Microservices

Course Documentation

Houssemeddine Werhani
Hamza Ben Younes
Rayen Dhaouadi
Naim Atai
Aziz Sahli
Rania Chebbi

Contents

Part I

Introduction and Theory

Chapter 1: Introduction

Day by day, the number of internet users is increasing. Some websites are frequently accessed and receive a massive number of requests. This high load can lead to system crashes or server failures, especially if the system is misconfigured or not designed to scale effectively. Microservices offer an approach to address this issue. That's why, in this course, we are going to explore what microservices are, the benefits of using them, and how they solve the challenges of scalability and maintainability. By the end, we will go through a practical example to demonstrate how microservices architecture works in a real-world scenario.

> **Definition** Microservices — also known as the microservice architecture — is an architectural style that structures an application as a collection of two or more services that are:
>
> - Independently deployable
> - Loosely coupled
>
> Services are typically organized around business capabilities. Each service is often owned by a single, small team.[1]

Monolithic Architecture: The Predecessor to Microservices

Before delving into microservices in depth, we must first examine the monolithic approach they evolved from.

> **Definition** Monolithic architecture is a traditional software development model in which a single codebase executes multiple business functions. [2]

In a monolithic architecture, all components of an application—such as the user interface, business logic, and database access—are tightly integrated into a single, indivisible unit. While this approach is simple to develop, test, and deploy initially, it poses significant challenges as the application grows in complexity and user demand increases.

[1] `https://microservices.io/`
[2] `https://www.ibm.com/think/topics/monolithic-architecture`

Challenges of Monolithic Architecture

- **Scalability Issues**
  Scaling a monolithic application requires replicating the entire system, even if only one component faces high demand. This leads to inefficient resource utilization and higher costs.

- **Slow Development Cycles**
  Since all teams work on the same codebase, coordination becomes cumbersome. A small change in one module may require rebuilding and redeploying the entire application, slowing down development and deployment pipelines.

- **Limited Flexibility**
  Adopting new technologies or frameworks is difficult because the entire application must be refactored. This rigidity can hinder innovation and modernization efforts.

- **Single Point of Failure**
  If one component fails, the entire system can crash, leading to significant downtime and poor user experience.

How Microservices Solve These Challenges

Microservices address the limitations of monolithic architectures by decomposing applications into smaller, independent services. Here's how they tackle the problems mentioned earlier:

- **Improved Scalability**
  Services can be scaled independently based on specific needs, allowing for more efficient resource utilization. For instance, an e-commerce platform can scale its payment service during peak shopping seasons without scaling the entire application.

- **Enhanced Development Velocity**
  Development teams can work autonomously on different services, enabling parallel development and faster release cycles. This is particularly valuable for large organizations with multiple development teams.

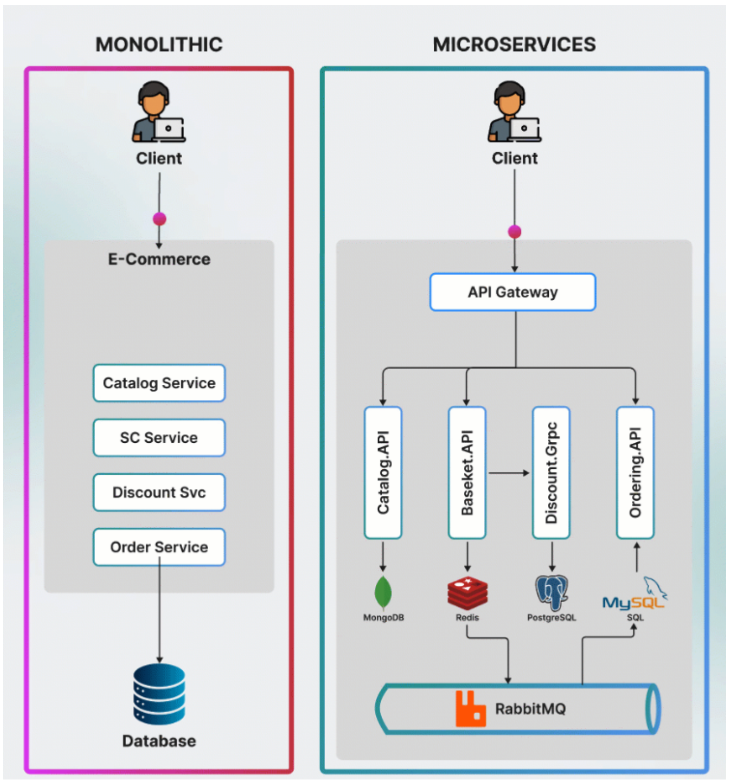
- **Technology Flexibility**
  Each microservice can use the most appropriate technology stack for its specific requirements. This allows organizations to leverage different programming languages, databases, and frameworks within a single application ecosystem.

- **Increased Resilience**
  The isolation of services prevents system-wide failures. If one service fails, others can continue functioning, often with graceful degradation rather than complete system outages.

- **Continuous Deployment**
  Individual services can be updated and deployed independently, reducing risk and enabling more frequent releases. This supports modern DevOps practices and CI/CD pipelines.

- **Organizational Alignment**
  The architecture naturally aligns with business capabilities, allowing teams to take ownership of specific domains. This often results in better alignment between technical and business units.

These benefits make microservices particularly suitable for complex, evolving applications that require high availability and rapid iteration.



Microservices Architecture Challenges

Despite its numerous advantages, microservices architecture introduces a range of challenges that must be carefully addressed to ensure successful implementation and operation.

- One of the foremost concerns is service communication complexity. Unlike monolithic systems where internal method calls suffice, microservices rely on interservice communication over networks, which introduces latency, potential message loss, and the need for reliable messaging protocols.

- Another significant challenge is data management and consistency. Microservices are typically designed with independent data stores, which complicates maintaining transactional integrity across services.

- Setting up and managing many services becomes more complex. You need strong tools and systems to help with automation, using containers like Docker, and organizing everything with tools like Kubernetes. These systems should also support CI/CD pipelines to keep things fast and flexible without breaking the system.

- Additionally, observability and monitoring are critical concerns. Traditional logging and monitoring tools are often insufficient in a distributed environment.
- Security also becomes more complex in microservices. Ensuring secure communication, authentication, and authorization across distributed services demands adherence to strict security protocols.
- Team coordination and governance can be difficult, particularly in large-scale systems where multiple teams manage different services.

Addressing these challenges is crucial to realizing the full potential of microservices while minimizing operational and developmental risks.

Conclusion

This chapter distinguished monolithic and microservices architectures, underscoring how microservices enhance scalability and maintainability through decentralized, loosely coupled components.

Chapter 2:

Microservices Architecture Core Components

Having established the fundamental advantages of microservices over monolithic architectures, we now turn to their structural foundations. This chapter examines the essential building blocks that enable microservices systems to achieve their promised benefits of scalability, flexibility, and resilience.

Microservices Examples

The microservices architectural pattern has been successfully adopted by numerous industry leaders to handle massive scale and rapid innovation. Below we examine prominent implementations:

- **Netflix**
  - Personalized recommendations
  - Video encoding/streaming
  - Payment processing
  - User analytics
- **Amazon** - Transitioned from monolith to microservices enabling:
  - Independent scaling of product catalog, cart, and checkout services
  - Continuous deployment
  - Polyglot persistence (DynamoDB, RDS, S3)

  This architectural shift was crucial in supporting Amazon's growth from online bookstore to cloud computing leader, with AWS itself being built as a collection of microservices.

- **Uber**
  - Dispatch service (matching drivers/riders)
  - Dynamic pricing engine
  - Trip tracking
  - Payment processing
- **Spotify**
  - Music recommendation
  - Playlist management
  - Social features
  - Ad serving

Core Components

To implement microservices effectively, a system must be underpinned by several architectural elements. These components are not optional add-ons but intrinsic necessities to achieve the promised benefits, microservices architectures rely on six fundamental components that work in concert. We will examine each through both theoretical and practical lenses:

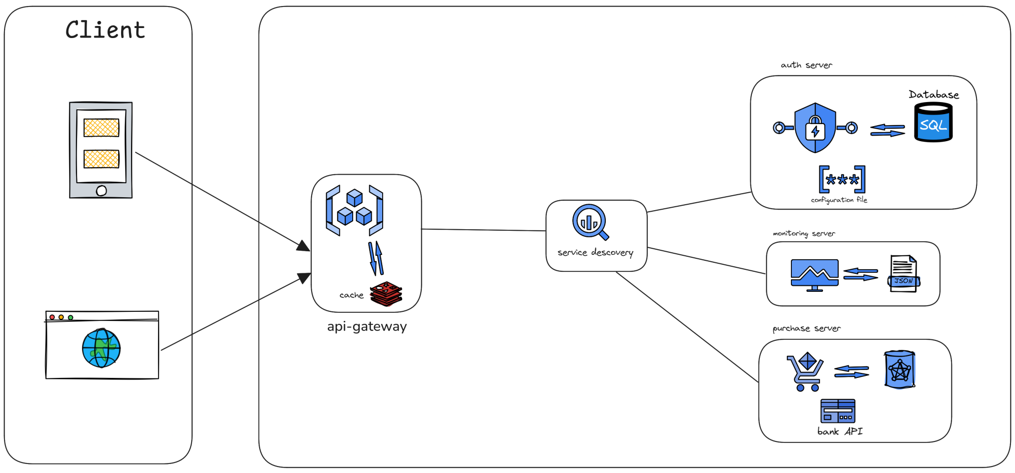


Figure 2.1: Interaction of Microservices Core Components

1. **Individual Microservices**
   - *Small, autonomous services* - Typically own a single business capability (e.g., user authentication, payment processing), allowing teams to develop, deploy and scale each service independently.
   - *Single responsibility principle implementation* - Each microservice focuses on one specific function, following Unix philosophy of "doing one thing well" to minimize complexity.
   - *Independently deployable units* - Can be updated without coordinating with other teams, enabling continuous delivery pipelines.

2. **API Gateway**
   - *Unified entry point for clients* - Acts as a reverse proxy that routes requests to appropriate services while providing a consistent interface to consumers.
   - *Handles routing, composition, and protocol translation* - Transforms between external REST/HTTP and internal protocols (like gRPC), and aggregates responses from multiple services.
   - *Implements security (OAuth2, JWT validation)* - Centralizes authentication/authorization, rate limiting, and DDoS protection at the edge of the system.
3. **Service Discovery**
   - *Dynamic registry of service instances* - Automatically tracks live service instances and their locations as they scale up/down or fail.
   - *Health check monitoring* - Continuously verifies service availability using heartbeat mechanisms (e.g., 30-second pings).
   - *Tools: Eureka* - Netflix's discovery service handles 100,000+ instance registrations with sub-second latency.[1]
4. **Inter-Service Communication**
   - *REST (synchronous) and messaging (asynchronous)* - RESTful APIs for request/response patterns, while message queues (e.g., Kafka) handle event-driven workflows.
   - *Protocol choices: HTTP, gRPC, AMQP, WebSockets, etc.*
5. **Database per Service**
   - *Each service owns its data schema* - Services can't directly access others' databases, ensuring loose coupling (e.g., Order Service has exclusive access to orders table).
   - *Database choices: (SQL + NoSQL)*
6. **Configuration Management**
   - *Externalized configuration* - Stores credentials, feature flags, and environment variables outside code (versioned in Git or secure vaults).

As visualized in Figure 2.1, these components create a distributed system that balances autonomy with coordination. The numbered sequence reflects the architectural decision flow when designing microservices solutions.

Note: API Gateway Overhead

While the API Gateway simplifies client interaction, it can become a performance bottleneck or single point of failure if not properly scaled or replicated. Load balancing and high availability configurations are essential to mitigate this risk.

[1]`https://github.com/Netflix/eureka/issues/834#issuecomment-244278197`

Conclusion

Having examined the core components of microservices through real-world examples, we now shift focus to the realization phase—where theory meets practice. The next part explores implementation strategies, and the challenges for building a microservices systems.

Summary of Microservices

In this chapter, we have explored the foundational concepts of microservices, examining their key components, benefits, and challenges.

- **Microservices** break down applications into independent, small services for scalability and flexibility.
- **Benefits**: Scalable, resilient, agile, and technology-agnostic.
- **Drawbacks**: Increased complexity, data consistency issues, and communication latency.
- **Core Components**: Independent services, API Gateway, service discovery, inter-service communication, and database per service.
- **API Gateway** centralizes client requests and manages routing and security.
- **Service Discovery** enables dynamic locating of services in a distributed system.
- **Data Consistency** relies on eventual consistency for reliability.

Part II

Implementation and Project Building

Chapter 3:

Introduction: Project Vision & Roadmap

After exploring our introduction to microservices and examining the different core components, it's time to apply our knowledge in practice. In this section, we'll build a real-world project to solidify your understanding and give you hands-on experience with microservice architecture.

What We're Going to Build

In this part of the course, we'll develop an **e-commerce web application**. The system has two main actors:

- **Admin:** Can add, manage, and delete items for sale.
- **Client:** Can register, browse items, and make purchases.

Some parts of the project are pre-built to save time. We'll cover how to download and set them up in a dedicated section.

Tech Stack Overview

This project is built using multiple programming languages and frameworks, each chosen for specific roles in our microservice ecosystem:

- **JavaScript:** Used for the client-side application and a Node.js-based authentication service.
- **Flask (Python):** Handles the product catalog and purchase services.
- **Spring Boot (Java):** Powers the Eureka service registry, API Gateway, and the review service. All these components will be built from scratch during the course.

Project Roadmap

The project will be divided into the following phases:

1. Environment setup and dependency installation
2. Microservices implementation, module by module
3. Service integration and communication
4. Testing and debugging
5. Running and monitoring the final system

Prerequisites

Before starting the project, please ensure the following tools and platforms are installed on your system:

- Java 21
- Node.js 18
- Python 3
- IntelliJ IDEA or VS Code (feel free to use your preferred editor)
- Git (optional but recommended)

Environment Verification

To ensure you have the required environments installed, please run the following commands:

- `nodejs --version` or `node --version`
  This command displays the installed Node.js version. Our project uses **Node.js 18.19**.
- `python3 --version` or `python --version`
  This will show the Python version available on your system. We require **Python 3.12**.
- `java --version`
  This command displays the Java Development Kit (JDK) version. We recommend installing **JDK 21**.
- `git --version` *(optional)*
  This command shows the Git version, which is optional but may be useful for version control.

Getting Started

To begin working on the project, you need to obtain the starter source code from the provided repository. You have two options:

- **Option 1: Clone the repository using Git**
  - Ensure Git is installed on your system.
  - Open a terminal or command prompt.
  - Run the following command:

    ```
    git clone https://github.com/werhani-
        houssemeddine/microservices-course
    ```

  - Navigate to the project directory:

    ```
    cd microservices-course
    ```

- **Option 2: Download the ZIP archive**
  - Go to the repository URL in your browser.
  - Click on the `Code` button and select `Download ZIP`.
  - Once downloaded, extract the ZIP archive to your desired location.
  - Open a terminal or command prompt and navigate to the extracted folder.

The following diagram presents a simple directory structure of the microservices:

```
microservices-course/
├── auth-server/
│   ├── server.js/
│   └── package.json
├── product-server/
│   ├── app.py/
│   └── requirements.txt
└── client/
    ├── index.html/
    └── package.json
```

1. **Authentication Server Setup**
   - Navigate to the `auth-server` directory.
   - Install the required packages and run the server (default port is 3000):

     ```
     npm install
     npm start
     ```

2. **Client Setup**

   - Navigate to the `client` directory.
   - Install the necessary dependencies and run the client server (default port is 5173):

   ```
   npm install
   npm run dev
   ```

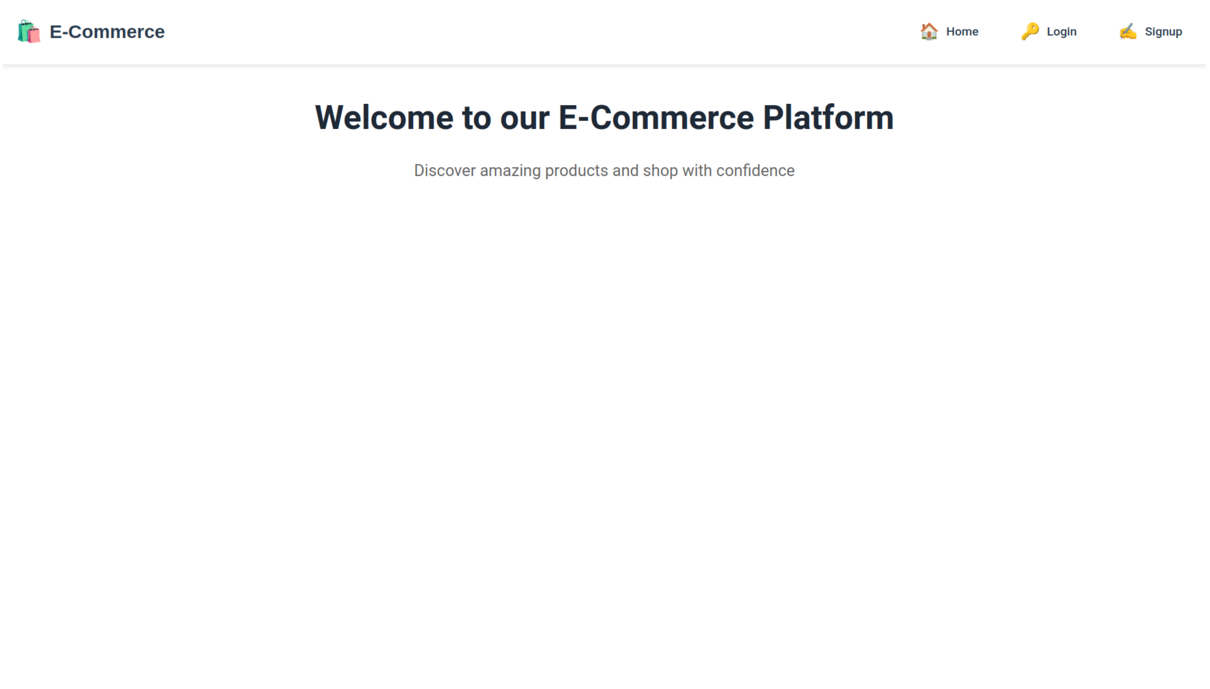
3. **Product Server Setup**

   - Navigate to the `product-server` directory.
   - Create a virtual environment, activate it, install the dependencies, and run the server (default port is 5000):

   ```
   # Create a virtual environment
   python -m venv venv  # or: python3 -m venv venv

   # Activate the virtual environment
   # On Linux/macOS:
   source venv/bin/activate
   # On Windows:
   venv\Scripts\activate

   # Install dependencies
   pip install -r requirements.txt

   # Run the server
   flask run
   ```

Accessing the Platform

- After successfully running all commands, open a web browser and navigate to the client URL
- You should see the following webpage:



Welcome to our E-Commerce Platform

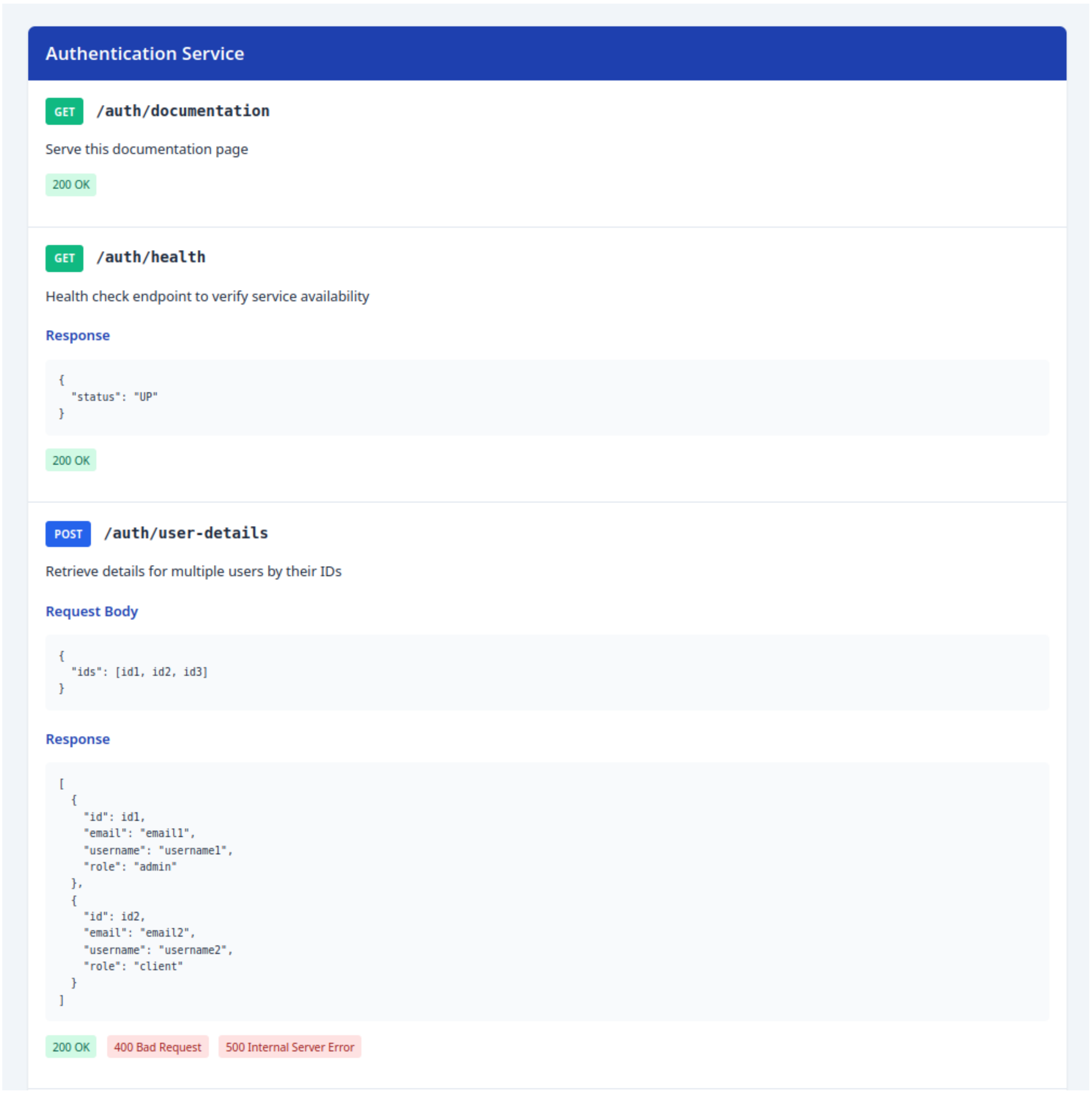
Discover amazing products and shop with confidence

Figure 3.1: Platform Home Page

API Documentation Access

- Each server provides its own API documentation. To access it, navigate to the corresponding URLs
  - Auth Service Documentation:
    `http://localhost:3000/auth/documentation`
  - Product Service Documentation:
    `http://localhost:5000/api/products/documentation`

  Note: Port `3000 and 5000` are the default ports. If you changed the port, replace it accordingly.


Authentication Service

GET /auth/documentation

Serve this documentation page

200 OK

GET /auth/health

Health check endpoint to verify service availability

Response

```
{
  "status": "UP"
}
```

200 OK

POST /auth/user-details

Retrieve details for multiple users by their IDs

Request Body

```
{
  "ids": [id1, id2, id3]
}
```

Response

```
[
  {
    "id": id1,
    "email": "email1",
    "username": "username1",
    "role": "admin"
  },
  {
    "id": id2,
    "email": "email2",
    "username": "username2",
    "role": "client"
  }
]
```

200 OK 400 Bad Request 500 Internal Server Error

Figure 3.2: API Server Documentation

Verification – Everything Is Fine

To verify that everything is working correctly, access the following pages. If you do not see any errors, then everything is fine and you can proceed to the next chapter. Otherwise, fix the problem first or ask one of our team members for help.

- Access Auth Service Documentation
- Access Product Service Documentation
- Access E-commerce Platform

Chapter 4:

Implementation

Introduction

After setting up everything, it's now time to start building the rest of the project. As mentioned earlier, we are going to build three services: the **API Gateway**, the **Discovery Service**, and the **Review Service**. All of these services will be developed using **Spring Boot**.

Setting Up Spring Boot Service Skeletons

Go to `https://start.spring.io/` and generate three services with the following details:

1. API Gateway

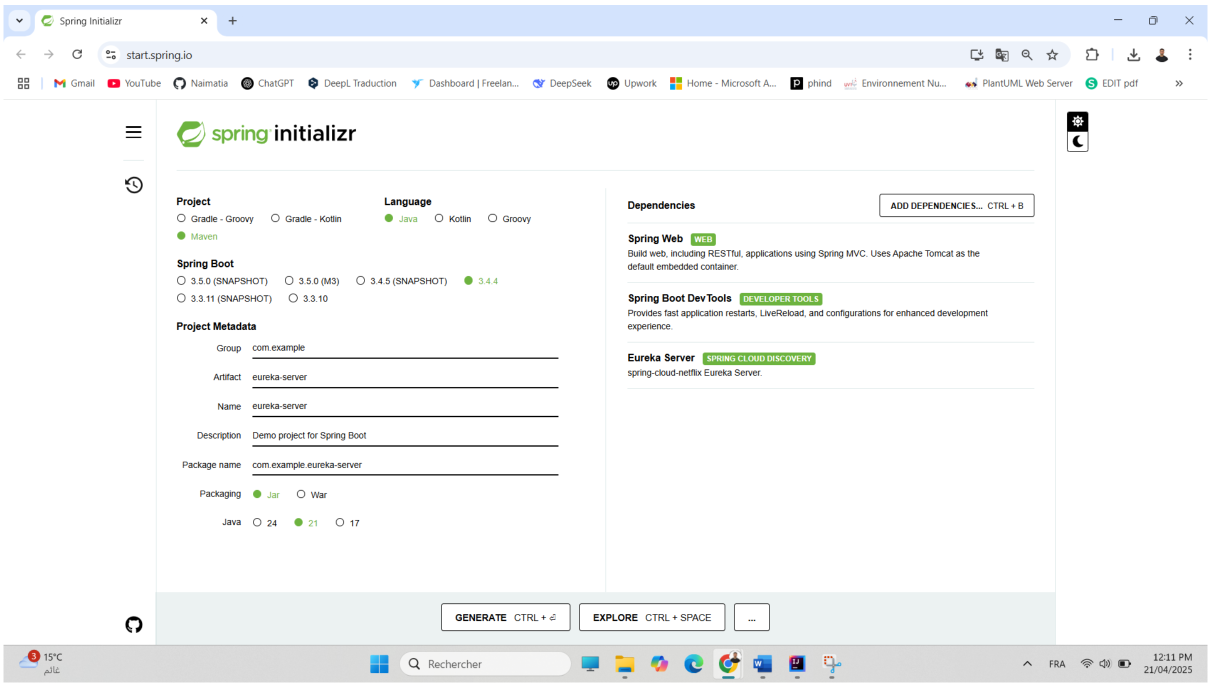
- **Name**: API Gateway
- **Group**: `com.microservices.gateway`
- **Artifact**: `api-gateway`
- **Language**: Java
- **Spring Boot Version**: 3.4.4
- **Java Version**: 21
- **Packaging**: JAR
- **Dependencies**:
  - Spring WebFlux (Required for reactive Gateway)
  - Spring Cloud Gateway
  - Eureka Client (For service discovery)
  - Spring Boot Actuator (Optional, for monitoring)

2. Discovery Service (Eureka Server)

- **Name**: Discovery Service
- **Group**: `com.microservices.discovery`
- **Artifact**: `discovery-service`
- **Language**: Java
- **Spring Boot Version**: 3.4.4
- **Java Version**: 21
- **Packaging**: JAR
- **Dependencies**:
  - Spring Web (Embedded Tomcat)
  - Spring Cloud Netflix Eureka Server

3. Review Service

- **Name**: Review Service
- **Group**: `com.microservices.review`
- **Artifact**: `review-service`
- **Language**: Java
- **Spring Boot Version**: 3.4.4
- **Java Version**: 21
- **Packaging**: JAR
- **Dependencies**:
  - Spring Web (REST API)
  - Spring Data JPA (Database access)
  - H2 Database (Embedded, for development)
  - Eureka Client (Service registration)
  - Lombok (Optional, for boilerplate reduction)
  - Spring Boot Validation (Input validation)



Project
Gradle - Groovy Gradle - Kotlin [✓ Maven]

Language
[✓ Java] Kotlin Groovy

Spring Boot
3.5.0 (SNAPSHOT) 3.5.0 (M3) 3.4.5 (SNAPSHOT) [✓ 3.4.4] 3.3.11 (SNAPSHOT) 3.3.10

Project Metadata

Group: com.example
Artifact: eureka-server
Name: eureka-server
Description: Demo project for Spring Boot
Package name: com.example.eureka-server
Packaging: [✓ Jar] War
Java: 24 [✓ 21] 17

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Eureka Server SPRING CLOUD DISCOVERY
spring-cloud-netflix Eureka Server.

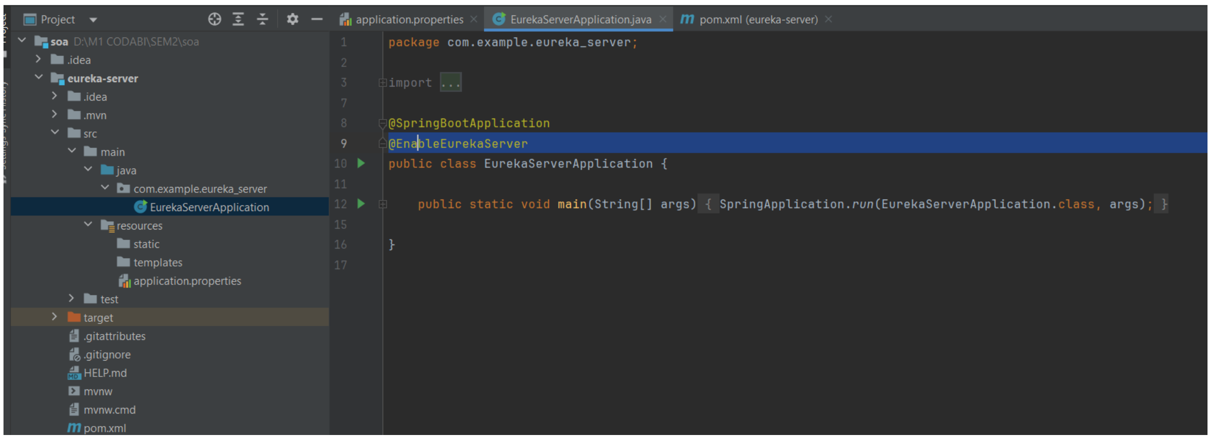
Figure 4.1: Example of creating eureka-service

Discovery Service Configuration

- Open the discovery service project.
- Inside the folder `src/main/resources`, create an `application.yml` file with the following configuration:

```yaml
server:
    port: 8761

spring:
  application:
    name: eureka-server

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    service-url:
      defaultZone: http://localhost:8761/eureka/
  server:
    wait-time-in-ms-when-sync-empty: 0
    enable-self-preservation: false
```

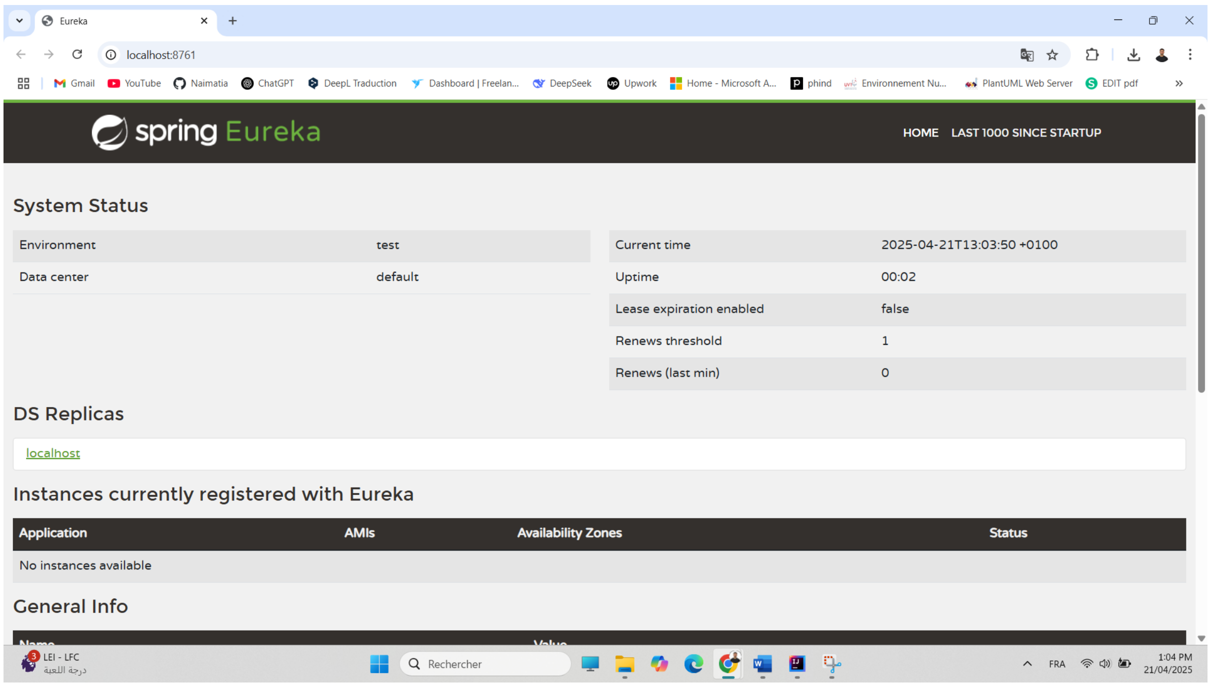
- Inside the main file `com/example/eureka/EurekaApplication.java` annotate the main class with `@EnableEurekaServer`



```java
package com.example.eureka_server;

import ...

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

    public static void main(String[] args) { SpringApplication.run(EurekaServerApplication.class, args); }

}
```

Figure 4.2: Enable eureka server

- Run the server and visit this URL `http://localhost:8761/`, you should see the next webpage



System Status

Environment	test
Data center	default

Current time	2025-04-21T13:03:50 +0100
Uptime	00:02
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

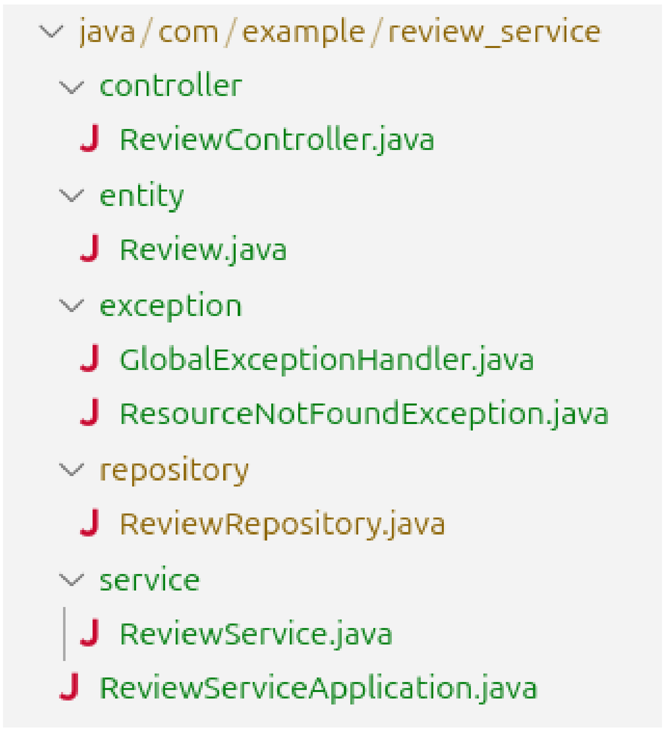
Figure 4.3: Eureka website

Review Service

- Open the review service project.
- Inside the folder `src/main/resources`, create an `application.yml` file with the following configuration:

```
server:
  port: 10000

spring:
  datasource:
    url: jdbc:h2:file:./data/review;DB_CLOSE_DELAY=-1
    driver-class-name: org.h2.Driver
    username: sa
    password: ""

  h2:
    console:
      enabled: true
      path: /h2-console

  jpa:
    hibernate:
      ddl-auto: update

  application:
    name: review-service

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
```

- Create 5 new package and name them `controller`, `entity`, `exception`, `repository`, and finally `service`, then create the files as shown the next figure:



```
java / com / example / review_service
  controller
    ReviewController.java
  entity
    Review.java
  exception
    GlobalExceptionHandler.java
    ResourceNotFoundException.java
  repository
    ReviewRepository.java
  service
    ReviewService.java
  ReviewServiceApplication.java
```

Figure 4.4: Review Service files structure

- Write the following code inside `ReviewController.java`

```java
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@RestController
@RequestMapping("/api/reviews")
public class ReviewController {
    private final ReviewService reviewService;

    public ReviewController(ReviewService reviewService) {
        this.reviewService = reviewService;
    }

    @GetMapping("/health")
    public ResponseEntity<Map<String, String>> health() {
        Map<String, String> response = new HashMap<>();
        response.put("STATUS", "UP");
        return ResponseEntity.ok(response);
    }

    @PostMapping
    public ResponseEntity<Review> createReview(
        @RequestHeader("User-Id") Long userId,
        @Valid @RequestBody Review review
    ) {
        System.out.println("userId: " + userId);
        review.setUserId(userId);
        Review createdReview = reviewService.createReview(userId,
    review);
        return ResponseEntity.status(HttpStatus.CREATED).body(
    createdReview);
    }

    @GetMapping("/product/{productId}")
    public ResponseEntity<List<Review>> getReviewsByProductId(
    @PathVariable Long productId) {
        return ResponseEntity.ok(reviewService.
    getReviewsByProductId(productId));
    }

    @GetMapping("/product/{productId}/average-rating")
    public ResponseEntity<Double> getAverageRating(@PathVariable
    Long productId) {
        return ResponseEntity.ok(reviewService.
    getAverageRatingForProduct(productId));
    }

    // for testing
    @GetMapping
    public ResponseEntity<List<Review>> getAllReviews() {
        return ResponseEntity.ok(reviewService.getAllReviews());
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteReview(@PathVariable Long
    id) {
        reviewService.deleteReview(id);
        return ResponseEntity.noContent().build();
    }
}
```

Listing 4.1: ReviewController.java

- Write the following code inside `Review.java`

```
import jakarta.persistence.*;
import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

import java.time.LocalDateTime;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
```

```
@Table(name = "reviews")
public class Review {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull(message = "Product ID cannot be null")
    @Column(name = "product_id", nullable = false)
    private Long productId;

    @NotNull(message = "User ID cannot be null")
    @Column(name = "user_id", nullable = false)
    private Long userId;

    @NotBlank(message = "Comment cannot be blank")
    @Column(nullable = false, length = 1000)
    private String comment;

    @NotNull(message = "Rating cannot be null")
    @Min(value = 1, message = "Rating must be at least 1")
    @Max(value = 5, message = "Rating must be at most 5")
    @Column(nullable = false)
    private Integer rating;

    @CreationTimestamp
    @Column(name = "created_at", updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    @Column(name = "updated_at")
    private LocalDateTime updatedAt;
}
```

Listing 4.2: ReviewController.java

- Write the following code inside `Review.java`

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleResourceNotFound(
   ResourceNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(
   ex.getMessage());
    }
}
```

Listing 4.3: GlobalExceptionHandler.java

- Write the following code inside `ResourceNotFoundException.java`

```
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

Listing 4.4: ResourceNotFoundException.java

- Write the following code inside `ReviewRepository.java`

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

public interface ReviewRepository extends JpaRepository<Review,
   Long> {
    List<Review> findByProductId(Long productId);
    List<Review> findByUserId(Long userId);

    @Query("SELECT AVG(r.rating) FROM Review r WHERE r.productId
   = :productId")
    Double findAverageRatingByProductId(@Param("productId") Long
   productId);

    List<Review> findByProductIdAndUserId(Long productId, Long
   userId);
}
```

Listing 4.5: ReviewRepository.java

- Write the following code inside `ReviewService.java`

```
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;


@Service
public class ReviewService {

    private final ReviewRepository reviewRepository;

    public ReviewService(ReviewRepository reviewRepository) {
        this.reviewRepository = reviewRepository;
    }

    public Review createReview(Long userId, Review review) {
        review.setUserId(userId);
        return reviewRepository.save(review);
    }

    public List<Review> getReviewsByProductId(Long productId) {
        return reviewRepository.findByProductId(productId);
    }

    public Double getAverageRatingForProduct(Long productId) {
        return reviewRepository.findAverageRatingByProductId(
   productId);
    }

    public List<Review> getAllReviews() {
        return reviewRepository.findAll();
    }

    public void deleteReview(Long id) {
        if (!reviewRepository.existsById(id)) {
            throw new ResourceNotFoundException("Review not found
    with id: " + id);
        }
        reviewRepository.deleteById(id);
    }
}
```

Listing 4.6: ReviewService.java

- **Note:** Double-check that all necessary imports are included and there are no unresolved references.
- **Note:** We have completed the setup—now you can run the server.

API Gateway

- Open the api gateway project.
- Inside the folder `src/main/resources`, create an `application.yml` file with the following configuration:

```yaml
server:
  port: 8080

spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        - id: product-service
          uri: lb://PRODUCT-SERVICE
          predicates:
            - Path=/api/products/**
            - Path=/api/orders/**
          filters:
            - StripPrefix=1
        - id: user-service
          uri: lb://AUTH-SERVICE
          predicates:
            - Path=/api/auth/**
          filters:
            - StripPrefix=1
        - id: review-service
          uri: lb://REVIEW-SERVICE
          predicates:
            - Path=/api/reviews/**
          filters:
            - StripPrefix=1

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
```

- Inside the main file `com/example/gateway/GatewayApplication.java` annotate the main class with `@EnableDiscoveryClient`

- Add the following code inside the class

```
    @Autowired
    private AuthenticationFilter authenticationFilter;

    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder
   builder) {
        return builder.routes()
                .route("product-service", r -> r.path("/api/
   products/**")
                        .filters(f -> f.filter(
   authenticationFilter))
                        .uri("lb://PRODUCT-SERVICE"))
                .route("product-service", r -> r.path("/api/
   orders/**")
                        .filters(f -> f.filter(
   authenticationFilter))
                        .uri("lb://PRODUCT-SERVICE"))
        .route("review-service", r -> r.path("/api/reviews/**")
            .filters(f -> f.filter(authenticationFilter))
            .uri("lb://REVIEW-SERVICE"))
                .build();
    }

  @Bean
  public CorsWebFilter corsWebFilter() {
    final CorsConfiguration corsConfig = new CorsConfiguration();
    corsConfig.setAllowedOrigins(Collections.singletonList("*"));
    corsConfig.setMaxAge(3600L);
    corsConfig.setAllowedMethods(Arrays.asList("GET", "POST", "
   PUT", "DELETE", "OPTIONS"));
    corsConfig.addAllowedHeader("*");

    final UrlBasedCorsConfigurationSource source = new
   UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", corsConfig);

    return new CorsWebFilter(source);
  }
```

Listing 4.7: Complete API gateway class

- Create a new package and name it `filter`
- Inside the filter package, define a class named `AuthValidationResponse` and add the next code:

```
public class AuthValidationResponse {

    private boolean valid;
    private String id;

    public AuthValidationResponse(boolean valid, String id) {
        this.valid = valid;
        this.id = id;
    }

    public boolean isValid() {
        return valid;
    }

    public void setValid(boolean valid) {
        this.valid = valid;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
}
```

Listing 4.8: AuthValidationResponse class

- In the same package, add another class called `AuthenticationFilter`

```
import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.
    GatewayFilterChain;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient
    ;
import org.springframework.web.server.ServerWebExchange;

import reactor.core.publisher.Mono;

@Component
public class AuthenticationFilter implements GatewayFilter,
    Ordered {
     private final WebClient webClient;


     public AuthenticationFilter(WebClient.Builder
    webClientBuilder) {
         this.webClient = webClientBuilder.baseUrl("http://
    localhost:3000").build();
     }

     public static String processPath(String path) {
         if (path != null && path.contains("products") && path.
    startsWith("/api")) {
             return path.substring(4);
         }
         return path;
     }

     @Override
public Mono<Void> filter(ServerWebExchange exchange,
    GatewayFilterChain chain) {
     String path = exchange.getRequest().getPath().toString();

     if (path.startsWith("/api/orders") || path.startsWith("/api/
    products") || path.startsWith("/api/reviews")) {
         String authHeader = exchange.getRequest().getHeaders().
    getFirst("Authorization");

         if (authHeader == null || !authHeader.startsWith("Bearer
    ")) {
             System.out.println("Missing Authorization header");
             exchange.getResponse().setStatusCode(org.
    springframework.http.HttpStatus.UNAUTHORIZED);
             return exchange.getResponse().setComplete();
         }

         String token = authHeader.substring(7);
         System.out.println("Path: " + path);
         return webClient
             .get()
             .uri("/auth/validate")
             .header("Authorization", "Bearer " + token)
```

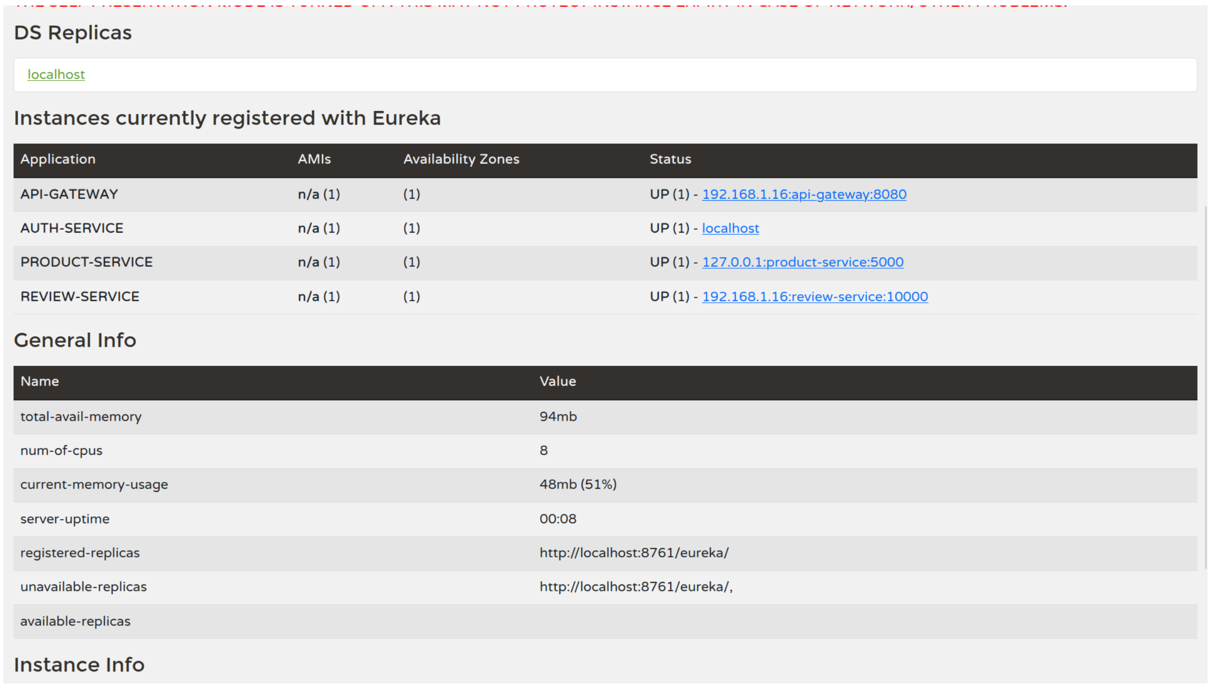
```
                .retrieve()
                .bodyToMono(AuthValidationResponse.class)
                .flatMap(response -> {
                    System.out.println("response: " + response.getId
    ());
                    if (response != null && response.isValid()) {
                        ServerWebExchange mutatedExchange = exchange.
    mutate()
                            .request(builder -> builder.header("user-
    id", response.getId()))
                            .build();
                        return chain.filter(mutatedExchange);
                    } else {
                        exchange.getResponse().setStatusCode(org.
    springframework.http.HttpStatus.UNAUTHORIZED);
                        return exchange.getResponse().setComplete();
                    }
                })
                .onErrorResume(error -> {
                    exchange.getResponse().setStatusCode(org.
    springframework.http.HttpStatus.UNAUTHORIZED);
                    return exchange.getResponse().setComplete();
                });
     }

     return chain.filter(exchange);
   }
     @Override
     public int getOrder() {
         return -1;
     }

}
```

Listing 4.9: AuthenticationFilter class

- **Note:** We have completed the setup—now you can run the server.

Active Instances in Eureka

After developing all the services, make sure to run them so that each instance becomes active and gets registered in the Eureka service, as shown in the next figure:



DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - 192.168.1.16:api-gateway:8080
AUTH-SERVICE	n/a (1)	(1)	UP (1) - localhost
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - 127.0.0.1:product-service:5000
REVIEW-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.16:review-service:10000

General Info

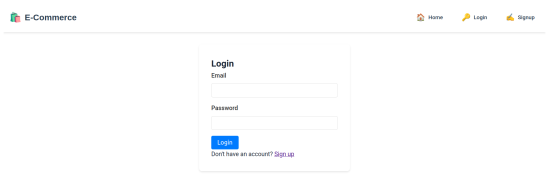
Name	Value
total-avail-memory	94mb
num-of-cpus	8
current-memory-usage	48mb (51%)
server-uptime	00:08
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/,
available-replicas	

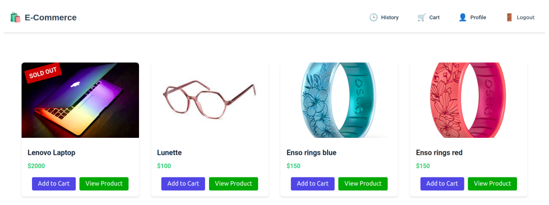
Instance Info

Figure 4.5: Eureka active instances

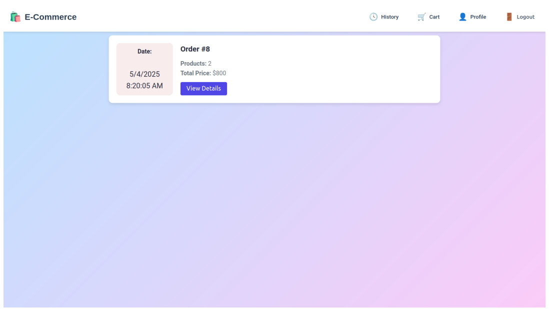
Final Project

Here are some pictures from the final project that show how everything came together.

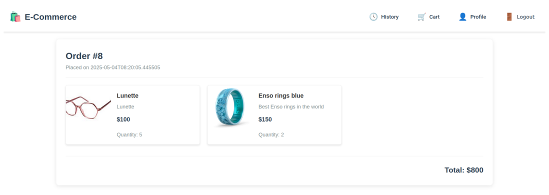




Lenovo Laptop $2000
Lunette $100
Enso rings blue $150
Enso rings red $150



Date: 5/4/2025 8:20:05 AM
Order #8
Products: 2
Total Price: $800

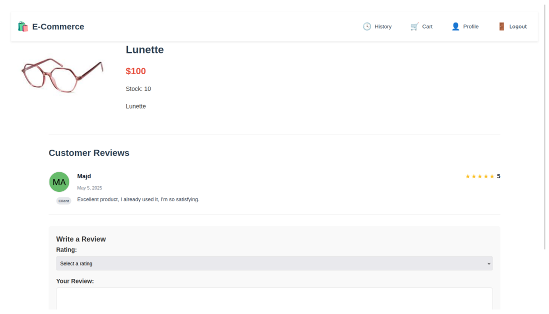


Order #8
Placed on 2025-05-04T08:20:05.445505

Lunette
Lunette
$100
Quantity: 5

Enso rings blue
Best Enso rings in the world
$150
Quantity: 2

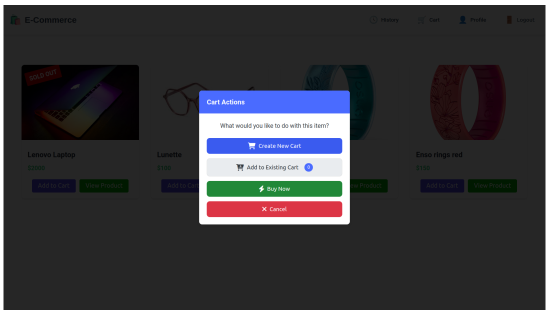
Total: $800

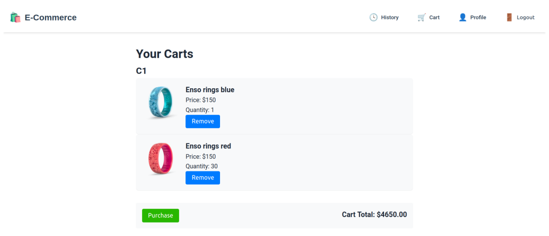


Lunette
$100
Stock: 10
Lunette

Customer Reviews

MA Majd ★★★★★ 5
May 5, 2025
Client Excellent product, I already used it, I'm so satisfying.



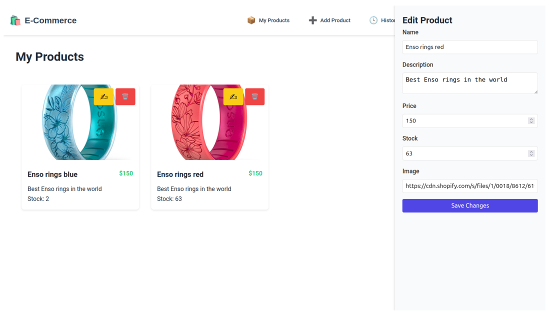


Your Carts
C1

Enso rings blue
Price: $150
Quantity: 1

Enso rings red
Price: $150
Quantity: 30

Cart Total: $4650.00



My Products

Enso rings blue $150
Best Enso rings in the world
Stock: 2

Enso rings red $150
Best Enso rings in the world
Stock: 63

Edit Product
Name: Enso rings red
Description: Best Enso rings in the world
Price: 150
Stock: 63
Image: https://cdn.shopify.com/s/files/1/0018/8612/61

Figure 4.6: Selected screenshots from the final project.