML pages and images. For this, CDNs operate server nodes on the internet, allowing pages and images to be loaded from a server nearby for every user and reducing the load times. With ESI, the CDNs can at least cache fragments of dynamic pages.



ESI: Composing HTML fragments to HTML pages

[7] https://www.w3.org/TR/esi-lang

Thus, ESI composes an HTML page from several HTML fragments, which can be supplied by various microservices.

An example of an ESI integration is available at https://github.com/ewolff/SCS-ESI. There is also a guide how to run the example[8].

In the example, ESI integration is used to integrate fragments of a common microservice into the web pages of other microservices. The example contains only one implemented microservice, the order microservice. The order microservice is a Spring Boot application and written in Java while the common microservice is written in Go. It shows that even very different technologies can be integrated in the frontend.

Listing 1: HTML output by the Order Microservice

```
<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>
```

The Order Microservice returns an HTML page as shown in listing 1. Such a page is available at http://localhost:8090/ when the Docker container runs on the local computer. If you look at this page in the browser, you see that the browser does not interpret the ESI tags, it rather displays a garbled web page.

[8] https://github.com/ewolff/SCS-ESI/blob/master/HOW-TO-RUN.md

The example uses the web cache Varnish[9] as an ESI implementation. The common microservice provides the content for the ESI tags. The Varnish runs at http://localhost:8080/ when the Docker container runs on the local computer. Listing 2 shows the HTML that Varnish returns.

Listing 2: Varnish's HTML output

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

<body>
  <div class="container">
    <a class="brand"
     href="https://github.com/ultraq/thymeleaf-layout-dia\
lect">
     Thymeleaf - Layout </a>
    Mon Sep 18 2017 17:52:01 </div></div>
    ...
  </div>
  <script
   src="/common/bootstrap-3.3.7/bootstrap.js" />
</body>
</html>
```

As you can see, the common microservice adds headers and footers as well as a navigation bar. The common microservice also implements a kind of asset server: It makes shared libraries like Bootstrap available.

[9] https://varnish-cache.org/

If a new version of Bootstrap should be used, only the HTML fragment in the common microservice must be changed and the new Bootstrap version must be provided by the common microservice. However, in a productive system this is hardly sufficient because the user interface of the order microservice needs to be tested with the new Bootstrap version.

Caching and Resilience

Because the system uses a Varnish cache, the HTML fragments are cached for 30 seconds. That can be seen by looking at the time in the navigation bar, which changes only every 30 seconds. If one of the microservices fails, the time for caching is even extended to 15 minutes. The configuration for Varnish can be found in the file `default.vcl` in the directory `docker/varnish/` in the example.

So the Varnish cache improves not just the performance of the system, but also the resilience because the system still works at least for 15 minutes even if the microservices fail.

With server-side integration the whole HTML page with all fragments is always sent to the client. In the example, in fact all fragments of the page need to be present: Without the frame and the Bootstrap library the page is not really usable. An optional information such as the number of items in the shopping cart does not necessarily have to be integrated with ESI.

Alternative: Server-Side Includes (SSI)

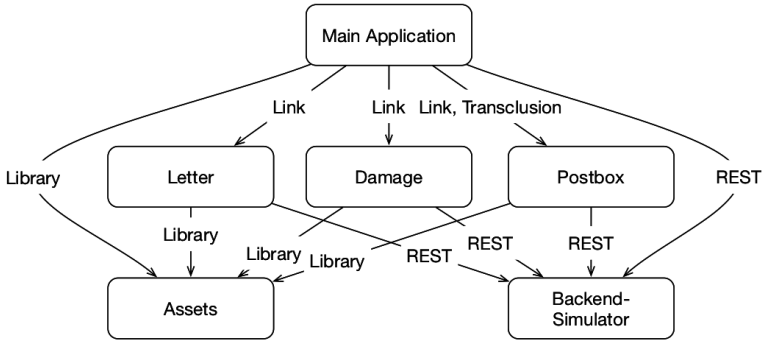
Another option for server-side frontend integration is SSI[10] (Server-side Includes). This is a feature that most web servers provide. Here, for the integration the Varnish cache is replaced by a web server. This has the advantage that a web server may already be installed for example for TLS / SSL termination. In that case, the additional effort for server-side integration is much lower

[10] https://en.wikipedia.org/wiki/Server_Side_Includes

because no additional software needs to be installed. However, the advantages of the cache concerning performance and resilience are lost. https://scs-commerce.github.io/ is an example of a system that uses SSI with nginx for frontend integration.

Alternative Recipes: Links and JavaScript

The Crimson Assurance example uses a completely different approach to frontend integration. The example was created as a prototype for a German insurance to show how a web application with frontend integration can be implemented. The two INNOQ consultants Lucas Dohmen and Marc Jansing implemented this example.



Integration with Links and JavaScript

This example is available on the internet[11]. You can also run the example on your own laptop as a set of Docker containers provided by a project on Github[12]. A guide[13] explains how to run the example.

This example implements an application for a clerk at an insurance.

[11]https://crimson-portal.herokuapp.com/

[12]https://github.com/ewolff/crimson-assurance-demo

[13]https://github.com/ewolff/crimson-assurance-demo/blob/master/HOW-TO-RUN.md

The main application `crimson-portal` has links to the applications for writing letters `crimson-letter`, for reporting damage `crimson-damage` and the REST backend simulator `crimson-backend`. These links use parameters to pass information like the contract ID to the other applications.

Only for the integration of the postbox an additional client-side integration is implemented in about 50 lines of JavaScript. So the postbox can also be included in the main page of the portal.

All of the applications have a consistent look & feel that is supported through shared assets in the project `crimson-styleguide`. The assets are integrated into the projects as a library during the build.

This example shows how far you can get with a simple integration with links. In addition, this example also illustrates the integration of very different systems: The main application, letter and damage are implemented with NodeJS, while the postbox is implemented with Java and Spring Boot.

Another example of client-side integration is https://github.com/ewolff/SCS-jQuery. It implements a very simple client-side integration with JavaScript. From a user perspective the example is identical to the ESI example.

Both projects use links. The linked pages will be transcluded into the original page through JavaScript. Even if the JavaScript can not be executed because there is an error in the code or the linked page is not available, the system still works: It just displays a link instead of the messages in the postbox.

Because client-side integration is implemented in jQuery in both examples, each system must integrate this JavaScript library in a version that works with that integration. This leads to a restriction of the technology freedom. An implementation with pure JavaScript would be better in this regard.

Conclusion

Frontend integration leads to a very loose coupling. In many cases, links are sufficient. In this case, the systems only need to know the URLs of the linked pages. If a web page is to be composed of fragments from different systems, then the necessary integration can take place on the server. If a cache is installed, ESI can be used. The cache allows HTML fragments to be cached. That benefits performance and resilience. Web servers can implement SSI. If a web server is already in use, then the additional infrastructure of a cache can be avoided. Finally, a client-side integration can load optional content, such as the overview of the postbox.

Experiments

- Start the ESI example. Please refer to https://github.com/ewolff/SCS-ESI/blob/master/HOW-TO-RUN.md.
- Look at the output of the Varnish cache at http://localhost:8080/ and compare it to the output of the order microservice at http://localhost:8090/. Take a look at the source code of the returned HTML pages with your browser. How can you access the HTML fragments of the common microservice?
- Try the user interface. Stop the microservices with `docker-compose up --scale common=0` or `docker-compose up --scale order=0`. Which microservices are still usable?

Concept: Asynchronous Microservices

Microservices can exchange messages. Asynchronous communication allows loose coupling and good resilience.

Definition

Asynchronous microservices differ from synchronous microservices. The next chapter describes synchronous microservices in detail. The term "synchronous microservices" means the following:

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

Asynchronous microservices do not wait for responses from other systems while they are processing a request themselves. There are two ways to do this:

- The microservice does not communicate with other systems while processing a request. In this case, the microservice will typically communicate with the other systems at a different time. For example, the microservice can replicate data that is used when processing a request. Thus, customer data can be replicated in order to access the locally stored customer data when processing an order.
- The microservice sends a request to another microservice, but does not wait for a response. A microservice for processing an

order can send a message to another microservice that creates the invoice. An response to this message is not necessary and therefore does not have to be waited for.

Why Asynchronous Microservices?

Asynchronous microservices have several advantages:

- If a communication partner fails, the message is still transmitted once the communication partner is available again. So asynchronous communication *provides resilience*, i.e, a protection against the failure of parts of the system.
- The transmission and processing of a message can almost always be *guaranteed*: The messages are stored. At some point they will be processed. The fact that they are processed can for example be ensured by the recipient acknowledging the message.
- Asynchronous microservices can implement *events*. Events provide better decoupling. For example, an event could be "order received". Each microservice can decide for itself how it reacts to the event. For example, one microservice can create an invoice and another can initiate delivery. If additional microservices are added e.g. for a bonus program, they only have to respond appropriately to the existing event. So the system is very easy to extend.

Recipe: Messaging with Kafka

Kafka is an example of message-oriented middleware (MOM). A MOM sends messages and ensures that the messages arrive at the recipient. MOMs are asynchronous. So you do not implement a request / reply approach as with synchronous communication protocols but only send messages.

Basic Kafka Concepts

Kafka[14] differs from other MOMs mainly in the fact that it permanently stores the messages it transmits instead of discarding them after transmission.

The main concepts of Kafka are:

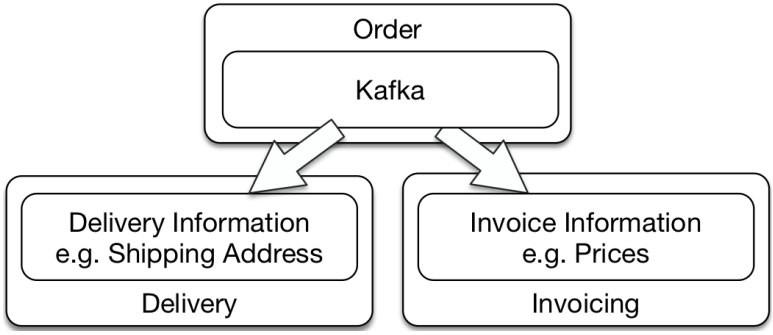
- There are three *APIs*: the *producer API* for sending data, the *consumer API* for receiving data and the *streams API* to transform the data.
- Kafka organizes data in *records*. They contain the transmitted data as *value*. Records also have a *key* and a *timestamp*.
- *Topics* contains records. Usually, records of a certain kind are sent as part of a topic.
- Topics are divided into *partitions*. When a producer creates a new record, the record is appended to a partition of the topic. The distribution of records to partitions is implemented by the producer and usually based on the key of the record.
- Kafka stores the *offset* for each consumer in each partition. This offset indicates which record the consumer last read in the partition. If a consumer has processed a record, the consumer can commit a new offset. For every consumer Kafka only needs to store the offset in each partition which is relatively lightweight.
- In a *consumer group* there is exactly one consumer for each partition. This ensures that a record is processed by only one consumer: The record is stored in a partition, which is processed by one consumer thanks to the consumer group.
- *Log compaction* is a mechanism that can be used to delete old records: Â Â If there are multiple records with the same ID, a log compaction deletes all of these records except the last one. This can be used to remove events that are superseded by newer events.

[14] https://kafka.apache.org/

Kafka can be operated in a cluster to provide reliability and scalability.

The Kafka Example

The example can be found at GitHub[15]. The guide[16] contains an extensive documentation that explains step by step how to install and start the example.



Overview of the Kafka example

If the Docker containers run on the local machine, the web application is available at http://localhost:8080/. The web interface is served by an Apache httpd web server acting as a reverse proxy. It forwards the HTTP requests to the microservices.

Split of the Example into Microservices

The system consists of a microservice `order`, which accepts orders via the web interface. The `order` microservice then sends the order data as a record via Kafka to the `shipping` microservice and the `invoicing` microservice. The order is transferred as JSON. Because of JSON's flexibility, the invoicing microservice and the shipping microservice can read just the data that is relevant for the respective microservice from the JSON data structure.

[15] https://github.com/ewolff/microservice-kafka
[16] https://github.com/ewolff/microservice-kafka/blob/master/HOW-TO-RUN.md

All the shipping microservice instances and all the invoicing microservice instances are each organized in a consumer group. This means that the records for the orders are load balanced over all consumers, but each record is sent to just one consumer. This ensures that only one invoicing microservice writes an invoice for an order and only one shipping microservice delivers the goods.

The shipping microservice and the invoicing microservice store the information from the records in their own database schemas. All microservices use the same Postgres database.

Each Kafka record contains an order. The key is the ID of the order with the suffix `created`, for example `1created`.

Avro: An Alternative Data Format

An alternative data format is Avro[17]. It provides a binary protocol but also a JSON representation. Avro can define a schema for the data. It is also possible, for example with default values, to convert data from an old version of the schema to a new version of the schema. This allows old events to be processed even if the schema has changed in the meantime.

Alternative Recipe: REST with Atom

Asynchronous microservices can also be implemented with REST. For example, it is possible to provide orders as an Atom Feed[18]. Atom is a data format originally developed to make blogs available to readers. Just as a new entry in an Atom document is created for each new blog post, the same is possible for every new order. A client must then periodically poll the Atom document and process new entries. That's not very efficient. It can be optimized by HTTP caching. Then data will only be transferred if there are really new

[17] http://avro.apache.org/
[18] https://validator.w3.org/feed/docs/atom.html

entries. Pagination can also ensure that only the most recent entries are transmitted, not all.

An example of an asynchronous integration of microservices with Atom can be found at https://github.com/ewolff/microservice-atom. https://github.com/ewolff/microservice-atom/blob/master/HOW-TO-RUN.md explains in detail the necessary steps to run the example.

Atom has the advantage of being based on REST and HTTP. As a result, no MOM must be operated. In most cases, teams already have experiences with HTTP and web servers. Thus, the operations can be ensured even with large amounts of data.

Unfortunately, this type of communication can not ensure that an order is only received and processed by a single microservice instance. However, if one of the microservices instances in the example application reads a new order from the Atom feed, it first checks whether there is already an entry for this order in the database, and it will only create an entry itself if this is not the case. Therefore, only one entry in the database is created for each order.

It is not mandatory to use the Atom format. You can also use your own format to make the changes available as a list and then provide details with links. Likewise, a different feed format such as RSS[19] or JSON Feed[20] can be used.

Other MOMs

Of course, also other MOMs than Kafka can be used. For example, there are JMS implementations[21], the Java messaging service standard (JMS[22]) or implementations[23] of the AMQP[24] (Advanced Message Queuing Protocol). But here the microservices have to deal with the fact that old events will not be available after some time.

[19] http://web.resource.org/rss/1.0/spec
[20] http://jsonfeed.org/
[21] https://en.wikipedia.org/wiki/Java_Message_Service#Provider_implementations
[22] https://jcp.org/aboutJava/communityprocess/final/jsr914/index.html
[23] https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol#Implementations
[24] https://www.amqp.org/

Conclusion

Asynchronous microservices offer advantages in resilience but also in decoupling. Kafka is an interesting alternative for asynchronous microservices because the history of the events is stored long-term. In addition, it can also support a large number of clients, without consuming too many resources.

An HTTP / REST based system that offers changes as an Atom feed or in a different data format has the advantage over Kafka that it does not need an additional server. However, it is not that easy to send a message to a single recipient, because the protocol does not have a direct support for this.

Experiments

- Start the Kafka example. See https://github.com/ewolff/microservice-kafka/blob/master/HOW-TO-RUN.md.
- It is possible to start multiple instances of the shipping and invoicing microservice. This can be done with `docker-compose up -d --scale shipping=2` or `docker-compose up -d --scale invoicing=2`. With `docker logs mskafka_invoicing_2` you can look at the logs. In the logs the microservice indicates which Kafka partitions it is accepting records from.

Concept: Synchronous Microservices

Many microservice systems use synchronous communication. This chapter shows how synchronous microservices can be implemented with different technologies.

Definition

The last chapter already defined synchronous microservices:

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

Thus, a synchronous order microservice can request customer data from another microservice while processing a request for an order.

Many microservice systems use synchronous communication. This chapter shows how synchronous microservices can be implemented with different technologies.

Why Synchronous Microservices?

The reasons for using synchronous microservices are:

- Synchronous microservices are *easy to understand*. Instead of a local method call, functionality is called in another

microservice. This is quite close to what programmers are used to.
- Better *consistency* can be achieved. For each call the latest information is retrieved from all other services. So the data is up-to-date unless a last-minute change has occurred.

But *resilience* is more complex: If the called microservice is currently not available, the caller must deal with the failure in a way that ensures that it does not fail as well. For this, the caller can use data from a cache or resort to a simplified algorithm that does not need the information from the other microservice.

Challenges

For synchronous communication some challenges have to be solved:

- A microservice typically provides its interface via TCP / IP at a specific IP address and port. The caller must get this information. *Service discovery* solves this challenge.
- For each microservice, multiple instances can run. *Load balancing* must distribute the calls to all instances.
- To external users all microservices should be perceived as part of a system and be available under one URL. *Routing* ensures that calls are forwarded to the correct microservice.
- As already mentioned, *resilience* presents a particular challenge that must also be addressed.

A technology for implementing synchronous microservices must provide a solution to each of these challenges.

Recipe: Kubernetes

Kubernetes[25] is becoming increasingly important as the environment for the development and operations of microservices.

Docker

Kubernetes is based on Docker[26]. Docker makes it possible to decouple processes from each other in a Linux system: *Docker containers* provide an operating system process with its own file system, its own network interface, and its own IP address. Unlike a virtual machine, however, all Docker containers use the same Linux kernel. Therefore, a Docker container consumes hardly more ressources than a Linux process. It is easily possible to run hundreds of Docker containers on a laptop.

File systems in Docker containers are based on *Docker images*. The images contain all the files that the Docker container needs. This can include a Linux distribution or a Java runtime environment. Docker images have layers. The Linux distribution can be one layer and the Java runtime environment another. All Java microservices can share these two layers. These layers are stored only once on the Docker host. This significantly reduces the storage space occupied by Docker images.

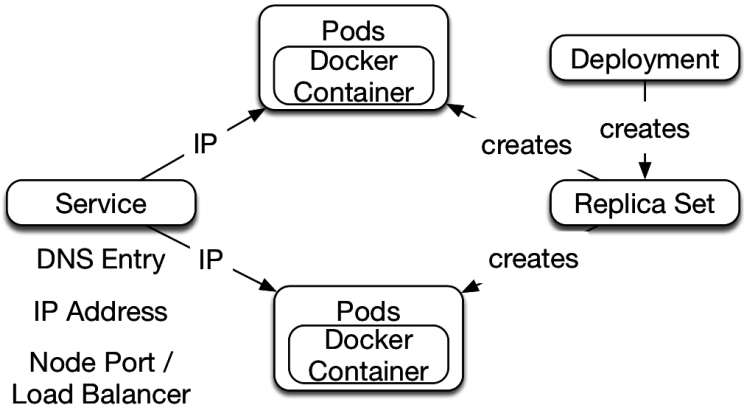
Kubernetes is a Docker Scheduler.

Running Docker containers on a single Docker host is not sufficient. If the Docker host fails, all Docker containers will fail. In addition, the scalability is limited by the performance of the Docker host.

To run Docker containers on a cluster of machines, there are schedulers like *Kubernetes*. Kubernetes introduces some new concepts:

[25] https://kubernetes.io/
[26] https://www.docker.com/

- *Nodes* are the servers Kubernetes is running on. They are organized in a cluster.
- *Pods* are multiple Docker containers that provide a service together. This could be, for example, a container with a microservice together with a container for log processing.
- A *replica set* ensures that there is always a certain number of instances running for each pod.
- A *deployment* creates a replica set and provides the required Docker images.
- *Services* make pods accessible. The services are registered under a name in the DNS and have a fixed IP address under which they can be contacted throughout the cluster. In addition, the service enables the routing of requests from the outside to a service instance.



Kubernetes Concepts

The figure illustrates all the Kubernetes concepts: A deployment creates a replica set. The replica set does not just create the Kubernetes pods, but also launches new ones, in case some of the pods fail. The pods include one or more Docker containers.

The Kubernetes service creates the DNS record and makes the microservice available at an IP address that is unique throughout

the cluster. Finally, the server creates a *node port*. Under this port the service can be reached on all Kubernetes nodes. Instead of a node port, a service can also create a *load balancer*. This is a load balancer offered by the infrastructure. For example, if Kubernetes is running in the Amazon Cloud, Kubernetes would create an Amazon Elastic Load Balancer (ELB).

Synchronous Microservices with Kubernetes

Kubernetes solves the challenges of synchronous microservices as follows:

- For *service discovery* Kubernetes uses DNS. The Kubernetes service sets up the corresponding DNS entry. Other microservices can then access the service by its host name.
- Kubernetes implements *load balancing* at the IP level. The Kubernetes service has an IP address. Behind the scenes, traffic to the IP address is load balanced across all service instances.
- Concerning *routing* the Kubernetes service can be implemented either via the node port or via a load balancer. It depends on how the service is configured and if the infrastructure offers a load balancer. An external request is either sent to the load balancer or to the node port to reach the microservice.
- For *resilience* Kubernetes has no solution. Of course, Kubernetes can start additional pods in case of a failure, but further resilience patterns like timeout or circuit breaker are not implemented by Kubernetes.

The solutions that Kubernetes offers for the challenges of synchronous microservices do not lead to any code dependencies on Kubernetes. If a microservice invokes another, it must resolve the name using DNS and communicate with the returned IP address. This is no different from communicating with any other server. For

routing, an external system uses a port on a Kubernetes host or a load balancer. Even in this case, it is transparent that Kubernetes is at work behind the scenes.

The example with Kubernetes

The example is available on GitHub[27]. A guide[28] explains in detail the necessary steps to install the software and run the example.

The example consists of three microservices: order, customer and catalog. Order uses catalog and customer via the REST interface. In addition, every microservice provides some HTML pages.

In the example also an Apache web server is installed, which provides the users with a website to facilitate using the system.

Finally, a Hystrix dashboard is available as a separate Kubernetes pod. The example uses the Java library Hystrix[29] for resilience. Among other things, this library runs calls in a separate thread pool, and it implements a timeout for the calls.

On a laptop, you can use Minikube[30] to run the example. This Kubernetes distribution is very easy to install. However, it provides no load balancer, so the services are only accessible over a node port.

The script `docker-build.sh` creates the Docker images for the microservices and uploads them to the public Docker hub. This step is optional because the images are already stored on the Docker hub.

The script `kubernets-deploy.sh` deploys the images from the public Docker hub. To do this, the script uses the tool `kubectl`. `kubectl run` starts the image. The image is downloaded from the specified URL at the Docker hub. In addition, this command defines which ports the Docker containers will provide. `kubectl run` creates the

[27] https://github.com/ewolff/microservice-kubernetes
[28] https://github.com/ewolff/microservice-kubernetes/HOW-TO-RUN.md
[29] https://github.com/Netflix/Hystrix/
[30] https://github.com/kubernetes/minikube

deployment which constructs the replica set and thus the pods. `kubectl expose` creates the service that provides accesses to the replica set and create an IP address, a node port or load balancer and a DNS entry.

This excerpt from `kubernetes-deploy.sh` shows how the tools are used to deploy and run the catalog microservice:

```
#!/bin/sh
if [ -z "$DOCKER_ACCOUNT" ]; then
  DOCKER_ACCOUNT=ewolff
fi;
...
kubectl run catalog \\
 --image=docker.io/$DOCKER_ACCOUNT/microservice-kubernete\
s-demo-catalog:latest
  \\
 --port=80
kubectl expose deployment/catalog --type="LoadBalancer" -\
-port 80
...
```

Alternative Recipes: Netflix, Consul, Cloud Foundry

In addition to Kubernetes, there are several other solutions for synchronous microservices:

- *Cloud Foundry* also uses Docker like Kubernetes. However, Cloud Foundry is a PaaS (Platform as a Service). It provides a complete platform for the application. That's why it is not necessary to create Docker containers. It is sufficient to just provide a Java application.

  - Cloud Foundry also implements *service discovery* with DNS.
  - The platform implements *load balancing* at the network level.
  - For *routing* of requests from external systems it is sufficient to use the DNS name of the microservice.
  - Much like Kubernetes Cloud Foundry does not really support *resilience*.

    The Cloud Foundry demo[31] implements an example that is basically identical with the Kubernetes example. There is a detailed guide[32] on how to run the example.
- *Consul* is actually a service discovery technology. However, it can be combined with some other technologies to provide a complete solution for microservices.
  - Consul also offers a DNS interface for *service discovery*. It also has a separate interface for service discovery that can be used to add and read the information.
  - For *routing* Consul itself offers no solution. But Consul Template[33] can fill out a template with information about the microservices to create a configuration file. For example, a web server can be configured in this way to receive HTTP requests from the outside and send them to the microservices. The web server reads the configuration file provided by Consul Template. It does not need to implement any interface to Consul.
  - *Load balancing* can be implemented just like routing with a web server and Consul Template. An alternative is a Java library like Ribbon[34]. It implements load balancing in the calling microservice.
  - *Resilience* needs to be implemented with an additional library.

[31] https://github.com/ewolff/microservice-cloudfoundry
[32] https://github.com/ewolff/microservice-cloudfoundry/blob/master/HOW-TO-RUN.md
[33] https://github.com/hashicorp/consul-template
[34] https://github.com/Netflix/ribbon/wiki

The Consul example[35] uses Spring Cloud to register the microservices and the Ribbon library for load balancing. Hystrix provides resilience. Apache httpd implements routing and Consul Template configures Apache httpd. An alternative would be Registrator[36]. It automatically registers Docker containers in Consul. Together with access to Consul via DNS, Consul can be just as transparently used as Kubernetes or Cloud Foundry. The Consul DNS example[37] implements this approach.

- The *Netflix Stack* provides a complete solution for synchronous Microservices:
  - Eureka implements *service discovery*. It has a REST interface, and the Eureka Java client library also implements a cache on the client.
  - Ribbon is the *load balancer* of the Netflix stack. This is a Java library that selects one of the service instances registered at Eureka.
  - Zuul is a proxy for *routing* written in Java. Zuul can be supplemented with custom filters, which can be written in Java or Groovy. Therefore, Zuul can be extended very flexibly.
- For *resilience* the Netflix stack uses Hystrix.

  The Netflix example[38] uses Spring Cloud to integrate the Netflix stack into Java applications. The microservices system implements the same scenario as the other examples for synchronous microservices.

The Kubernetes and Cloud Foundry examples have no code dependencies. Such a solution can also be implemented with Consul. In this way, the microservice systems can easily use other technologies than Java. This supports technology freedom, a major benefit of microservices.

[35] https://github.com/ewolff/microservice-consul/
[36] https://github.com/gliderlabs/registrator
[37] https://github.com/ewolff/microservice-consul-dns/
[38] https://github.com/ewolff/microservice

Conclusion

Kubernetes offers a very powerful solution for synchronous microservices that also covers the operations of microservices. PaaS like Cloud Foundry provide a higher level of abstraction, thus the user does not have to deal with Docker. But both, Kubernetes and Cloud Foundry, force users to run a different runtime environment. It is not possible to stick to bare metal or virtual systems, instead Kubernetes or a PaaS like Cloud Foundry must be used. This is not the case with Consul and Netflix: Both systems can be used with Docker containers as well as with virtual machines or physical servers. Of those two, Consul offers a lot more features.

Experiments

- Start the Kubernetes example as described in the guide[39].
  - Open the Apache httpd website with `minikube service apache`.
  - Open the Kubernetes dashboard with `minikube dashboard`.
- Test the load balancing in the example:
  - `kubectl scale` changes the number of pods in a replica set. `kubectl scale -h` shows the options of the command. For example, scale the replica set `catalog`.
  - `kubectl get deployments` shows how many pods are running in each deployment.

[39] https://github.com/ewolff/microservice-kubernetes/blob/master/HOW-TO-RUN.md

What next?

This booklet can only provide a brief introduction to microservices.

The book Microservices - A Practical Guide[40] contains a detailed description of the examples in this brochure. In addition, it contains an introduction to microservices and an overview of technologies for the monitoring of microservices. There is also a German version[41].

The book Microservices[42] describes the motivation, architecture, and concepts of microservices. There is also a German version[43].

Finally, the free Microservices Primer[44] provides an overview of the basic motivation and architecture of microservices. There is also a German version[45].

[40] http://practical-microservices.com/
[41] http://microservices-praxisbuch.de/
[42] http://microservices-book.com/
[43] http://microservices-buch.de/
[44] http://microservices-book.com/primer.html
[45] http://microservices-buch.de/ueberblick.html