

Application Design

Austrian Tourism Blogger / ReisePortal



September 23, 2021

FH Technikum

Chahed Rajoub (if19b166)

Stefan Tirea (if19b207)

Table of Contents

[Introduction: 2](#_Toc90548391)

[GitHub repository: 2](#_Toc90548392)

[Implementations: 2](#_Toc90548393)

[Use Case Diagram: 3](#_Toc90548394)

[UML Diagram: 4](#_Toc90548395)

[Database Design: 5](#_Toc90548396)

[Backend: 6](#_Toc90548397)

[Discovery: 6](#_Toc90548398)

[Config Server: 7](#_Toc90548399)

[Gateway 7](#_Toc90548400)

[Services: 8](#_Toc90548401)

[Zipkin Server: 9](#_Toc90548402)

[Frontend: 10](#_Toc90548403)

[API Contract/Specifications: 11](#_Toc90548404)

# Introduction:

* This travel portal is used to stimulate Austrian tourism blogger; authors can publish their sensational blog posts on given Austrian sights. These articles are designed to get the reader interested in having a vacation in the Alpine republic.
* 1 cent is to be paid out per article view. The commission is paid to the authors monthly.
* Data is collected statistically. readers can see where blogs are particularly popular right now.
* It is implemented in the form of a microservice architecture
* UML diagram is developed based on this use case.

## GitHub repository:

<https://github.com/Shahed-RAJOUB/ReiseProtal>

# Implementations:

* Java
* Maven
* Java Persistence API
* Spring Boot
* Spring Cloud Config
* Eureka
* Spring Cloud Load Balancer
* Spring Cloud Gateway
* Kafka
* PostgreSQL
* Zipkin server
* Docker
* Angular

# Use Case Diagram:

Diagram

Description automatically generated

# UML Diagram:

Diagram

Description automatically generated

# Database Design:

Diagram

Description automatically generated

* Three different tables: Author, Location, Blogs
* Each Blog has two foreign keys one for an author and other for location to simplify the design
* Docker container for common Postgres Database is used for this design and services are connected to the same container
* Flyway keeps track of our schema and implementation to make working in a team easier
* All the configurations are saved in docker-compose.yml outside the projects and it will go online and connect just by clicking run
* SQL script es added to initialize the container with few examples

# Backend:

To run Backend the sequence is:

Discovery ->Config-server -> Gateway -> Services -> Zipkin Server

## Discovery:

The following technologies are used in this project:

* spring-cloud-starter-netflix-eureka-server
* spring-boot-starter-test
* spring-cloud-dependencies
* maven

first step was to enable Eureka Server in a @SpringBootApplication by annotating it with @EnableEurekaServer. Running the application will allow the discovery layer to be displayed in <http://localhost:8761>. All services will be added in dashboard at different ports:

A screenshot of a computer

Description automatically generated

Services , Gateway and Config must have eureka Client to be able to connect and we need to annotate a @Configuration with @EnableEurekaClient

## Config Server:

* spring-cloud-config-server
* spring-cloud-starter-netflix-eureka-client
* spring-boot-starter-test
* spring-cloud-dependencies
* maven

This configuration store is ideally versioned under Git version control and can be modified at application runtime. Bootstrap yml file is added to point at it

Public project: <https://github.com/Shahed-RAJOUB/config-sever.git>

It serves configuration in YAML or Properties format instead of JSON – also with placeholders resolved. Which can be useful, when using it in non-Spring environments, where the configuration is not directly mapped to a Property Source.

Gateway:

* spring-cloud-starter-gateway
* spring-cloud-starter-netflix-eureka-client
* spring-cloud-starter-bootstrap
* spring-boot-starter-test
* spring-cloud-dependencies
* maven

The tool provides out-of-the-box routing mechanisms often used in microservices applications as a way of hiding multiple services behind a single facade. The Spring Cloud Gateway forwards requests to a Gateway Handler Mapping – which determines what should be done with requests matching a specific route.

This means every request will call Gateway port 9090 and our Gateway will distribute the calls between services:

Text

Description automatically generated

## Services:

Each service has the following dependencies:

* spring-boot-starter-data-jpa
* postgresql
* spring-cloud-starter-netflix-eureka-client
* spring-cloud-starter-bootstrap
* spring-cloud-sleuth-zipkin
* spring-cloud-starter-loadbalancer
* resilience4j-spring-boot2
* resilience4j-circuitbreaker
* spring-boot-starter-web
* spring-cloud-starter-openfeign
* spring-boot-devtools
* flyway-core
* maven

🡪data JPA, Flyway, postgresql are added to manage our Database connection and Schema. Ports are configured in application yml to ensure the connection and other options.

🡪To fill our client application with some sample-life, we'll also include the spring-boot-starter-web package in the pom.xml and implement a REST controller.

🡪 Feign makes writing web service clients easier with pluggable annotation support, which includes Feign annotations and JAX-RS annotations. This means easier communication with other services

🡪 Client – LoadBalancerFeignClient or default Feign client

This means the traffic will be managed through this technology and if a service is down, it will be present in the exception:

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence🡪After calling the server more than 5 times , circuit breaker will break the calls and will not allow further calls.:

## Zipkin Server:

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.

By opening CMD and tipping: *java -jar zipkin-server-2.23.4-exec.jar*

It will start tracking every single call like the photos:

A screenshot of a computer

Description automatically generated

Graphical user interface, application

Description automatically generated

# Frontend:

It is built using Angular, Bulma, with a proxy configuration to call our Gate way, in packages.json:

{

"start": "ng serve --proxy-config proxy.config.json",

},

Three pages are displayed, and we can navigate through the navbar easily:

Graphical user interface, website

Description automatically generated

Author view is a form where he can add all the details to insert them:

A picture containing text

Description automatically generated

User view is a list of the blogs, and the user can click on the blog to see it. Each fetch will increment the views of the blog:

Graphical user interface

Description automatically generated Text

Description automatically generated

Statistics page that shows popular location, popular blog, number of views, Authors number to give an insight in the data of this application:

Text

Description automatically generated

# API Contract/Specifications:

Postman collection documentation: <https://documenter.getpostman.com/view/12261877/UVR4Pqdm>