Cookie-Shop

Modul 321

Rinaldo Lanza

Aakash Sethi, Delvin Ngauv, Lucas Heroin

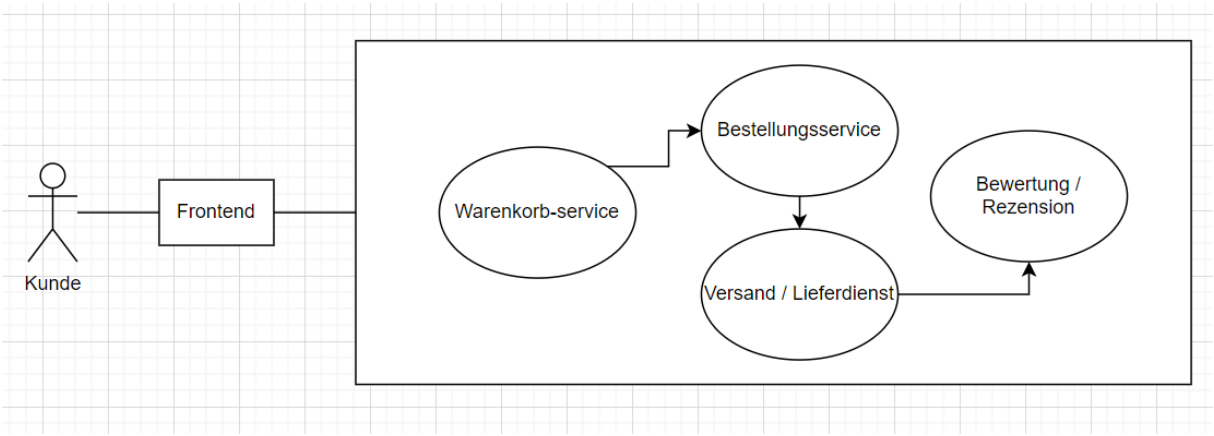
Github: https://github.com/delvindylan/Microservices_Cookies

Contents

Projektinhalt

In diesem Projekt geht es darum sich Gedanken über Mikroservices zu machen. Für diesen Auftrag haben wir ein Unternehmen gegründet, welche Kekse verkauft. Dieser verfügt im Backend über mehrere Services. Wir sind noch nicht soweit mit der Implementierung unserer Online-Präsenz, weswegen wir nur eine Planung erstellen

Architekturskizze



Der Kunde kann über das Frontend Produkte dem Warenkorb-Service hinzufügen. Anschliessend gelangt er zum Bestell-Service, wo er seine Kundendaten eingeben kann. Für die Auswahl des Versanddienstleisters und der Versandart steht der Versand-/Lieferdienst-Service zur Verfügung. Schliesslich hat der Kunde die Möglichkeit, eine Bewertung abzugeben.

Warenkorb Service

Zweck: Durch den Warenkorb können Kunden Produkte temporär in einer Kaufliste stellen. Der Virtuelle Warenkorb dient als normaler Warenkorb.

Technologien: Spring Boot, Spring Data JPA, und eine relationale Datenbank.

Endpunkte:

POST /basket: Ein Produkt zum Warenkorb hinzufügen.

GET /basket: Alle Produkte im Warenkorb abrufen.

GET /basket/{id}: Ein Produkte im Warenkorb abrufen.

DELETE /basket/{id}: Ein Produkt aus dem Warenkorb löschen.

Bestellungsservice

Zweck: Verarbeitet Bestellungen von Kunden, inklusive Bestelldetails und Zahlungsinformationen.

Technologien: Spring Boot, Spring Data JPA, und eine relationale Datenbank.

Endpoints:

POST /orders: Eine neue Bestellung erstellen.

GET /orders: Alle Bestellungen abrufen.

GET /orders/{id}: Eine Bestellung anhand der ID abrufen.

PUT /orders/{id}: Eine Bestellung aktualisieren (z.B. Bestellstatus).

DELETE /orders/{id}: Eine Bestellung löschen.

Versand / Lieferdienst

Zweck: Dieser Service kümmert sich um die Logistik der Bestellung, wie z.B. Versandoptionen, Tracking-Informationen und Lieferzeitberechnung.

Technologien: Spring Boot und eine relationale Datenbank.

Endpoints:

POST /shipping: Versandinformationen hinzufügen.

GET /shipping: Alle Versandinformationen abrufen.

GET /shipping/{id}: Versandinformation anhand der ID abrufen.

PUT /shipping/{id}: Versandinformation aktualisieren.

DELETE /shipping/{id}: Versandinformation löschen.

Bewertungs- und Rezensions-Service

Zweck: Der Bewertungs- und Rezensions-Service ermöglicht es Kunden, Bewertungen und Rezensionen für verschiedene Produkte zu hinterlassen. Dies dient dazu, die Qualität und Zufriedenheit der Produkte zu bewerten und Feedback für Verbesserungen zu sammeln.

Technologien: Der Service wird mit Spring Boot als Framework für die Entwicklung des Backends implementiert. Spring Data JPA wird verwendet, um die Interaktion mit der relationalen Datenbank zu vereinfachen. Die Datenbank selbst ist ebenfalls relational und speichert Informationen über Kunden, Produkte und deren Bewertungen.

Endpoints:

POST /reviews: Eine Bewertung hinzufügen.

GET /reviews: Alle Bewertungen abrufen.

GET /reviews/{id}: Eine Bewertung anhand der ID abrufen.

DELETE /reviews/{id}: Eine Bewertung löschen.

PUT /reviews/{id}: Eine Bewertung aktualisieren.

Web-Applikation - ReviewsAG

Microservice

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.web.client.HttpClientErrorException;

@Service
public class ProductChoiceListener {

    private static final Logger logger = LoggerFactory.getLogger(ProductChoiceListener.class);

    @Autowired
    private InventoryClient inventoryClient; // Assuming this is your Feign Client

    @KafkaListener(topics = "product-choice-topic", groupId = "shopping-basket-group")
    public void listen(ConsumerRecord<String, ShoppingBasketUpdate> record) {
        ShoppingBasketUpdate update = record.value();
        String productId = update.getProductId();

        try {
            // Fetch product details using Feign Client
            Product product = inventoryClient.getProductById(Long.parseLong(productId));

            // Process the product choice update
            processProductChoice(update, product);
        } catch (HttpClientErrorException | NumberFormatException e) {
            logger.error("Error processing product choice update: {}", e.getMessage());
            // Handle the error appropriately, e.g., retry, log, notify
        }
    }

    private void processProductChoice(ShoppingBasketUpdate update, Product product) {
        // Implement your business logic here
        // For example, validate the product, update the shopping basket, etc.
        logger.info("Processed product choice for product ID: {}, name: {}", product.getId(), product.getName());
    }
}
```

Dieser Microservice brauchen wir für Waren die in den Warenkorb getan werden. Es verwendet SLF4J für das Logging und bietet eine bessere Kontrolle über die Ausgabe und Ebenen der Logs. Es integriert sich mit einem externen Dienst (InventoryClient), um Produktinformationen basierend auf der im Kafka-Nachricht empfangenen Produktnummer abzurufen. Dies zeigt, wie synchrone Aufrufe innerhalb eines asynchronen Listeners durchgeführt werden können. Es beinhaltet Fehlerbehandlung für potenzielle Probleme während des Abrufs von Produktinformationen oder beim Parsen der Produktnummer. Dieser Ansatz macht den Listener widerstandsfähiger und befähigt ihn zur Interaktion mit anderen Diensten, wodurch seine Funktionalität und Zuverlässigkeit in einer Mikroservices-Architektur verbessert wird.

Feign-Client

```
@FeignClient(name = "inventory-service", fallback = InventoryClientFallback.class)
public interface InventoryClient {

    @GetMapping("/products/{id}")
    Product getProductById(@PathVariable("id") Long id);

    @PostMapping("/products")
    Product createProduct(@RequestBody Product product);

    @PutMapping("/products/{id}")
    Product updateProduct(@PathVariable("id") Long id, @RequestBody Product product);

    @DeleteMapping("/products/{id}")
    void deleteProduct(@PathVariable("id") Long id);
}
```

Der erweiterte Feign-Client ermöglicht es uns, nicht nur Produkte abzurufen, sondern auch neue Produkte zu erstellen, vorhandene zu aktualisieren und zu löschen. Durch die Verwendung eines Fallback-Clients stellen wir sicher, dass unser Service auch dann noch funktioniert, wenn der Inventory-Dienst vorübergehend nicht verfügbar ist. Dies erhöht die Robustheit unserer Anwendung.

Die Integration mit dem Service Discovery (z.B. Eureka) ermöglicht es unserem Feign-Client, dynamisch die Adresse des Inventory-Dienstes zu finden, was besonders nützlich ist in Umgebungen mit häufig wechselnden Netzwerkkonfigurationen oder bei Skalierungsvorgängen.

Durch die Konfiguration von benutzerdefinierten Encodern und Decodern können wir die Art und Weise steuern, wie Daten zwischen unseren Services übertragen werden, was uns eine grössere Flexibilität bei der Handhabung verschiedener Datentypen und Formate gibt.

Diese Verbesserungen machen unseren Feign-Client nicht nur mächtiger, sondern auch besser geeignet für die Nutzung in komplexen Mikroservice-Architekturen, indem sie eine effiziente und zuverlässige Kommunikation zwischen den Services gewährleisten.

Cricuit-Breaker-Pattern & Messaging mit Kafka

```
@Configuration
public class CircuitBreakerConfig {

    @Bean
    public CircuitBreaker circuitBreaker() {
        CircuitBreakerConfig config = CircuitBreakerConfig.custom()
                .failureRateThreshold(50)
                .waitDurationInOpenState(Duration.ofMillis(10000))
                .ringBufferSizeInHalfOpenState(10)
                .ringBufferSizeInClosedState(20)
                .build();

        return CircuitBreaker.of("backendService", config);
    }
}
```

```java
@Service
public class ProductService {

    private final CircuitBreaker circuitBreaker;
    private final KafkaTemplate<String, Product> kafkaTemplate;

    public ProductService(CircuitBreaker circuitBreaker, KafkaTemplate<String, Product> kafkaTemplate) {
        this.circuitBreaker = circuitBreaker;
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendProductMessage(Product product) {
        circuitBreaker.executeRunnable(() -> {
            kafkaTemplate.send("product-topic", product);
        });
    }
}
```

```java
@Configuration
public class KafkaProducerConfig {

    @Value("${kafka.bootstrap.servers}")
    private String bootstrapServers;

    @Bean
    public ProducerFactory<String, Product> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
        return new DefaultKafkaProducerFactory<>(configProps);
    }

    @Bean
    public KafkaTemplate<String, Product> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

In diesem Setup verwenden wir Resilience4j, um ein Circuit Breaker Pattern zu implementieren, das automatisch erkennt, wann ein Service fehlschlägt, und entsprechende Massnahmen ergreift, um weitere Fehler zu verhindern. Der Circuit Breaker öffnet sich nach einer bestimmten Anzahl von Fehlern und blockiert alle weiteren Anfragen für eine festgelegte Dauer, bevor er versucht, wieder zu schliessen.

Gleichzeitig nutzen wir Spring Kafka, um Nachrichten zwischen verschiedenen Teilen unserer Anwendung zu übertragen. Die Kombination dieser beiden Technologien ermöglicht es uns, ein robustes und widerstandsfähiges System zu erstellen, das sowohl in der Lage ist, mit temporären Ausfällen umzugehen, als auch effizient Nachrichten zu übertragen.

Durch die Integration des Circuit Breakers direkt in den Service, der Kafka-Nachrichten sendet, stellen wir sicher, dass unsere Nachrichtenübertragung ebenfalls vor Überlastung geschützt ist und dass wir proaktiv auf mögliche Probleme reagieren können, bevor sie kritische Auswirkungen auf unser gesamtes System haben.

Eureka-Service-Discovery und Gateway

```
eureka.client.serviceUrl.defaultZone=http://{eureka-host}:{port}/eureka/
```

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

Application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: product_service_route
        uri: lb://PRODUCT-SERVICE
        predicates:
        - Path=/products/**
```

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

```properties
server.port=8761
eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false
```

Mit diesen Schritten haben Sie ein Gateway eingerichtet, das als Eureka Client fungiert und automatisch Services entdeckt, die sich beim Eureka Server registrieren lassen. Das Gateway leitet eingehenden Datenverkehr basierend auf definierten Routen an die entsprechenden Mikroservices weiter. Durch die Integration von Eureka ermöglichen Sie Ihrem Gateway, dynamisch auf Änderungen in der Service-Landschaft zu reagieren, wie z.B. das Hinzufügen neuer Service-Instanzen oder das Entfernen von nicht mehr aktiven Instanzen. Dies trägt wesentlich zur Skalierbarkeit und Wartbarkeit Ihrer Mikroservice-Architektur bei.

Sicherheitsaspekte

Für ein sicheres Cookie-Shop-Erlebnis sind folgende Kernpunkte entscheidend:

1. **Verschlüsselung**: Alle Datenübertragungen, insbesondere solche mit persönlichen Informationen, müssen über HTTPS erfolgen, um die Daten während der Übertragung zu schützen.

2. **Regelmässige Updates**: Software und Systeme müssen regelmässig aktualisiert werden, um bekannte Sicherheitslücken zu schliessen.

3. **Zugangskontrolle**: Starke Passwörter und zwei-Faktor-Authentifizierung für den Zugang zu Konten und Backend-Areas sind erforderlich, um unbefugten Zugriff zu verhindern.

4. **Backup und Datensicherheit**: Regelmässige Backups aller wichtigen Daten sowie die Einhaltung der DSGVO-Anforderungen für die Speicherung und Verarbeitung personenbezogener Daten sind unerlässlich.

5. **Schutz vor externen Netzwerken**: Bei der Arbeit unterwegs oder im öffentlichen WLAN sollten zusätzliche Sicherheitsmassnahmen wie VPNs verwendet werden, um die Datenintegrität zu gewährleisten.

Diese Massnahmen bilden die Grundlage für ein sicheres Online-Shop-Umfeld, indem sie sowohl die physische als auch die digitale Sicherheit abdecken.

Reflexion

Trotz der begrenzten Zeit, die uns zur Verfügung stand, haben wir intensiv daran gearbeitet, unser Maximum aus dieser Situation herauszuholen. Wir haben als Trio effizient zusammengearbeitet, um die Aufgabenliste zu bewältigen und dabei wertvolle Erfahrungen gesammelt. Zu unseren Ressourcen gehörten sowohl formelle Dokumentationen als auch praktische Unterstützung durch ChatGPT für die Korrektur von Texten.

Trotz der begrenzten Zeit, die uns zur Verfügung stand, haben wir intensiv daran gearbeitet, unser Maximum aus dieser Situation herauszuholen. Wir haben als Trio effizient zusammengearbeitet, um die Aufgabenliste zu bewältigen und dabei wertvolle Erfahrungen gesammelt. Zu unseren Ressourcen gehörten sowohl formelle Dokumentationen als auch praktische Unterstützung durch ChatGPT für die Korrektur von Texten.