API gateway in Spring boot

Ankitha Gowda · Follow
7 min read · Oct 7, 2023

Photo by Gabriele Stravinskaite on Unsplash

APIs are a common way of communication between applications. In the case of microservice architecture, there will be a number of services and the client has to know the hostnames of all underlying applications to invoke them.

To simplify this communication, we prefer a component between client and server to manage all API requests called API Gateway. Additionally, we can have other features which include:

- **Security** — Authentication, authorization
- **Routing** — routing, request/response manipulation, circuit breaker
- **Observability** — metric aggregation, logging, tracing

Architectural benefits of API Gateway:

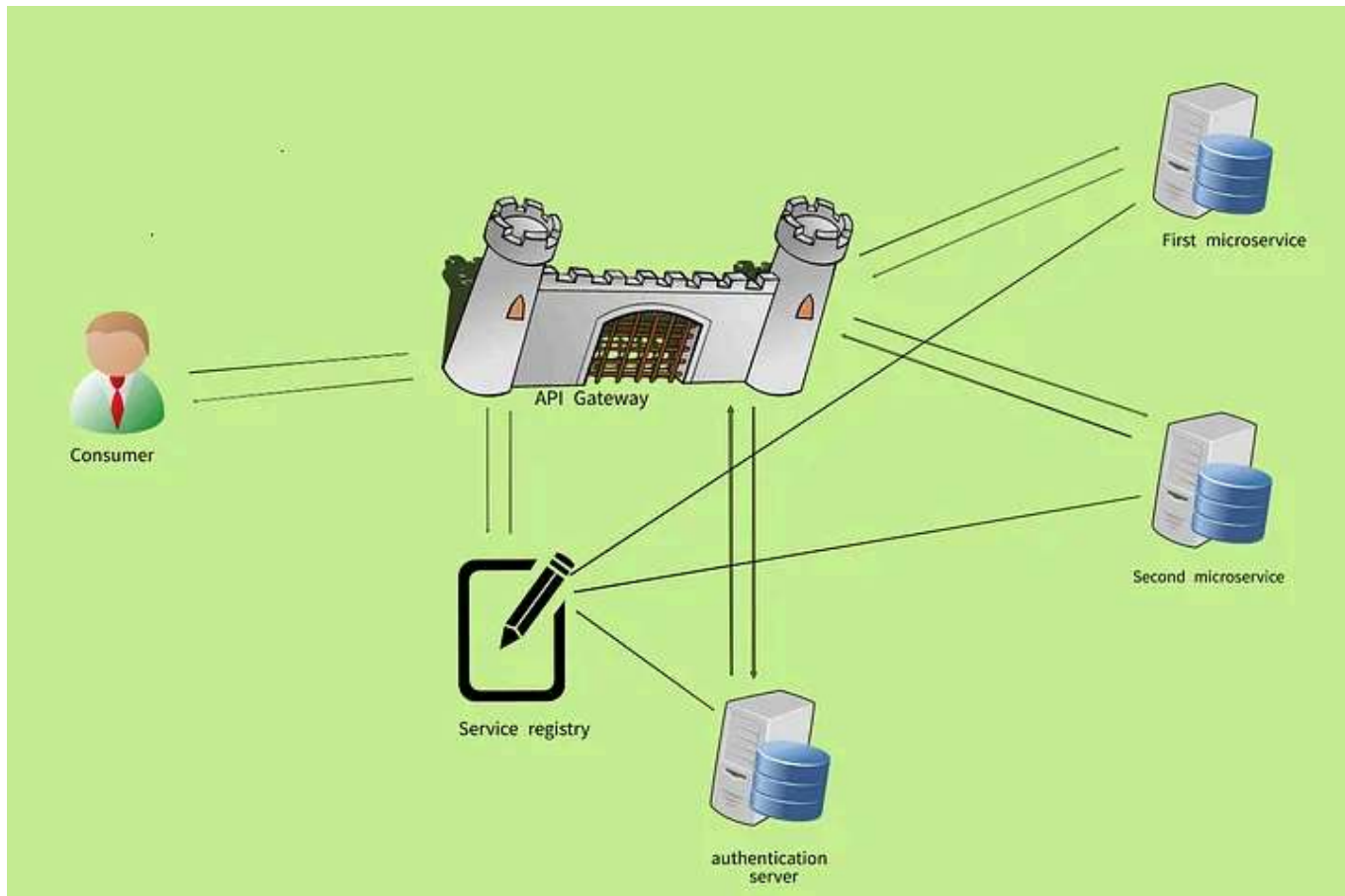
- Reduced complexity
- Centralized control of policies
- Simplified troubleshooting

There are many types of implementations available for API Gateway which include — Spring Cloud Gateway, Zuul API Gateway, APIGee, EAG (Enterprise API Gateway)

In this article, we will see how to implement the Spring Cloud API gateway, filter incoming requests, manipulate requests/responses, and handle authentication.

. . .

We can visualize the whole ecosystem as below:



Dataflow

In the above diagram, we have 5 services

- **Service Registry** — The application that keeps track of the available instances of each microservice in a project.
- **API Gateway** — receives incoming requests, performs authentication (if enabled) and forwards requests to actual microservice. On getting the response, return it to the consumer.
- **Authentication server** — The application that takes care of authentication
- **First and Second Microservices** — Two normal internal applications with different functionalities.

All the applications on startup register themselves into `Service Registry`. Below are the steps that occur upon receiving any API request:

1. The consumer calls any application via the API gateway.

2. API gateway will check if the incoming URL needs authentication. If yes, it calls the Authentication server to validate it.
3. If it is a valid token, it forwards the requests to the corresponding application after applying filters.
4. If it is an invalid token, respond to the consumer as unauthorized.
5. Once it receives a response from an internal microservice, it returns it back to the consumer after applying filters.

The filters in Gateway can include operations like logging or manipulating/customizing the request/response details.

> Note: *Here, the individual services will be residing in the intranet and we will be exposing only the APIGateway service for consumers.*

. . .

Service registry

An application that serves as an Eureka server. This will have the below dependency in `pom.xml`

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

And `@EnableEurekaServer` annotation in the main class.

By default, the Eureka server will register itself into discovery. We need to disable it by including the below properties in `application.properties`

```
eureka.client.registerWithEureka = false
eureka.client.fetchRegistry = false
```

Make other applications as Eureka clients by including the below dependency in `pom.xml`:

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  </dependency>
```

and, provide the Eureka server URL in `application.properties`:

```
eureka.client.serviceUrl.defaultZone= <host-and-port-where-eureka-server-running
```

API Gateway:

For the Spring Cloud API gateway, we need the below dependencies in `pom.xml`

```
<dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-gateway</artifactId>
  </dependency>
</dependencies>
 <dependencyManagement>
  <dependencies>
   <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-dependencies</artifactId>
    <version>${spring-cloud.version}</version>
    <type>pom</type>
    <scope>import</scope>
   </dependency>
  </dependencies>
 </dependencyManagement>
```

In `application.yml` file, provide details of all internal microservice names, paths, and uri as below:

```
spring:
  application:
    name: gateway-service
  cloud:
    gateway:
      routes:
        - id: first
          predicates:
            - Path=/first/
          uri: localhost:8081
        - id: second
          predicates:
            - Path=/second/
          uri: localhost:8082
        - id : auth-server
          predicates:
            - Path=/login/
          uri: localhost:8088
```

And we need to have a bean of `RouteLocator` type to provide all the routes the Gateway serves. We can include any filter if we want to process the request/response:

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
            // adding 2 rotes to first microservice as we need to log reques
    return builder.routes()
            .route("first-microservice",r -> r.path("/first")
                    .and().method("POST")
                    .and().readBody(Student.class, s -> true).filters(f -> f
                    .uri("http://localhost:8081"))
            .route("first-microservice",r -> r.path("/first")
                    .and().method("GET").filters(f-> f.filters(authFilter))
                    .uri("http://localhost:8081"))
            .route("second-microservice",r -> r.path("/second")
                    .and().method("POST")
                    .and().readBody(Company.class, s -> true).filters(f -> f
                    .uri("http://localhost:8082"))
            .route("second-microservice",r -> r.path("/second")
                    .and().method("GET").filters(f-> f.filters(authFilter))
                    .uri("http://localhost:8082"))
            .route("auth-server",r -> r.path("/login")
                    .uri("http://localhost:8088"))
            .build();
}
```

Filters:

A. *To log request body:*

To read the body, in ResourceLocator bean we need to make readBody() as `true`. This makes, the ServerWebExchange object cache the request body in attribute — “cachedRequestBodyObject”.

```
package com.example.springcloudgatewayoverview.filter;

import com.example.springcloudgatewayoverview.model.Company;
import com.example.springcloudgatewayoverview.model.Student;
import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component
public class RequestFilter implements GatewayFilter {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chai
        Object body = exchange.getAttribute("cachedRequestBodyObject");
        System.out.println("in request filter");
        if(body instanceof Student) {
            System.out.println("body:" + (Student) body);
        }
        else if(body instanceof Company) {
            System.out.println("body:" + (Company) body);
        }
        return chain.filter(exchange);
    }
}
```

B. *To log response body:*

```
package com.example.springcloudgatewayoverview.filter;

import com.example.springcloudgatewayoverview.model.Company;
import com.example.springcloudgatewayoverview.model.Student;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.reactivestreams.Publisher;
import org.springframework.core.io.buffer.DataBuffer;
import org.springframework.core.io.buffer.DataBufferFactory;
import org.springframework.core.io.buffer.DefaultDataBuffer;
import org.springframework.core.io.buffer.DefaultDataBufferFactory;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.http.server.reactive.ServerHttpResponse;
import org.springframework.http.server.reactive.ServerHttpResponseDecorator;
import org.springframework.web.server.ServerWebExchange;
import org.springframework.web.server.WebFilter;
import org.springframework.web.server.WebFilterChain;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.nio.charset.StandardCharsets;
import java.util.List;

public class PostGlobalFilter implements WebFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {
        String path = exchange.getRequest().getPath().toString();
        ServerHttpResponse response = exchange.getResponse();
        ServerHttpRequest request = exchange.getRequest();
        DataBufferFactory dataBufferFactory = response.bufferFactory();
        ServerHttpResponseDecorator decoratedResponse = getDecoratedResponse(pat
        return chain.filter(exchange.mutate().response(decoratedResponse).build(
    }

    private ServerHttpResponseDecorator getDecoratedResponse(String path, Server
        return new ServerHttpResponseDecorator(response) {

            @Override
            public Mono<Void> writeWith(final Publisher<? extends DataBuffer> bo

                if (body instanceof Flux) {

                    Flux<? extends DataBuffer> fluxBody = (Flux<? extends DataBu

                    return super.writeWith(fluxBody.buffer().map(dataBuffers ->

                        DefaultDataBuffer joinedBuffers = new DefaultDataBufferF
                        byte[] content = new byte[joinedBuffers.readableByteCoun
                        joinedBuffers.read(content);
                        String responseBody = new String(content, StandardCharse
                        System.out.println("requestId: "+request.getId()+", meth
                        try {
                            if(request.getURI().getPath().equals("/first") && re
                                List<Student> student = new ObjectMapper().readV
                                System.out.println("student:" + student);
                            }
                            else if(request.getURI().getPath().equals("/second")
                                List<Company> companies = new ObjectMapper().rea
                                System.out.println("companies:" + companies);
                            }
                        } catch (JsonProcessingException e) {
                            throw new RuntimeException(e);
                        }
                        return dataBufferFactory.wrap(responseBody.getBytes());
                    })).onErrorResume(err -> {

                        System.out.println("error while decorating Response: {}"
                        return Mono.empty();
                    });
```

```
                }
                return super.writeWith(body);
            }
        };
    }

}
```

C. To authenticate before API invocations:

Create authentication filter as below:

```
package com.example.springcloudgatewayoverview.filter;

import com.example.springcloudgatewayoverview.util.AuthUtil;
import com.example.springcloudgatewayoverview.util.JWTUtil;
import com.example.springcloudgatewayoverview.validator.RouteValidator;
import io.jsonwebtoken.Claims;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.http.HttpStatus;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.http.server.reactive.ServerHttpResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component
@RefreshScope
public class AuthFilter implements GatewayFilter {

    @Autowired
    RouteValidator routeValidator;

    @Autowired
    private JWTUtil jwtUtil;

    @Autowired
    private AuthUtil authUtil;

    @Value("${authentication.enabled}")
    private boolean authEnabled;

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chai
        if(!authEnabled) {
            System.out.println("Authentication is disabled. To enable it, make \
            return chain.filter(exchange);
        }
        String token ="";
        ServerHttpRequest request = exchange.getRequest();

        if(routeValidator.isSecured.test(request)) {
            System.out.println("validating authentication token");
            if(this.isCredsMissing(request)) {
                System.out.println("in error");
                return this.onError(exchange,"Credentials missing",HttpStatus.UN
            }
            if (request.getHeaders().containsKey("userName") && request.getHeade
                token = authUtil.getToken(request.getHeaders().get("userName").t
            }
            else {
                token = request.getHeaders().get("Authorization").toString().spl
            }

            if(jwtUtil.isInvalid(token)) {
```

```
                return this.onError(exchange,"Auth header invalid",HttpStatus.UN
            }
            else {
                System.out.println("Authentication is successful");
            }

            this.populateRequestWithHeaders(exchange,token);
        }
        return chain.filter(exchange);
    }

    private Mono<Void> onError(ServerWebExchange exchange, String err, HttpStatu
        ServerHttpResponse response = exchange.getResponse();
        response.setStatusCode(httpStatus);
        return response.setComplete();
    }

    private String getAuthHeader(ServerHttpRequest request) {
        return  request.getHeaders().getOrEmpty("Authorization").get(0);
    }


    private boolean isCredsMissing(ServerHttpRequest request) {
        return !(request.getHeaders().containsKey("userName") && request.getHead
    }

    private void populateRequestWithHeaders(ServerWebExchange exchange, String t
        Claims claims = jwtUtil.getALlClaims(token);
        exchange.getRequest()
                .mutate()
                .header("id",String.valueOf(claims.get("id")))
                .header("role", String.valueOf(claims.get("role")))
                .build();
    }
}
```

Some endpoints need to be unprotected i.e., to allow invocation without a token (e.g.: login URL, health check URL, etc.). We will add them to the below list :

```
package com.example.springcloudgatewayoverview.validator;

import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.stereotype.Component;

import java.util.List;
import java.util.function.Predicate;

@Component
public class RouteValidator {
    public static final List<String> unprotectedURLs = List.of("/login");

    public Predicate<ServerHttpRequest> isSecured = request -> unprotectedURLs.s
}
```

JWTUtil:

```
package com.example.springcloudgatewayoverview.util;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
```

```
import java.util.Date;

@Component
public class JWTUtil {

    @Value("${jwt.secret}")
    private String secret;

    public Claims getALlClaims(String token) {
        return Jwts.parserBuilder().setSigningKey(secret).build().parseClaimsJws
    }

    private boolean isTokenExpired(String token ) {
        return this.getALlClaims(token).getExpiration().before(new Date());
    }

    public boolean isInvalid(String token) {
        return this.isTokenExpired(token);
    }

}
```

Authentication server:

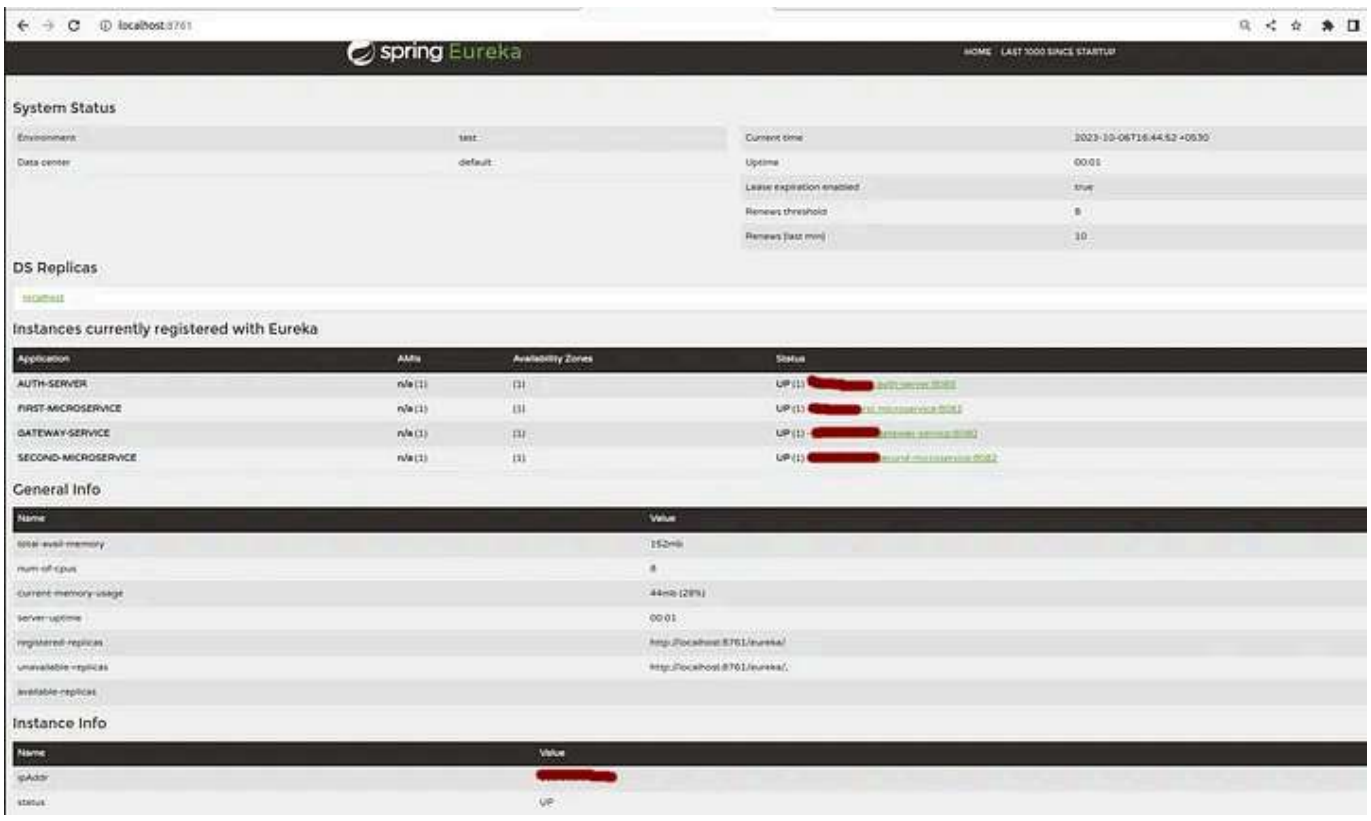
This service will provide you with the token for access to internal microservices.

> *Note: Authentication in itself is a huge topic. To make our example simpler, I have included a simple endpoint to return token. For more details on JWT authentication you can refer the article* ***JWT authentication and role-based authorization in Springboot*** *and the code here.*

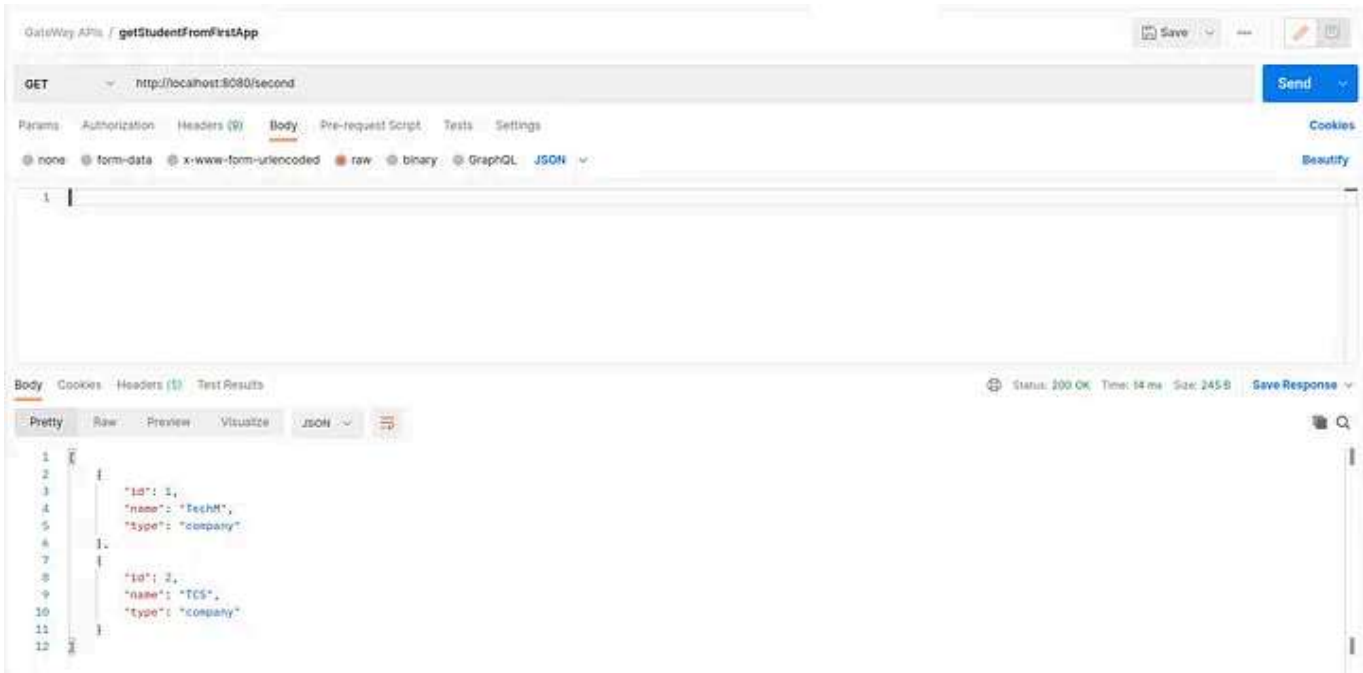
. . .

Execution:

Once all applications are running we can invoke the service registry using `http://localhost:8761` from browser. It will have all the details about currently running services:



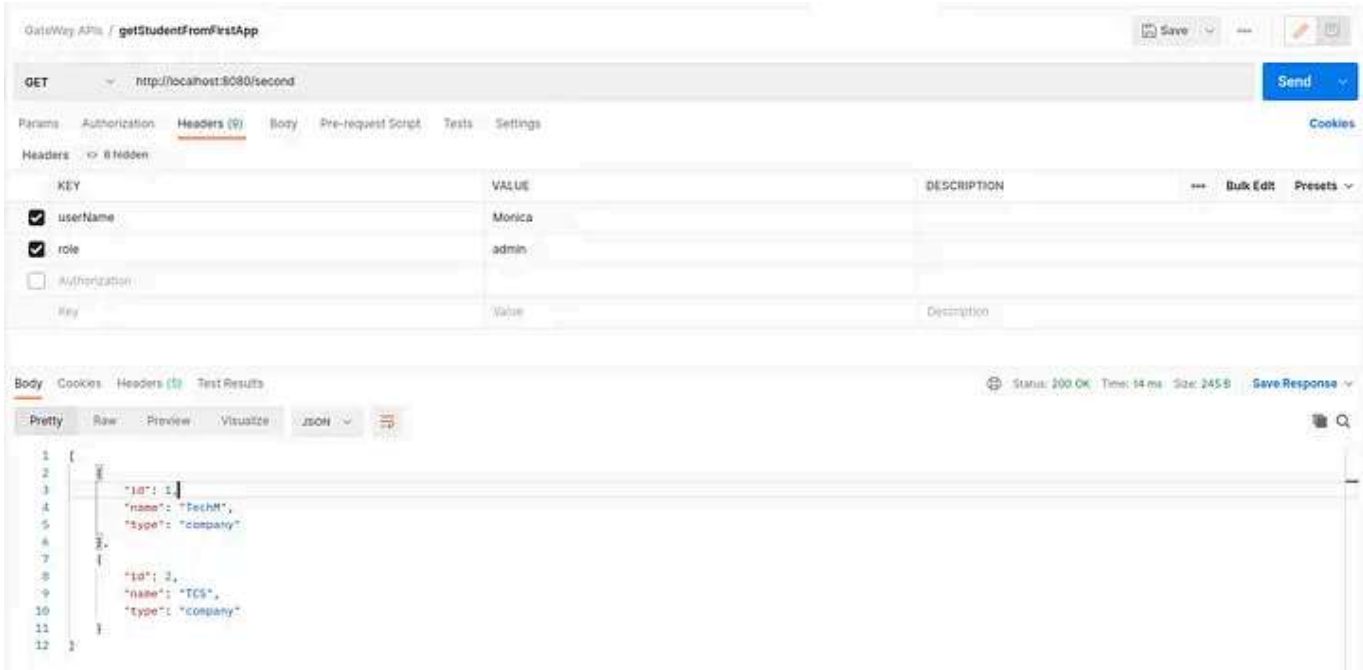
As the API Gateway is running on localhost at port 8080, you can invoke the endpoints of the First or Second microservice from `http://localhost:8080`



Request/Response logs in the APIGateway console:

> *Note: In this example, I have disabled authentication from API Gateway application. You can enable it from application.properties. If you enable it, provide "userName" and "role" in the request header.*

Sample request with authentication:



The completed codebase of all the services is available here.

Thank you for reading. Happy exploring!!!

Api Gateway | Java | Spring Cloud Gateway | Spring Boot | Microservices

Written by Ankitha Gowda

321 Followers · 26 Following

Curiosity and Discipline make miracles

Follow

Responses (4)

What are your thoughts?

Respond

Siwakhile Mpandza
about 1 year ago

Very clear guide, thank you.

2 1 reply Reply

Alex Pliutau
6 months ago

Loved this! We have a newsletter with similar content on Microservices, Backend, Distributed Systems and more, check it out if you're interested: https://packagemain.tech

1 1 reply Reply

Shally Jindal
6 months ago

good one

1 1 reply Reply

See all responses

More from Ankitha Gowda

Ankitha Gowda

Spring Retry Overview

In the software world, there is a chance that some part of the code might not work at a...

Nov 27, 2023

Ankitha Gowda

Log request and responses of REST APIs in SpringBoot

Today we will see a way to log requests and responses of APIs without inserting logger...

Apr 16, 2022

Ankitha Gowda

Convert .xsd files to Java classes

XSD (XML Schema Definition) files describe the structure of an XML document. It contai...

May 26, 2024

Ankitha Gowda

API Code generation using Swagger in Springboot

API allows two systems to talk to each other. We can perform required CRUD operations...

Jul 11, 2022

See all from Ankitha Gowda

Recommended from Medium

Ramesh Fadatare

Spring Boot Microservices API Gateway Example

In this tutorial, we'll create two Spring Boot microservices, an API Gateway, and an...

Sep 19, 2024 •••

Vishal

Microservices Architecture & Comparison: A Practical Guide...

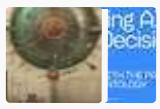
Microservices have evolved as a popular architectural style in the software...

Aug 26, 2024 •••

Lists

General Coding Knowledge

20 stories · 1852 saves



data science and AI

40 stories · 313 saves

Balian's techologies and innovation lab

API Gateway Patterns with Spring Cloud Gateway: Routing, Load...

In microservices architectures, an API Gateway acts as the single entry point for all...

Nov 12, 2024 •••

In DevOps.dev by Nithidol Vacharotayan

Spring Boot 3 API Gateway with Spring Cloud Netflix Eureka Server

Discover how to integrate Spring Boot 3 API Gateway with Spring Cloud Netflix Eureka...

Sep 6, 2024 •••

In Javarevisited by Dylan Smith

Mikhail Potter

Because I Didn't Know the Difference Between Exception an...

My articles are open to everyone; non-member readers can read the full article by...

Dec 9, 2024

Managing Spring Cloud Gateway Configurations at Scale

Spring Cloud Gateway offers a powerful way to manage routing rules through external...

Oct 12, 2024

See more recommendations