ENSET Mohammedia
Hassan II University of Casablanca
Master's Program in Software Engineering

Microservices Architecture with Spring Cloud and Angular

Distributed and Parallel Systems Laboratory

Prepared by:
Otmane OUHAMI

Academic Year:
2024 – 2025

December 2024

Abstract

This report presents a comprehensive study and implementation of a microservices-based application using Spring Boot and Spring Cloud technologies. The project demonstrates the practical application of distributed systems concepts including service discovery, centralized configuration management, API gateway patterns, and inter-service communication.

The system consists of multiple independent services working together to provide a cohesive billing and inventory management solution, complemented by a modern Angular frontend that interacts with the backend through a unified API gateway.

Key technologies explored include Netflix Eureka for service discovery, Spring Cloud Config for centralized configuration, Spring Cloud Gateway for API routing, and OpenFeign for declarative REST clients. The frontend leverages Angular 19 with its modern signals-based reactive architecture.

Keywords: Microservices, Spring Boot, Spring Cloud, Angular, Service Discovery, API Gateway, Distributed Systems

Contents

1 Introduction

The evolution of software architecture has led to the widespread adoption of microservices as a preferred approach for building complex, scalable applications. Unlike monolithic architectures where all functionality resides within a single deployable unit, microservices decompose applications into small, independently deployable services that communicate over well-defined APIs.

This laboratory project explores the practical implementation of a microservices ecosystem using the Spring Cloud framework, which provides essential tools and patterns for building distributed systems. The application simulates a simplified e-commerce scenario with customer management, product inventory tracking, and billing functionalities, each implemented as separate microservices.

The primary objectives of this project include understanding service discovery mechanisms and their role in dynamic service location, implementing centralized configuration management for consistent application settings across services, utilizing an API gateway for request routing and load balancing, establishing inter-service communication using declarative REST clients, and developing a responsive frontend application using Angular with modern design patterns.

Through this implementation, we gain practical experience with the challenges and solutions inherent in distributed computing, including network latency, partial failures, and data consistency across service boundaries.

2 System Architecture

The architecture of this application follows the microservices pattern with several supporting infrastructure components that facilitate service coordination and communication. Understanding this architecture is fundamental to appreciating how the individual components work together as a cohesive system.

2.1 Architecture Overview

The system comprises six distinct services organized into two categories: infrastructure services that provide supporting functionality, and business services that implement the core application logic.

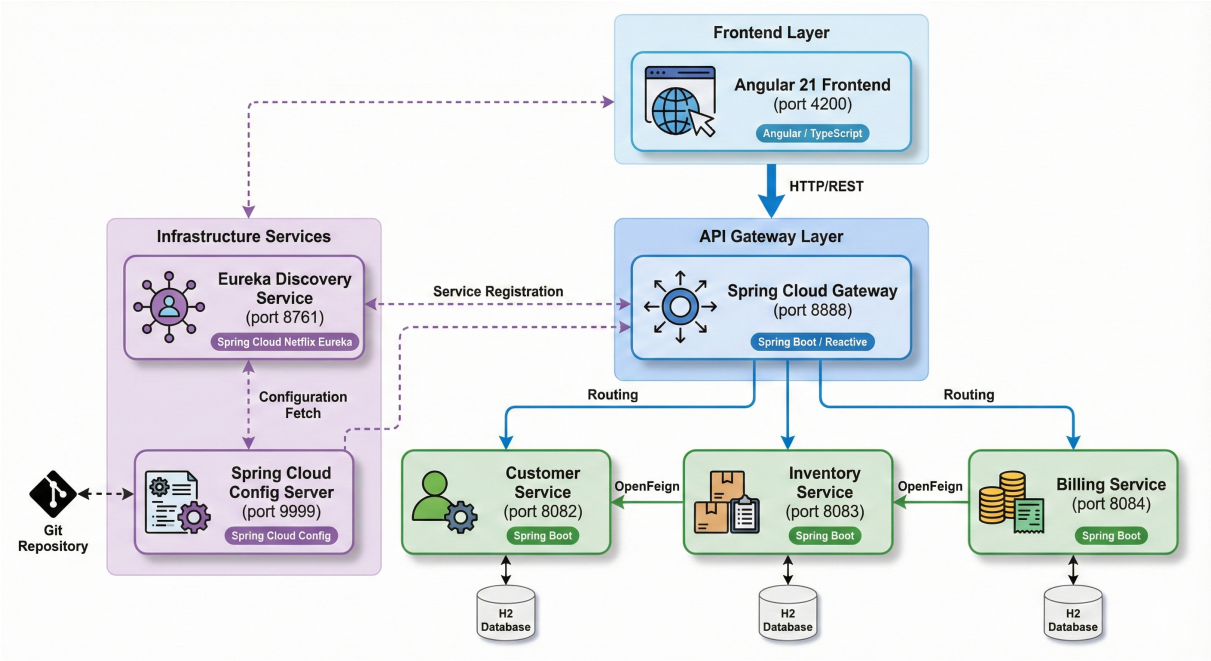

Figure 1: Complete microservices architecture showing all components and their interactions

As illustrated in Figure 1, the architecture demonstrates several key patterns. The Discovery Service acts as a service registry where all microservices register themselves upon startup. The Config Service provides externalized configuration, allowing services to retrieve their settings from a centralized Git repository. The Gateway Service serves as the single entry point for all client requests, routing them to appropriate backend services. The three business services, namely Customer Service, Inventory Service, and Billing Service, each maintain their own database and expose REST APIs for their respective domains.

2.2 Technology Stack

The implementation leverages a carefully selected technology stack that provides robust support for microservices development. The backend is built entirely with Spring Boot 3.5.6, utilizing Java 21 as the runtime environment. Spring Cloud 2025.0.0 provides the distributed systems infrastructure components. Each service uses Spring Data JPA with H2 in-memory databases for data persistence, while Spring Data REST automatically exposes repository methods as RESTful endpoints.

The frontend application is developed using Angular 19 with TypeScript, employing a signals-based reactive architecture for optimal performance. The styling follows a shadcn-inspired design system implemented in pure CSS, providing a modern and professional user interface.

3 Infrastructure Services

Infrastructure services form the backbone of the microservices ecosystem, providing essential capabilities that enable the business services to function effectively in a distributed environment.

3.1 Discovery Service with Netflix Eureka

Service discovery is a fundamental requirement in microservices architectures where services need to locate and communicate with each other dynamically. Netflix Eureka, integrated through Spring Cloud Netflix, provides this capability by maintaining a registry of available service instances.

The Discovery Service is configured as an Eureka Server, as shown in the following Spring Boot application class:

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(DiscoveryServiceApplication.class,
    args);
    }
}
```

Listing 1: Discovery Service Main Application

The `@EnableEurekaServer` annotation transforms this Spring Boot application into a service registry. When other services start, they automatically register with this server and periodically send heartbeat signals to indicate their availability. If a service fails to send heartbeats, Eureka marks it as unavailable and eventually removes it from the registry.

The configuration for the Discovery Service specifies that it should not register itself or fetch the registry, as it is the registry itself:

```
server.port=8761
spring.application.name=discovery-service
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Listing 2: Discovery Service Configuration

3.2 Config Service with Spring Cloud Config

Managing configuration across multiple services presents significant challenges, particularly when configurations need to change without redeploying services. Spring Cloud Config addresses this by providing a centralized configuration server that serves configuration properties from a Git repository.

```java
@SpringBootApplication
@EnableConfigServer
public class ConfigServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServiceApplication.class,
    args);
    }
}
```

Listing 3: Config Service Main Application

The configuration server connects to a Git repository containing property files for each service. When a service requests its configuration, the Config Server retrieves the appropriate file based on the service name and active profiles.

```
server.port=9999
spring.application.name=config-service
spring.cloud.config.server.git.uri=file:///path/to/config-repo
```

Listing 4: Config Service Properties

This approach enables configuration changes without service redeployment, version control for configuration history, environment-specific configurations through profile support, and consistent configuration management across all services.

3.3 Gateway Service with Spring Cloud Gateway

The API Gateway pattern provides a single entry point for all client requests, offering benefits such as request routing, protocol translation, and cross-cutting concerns like authentication and rate limiting. Spring Cloud Gateway implements this pattern using a reactive, non-blocking architecture built on Project Reactor.

```java
@SpringBootApplication
public class GatewayServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayServiceApplication.class,
    args);
    }

    @Bean
    DiscoveryClientRouteDefinitionLocator routeDefinitionLocator
    (
            ReactiveDiscoveryClient discoveryClient,
            DiscoveryLocatorProperties properties) {
        return new DiscoveryClientRouteDefinitionLocator(
            discoveryClient, properties);
    }
}
```

Listing 5: Gateway Service Configuration

The `DiscoveryClientRouteDefinitionLocator` bean enables dynamic route discovery. Rather than manually configuring routes for each service, the gateway automatically creates routes based on services registered with Eureka. When a request arrives at `/customer-service/api/customers`, the gateway resolves the customer-service through Eureka and forwards the request to an available instance.

Cross-Origin Resource Sharing (CORS) configuration is essential when the frontend application runs on a different origin than the backend services:

```
@Configuration
public class CorsConfig {
    @Bean
    public CorsWebFilter corsWebFilter() {
        CorsConfiguration corsConfig = new CorsConfiguration();
        corsConfig.setAllowedOrigins(List.of("http://localhost
    :4200"));
        corsConfig.setAllowedMethods(Arrays.asList(
            "GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS"))
    ;
        corsConfig.setAllowedHeaders(List.of("*"));
        corsConfig.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source =
            new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", corsConfig);
        return new CorsWebFilter(source);
    }
}
```

Listing 6: CORS Configuration for Gateway

4 Business Services

The business services implement the core functionality of the application, each responsible for a specific domain within the system. Following the microservices principle of single responsibility, each service manages its own data and exposes a well-defined API.

4.1 Customer Service

The Customer Service manages customer information, providing operations for creating, reading, updating, and deleting customer records. The service uses Spring Data REST to automatically expose repository methods as RESTful endpoints.

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Getter @Setter @Builder
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
}
```

Listing 7: Customer Entity Definition

The repository interface extends `JpaRepository` and is annotated with `@RepositoryRestResource`, which instructs Spring Data REST to expose CRUD operations automatically:

```
@RepositoryRestResource
public interface CustomerRepository
        extends JpaRepository<Customer, Long> {
}
```

Listing 8: Customer Repository with REST Exposure

This minimal configuration generates a complete REST API with endpoints for listing all customers, retrieving individual customers by ID, creating new customers, updating existing customers, and deleting customers. The API follows HATEOAS principles, including hypermedia links in responses for discoverability.

4.2 Inventory Service

The Inventory Service manages the product catalog and stock levels. Products are identified by UUID to ensure globally unique identifiers across distributed systems.

```
@Entity
```

```
@NoArgsConstructor
@AllArgsConstructor
@Getter @Setter @Builder
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;
    private String name;
    private double price;
    private int quantity;
}
```

Listing 9: Product Entity with UUID Identifier

Beyond the standard CRUD operations provided by Spring Data REST, the Inventory Service exposes additional endpoints for inventory management operations such as updating stock quantities:

```
@RestController
@RequestMapping("/inventory")
@RequiredArgsConstructor
public class ProductRestController {
    private final ProductRepository productRepository;

    @PostMapping("/products/{id}/update-quantity")
    public ResponseEntity<Product> updateQuantity(
            @PathVariable UUID id,
            @RequestParam int delta) {
        return productRepository.findById(id)
            .map(product -> {
                int newQuantity = product.getQuantity() + delta;
                if (newQuantity < 0) {
                    return ResponseEntity.badRequest().<Product>
    build();
                }
                product.setQuantity(newQuantity);
                return ResponseEntity.ok(productRepository.save(
    product));
            })
            .orElse(ResponseEntity.notFound().build());
    }
}
```

Listing 10: Inventory Controller for Stock Management

4.3 Billing Service

The Billing Service represents the most complex component, managing bills and their associated line items. This service demonstrates inter-service communication by fetching customer and product details from other services.

```java
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Getter @Setter @Builder
public class Bill {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date billingDate;
    private Long customerId;

    @OneToMany(mappedBy = "bill")
    private List<ProductItem> productItems = new ArrayList<>();

    @Transient
    private Customer customer;
}
```

Listing 11: Bill Entity with Relationships

The `@Transient` annotation on the customer field indicates that this property is not persisted in the database. Instead, it is populated at runtime by fetching data from the Customer Service.

4.4 Inter-Service Communication with OpenFeign

OpenFeign provides a declarative approach to building REST clients. Rather than manually constructing HTTP requests, developers define interfaces annotated with Spring MVC annotations, and Feign generates the implementation.

```java
@FeignClient(name = "customer-service")
public interface CustomerRestClient {
    @GetMapping("/api/customers/{id}")
    Customer getCustomerById(@PathVariable("id") Long id);

    @GetMapping("/api/customers")
    PagedModel<Customer> getCustomers();
}
```

Listing 12: Feign Client for Customer Service

The `name` attribute specifies the service name as registered in Eureka. Feign integrates with the service discovery mechanism to resolve the actual service location at runtime. This abstraction means that the calling code does not need to know the physical address of the target service.

The Billing Service uses these Feign clients to enrich bill data with customer and product information:

```java
@RestController
```

```java
@RequiredArgsConstructor
public class BillRestController {
    private final BillRepository billRepository;
    private final ProductItemRepository productItemRepository;
    private final CustomerRestClient customerRestClient;
    private final ProductRestClient productRestClient;

    @GetMapping("/bills/full/{id}")
    public Bill getBill(@PathVariable Long id) {
        Bill bill = billRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Bill not
    found"));

        bill.setCustomer(customerRestClient.getCustomerById(
            bill.getCustomerId()));

        bill.getProductItems().forEach(item -> {
            item.setProduct(productRestClient.getProductById(
                item.getProductId()));
        });

        return bill;
    }
}
```

Listing 13: Bill Controller with Service Integration

5 Frontend Application

The frontend provides a user-friendly interface for interacting with the microservices backend. Built with Angular 19, it employs modern development practices including standalone components, signals for reactive state management, and a custom CSS design system inspired by shadcn/ui.

5.1 Application Structure

The Angular application follows a modular structure that promotes maintainability and scalability. Services encapsulate HTTP communication logic, while components manage presentation and user interaction.

```
@Injectable({ providedIn: 'root' })
export class CustomerService {
  private http = inject(HttpClient);
  private baseUrl = `${environment.apiBaseUrl}/customer-service/api/customers`;

  getAll(): Observable<Customer[]> {
    return this.http.get<PagedResponse<Customer>>(this.baseUrl).pipe(
      map(response => response._embedded?.customers || [])
    );
  }

  create(customer: Customer): Observable<Customer> {
    return this.http.post<Customer>(this.baseUrl, customer);
  }

  update(id: number, customer: Customer): Observable<Customer> {
    return this.http.put<Customer>(`${this.baseUrl}/${id}`, customer);
  }

  delete(id: number): Observable<void> {
    return this.http.delete<void>(`${this.baseUrl}/${id}`);
  }
}
```

Listing 14: Customer Service in Angular (TypeScript)

All HTTP requests are routed through the API Gateway at port 8888, which forwards them to the appropriate backend service. This approach provides a single point of entry, simplifying CORS configuration and enabling future enhancements such as authentication middleware.

5.2 Reactive State Management with Signals

Angular signals provide a reactive primitive for managing component state. Unlike the traditional change detection approach that relies on Zone.js, signals explicitly track dependencies and update only the affected parts of the view.

```typescript
@Component({
  selector: 'app-customers',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './customers.component.html'
})
export class CustomersComponent implements OnInit {
  private customerService = inject(CustomerService);

  customers = signal<Customer[]>([]);
  loading = signal(true);
  showModal = signal(false);

  loadCustomers() {
    this.loading.set(true);
    this.customerService.getAll().subscribe({
      next: (data) => {
        this.customers.set(data);
        this.loading.set(false);
      },
      error: (err) => {
        console.error('Error loading customers:', err);
        this.loading.set(false);
      }
    });
  }
}
```

Listing 15: Component Using Signals (TypeScript)

This approach eliminates the need for Zone.js and provides more predictable change detection behavior, improving application performance.

5.3 Design System and Styling

The application employs a custom CSS design system that provides consistent styling across all components. Using CSS custom properties enables easy theming and maintains visual consistency.

```css
:root {
  --background: 0 0% 100%;
  --foreground: 240 10% 3.9%;
  --primary: 240 5.9% 10%;
  --primary-foreground: 0 0% 98%;
  --muted: 240 4.8% 95.9%;
```

```
    --border: 240 5.9% 90%;
    --radius: 0.5rem;
}

.btn-primary {
    background: hsl(var(--primary));
    color: hsl(var(--primary-foreground));
    padding: 0.5rem 1rem;
    border-radius: var(--radius);
    transition: opacity 0.2s;
}
```

Listing 16: CSS Design System Variables

5.4 User Interface

The frontend provides intuitive interfaces for managing customers, products, and bills. Each view includes data tables with action buttons, modal forms for creating and editing records, and visual feedback for loading states.

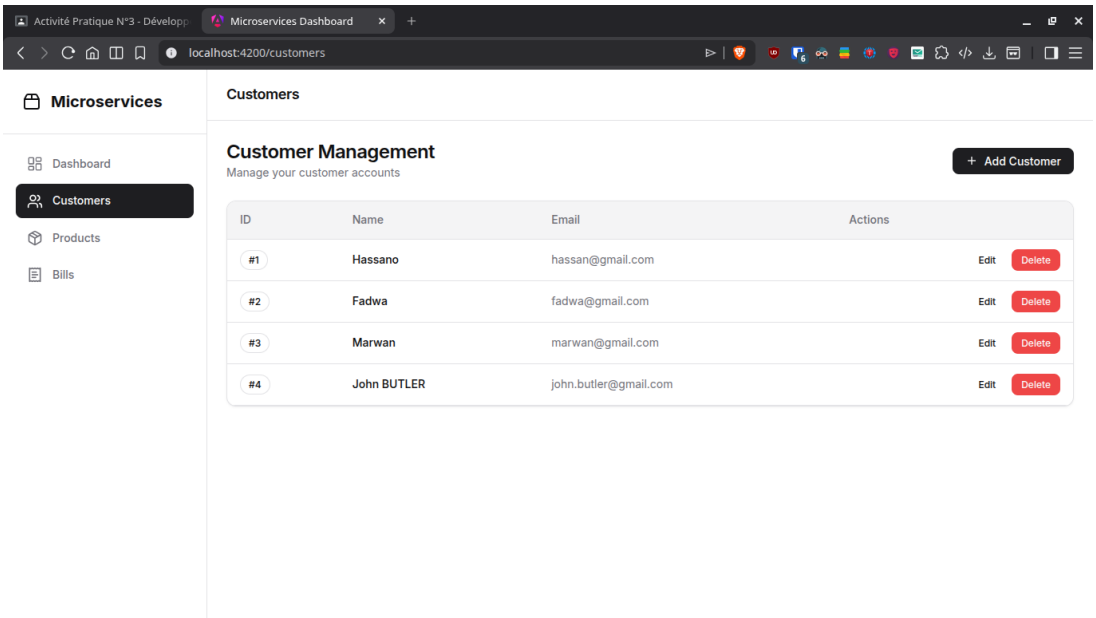

Figure 2: Customer management interface displaying the list of customers with action buttons

The customer management interface shown in Figure 2 presents customers in a clean table format with options to edit or delete each record. The interface follows accessibility best practices with clear labels and consistent interaction patterns.

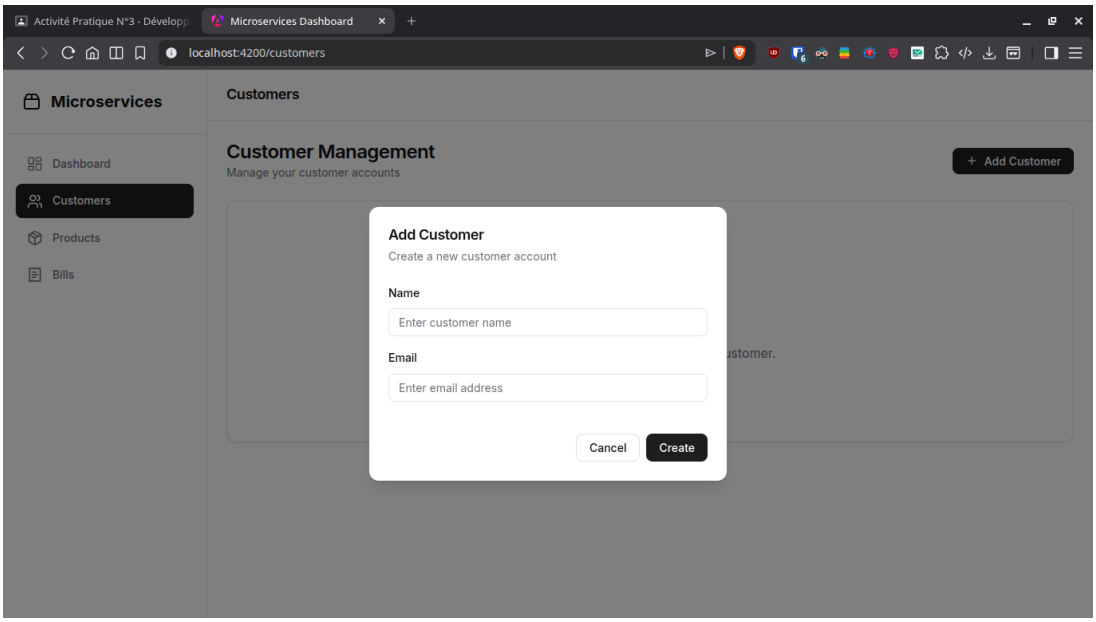

Figure 3: Modal dialog for adding a new customer

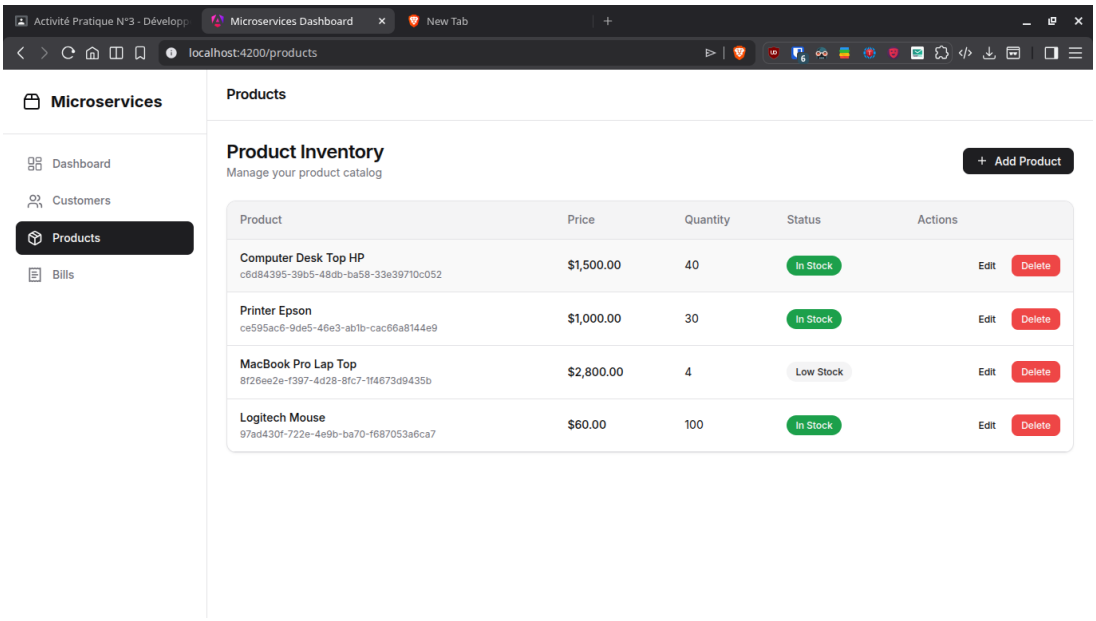

Figure 4: Product inventory interface showing stock levels and product details

The product management interface in Figure 4 displays the inventory with stock status indicators. Products with sufficient stock are marked as "In Stock" while those with low quantities receive appropriate visual warnings.

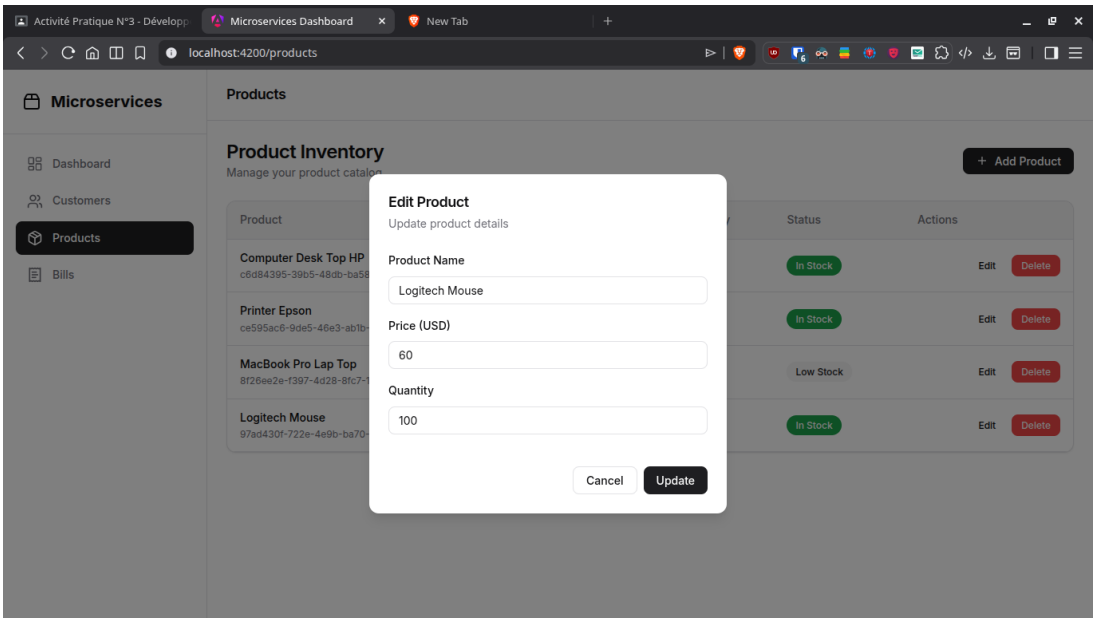

Figure 5: Product editing form with price and quantity fields

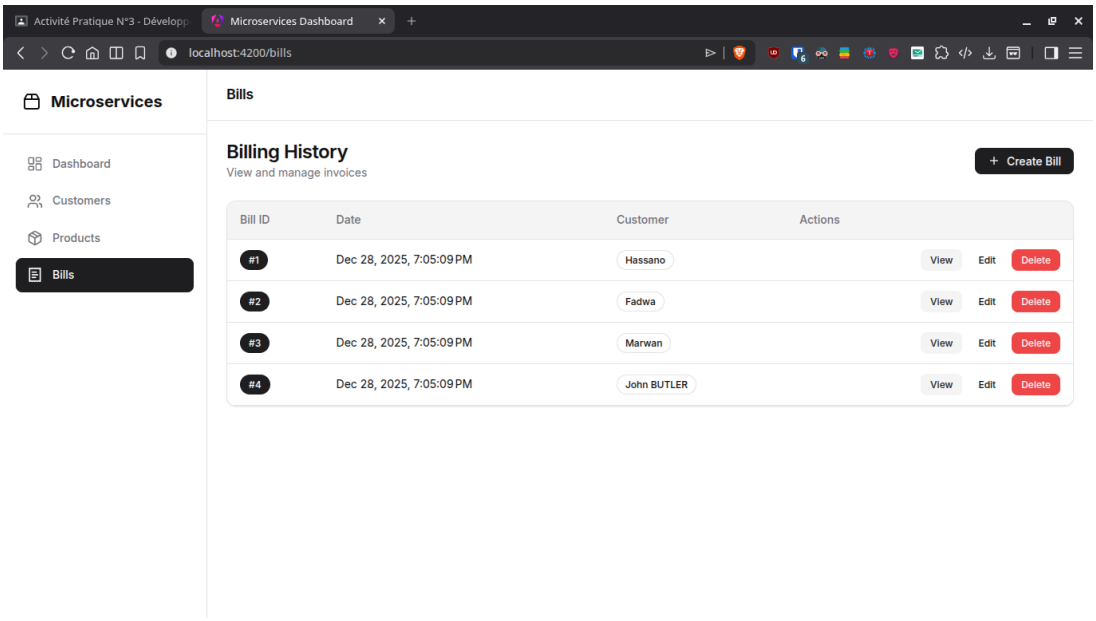

Figure 6: Bills listing interface showing all invoices

The billing interface in Figure 6 presents all invoices with their associated customers. Users can view detailed bill information including individual line items and their totals.

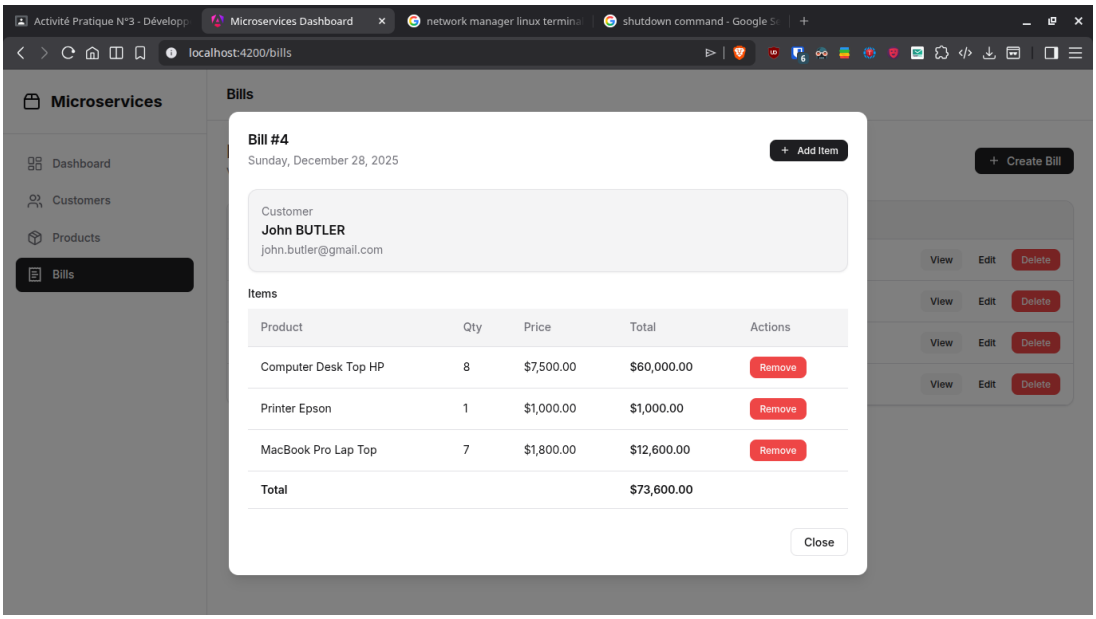

Figure 7: Detailed bill view with product items and total calculation

Figure 7 shows the bill detail view where users can see all items included in a bill, their quantities, unit prices, and the computed total. This view also allows adding new items to an existing bill or removing items, with automatic inventory updates.

6 API Documentation with OpenAPI

The project integrates SpringDoc OpenAPI to provide interactive API documentation. Each service exposes its API specification through Swagger UI, accessible at the `/swagger-ui.html` endpoint.

The Gateway Service aggregates documentation from all downstream services, providing a unified view of the entire API surface. This configuration is achieved through a dedicated SwaggerConfig class that registers each service's documentation endpoint:

```java
@Configuration
public class SwaggerConfig {
    @Bean
    @Lazy(false)
    public Set<SwaggerUrl> swaggerUrls(
            RouteDefinitionLocator locator,
            SwaggerUiConfigProperties swaggerUiConfigProperties)
    {
        Set<SwaggerUrl> urls = new HashSet<>();

        List<String> services = List.of(
            "customer-service",
            "inventory-service",
            "billing-service"
        );

        services.forEach(service -> {
            SwaggerUrl swaggerUrl = new SwaggerUrl();
            swaggerUrl.setName(service);
            swaggerUrl.setUrl("/" + service + "/v3/api-docs");
            urls.add(swaggerUrl);
        });

        swaggerUiConfigProperties.setUrls(urls);
        return urls;
    }
}
```

Listing 17: Swagger Aggregation Configuration

Accessing the Gateway's Swagger UI at `http://localhost:8888/swagger-ui.html` presents a dropdown menu allowing users to select and explore the documentation for each individual service.

7 Conclusion

This project has provided valuable hands-on experience with microservices architecture using Spring Cloud and Angular. Through the implementation of a complete distributed system, we have explored key concepts and patterns that are essential in modern software development.

The service discovery pattern, implemented using Netflix Eureka, demonstrated how services can dynamically locate each other without hardcoded addresses. This capability is fundamental in cloud environments where service instances may scale up or down and their addresses may change frequently. The centralized configuration approach with Spring Cloud Config showed how application settings can be managed consistently across all services, enabling configuration changes without redeployment.

The API Gateway pattern proved invaluable for simplifying client communication with the backend services. By providing a single entry point, the gateway eliminated the need for clients to know the locations of individual services and enabled centralized handling of cross-cutting concerns such as CORS. The inter-service communication through OpenFeign demonstrated how microservices can collaborate while remaining loosely coupled, with the billing service fetching data from customer and inventory services transparently.

On the frontend, Angular's modern features including standalone components and signals provided an efficient approach to building reactive user interfaces. The separation of concerns between services, components, and styling facilitated maintainable code organization, while the custom CSS design system ensured visual consistency throughout the application.

Looking forward, this architecture could be extended with additional capabilities such as distributed tracing for debugging request flows across services, circuit breakers for improved resilience against service failures, message queues for asynchronous communication, and containerization with Docker and Kubernetes for deployment. The solid foundation established in this project provides a starting point for exploring these advanced topics in distributed systems.

The experience gained through this implementation reinforces the importance of understanding distributed systems fundamentals. While microservices offer significant benefits in terms of scalability and team autonomy, they also introduce complexity that must be carefully managed through proper architectural patterns and infrastructure components.