Design and Implementation of a Service Discovery System with Dynamic Routing

Bachelor's Thesis

Björn Vonheiden

September 27, 2019

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Nelson Tavares de Sousa, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 27. September 2019

Abstract

Traditional applications, such as monoliths, are available at a static network location. By using cloud based microservices, however, these services become more dynamic. In contrast to those monolithic applications, cloud based services cannot directly call other services through methods. Instead, they use defined APIs which are called through HTTP or other mechanisms. In the cloud, the microservices get scaled, updated and replaced on failures. Thus, their network locations change and, therefore, the different services need a mechanism to find and reach each other. This is solved by a service discovery mechanism, in which a service registry contains all network locations of these services. The service discovery can be used in GeRDI to integrate and find new store services automatically.

The different services of a microservice architecture are distributed. Clients interact with these different services and, therefore, need to know all locations. When the API of a service changes, the client also needs to change. An API gateway encapsulates services in that way that it routes to the different services and provides the client a single entry point to the backend. Moreover, the gateway may utilize service discovery for dynamically providing services. The gateway can be used in GeRDI to expose new store services automatically to the frontend.

In this thesis, we will implement different service discovery mechanisms and API gateways. These will be integrated in the GeRDI store service and later the different solutions will be evaluated.

Contents

List of Acronyms

API	Application programming interface
AOP	Aspect Oriented Programming
DI	Dependency Injection
DNS	Domain Name System
GeRDI	Generic Research Data Infrastructure
HTTP	Hypertext Transfer Protocol
I/O	Input/output
IoC	Inversion of Control
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter process communication
JPA	Java Persistence API
MTPR	Mean time per request
REST	Representational State Transfer
RDM	Research Data Management
RPS	Requests per second
SCG	Spring Cloud Gateway
SCS	Self-contained System
UI	User Interface
URL	Uniform Resource Locator
VIP	Virtual IP
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

1.1 Motivation

In the microservice architecture pattern, or in this case Self-contained Systems (SCSs) which are closely related to microservices, exists multiple services that need to communicate with each other. There are many instances of each service and new services may be added while already existing ones may be removed at runtime. Services need to know which of those are still available and where the endpoints for them are located. Therefore, service discovery mechanisms are used to solve this problem. Services are, thus, registered in a registry and may also query other services. Furthermore, external clients also need to access the different services, which may be solved by using an API Gateway. It provides the capability to use the service discovery and route dynamically to the services.

The Generic Research Data Infrastructure uses the SCS architecture. Currently there is no mechanism to route dynamically to the existing services and the routes for a client are statically defined. With new services the configuration needs to be modified manually. Implementing a service discovery and an API Gateway solve these problems.

1.2 Goals

This section covers the goals that should be achieved. The main goal is to implement a service discovery mechanism and an API gateway for the store services in the Generic Research Data Infrastructure (GeRDI). Ultimately, different approaches, that are used to achieve the main goal, are evaluated.

1.2.1 G1: Service Discovery for the Store Services

When starting or stopping a store service the configuration of the endpoint and availability of this service needs to be updated manually. An Apache[1] reverse proxy is used to expose a store service to the frontend. In the reverse proxy, the endpoint API URL for a service is statically defined. Similarly, the frontend contains a list of store services with their endpoint URLs which is also statically defined.

[1] https://httpd.apache.org

Currently there are JupyterHub and WebDAV as store services while other services that store research data may also exist in the future. The routes and existing services are modified manually in the current configuration. Furthermore, the locations of the services are fixed and, therefore, it is not easy to move the service to a different location.

The solution is to implement a service discovery mechanism. A service registry contains all services with their name and the locations of the service instances. There are two ways in which the different services may be recorded in the service registry: on the one hand, the different services may register themselves actively, or on the other hand, are registered by a registrar. The different store services, then, can be queried and used in the front end.

We expect to have different services for storing in the future. Furthermore, services may scale on purpose, may change their location or may be unavailable. By using the service discovery, the available services are configured automatically and no manual configuration is needed anymore. In addition, other services may use the service discovery as well.

1.2.2 G2: API Gateway with Dynamic Routing for the Store Service

The API endpoints of the store services are defined statically in an Apache reverse proxy. For a new service as well as for a removed service, an endpoint either needs to be added or deleted manually.

Available services should be propagated automatically and requests should be routed dynamically to one instance of the requested service.

The solution is to implement an API gateway: It provides the current store services for the front end by using the service registry. Furthermore, the gateway routes the requests to one instance of a service using the service discovery.

One reason for this implementation is that there may be new services available in the future. These would then be added as a new route and, thus, become available automatically. Another reason is that service endpoints may also change which would have to be handled manually. Further, it simplifies the client, because the endpoints get aligned and follow a consistent scheme.

1.2.3 G3: Evaluation of Different Solutions for Service Discovery and API Gateway

For the goals in Section 1.2.1 and Section 1.2.2 we want to implement and evaluate different approaches.

We execute a benchmark test to explore the additional overhead from the gateways and service discoveries. Therefore we connect the different API gateways to the service discovery mechanisms. Then we execute the benchmark on the different solutions. To get a reference we execute the benchmark also with the direct access. Dependent on this we compare the different results.

Further the service discovery mechanisms and API gateways will be compared regarding their provided functionalities.

1.3 Document Structure

In Chapter 2, the theoretical background for the thesis is provided. An analysis of the current architecure and an outline of the new infrastucture follow in Chapter 3. Chapter 4 deals with the service discovery implementations. We implement the API gateways and connect them to the service discovery mechanisms in Chapter 5. The adaption for the frontend is further shown in Chapter 6. Chapter 7 then discusses different solutions by comparing them with regard to their features. Chapter 8 deals with related works. Ultimately, Chapter 9 provides the conclusion of the thesis and gives and outlook for future work.

Chapter 2

Foundations and Technologies

This chapter covers essential foundations and technologies.

2.1 Microservices

The microservice architecture pattern is an architectural style to build software. It defines an implementation view for multiple components of a system. One component of the system is a service. Communication protocols connect these services with each other. An application, then, is composed by a set of independent services [Hasselbring 2016; Richardson 2019; Namiot and Sneps-Sneppe 2014].

Chris Richardson defines a service as the following:

> A service is a standalone, independently deployable software component that implements some useful functionality. (Richardson 2019)

A service implements a specific business capability. It is a full-stack implementation for this business area with its own database and business logic [Hasselbring 2016; Richardson 2019].

Services are standalone and independent software components. In this way, they should be loosely coupled to maintain their independence. Services interact with each other through their APIs, meaning that there are no shared databases [Richardson 2019]. In order to avoid dependencies between microservices, code should not be shared between them [Hasselbring and Steinacker 2017].

The term micro in microservices, however, is no measure for the size of a service. A small team should be able to develop a microservice "with minimal lead time and with minimal collaboration with other teams" (Richardson 2019).

2.2 Self-contained Systems (SCSs)

SCS is an architectural pattern that divides functions of a system into own applications. The idea is to split a monolithic application vertically. A single function of the system is an application with its own User Interface (UI), business logic, and persistence. The application is totally independent and should only be accessed over defined interfaces from the outside. The favored communication between SCSs, if necessary, is asynchronously over

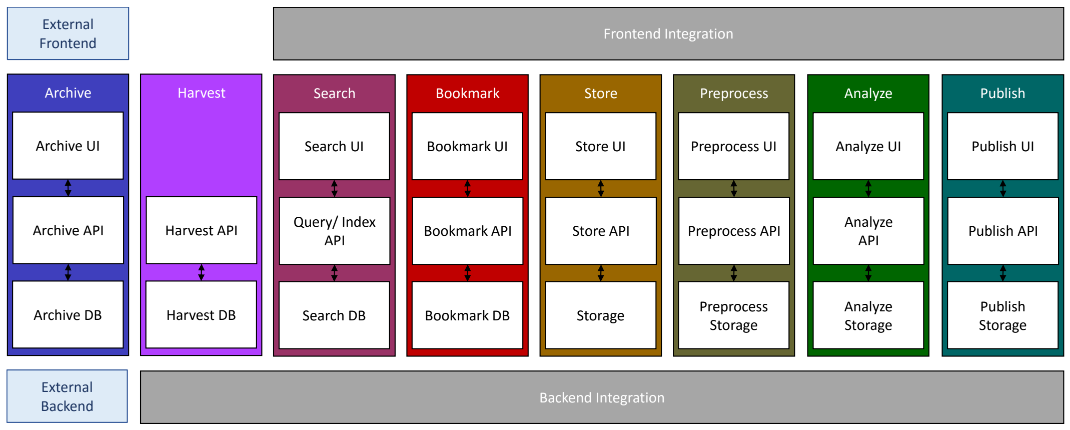

Figure 2.1. Architecture from GeRDI with the different SCSs

REST-interfaces. The parts of a SCS could be implemented as microservices [Tavares de Sousa and Hasselbring 2018]. In comparison to the microservices explained in Section 2.1, the code-base is shared inside a SCS between UI and the microservices [Cerny et al. 2017].

2.3 Generic Research Data Infrastructure (GeRDI)

The Generic Research Data Infrastructure (GeRDI) project is about building a Research Data Management (RDM) system. The platform interconnects existing research data repositories, facilitates search on them ,and provides capabilities to work on the research data [Grunzke et al. 2017].

Domain driven design has been used to model the system. It is split into different bounded contexts. The required functionality of GeRDI has been clustered into different abstract services that have been used for the different domains. Every SCS implements a domain and the different SCSs are depicted in Figure 2.1. The store service is one SCS from GeRDI. The function is to store collected research data to a local machine or in a remote storage. GeRDI has a frontend and a backend integration besides the SCSs [Tavares de Sousa et al. 2018].

2.4 Service Discovery

Service discovery is about finding instances of a service. The key component of the service discovery is the service registry. The registry is a database that stores the location of instances from services. It provides an interface for registration, health check and exposing services [Tang et al. 2019; Richardson 2019].

As soon as an application starts or stops, the service discovery mechanism updates the registry. When the client want to invoke a service the service discovery mechanism queries the registry for instances. Then, the request is routed to one of them. There are two main approaches for implementing the service discovery: In the first approach, clients and services interact directly with the registry. In the second, the deployment platform is in charge to handle the discovery [Richardson 2019].

In the following, application-level and platform-provided service discoveries are explained in more detail.

2.4.1 Application-Level Service Discovery

Services and clients of the application interact directly with the service registry. A service registers its network location within the service registry. In order to send a request to a service the client first queries the service registry for a list of instances. Then it sends the request to one of those instances.

The self registration and client-side discovery patterns are used in this approach. For the self registration, a service invokes the API of the service registry and registers its network location and optional a health check URL. The health check URL is used to obtain the status of the service. Further, the registry may require the service to send a periodic heartbeat in order to show its availability. A service may be deregistered when its health check URL is not reachable, when it does not send the heartbeat for a few times or when it deregisters itself.

When a client wants to invoke a service, it uses the client-side discovery pattern. The client queries the service registry for a list of service instances. It chooses an instance with a load balancing algorithm and sends the request to the selected one [Richardson 2019].

2.4.2 Platform-Provided Service Discovery

The deployment platform incorporates a service registry and discovery mechanisms. A Domain Name System (DNS) name and a Virtual IP (VIP) address is provided for each service. The DNS name resolves to the VIP address. Clients send requests to the DNS name or VIP address and the platform then routes to one of the running instances of the service.

The platform has a registry with the IP addresses of the service instances. Requests to services are load balanced by the platform to one of the instances.

However, the services do not register themselves. Instead, a registrar registers these services in the registry. This pattern is called 3rd party registration.

The server-side discovery is another pattern, in which the service registry is not requested directly by the clients. Instead, they send their requests to a DNS name, which is further resolved to a request router. The request router then queries the services registry and load balances the requests [Richardson 2019].

2.5 Netflix Eureka - A Service Registry

Eureka[1] is a service registry developed by Netflix and publicly available at GitHub[2]. It is used for service discovery to enable mid-tier load balancing and resilience. Components of Eureka are the Eureka server and the Eureka client. The server is a service registry and the service is REST based. It contains all registered services with their instances. For every instance the host name, IP address, and metadata are stored. The client interacts with the registry and is used for registering a service and fetching the available services. A load balancer with a round-robin algorithm is embedded in the Eureka client but own load balancing solutions are possible. [Eureka 2014].

Figure 2.2 shows a possible architecture for Eureka. There is a region with one Eureka cluster depicted. The region is partitioned into several zones. There should be at least one Eureka server per zone to handle zone failures. Application services and clients contain a Eureka client. An application service registers with the Eureka server in its zone. It then sends heartbeats every 30 seconds to renew the lease, and when the service fails to send heartbeats, it is removed within 90 seconds. The Eureka servers in a cluster replicate their information. Application clients can lookup the registry on any Eureka server from the cluster. A lookup happens every 30 s.

2.6 Docker - A Container Platform

Containers are an operating-system-level virtualization mechanism. A process in a container runs in its own sandbox, isolated from other containers. For a running process it is like an own machine. Each container has a root file system [Richardson 2019].

Docker[3] is a container runtime and a platform to package, distribute and execute applications. Containers are created by instantiating Docker images. A container image defines a file system image with the application and the needed software to run this application. With a Dockerfile we describe how to build an image and create container images. A base image defines the starting point for a new image in the Dockerfile. Different instructions build new layers on top of this image. Instructions can install software or configure the container. The last instruction in a Dockerfile is a shell command that is executed at the start of a container [Richardson 2019; Luksa 2018].

2.7 Kubernetes - An Orchestration Platform

Kubernetes[4] is a docker orchestration platform. It runs on a cluster of machines and abstracts the resources of the machines into a single machine. Functions of the orches-

[1] `https://github.com/Netflix/eureka`
[2] `https://github.com`
[3] `https://www.docker.com`
[4] `https://kubernetes.io`

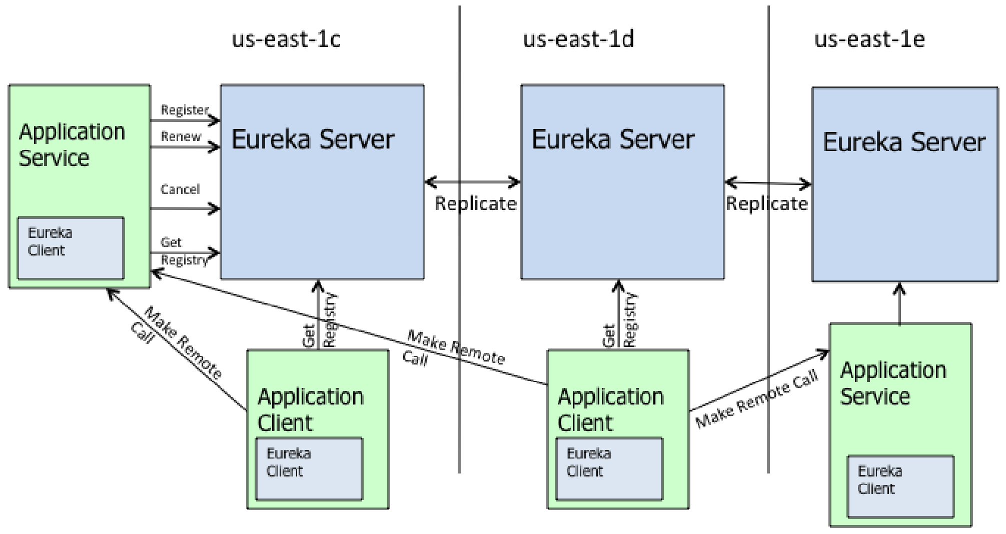

Figure 2.2. High level architecture of Eureka [Eureka 2014] with a region and multiple zones

tration platform are resource management, scheduling, and service management. Pods, deployments, and services are important components of Kubernetes.

A pod is the basic deployment unit in Kubernetes. One ore more containers are part of a pod. These containers share the same IP address and storage.

A deployment is a declarative specification of a pod. A controller exists for a deployment and ensures to have the desired state, like the number of pod instances, in Kubernetes.

A service abstracts a number of pods to a single service and has a static IP address, port and DNS name. The IP address and the DNS name are accessible within the cluster and fixed for the lifespan of a service [Richardson 2019].

Services are defined in YAML files. A `selector` defines which pods are part of the service and ,therefore, uses the labels of pods. Further, the port of the service, the protocol, and the target port of the pods are defined in the YAML file.

A new service creates an endpoint in Kubernetes. The endpoint has the same name as the service and contains a list of IPs with ports. A service controller is in charge to update the endpoint on changes. It is possible to create a service without a selector. Hence, it is necessary to create an endpoint manually.

Every node in Kubernetes has a **kube-proxy** installed. The proxy uses the IP address of a service to route and load balance to the belonging pods. Different proxy modes enable different load balancing strategies.

Within Kubernetes a built in DNS service is running that is used automatically by the pods. This service creates DNS entry for every service. The entry has the form `<service>.<namespace>.svc.<zone>` where `<service>` is the name of the service, the `<namespace>` where the service is deployed and `<zone>` the cluster domain (typically `cluster.local`). All pods in the cluster can reach the service with this name [Kubernetes 2019].

2.8 API Gateway

The API gateway is the entry point to an application from the outside world. All API requests from external clients to the application go to the API gateway. Aftewards, they are routed to the corresponding service. The gateway encapsulates the inner architecture of the application. The main functions are request routing, API composition and protocol translation. Further, it could provide edge functionalities like authentication, authorization, rate limiting, caching, metrics collection and request logging [Richardson 2019].

2.9 Netflix Zuul - A Gateway Service

Zuul[5] is a gateway service developed by Netflix. It provides dynamic routing, monitoring, resiliency, security, and more. Two versions of Zuul currently exist, i.e. version 1.x and 2.x. Further, Zuul utilizes filters in order to add functionality [Zuul 2018].

Pre, routing, post, and error filters exist in Zuul 1. The pre filter is executed before the request is routed to the endpoint. In the routing filter a request is sent to an endpoint and the post filter is executed after the routing.

If an error occurs in one of the filters, the error filter is executed. Further, the filters have some characteristics in common: All filters incorporate a type, an execution order, a criteria and an action. The type defines when to run the filter and the execution order the order to run relative to the other filters. A filter is only run when its criteria are met. The action is executed during the run of the filter [Zuul 2013].

2.10 Spring Framework

Spring[6] is an open source framework for developing Java applications. It utilizes Inversion of Control (IoC), Dependency Injection (DI), Aspect Oriented Programming (AOP), and the

[5]`https://github.com/Netflix/zuul`

[6]`https://spring.io`

Java Persistence API (JPA).

The development of a Spring Applications is simplified by using Spring Boot which employs starter dependencies to integrate required dependencies automatically and configures the application automatically as a Spring application [Jovanovic et al. 2017].

Chapter 3

Approach

3.1 Current Approach

Two store services exist in the current architecture. One is WebDAV and the other JupyterHub which are both deployed in Kubernetes and exposed with a Kubernetes service. An Apache HTTP Server is run independently and outside of Kubernetes. The Apache is accessible over the internet and works as a reverse proxy. For this, the Apache is connected to the network of the Kubernetes cluster. In Listing 3.1 the proxy definitions for the store services are outlined. As seen, these are statically defined and need to be modified manually on changes.

Listing 3.1. Extract of the Apache reverse proxy configuration for the store services

```
ProxyPass         /api/v1/store       http://webdav-store.default.svc.cluster.
    local:5678
ProxyPassReverse  /api/v1/store       http://webdav-store.default.svc.cluster.
    local:5678
ProxyPass         /api/v1/store-jhub  http://jhub-store.default.svc.cluster.
    local:5678
ProxyPassReverse  /api/v1/store-jhub  http://jhub-store.default.svc.cluster.
    local:5678
```

Figure 3.1 shows the communication of the frontend with the backend. In the first step the frontend sends a store request to the backend (1). The Apache processes this request and routes depending on the requested URL to the desired store service (2a/b). The store service sends the response to the Apache reverse proxy (3a/b) which routes the response back to the client (4).

3.2 Envisioned Approach

We add a service registry for the service discovery and an API gateway for the dynamic routing to accomplish the goals. Both will be deployed in Kubernetes to follow the existing practice. The remaining architecture stays the same.

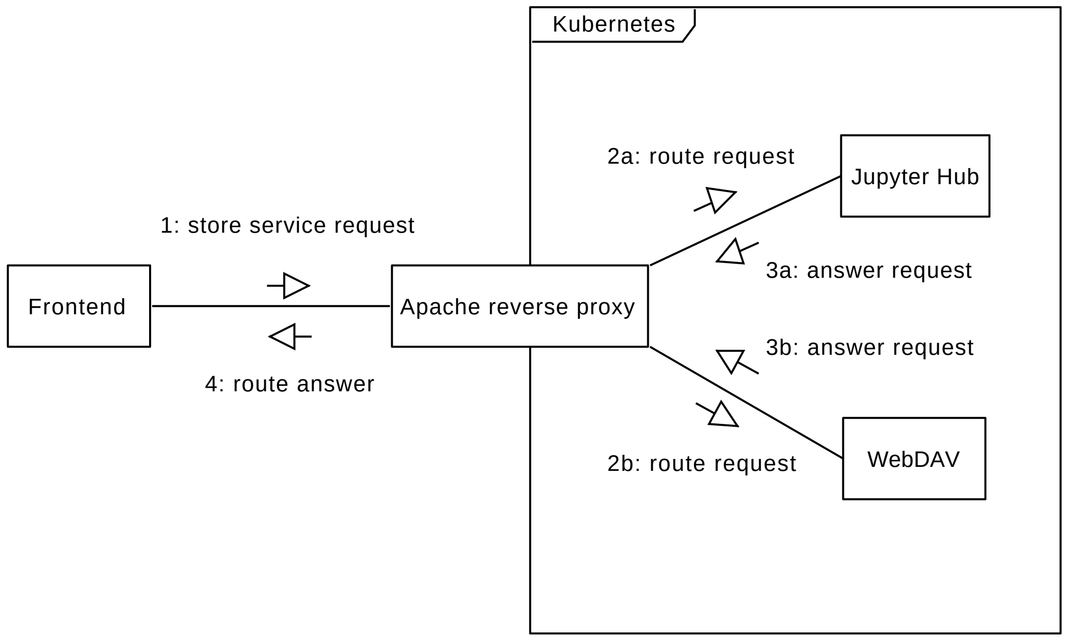

Figure 3.1. Communication for a request to a store service in the existing architecture

The new communication path for the interaction is shown in Figure 3.2. First, the applications register themselves in the registry (1). New, but yet unknown services are depicted as **Store Service X** which are also registered in the registry (1). When the frontend loads, it requests the available store services (2). The Apache is still the access point from outside, but it only forwards all requests to and from the API gateway and the frontend (3,7,9,15). The API gateway requests the services from the registry (4,5) and returns a list of available services to the frontend (6). Depending on this, the frontend adds the available store options.

In this way, the frontend is then able to send store service requests to the backend. On a store service request (8) the API gateway uses the service discovery to find available instances of the demanded service (10,11). Accordingly, it selects an instance and sends the request to it (12a/b/c). The response is forwarded back to the frontend (13a/b/c, 14).

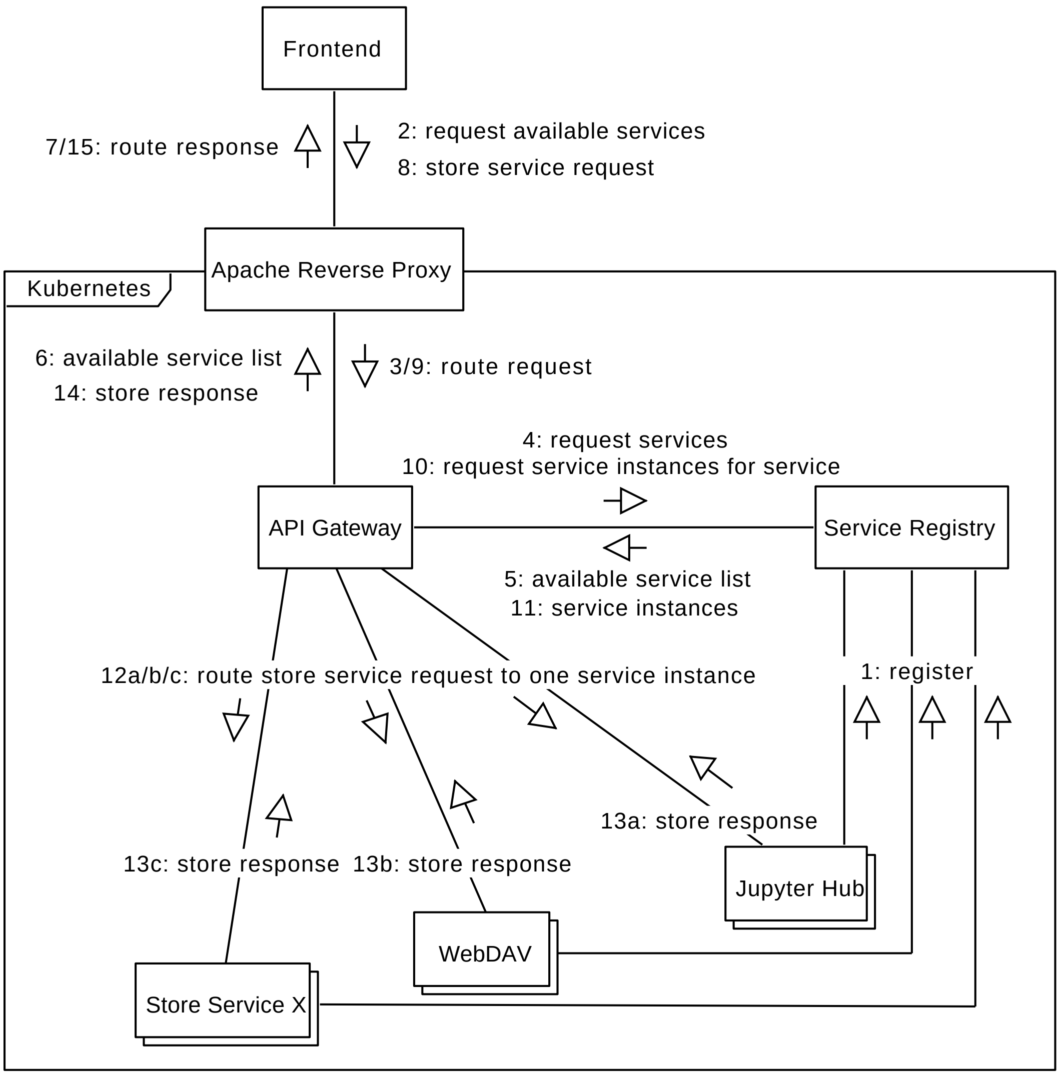

Figure 3.2. Communication with the different services using the API gateway and service discovery

Chapter 4

Service Discovery

This section covers the implementation of different service discovery mechanisms.

4.1 Netflix Eureka

4.1.1 Eureka Server

The Spring Cloud Netflix[1] project provides an out of the box Eureka server. We create the project with the Spring Initializr[2]. This is a tool for initializing Spring Boot applications. We choose Gradle[3] as the build tool and Spring Boot version 2.1.5. The Eureka server dependency is added to get the registry. Java version 8 is required to build the server because the server depends on dependencies that are removed in Java 11. We add the `@EnableEurekaServer` annotation to the Spring Boot application to enable the Eureka Server.

Afterwards, we configure the server. The configuration is shown in Listing 4.1. In line 2 we set the port of the integrated Tomcat server. The server is the registry itself and should not be listed as a service. Further, it does not need to fetch the registry. We disable these functionalities of the client in line 6 and 7.

Listing 4.1. Application YAML file for Eureka Server

```
1 server:
2   port: 8761
3 
4 eureka:
5   client:
6     register-with-eureka: false
7     fetch-registry: false
```

With `gradle bootRun` or executing the jar, the server is started. Consequently, an overview page becomes available at `http://localhost:8761`. The Eureka Representational State Transfer (REST) API is available at `http://localhost:8761/eureka/`.

[1] https://spring.io/projects/spring-cloud-netflix
[2] https://start.spring.io
[3] https://gradle.org

We create a Dockerfile and a Kubernetes deployment and service to start the Eureka server in Kubernetes.

4.1.2 Eureka Client

The JupyterHub Store, WebDAV Store, and prospective stores need to register to the Eureka server. An abstract store service library exists for the stores. This library defines the REST API, creates a server and defines functions that need to be implemented for store services. The server uses the Java Spark[4] framework. To use the library the `AbstractStoreService` class need to be subclassed. JupyterHub and WebDAV uses this library. To integrate the Eureka client we implement it in the store service library.

The `EurekaClient` needs an `ApplicationInfoManager` and an `EurekaClientConfig` object. The `ApplicationInfoManager` contains information about the application for the registration with the Eureka server. We have two options to configure the `ApplicationInfoManager`. One is to use a `eureka-client.properties` file and create a `MyDataCenterInstanceConfig` object. The configuration is then loaded into the object from the file. All stores that use the service library need to have an own configuration file. The other option is to inherit from `MyDataCenterInstanceConfig` and overwrite the fields of interest. We choose the second option because it is easier to maintain common values for store services with it. We set the application name and the port of the application in the configuration class. To obtain the application name we modify the constructor of the `AbstractStoreService`. This receives a name for the application and uses it in the configuration.

The `EurekaClientConfig` contains information how to register with the Eureka servers. We can also choose between a configuration file or an own configuration class. With the same reason as above we choose a configuration class. In the configuration class we add the URL of the Eureka server.

The `EurekaClient` automatically registers with the defined Eureka server and sends a regular heartbeat. To deregister from the Eureka server when the application is closed we add a shutdown hook. There we shutdown the `EurekaClient` which deregisters the application from the Eureka server.

4.2 Kubernetes Service Discovery

As described in Section 2.7, Kubernetes allows to define services in a YAML file. Services are already defined and ready to use for JupyterHub and WebDAV. Therefore, nothing has to be done to register the services in Kubernetes. The configuration for the JupyterHub service is shown in Listing 4.2.

Listing 4.2. Service definition in Kubernetes for the JupyterHub store

```
apiVersion: v1
```

[4] `http://sparkjava.com`

```
 2 kind: Service
 3 metadata:
 4   name: jhub-store
 5   annotations:
 6     type: store
 7     name: Jupyter Hub
 8     serviceid: jhub
 9 spec:
10   ports:
11     - port: 5678
12       protocol: TCP
13   selector:
14     app: jhub-store
```

In line 2 the Kubernetes object is defined as a service. The name of the service is set in line 4. Metadata, in form of annotations, are set from lines 5 to 8. The selector in line 14 defines that all pods with the label `app: jhub-store` belong to this service.

The DNS entry `jhub-store.default.svc.cluster.local` is automatically created for the service. Requests can be sent to it and they get load balanced to the pods of the service. Further, Kubernetes provides a REST API. Because of the API, it is possible to list all services and get the pods that belong to the service.

Chapter 5

API Gateway with Service Discovery

This section covers the implementation of different API gateways. Afterwards, we connect them to the different service discovery mechanisms from Chapter 4 and, thus, enable dynamic routing to the services.

We do not want all services to be exposed and, therefore, we use metadata in the services to distinguish them. In Kubernetes we add an annotation to the services and in Eureka we send metadata with the Eureka client. Both service discovery mechanisms use key-value pairs for the metadata. We add the key **type** with the value **store** in Eureka and Kubernetes. Only services that have the key **type** will be routed with the API gateway and the value is used as a prefix for the service path. In Listing 4.2 we have already added the annotations to the Kubernetes service. We subclass `DefaultEurekaClientConfig` to set metadata in Eureka.

5.1 Netflix Zuul Implementation

In order to add the Zuul server to a Spring Boot application, we need to add `spring-cloud-starter-netflix-zuul` to the dependencies first. To activate Zuul we add the `@EnableZuulProxy` annotation on the Spring Boot application. The `@EnableZuulProxy` annotation adds predefined mechanisms for locating the routes and enables filters. This will enable us to use the service discovery, implemented in Chapter 4, later.

5.2 Spring Cloud Gateway Implementation

Spring Cloud Gateway (SCG)[1] is an API gateway that has been built on top of the Spring Ecosystem. It provides routing and cross cutting concerns like security, monitoring, and resiliency. The SCG utilizes routes that consist of an ID, a destination URL as well as collections of predicates and filters. The predicates determine if a route is chosen. Any attribute of a HTTP request can be matched with the predicates. When a route is selected, the pre filters are executed. A proxy request is made before the post filters are executed. The filters can modify the request and the response [Spring 2019].

[1] `https://spring.io/projects/spring-cloud-gateway`

To enable the SCG we need to add the `spring-cloud-starter-gateway` dependency in a Spring Boot application.

5.3 Spring Cloud Gateway with Kubernetes

We use the SCG with the Kubernetes service discovery. The Spring Cloud Kubernetes project provides a starter dependency for a discovery client that can be used to query Kubernetes for services and service instances.

We add the `spring-cloud-starter-kubernetes` dependency to include the Kubernetes discovery client. We further need to add the `@EnableDiscoveryClient` annotation to the Spring Boot application. This initializes the discovery client in the application. Then, we need to add `spring.cloud.gateway.discovery.locator.enabled: true` to the configuration file so that the discovery client is used to locate the routes in SCG (Listing 5.1, Line 7). This configuration only adds services that are active when the gateway is created. In order to update the available services automatically, we add the `@EnableScheduling` annotation to the application. The `KubernetesCatalogWatch` class has a method that is annotated with `@Scheduled`. With `@EnableScheduling` the method is executed in a regular interval. The interval length is 30 seconds by default. When the method is executed, it checks if the list of services changed and ,in case of changes, updates the routes.

For routing the requests to the correct services and discover the instances we add the `spring-cloud-starter-kubernetes-ribbon` dependency which includes a Ribbon[2] client. Ribbon is an Inter process communication (IPC) library from Netflix. We use the client with Kubernetes integration to discover the instances and load balance across them.

Further, we want to include only particular services, define the path for a service, and rewrite the path for the destination service. The configuration for this is shown in Listing 5.1.

Listing 5.1. Application YAML file for Spring Cloud Gateway with Kubernetes

```
 1 spring:
 2   application.name: spring-gateway
 3   cloud:
 4     gateway:
 5       discovery:
 6         locator:
 7           enabled: true
 8           include-expression: metadata['type'] != null
 9           predicates:
10             - name: Path
11               args[pattern]: "'/' + metadata['type'] + '/' + metadata['serviceid']
                      + '/**'"
```

[2] `https://github.com/Netflix/ribbon`

```
12          filters:
13            - name: RewritePath
14              args[regexp]: "'/' + metadata['type'] + '/' + metadata['serviceid']
                    + '/(?<remaining>.*)'"
15              args[replacement]: "'/${remaining}'"
```

Only routes with the key **type** should be included. SCG provides an `include-expression` to configure this and we define it so that the service needs to have the key **type** in its metadata (line 8).

With `spring.cloud.gateway.discovery.locator.predicates[x]` and `spring.cloud.gateway.discovery.locator.predicates[y]` we can customize the predicates and filters of the routes that are added with the service discovery. A predicate can be defined in order to add the prefix to the routes. We use the **type** value and the **serviceid** value to create the path (lines 10,11). When a route is matched their filters are applied. To route to the correct URL on the downstream request, we need to rewrite the path (lines 13-15).

5.4 Zuul with Kubernetes

To use Spring Cloud Zuul with Kubernetes service discovery we add the same dependencies as in Section 5.3. We also add the `@EnableDiscoveryClient` and `@EnableScheduling` annotations to the Spring Boot application.

Zuul should not expose all services. It provides an `ignoredServices` property to blacklist services. We use the property in line 2 in Listing 5.2 to blacklist all services by default.

Listing 5.2. Application YAML file for Zuul with Kubernetes

```
1  zuul:
2    ignoredServices: '*'
3    sensitiveHeaders:
4    ribbon:
5      eager-load:
6        enabled: true
7
8  ribbon:
9    NIWSServerListClassName: org.springframework.cloud.kubernetes.ribbon.
         KubernetesServerList
10   KubernetesNamespace: default
11   ConnectTimeout: 1000
12   ReadTimeout: 5000
```

The * is a wildcard to blacklist all services.

We create the `OwnDiscoveryClientRouteLocator` class and subclass `DiscoveryClientRouteLocator` to add our desired routes programmatically. After the `DiscoveryClientRouteLocator` loads routes from the configuration file, it loads services with the discovery client. Zuul filters these services with `ignoredServices` property and creates routes for the remaining services. The `OwnDiscoveryClientRouteLocator` first calls its super class to locate the routes (Listing 5.3, line 4). Then it loads all available services and filters for the key **type** in the annotations. The path is composed of the **type** and the **serviceid** values from the annotations (line 17). For the JupyterHub store service definition, illustrated in Listing 4.2, the created path would be `/store/jhub/**`. Listing 5.3 shows how the `ZuulRoute` is created. The filter function is shown in lines 7 to 12 and the function for adding routes in lines 15 to 19.

Listing 5.3. Method for finding routes in the OwnDiscoveryClientRouteLocator class

```
 1 @Override
 2 protected LinkedHashMap<String, ZuulRoute> locateRoutes() {
 3     //get routes with existing superclass locator
 4     LinkedHashMap<String, ZuulRoute> routesMap = super.locateRoutes();
 5
 6     // Filter all services that has metadata type
 7     Predicate<Service> filter = (Service s) -> {
 8         return Optional.ofNullable(s.getMetadata())
 9                 .map(ObjectMeta::getAnnotations)
10                 .filter(map -> map.containsKey("type"))
11                 .isPresent();
12     };
13
14     // Adds all services to the routes
15     Consumer<Service> consume = (Service s) -> {
16         String serviceID = s.getMetadata().getName();
17         String path = ServiceUtil.path(s) + "/**";
18         routesMap.put(path, new ZuulRoute(path, serviceID));
19     };
20
21     discovery.getServiceInstances(filter).stream().forEach(consume);
22
23     return routesMap;
24 }
```

With the `@EnableZuulProxy` annotation from Section 2.9, a `RibbonRoutingFilter` is activated. This filter routes the requests to the correct service using Ribbon. We need to configure Ribbon to use the Kubernetes service discovery. In the configuration file (Listing 5.2) we set the `ribbon.NIWSServerListClassName` property to use the `KubernetesServerList` (line 9). To find our store services, we also need to add the Kubernetes namespace where to search (line 10). In this case it is the `default` namespace.

The store services depend on authorization headers. By default, Zuul does not route a predefined set of headers. These are `Cookie`, `Set-Cookie`, and `Authorization`. In order to allow Zuul to send the authorization header, we make the list of `sensitiveHeaders` empty (Listing 5.2, line 3).

On the test infrastructure there are problems with the connection to the JupyterHub store backend service. The first calls, after the gateway is started, return the error "Hystrix Readed time out". The problem is that the Ribbon client first needs to be initialized for the first call and, therefore, needs too much time for the request. Ribbon requests are embedded in a Hystrix[3] command. Hystrix is a framework from Netflix that implements the circuit breaker pattern. The Hystrix timeout is dependent of the `ReadTimeout` and `ConnectTimeout` from Ribbon. The formula for calculating the Hystrix timeout is the following:

$$(ribbon.ConnectTimeout + ribbon.ReadTimeout) * (ribbon.MaxAutoRetries + 1) \\ *(ribbon.MaxAutoRetriesNextServer + 1) = HystrixTimeout$$

For the first call this was not enough with the default values. We set higher timeout values for the ribbon client in lines 11 and 12. It is also possible to eager load the Ribbon client (line 6) which means it is loaded at the application startup.

5.5 Zuul with Eureka

To use Eureka as service discovery in Zuul, we add the `spring-cloud-starter-netflix-eureka-client` dependency to our Zuul application. Afterwards, we add the `@EnableDiscoveryClient` annotation at the Spring Boot application. The configuration for Zuul is shown in Listing 5.4. The Eureka client needs to know the URL of the Eureka server. To do so we can use the Kubernetes service URL of our Eureka server (line 4). Furthermore, Zuul should not be registered as a service in Eureka (line 5). Since routes are automatically refreshed, we do not need to configure this.

Listing 5.4. Application YAML file for Zuul with Eureka

```
1 eureka:
2   client:
3     serviceUrl:
4       defaultZone: http://eureka-server.default.svc.cluster.local:8761/eureka/
5     register-with-eureka: false
```

The data for an application instance in the registry is shown in Listing 5.5. In order to achieve that Spring Cloud Zuul sends requests to the services, the `appName` (line 6) and `vipAddress` (line 23) need to be the same for an instance in the Eureka server. Zuul fetches the service list for creating the routes. The route is composed of a path and the `appName` value. Ribbon, the load balancer, uses the `vipName` to find the instances of the service. Instead of the `vipAddress` Ribbon gets the `appName` from the created route. Therefore, the `appName`

[3] https://github.com/Netflix/Hystrix

and vipAddress need to be the same to find service instances. In our own implementation of MyDataCenterInstanceConfig from Section 4.1.2 we can use the application name for the appName and vipAddress property.

Listing 5.5. Extract of an application instance info for the JupyterHub store from the Eureka REST API

```
 1 <application>
 2   <name>JHUB-STORE</name>
 3   <instance>
 4     <instanceId>10.1.2.107</instanceId>
 5     <hostName>10.1.2.107</hostName>
 6     <app>JHUB-STORE</app>
 7     <ipAddr>10.1.2.107</ipAddr>
 8     <status>UP</status>
 9     <overriddenstatus>UNKNOWN</overriddenstatus>
10     <port enabled="true">5678</port>
11     <securePort enabled="false">443</securePort>
12     <countryId>1</countryId>
13 ...
14     <metadata>
15       <type>store</type>
16       <name>Jupyter Store</name>
17       <serviceid>jhub</serviceid>
18     </metadata>
19     <appGroupName>UNKNOWN</appGroupName>
20     <homePageUrl>http://10.1.2.107:5678/</homePageUrl>
21     <statusPageUrl>http://10.1.2.107:5678/Status</statusPageUrl>
22     <healthCheckUrl>http://10.1.2.107:5678/healthcheck</healthCheckUrl>
23     <vipAddress>jhub-store</vipAddress>
24 ...
25   </instance>
26 </application>
```

Spring Cloud Zuul uses the hostName attribute (line 5) of the instance to send requests. The problem is that the Eureka client uses the name of the pod when registering. The name of the pod cannot be resolved to an IP address in Kubernetes and, as a result, the request does not reach the service. In our own implementation of MyDataCenterInstanceConfig class we use the IP address instead for the hostName attribute.

To only allow a special set of services to be routed, we can use the same approach that we use in Section 5.4. We ignore all services and subclass the DiscoveryClientRouteLocator. The store applications can add own metadata when registering with Eureka. We filter the services for the key `type` metadata (line 15) and add them as routes.

5.6 Spring Cloud Gateway with Eureka

Using Eureka with the SCG is similar to Section 5.5. We add the Eureka dependency and enable the discovery client on the Spring Boot application. With the SCG we have similar problems using Eureka as with the Spring Cloud Zuul in Section 5.5.

When the `appName` and the `vipAddress` of the application are different, no routes are found at all. The `DiscoveryClientRouteDefinitionLocator` is used for locating routes. First it fetches all available services. Then it uses the service names to get the instances. The service name is the `appName` from Eureka, but the `vipAddress` is used to find instances. SCG needs the IP address in the `hostName` as well.

SCG uses interfaces for the `DiscoveryClient` and the contained data structures. For this reason we can use the same configuration from Listing 5.1 for filtering the services and adapting the services to the correct URL.

Chapter 6

Frontend Integration

The frontend should only provide services that are available. New and removed services should either be added or removed automatically. The frontend is implemented with the Vue.js[1] framework. Vue.js is a JavaScript framework for building UIs on the web.

The available services for storing are hard-coded in the frontend. A drop-down list shows the available store options. The user can select one of these options and send a store request. Depending on the selection, the URL of the endpoint is composed. Then the request is sent to the endpoint.

Our solution is to load the available services on every page load. Vue.js has a `created` method that is called upon page load. This function enables us to load resources dynamically.

At the gateway we create a static endpoint that provides the available store services. This service should return a list of JSON objects that has a `value` and a `text` attribute. The `value` contains the relative path to reach the service on the gateway. The `text` is later used by the frontend as service name. This list can directly be used by Vue.js to display the options in the frontend.

We use the WebFlux framework from the gateway to create an endpoint. A request to `/services/store` is forwarded to the handler method shown in Listing 6.1. This method gets a `ServerRequest` and returns a `ServerResponse`. We define a filter to get only services with value `store` for the key `type` in their metadata (lines 2-4). Further we create a function that maps the metadata to a `Service` object (lines 7-18). We first query all services with the discovery client (line 21). Then we fetch the instances of the services (line 22), filter out services that have no instance (line 23) and select the first instances of the remaining services (line 24). Upon these service we apply our filter for the store services (line 25) and then apply our mapper (line 26). In the `ServerResponse` we return the list of available services (30).

The implementation is the same for SCG and Zuul. It can be used with Eureka and Kubernetes service discovery because the `DiscoveryClient` interface is used.

Listing 6.1. Handler for serving all available store services using a DiscoveryClient

```
1 public Mono<ServerResponse> storeServices(ServerRequest serverRequest) {
2   Predicate<ServiceInstance> filter = (ServiceInstance instance) -> {
```

[1] https://vuejs.org

```
    return "store".equals((instance.getMetadata().get("type")));
  };

  //Create map with path for value and service name as text
  Function<ServiceInstance, Service> mapper = (ServiceInstance instance) -> {
    Map<String, String> metadata = instance.getMetadata();

    //Create a new service object for the instance
    Service service = new Service();

    //Set properties from metadata
    service.setValue("/"+ metadata.get("type") + "/" + metadata.get("serviceid"));
    service.setText(metadata.getOrDefault("name", "Store␣Service"));

    return service;
  };

  Mono<ServiceList> serviceList =
    Flux.fromIterable(discoveryClient.getServices())
          .map(discoveryClient::getInstances)
          .filter(instances -> !instances.isEmpty())
          .map(instances -> instances.get(0))
          .filter(filter)
          .map(mapper)
          .collectList()
          .map(services -> new ServiceList(services));

  return ServerResponse.ok().body(serviceList, ServiceList.class);
}
```

The new drop-down list does not initially provide store services. In the `created` method we call the API gateway to request the available services (Listing 6.2, line 3). We add the response as options to the drop-down list (lines 4,5).

Listing 6.2. Method to load the available service dynamically in the frontend

```
1 created() {
2   const self = this
3   axios.get('/api/v1/gateway/services/store')
4     .then(function (response) {
5       self.options = self.options.concat(response.data);
6     })
7     .catch(function (error) {
    console.error(error)
  });
}
```

When an option is selected the value is used to compose the endpoint of the API gateway (Listing 6.3, lines 3-7). Thus, the request is sent to it (line 7).

Listing 6.3. Method to initialize a store

```
 1 store(){
 2   const self = this
 3   var subdomain = this.selected
 4   if (subdomain === null) {
 5     return
 6   }
 7   axios.post('/api/v1/gateway' + subdomain + '/', self.storeData)
 8     .then(function (response) {
 9       location.href = subdomain + '/files/' + response.data.sessionId
10     })
11     .catch(function (error) {
12       console.error(error)
13     });
14 }
```

Chapter 7

Evaluation

In this chapter we evaluate the different approaches. In Section 7.1 we do a benchmark test. We use a benchmarking tool to send requests to a service directly and through API gateways. Subsequently, the results of the test are evaluated.

In Section 7.2 we compare Eureka with Kubernetes service discovery and Zuul with SCG. We take a look at the provided functionalities and how they work.

7.1 Benchmark Test

7.1.1 Experimental Setup

For the benchmark test we use the setup that is shown in Figure 7.1. There is one client which sends requests. The access points, services, and a Eureka server are deployed in the Kubernetes cluster. Access points are either directly the Kubernetes service or the API gateways. The services are test services for the benchmarking. They are implemented with Spark as the web framework. The service replies on a HTTP request with a small HTML page (479 Byte). *Kubernetes Service* is the service defined in Kubernetes. It is exposed externally with a `NodePort` type. This provides the direct access to the service instances. Spring cloud Zuul and SCG are the gateways under test. They are used once with the Kubernetes service discovery and once with the Eureka service discovery. *Test Service* is the vanilla implementation of the service. *Test Service + Eureka* has an additional Eureka client implemented. We make this distinction because the Eureka client may add some overhead to the service. *Zuul + Eureka* and *Gateway + Eureka* rely on the Eureka service discovery and therefore use this service. *Test Service* and *Test Service + Eureka* both have two instances. Real applications often have multiple instances and this enforces the API Gateway to use the load balancing.

The client sends requests to the access points. *Zuul + Kubernetes* and *Gateway + Kubernetes* use Kubernetes for service discovery and then send the request to one of the *Test Service* instances.

Zuul + Eureka and *Gateway + Eureka* use Eureka as service discovery mechanism and again send the request to one of the *Test Service + Eureka Client* instances.

We set up the Kubernetes cluster from Figure 7.1 on a MacBook Pro early 2015 with a 2.7 GHz Intel Core i5 processor (5257U) and 16 GB RAM. Docker desktop community

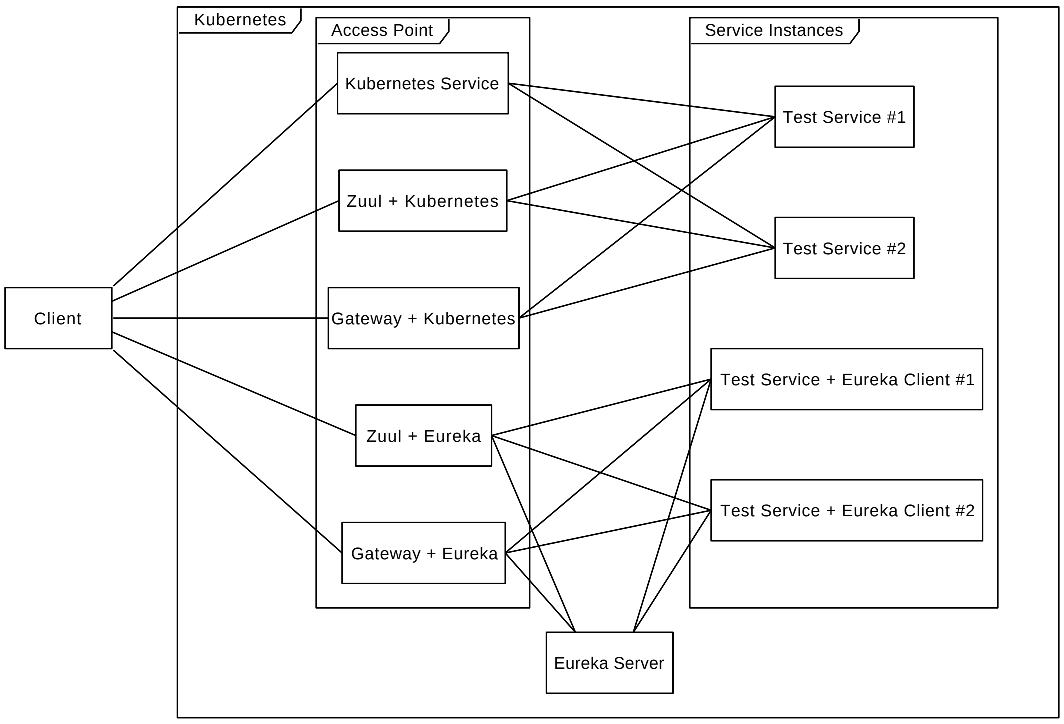

Figure 7.1. Setup for the Benchmark Test

version 2.1.0.2 (37877) with the integrated Kubernetes version v1.14.6 is installed on the machine.

We use the Apache HTTP server benchmarking tool (`ab`)[1] version 2.3 (1826891) for the benchmark test. To test an access point we execute

```
ab -n 10000 -c 100 <access point url>
```

on the command line. In total 10000 requests should be sent (`-n 10000`). The `-c` option defines that the requests should be sent with 100 concurrent threads. For better results we warm the machine up with executing the benchmarking multiple times before we use the benchmark values. Further, we do two consecutive runs for every gateway solution.

7.1.2 Results

The results of the execution are shown in Figure 7.2 and Figure 7.3. Most Requests per second (RPS) are sent with 610.11 RPS with the direct access. The highest value for a gateway

[1]`https://httpd.apache.org/docs/2.4/programs/ab.html`

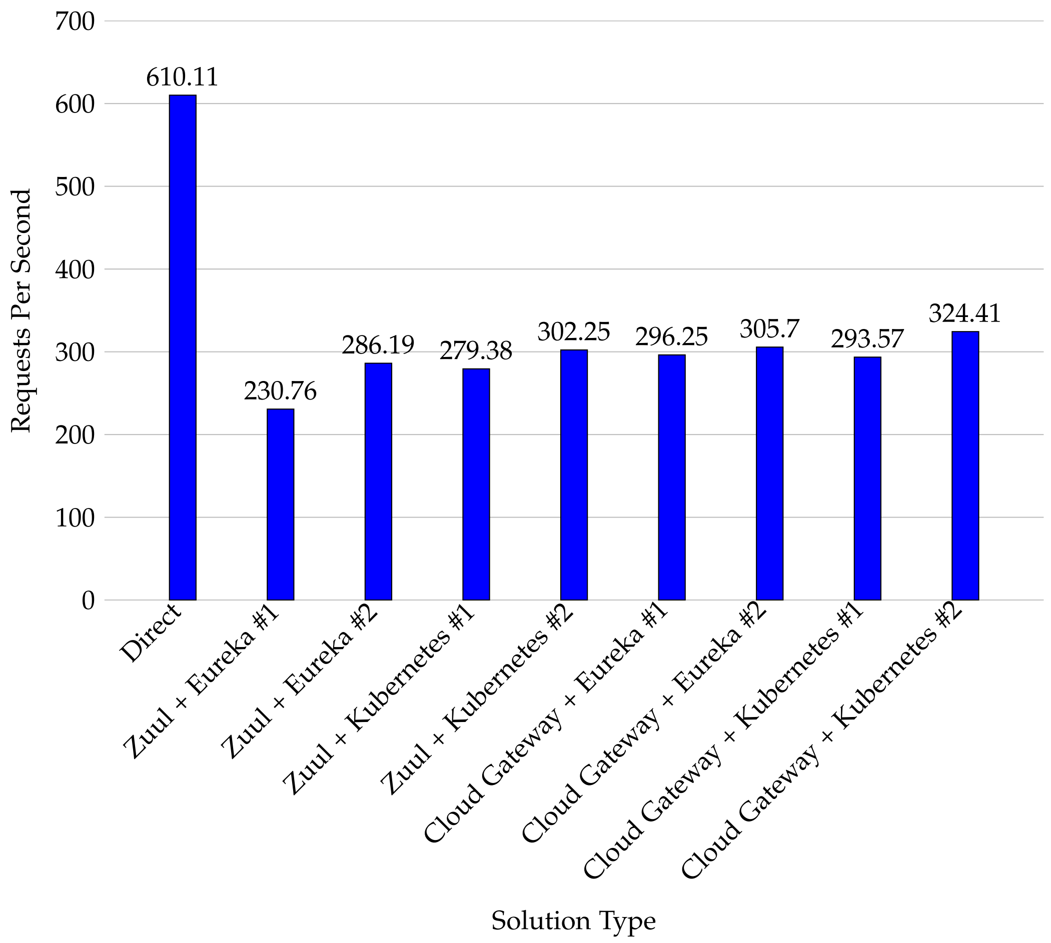

Figure 7.2. Comparison of the requests per second

solution is 324.41 RPS with SCG and Kubernetes. Zuul with Eureka has the lowest value with 230.76 RPS. So the maximum delta is 93.65 RPS between the gateway solutions.

The second run for every solution is always better then the first one with at least 9 RPS. The highest difference between two runs is with Zuul and Eureka. 55 more RPS are sent in the second run. Both gateways perform better in terms of more RPS with the Kubernetes service discovery than with Eureka.

Figure 7.3 details the Mean time per request (MTPR) for the different solutions. The results correlate to those in Figure 7.2. Direct access to the service is the fastest with 163.91 ms. The SCG with Kubernetes need the shortest time, i.e. 308.25 ms in mean per request. That is 144.34 ms longer and 1.88 times the time of the direct access. The slowest

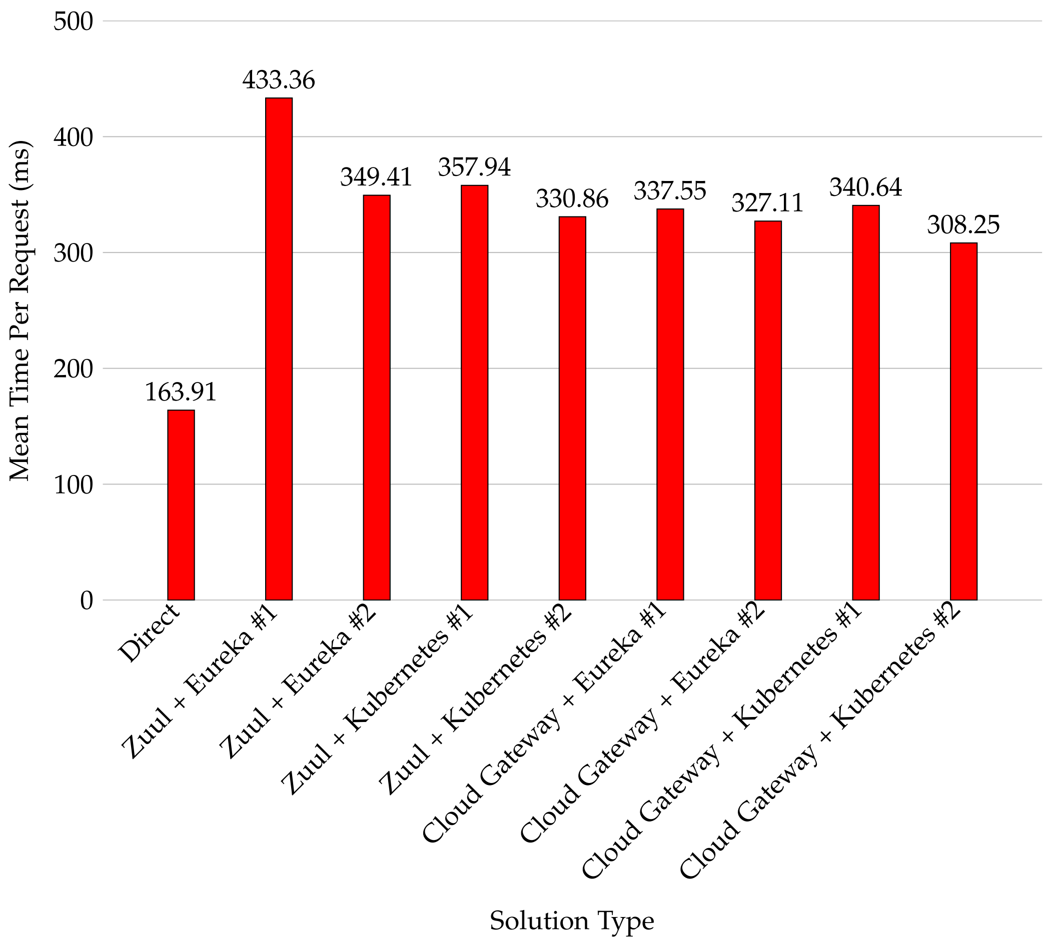

Figure 7.3. Comparison of the meantime per request

requests are performed by Zuul with Eureka and need 433.36 ms. That is 269.45 ms longer and 2.64 times the time of the direct access.

7.1.3 Discussion

The direct access is the fastest, because there are no extra hops for the request. It directly accesses the service in Kubernetes and Kubernetes does the load balancing. When using a gateway the request is sent to the gateway. The gateway needs to discover the service instances and then sends the request to one of them. On a response from the service, the gateway forwards it to the client. These are two extra hops introduced by the gateway.

Every hop adds latency to the request.

For the MTPR, the difference between the direct access is about twice as much as with the gateways. The computation of the service is not intensive. It returns a static HTML page. In comparison to the time that the service needs, the latency of the gateway could be a large share. Therefore, the impact would be smaller with services that do more computation and need more time for a response.

The consecutively second run of the benchmarking for every solution may be faster because of some optimizations made by the JVM.

Zuul is used with the Spring integration. The Spring infrastructure may add some extra overhead in comparison to a plain Zuul implementation.

In this setup, the SCG has a higher throughput and better performance than Spring Cloud Zuul.

7.1.4 Threats to Validity

The benchmarking is executed on a single machine. This machine could not replicate the real infrastructure of GeRDI. A benchmark on the existing Kubernetes cluster would deliver an accurate result for our demands.

The service used in this benchmark is created for the test. It does not represent the real store services. Using the real services with real requests would avoid this.

Requests are sent to the localhost domain and the localhost network is faster than the internet. Hence, there is less network latency. With longer network calls the impact of the gateway may be lower.

The gateways are designed to run million of requests and running on machines with higher operation facilities. To avoid this threat the benchmark should be executed on machines with different hardware configurations.

7.2 Comparison

7.2.1 Comparison of Eureka and Kubernetes Service Discovery

Eureka provides application-level service discovery and Kubernetes platform-provided service discovery. Eureka is independent from any deployment platform and works well when the applications are distributed on different platforms. When the applications are all deployed in Kubernetes the platform-provided discovery is favored. This applies for all infrastructures, where everything is deployed at one platform.

The Eureka service discovery mechanism needs a Eureka server which is an additional service in the infrastructure. The Eureka server always needs to be available for the service discovery. Further, services and clients need to implement the Eureka client. Therefore, existing applications need to be changed and new applications also need to implement the client. Besides, this is the force that enables the applications to be deployed on different platforms. Hence, they can be deployed in orchestration platforms like Kubernetes, in the cloud or standalone.

To enable the Kubernetes service discovery a service object needs to be defined. This can be defined alongside a deployment configuration. The upside is that applications do not need to be modified and there is no need for an extra service, because it is already integrated in the platform. Services outside of Kubernetes can be integrated in Kubernetes with `Endpoints` or with `ExternalName` objects. When moving an external service inside the cluster the applications do not need to be changed.

The downside is that everything needs to be deployed on the same platform. When moving the application to another platform, applications need to change because the service discovery may be different on the other platform.

The Eureka client is written in Java. Non Java applications can use the REST API of the Eureka server or run a side car with an embedded Eureka client for the service discovery.

The Kubernetes service discovery is primary with DNS. The applications know the service URLs, send requests to it and the platform does the service discovery. Kubernetes provides a REST API that can be used for client side discovery. It can be accessed either over HTTP, command line tools or client libraries. The client libraries are available in various programming languages.

All GeRDI applications are deployed in Kubernetes and for every application a service is defined. Therefore, everything is ready, to use the Kubernetes service discovery. With the Eureka service discovery we need to change the applications and need to add a Eureka server in the cluster. Further, we need to ensure that the IP address of the pod is used when sending a request, not the hostname of the pod.

Kubernetes is the most appropriate solution for GeRDI.

7.2.2 Comparison of Zuul and Spring Cloud Gateway

The functionality of Zuul is described in Section 2.9. Two versions exist for Zuul. These are Zuul 1 and Zuul 2. The Spring Cloud Netflix project provides Zuul as a Spring Boot integration. In Spring Cloud Netflix, Zuul 1 is used and Zuul 2 is not planned to be implemented.

In Section 5.2, the functionality of the SCG is described. The SCG is built using Spring 5, Spring Boot 2, Project Reactor, and Spring WebFlux.

Zuul 1 is based on the Servlet framework and ,therefore, is blocking. Zuul 2 uses Netty and SCG uses Reactor as well as WebFlux which are non-blocking. Blocking in Zuul 1 means there is one thread per request. When the thread executes an I/O operation the thread is blocked until the end of this operation. The non-blocking model is shown in Figure 7.4. In non-blocking systems like Zuul 2 and SCG, there is generally one thread per CPU for handling requests and responses. Requests are put in an event queue. An event loop handles the requests and calls the network asynchronously. In the meantime the event loop can handle other requests and responses. As the response comes back it is served to the client. On the one hand the costs for a connection in the non-blocking model is small in

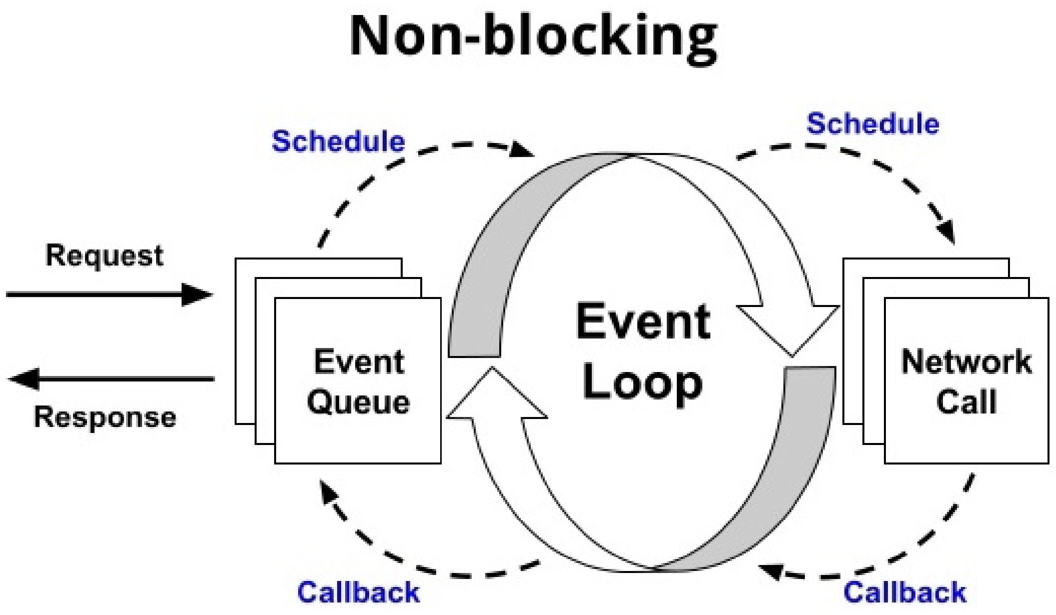

Figure 7.4. Non-blocking concept [Gibb and Tummidi 2017]

comparison to a new thread. Further, fewer context switches are necessary. On the other hand non-blocking systems are more complex to debug than blocking systems.

When the Zuul gateway is more CPU bound there is less efficiency gain with non-blocking. With a lower CPU bound, there is an efficiency gain visible with non-blocking in Zuul [Gibb and Tummidi 2017; Cohen et al. 2016].

The Spring Cloud project for Zuul defines some filters and the dynamic route locating is predefined. In Zuul without the Spring integration, predefined filters do not exist. Therefore, all required filters need to be implemented by ourselves.

SCG provides different predefined predicate and filter factories. The factories are used to create predicates and filters for routes. With the predicate factories predicates for the different HTTP attributes can be created. A few of the HTTP attributes are cookies, headers, host, method and path predicates. At the moment, 29 factories for filters exist. Examples for filters are the `Hystrix`, `AddRequestParameter`, `RewritePath`, and `RequestRateLimiter` filter. Apart from those, some global filters are defined [Spring 2019].

Zuul routes are matched against a path. So the path determines which service should be called. In SCG the HTTP attributes, including the path, are used to match a route. Thus, a more sophisticated routing can be achieved with SCG.

Implementing the service discovery in Spring Cloud Zuul or SCG needs the same effort. The `@EnableDiscoveryClient` annotation and their corresponding dependency needs to be added and some configuration need to be done.

SCG provides more functionalities out of the box than Zuul. The routing, load balancing, and discovery of services are preconfigured in SCG. With Zuul, at least a routing filter needs to be implemented. Thus, it is easier with SCG to get a running gateway.

Chapter 8

Related Work

Song et al. 2018 present an own API gateway solution and an auto scaling system for their API gateway. Benefits and a drawback of the gateway are presented. The intended purpose of an API Gateway is explained. They design and implement an API Gateway using Node.js[1] and Consul[2]. Consul is used for service discovery. Further, they design an auto scaling system for the gateway which utilizes Kubernetes and Prometheus[3]. The system adjusts the number of instances according to the workload.

Balalaie et al. 2016 discuss the migration toward a microservice architecture. They use Netflix Eureka for service discovery and Zuul as edge server. For the edge server and service discovery they describe why they used it. They abstract migration patterns and describe them in Balalaie et al. 2018.

Stubbs et al. 2015 review container technology and the challenge of service discovery in microservices. They introduce Serfnode as a solution to service discovery. Existing service discovery solutions are compared to Serfnode.

Abdollahi Vayghan et al. 2018 is about deploying microservices in Kubernetes and examine the availability with the default configuration of Kubernetes. Therefore, they present an architecture for deploying microservices with Kubernetes in a private cloud. They discuss a way to expose applications running in the private cloud using Kubernetes Ingress.

Lu et al. 2017 propose the use of microservices for Internet of Things (IoT) systems and vision an approach for them. Microservices should be used with complementary patterns, such as API gateway and service discovery. The patterns are explained generally and then how they could be used in the context of IoT systems.

[1]https://nodejs.org/en/

[2]https://www.consul.io

[3]https://prometheus.io

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In GeRDI the store services were configured manually in the frontend and backend. This configuration should be automated and the frontend should be able to provide the available store services dynamically.

To solve the problem we implemented service discovery with Eureka and used the integrated service discovery of Kubernetes. With Spring Cloud Zuul and Spring Cloud Gateway we implemented two different API gateways and used both with the different Service Discovery mechanisms.

The API gateways in combination with the service discovery mechanisms enable us to discover the available store services and to dynamically route to them. Available service endpoints are automatically added and removed in the gateways. In the frontend we use the possibility to discover the different services. We query the backend for these, when we load the page and then enable the different store options. On a store request, the request is sent to the gateway and routed to the correct service.

In GeRDI we use the Spring Cloud Gateway with the Kubernetes Service Discovery.

9.2 Future Work

Future work could extend the API gateway. The gateway could be used for all services in GeRDI which would streamline the external interface and reduce manual configurations.

Also, a load test and a benchmark should be done on the test system with the real services. This includes calling the services directly but also with different gateways and service discovery mechanisms. Based on the results, the gateway should be scaled and ,further, an evaluation should be done.

At the moment, the service for providing available services is part of the gateway. It could be extracted to an own service. This service may also provide services from other SCS.

In order to improve resilience, the circuit breaker pattern could be implemented in the gateway. SCG provides a filter factory to apply Hystrix as circuit breaker. It should be configured according to the needs of GeRDI.

Bibliography

[Abdollahi Vayghan et al. 2018] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, July 2018, pages 970–973. (Cited on page 41)

[Balalaie et al. 2016] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software* 33.3 (May 2016), pages 42–52. (Cited on page 41)

[Balalaie et al. 2018] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn. Microservices migration patterns. *Software: Practice and Experience* (July 2018). (Cited on page 41)

[Cerny et al. 2017] T. Cerny, M. J. Donahoo, and J. Pechanec. Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems. In: *Proceedings of the International Conference on Research in Adaptive and Convergent Systems - RACS '17*. ACM Press, Sept. 2017, pages 228–235. (Cited on page 6)

[Cohen et al. 2016] M. Cohen, M. Smith, S. Aroskar, A. Gonigberg, G. Varadarajan, and S. Vinukonda. *Zuul 2 : The Netflix Journey to Asynchronous, Non-Blocking Systems*. https://medium.com/netflix-techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c. (Accessed on 19.09.2019). Sept. 2016. (Cited on page 39)

[Eureka 2014] Eureka. *Netflix Eureka*. https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance. (Accessed on 16.09.2019). Dec. 2014. (Cited on pages 8, 9)

[Gibb and Tummidi 2017] S. Gibb and S. Tummidi. *Spring Cloud Gateway*. https://de.slideshare.net/Pivotal/spring-cloud-gateway. (Accessed on 19.09.2019). Dec. 2017. (Cited on page 39)

[Grunzke et al. 2017] R. Grunzke, T. Adolph, C. Biardzki, A. Bode, T. Borst, H.-J. Bungartz, A. Busch, A. Frank, C. Grimm, W. Hasselbring, et al. Challenges in creating a sustainable generic research data infrastructure. *Softwaretechnik-Trends* 37.2 (2017), pages 74–77. (Cited on page 6)

[Hasselbring 2016] W. Hasselbring. Microservices for Scalability. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*. ACM Press, 2016. (Cited on page 5)

[Hasselbring and Steinacker 2017] W. Hasselbring and G. Steinacker. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, Apr. 2017. (Cited on page 5)

[Jovanovic et al. 2017] Z. Jovanovic, D. Jagodic, D. Vujicic, and S. Ranđić. Java Spring Boot Rest WEB Service Integration with Java Artificial Intellgence Weka Framework. In: *UNITEH 2017, International Scientific Conference, Gabrovo, At Gabrovo, Bulgaria*. Nov. 2017. (Cited on page 11)

[Kubernetes 2019] Kubernetes. *Kubernetes Service*. `https://kubernetes.io/docs/concepts/services-networking/service/`. (Accessed on 16.09.2019). Aug. 2019. (Cited on page 10)

[Lu et al. 2017] D. Lu, D. Huang, A. Walenstein, and D. Medhi. A Secure Microservice Framework for IoT. In: *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. Apr. 2017, pages 9–18. (Cited on page 41)

[Luksa 2018] M. Luksa. *Kubernetes in Action*. Hanser Fachbuchverlag, July 9, 2018. 670 pages. (Cited on page 8)

[Namiot and Sneps-Sneppe 2014] D. Namiot and M. Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies* 2.9 (2014), pages 24–27. (Cited on page 5)

[Richardson 2019] C. Richardson. *Microservice Patterns*. Manning, May 1, 2019. 375 pages. (Cited on pages 5–10)

[Song et al. 2018] M. Song, C. Zhang, and E. Haihong. An Auto Scaling System for API Gateway Based on Kubernetes. In: *Proc. IEEE 9th Int. Conf. Software Engineering and Service Science (ICSESS)*. Nov. 2018, pages 109–112. (Cited on page 41)

[Spring 2019] Spring. *Spring Cloud Gateway*. `https://cloud.spring.io/spring-cloud-gateway/reference/html/`. (Accessed on 25.09.2019). Sept. 2019. (Cited on pages 21, 39)

[Stubbs et al. 2015] J. Stubbs, W. Moreira, and R. Dooley. Distributed Systems of Microservices Using Docker and Serfnode. In: *2015 7th International Workshop on Science Gateways*. IEEE, June 2015. (Cited on page 41)

[Tang et al. 2019] W. Tang, L. Wang, and G. Xue. Design of High Availability Service Discovery for Microservices Architecture. In: *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences - ICMSS 2019*. ACM Press, 2019. (Cited on page 6)

[Tavares de Sousa and Hasselbring 2018] N. Tavares de Sousa and W. Hasselbring. Skalierbare datenflussbasierte Architektur. *OBJEKTspektrum* 5 (2018), pages 28–33. (Cited on page 6)

[Tavares de Sousa et al. 2018] N. Tavares de Sousa, W. Hasselbring, T. Weber, and D. Kranzlmüller. Designing a Generic Research Data Infrastructure Architecture with Continuous Software Engineering. In: *Software Engineering Workshops 2018*. Volume Vol-2066. CEUR Workshop Proceedings. ARRAY(0x9f1e0bc), Mar. 2018, pages 85–88. URL: `http://oceanrep.geomar.de/42215/`. (Cited on page 6)

[Zuul 2013] Zuul. *Netflix Zuul - How it works*. `https://github.com/Netflix/zuul/wiki/How-it-Works`. (Accessed on 16.09.2019). Jan. 2013. (Cited on page 10)

[Zuul 2018] Zuul. *Netflix Zuul*. `https://github.com/Netflix/zuul/wiki`. (Accessed on 16.09.2019). Mar. 2018. (Cited on page 10)