Arquitetura de Referência para projetos em Java com Spring framework

Reinaldo Domingos de Freitas Junior

Orientador(a):
Prof. João Paulo Faria

25 de Fevereiro de 2024

Sumário

1. Introdução

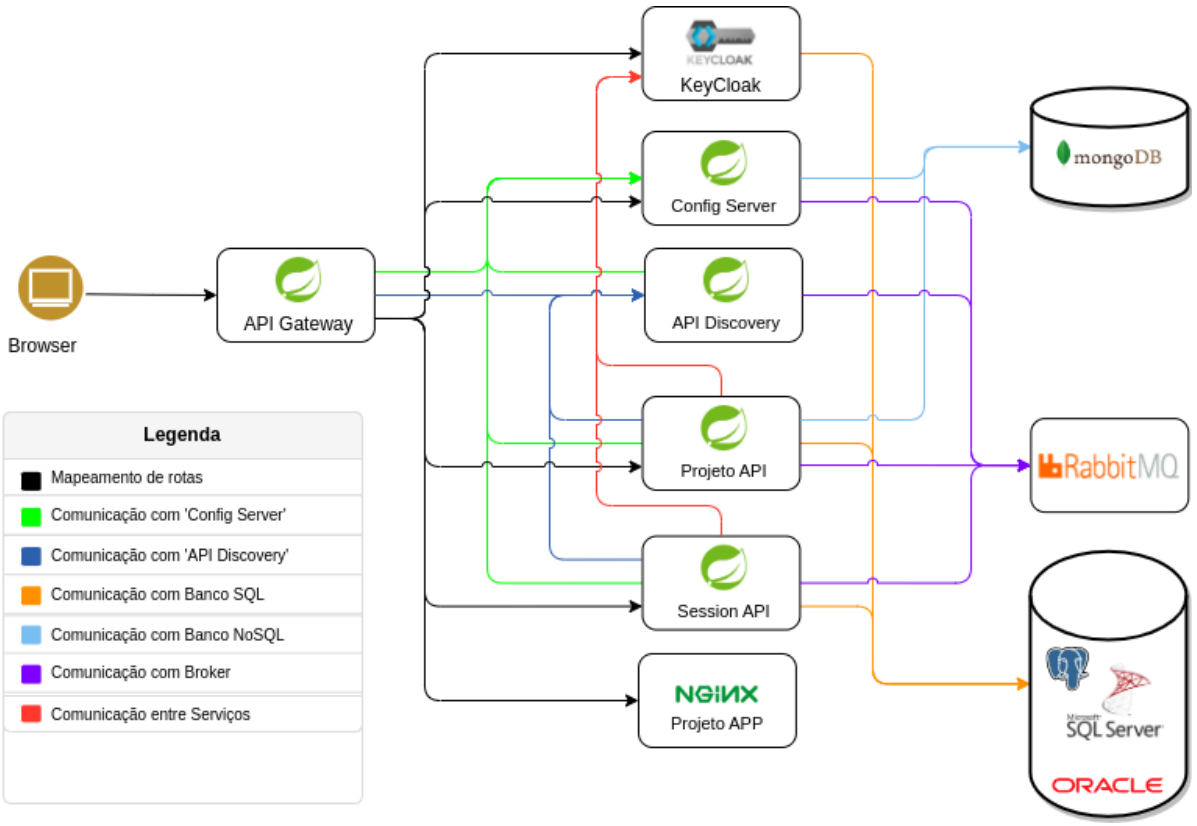
Neste documento, apresentamos uma arquitetura de referência para projetos em Java utilizando o framework Spring, voltada para a construção de sistemas distribuídos e escaláveis. A arquitetura proposta aborda diversos aspectos importantes, desde a estruturação dos microsserviços até a integração com tecnologias como bancos de dados relacionais e não relacionais, sistemas de mensageria e gerenciamento de identidade e acesso.

O objetivo desta arquitetura de referência é fornecer diretrizes e boas práticas para o desenvolvimento de sistemas baseados em microsserviços, utilizando tecnologias modernas e padrões amplamente adotados pela comunidade Java.

2. Diagrama arquitetural

Segue abaixo o diagrama arquitetural que representa a estrutura da arquitetura proposta para um produto chamado "Projeto", que consiste em dois módulos principais: "API" e "APP". Esses módulos são construídos utilizando a arquitetura voltada ao Spring Framework.

O diagrama ilustra as dependências entre os diferentes componentes do sistema, oferecendo uma visão geral da sua estrutura e organização. Vamos examinar detalhadamente cada parte do diagrama para entender como os diferentes componentes se relacionam e interagem entre si.



Componentes Principais no Diagrama:

- Microsserviços
- API Gateway
- API Discovery
- Bancos de Dados
- Config Server
- Mensageria
- Serviço de Autenticação e Autorização (SAA)

Cada um desses componentes será detalhado nos próximos tópicos, fornecendo explicações sobre sua função e importância na arquitetura.

3. API Discovery

O API Discovery é um componente fundamental em arquiteturas de microsserviços. Ele atua como um registro centralizado onde os microsserviços podem se registrar e os consumidores de serviços podem descobrir dinamicamente os serviços disponíveis. A principal função do API Discovery é facilitar a comunicação entre os microsserviços, permitindo que eles se encontrem uns aos outros de forma eficiente e transparente.

3.1. Funcionalidades Principais

Registro de Serviços: Os microsserviços se registram no API Discovery ao serem inicializados. Eles fornecem informações sobre sua localização, endpoints, versões e metadados relevantes.

Descoberta de Serviços: Os consumidores de serviços consultam o API Discovery para descobrir serviços disponíveis. Eles podem procurar serviços por nome, tipo, categoria ou outros critérios e obter informações sobre como acessá-los.

Balanceamento de Carga: O API Discovery pode fornecer funcionalidades de balanceamento de carga, distribuindo as requisições entre várias instâncias de um serviço para melhorar a disponibilidade e o desempenho.

Monitoramento e Resiliência: O API Discovery pode incluir recursos de monitoramento para verificar a saúde dos serviços registrados. Ele também pode implementar estratégias de resiliência, como roteamento inteligente de tráfego e failover, para lidar com falhas e interrupções.

3.2. Tecnologias Utilizadas

Spring Cloud Eureka: Uma implementação do padrão de API Discovery desenvolvida pelo Spring Cloud. O Spring Cloud Eureka simplifica o registro e a descoberta de serviços em um ambiente de microsserviços baseado em Spring Boot.

Netflix Eureka: O Eureka é uma implementação da Netflix do padrão de API Discovery, que pode ser integrada com o Spring Cloud para gerenciar o registro e a descoberta de serviços.

Exemplo de Utilização (Spring Cloud Eureka ou Netflix Eureka):

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

3.3. Benefícios

Desacoplamento: O API Discovery permite que os serviços sejam desenvolvidos e implantados de forma independente, sem a necessidade de configurações estáticas.

Escalabilidade: Facilita a adição ou remoção dinâmica de instâncias de serviços, permitindo que os sistemas se adaptem à carga de trabalho variável.

Resiliência: Ajuda a criar sistemas mais robustos, onde os serviços podem se recuperar automaticamente de falhas e interrupções.

O API Discovery desempenha um papel crucial na construção de arquiteturas de microsserviços flexíveis e escaláveis, permitindo uma comunicação eficiente e dinâmica entre os serviços

4. API Gateway

O API Gateway é um componente essencial em arquiteturas de microsserviços, atuando como um ponto de entrada único para todas as requisições do cliente. Ele desempenha várias funções importantes, incluindo roteamento de requisições, autenticação, autorização, balanceamento de carga e monitoramento.

4.1. Funcionalidades Principais

Roteamento de Requisições: O API Gateway recebe todas as requisições do cliente e as roteia para os serviços apropriados com base em URLs, cabeçalhos ou outros critérios de roteamento.

Autenticação e Autorização: Ele pode executar verificações de autenticação e autorização para garantir que apenas usuários autorizados tenham acesso aos serviços.

Balanceamento de Carga: O API Gateway pode distribuir as requisições entre várias instâncias de um serviço para melhorar a disponibilidade e o desempenho.

Cache de Requisições: Ele pode armazenar em cache respostas de serviços para reduzir o tempo de resposta e minimizar a carga nos serviços.

Logging e Monitoramento: O API Gateway pode registrar informações detalhadas sobre as requisições e respostas para fins de auditoria, monitoramento e análise de desempenho.

4.2. Tecnologias Utilizadas

Spring Cloud Gateway: Uma solução de gateway API construída sobre o Spring Framework, que fornece funcionalidades avançadas de roteamento e filtragem de requisições.

Netflix Zuul: Um gateway API da Netflix, que oferece recursos poderosos de roteamento, filtragem e monitoramento de tráfego.

NGINX: Um servidor web e proxy reverso popular que pode ser configurado como um gateway API para rotear e balancear o tráfego de entrada.

Exemplo de Utilização (Spring Cloud Gateway):

```
@SpringBootApplication
@EnableEurekaClient
public class GatewayApplication {

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

    @Bean
```

```
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("projeto-api", r -> r.path("/projeto/api/**")
                .uri("http://localhost:8081"))
            .route("projeto-app", r -> r.path("/projeto/**")
                .uri("http://localhost:8080"))
            .build();
    }
}
```

Repare que no exemplo, foi adicionado a anotação @EnableEurekaClient à classe de configuração do Spring Cloud Gateway para habilitar o registro do serviço no Eureka.

Com isso, o serviço do Spring Cloud Gateway será registrado no servidor do Eureka como um cliente, permitindo a descoberta de serviço. Certifique-se de que as dependências apropriadas estejam presentes no arquivo pom.xml do seu projeto para habilitar a integração do Eureka Client com o Spring Cloud Gateway.

4.3. Benefícios

Simplicidade na Fronteira: Simplifica a interface entre clientes e microsserviços, ocultando a complexidade da infraestrutura de serviços subjacentes.

Segurança Reforçada: Permite a implementação centralizada de políticas de segurança, incluindo autenticação, autorização e proteção contra ataques.

Escalabilidade e Desempenho: Facilita a escalabilidade horizontal e o balanceamento de carga para lidar com aumentos repentinos no tráfego.

Monitoramento e Análise: Oferece insights valiosos sobre o tráfego de entrada e saída, permitindo a detecção precoce de problemas e a otimização do desempenho.

O API Gateway desempenha um papel crucial na construção de sistemas distribuídos resilientes e seguros, fornecendo uma camada de abstração e controle sobre a infraestrutura de microsserviços subjacentes.

5. Spring Data

O Spring Framework oferece suporte ao Spring Data, que fornece uma abstração de banco de dados para simplificar o desenvolvimento de aplicações que utilizam bancos de dados relacionais. O Spring Data JPA é um dos módulos do Spring Data que permite o acesso a bancos de dados relacionais usando a API de persistência Java (JPA).

5.1. Banco relacional (SQL)

Um banco de dados relacional é um tipo de banco de dados que organiza e armazena dados em tabelas relacionadas. Ele é baseado no modelo relacional, introduzido por Edgar F. Codd na década de 1970, e utiliza um conjunto de conceitos matemáticos e estruturas de dados para representar e manipular informações.

Principais características de um banco de dados relacional:

Tabelas: Os dados são organizados em tabelas, onde cada tabela representa uma entidade ou conceito específico. Cada linha em uma tabela corresponde a uma instância individual dessa entidade, e cada coluna representa um atributo ou característica da instância.

Relacionamentos: Os bancos de dados relacionais permitem definir relacionamentos entre tabelas. Esses relacionamentos são estabelecidos por meio de chaves estrangeiras, que conectam uma tabela a outra com base em campos comuns. Isso permite representar associações entre diferentes entidades.

Integridade referencial: Os bancos de dados relacionais garantem a integridade referencial, o que significa que as relações entre tabelas são mantidas consistentes. Isso é alcançado por meio de restrições de chave estrangeira que garantem que não haja valores órfãos ou inválidos nas relações entre tabelas.

Normalização: A normalização é um processo usado para organizar os dados em tabelas de modo a reduzir a redundância e a inconsistência. Isso envolve dividir grandes tabelas em várias tabelas menores e relacionadas, de modo que cada tabela representa uma única unidade de informação.

SQL (Structured Query Language): A linguagem SQL é a linguagem padrão para interagir com bancos de dados relacionais. Ela permite realizar operações como consultas (queries), inserções, atualizações e exclusões de dados, além de definir estruturas de tabelas, índices e restrições.

ACID: Os bancos de dados relacionais seguem o princípio ACID (Atomicidade, Consistência, Isolamento e Durabilidade), que garante a confiabilidade e a consistência das transações realizadas no banco de dados, mesmo em condições de falha.

Em resumo, um banco de dados relacional oferece uma estrutura organizada e flexível para armazenar e manipular dados, permitindo a representação de relacionamentos complexos entre diferentes entidades. Isso o torna uma escolha popular para uma ampla gama de aplicativos e cenários de negócios.

5.2. Configuração do DataSource

O Spring Framework permite a configuração do DataSource, que é responsável por fornecer conexões com o banco de dados. Você pode configurar vários DataSources para diferentes bancos de dados e alternar entre eles com base em perfis de ambiente ou propriedades de configuração.

Exemplo de Configuração com Spring Boot:

```
@Configuration
public class DataSourceConfig {

    @Bean
    @Qualifier("dataSource")
    @ConfigurationProperties(prefix = "spring.datasource")
    public DataSource dataSource() {
        return DataSourceBuilder.create().build();
    }
}
```

Exemplo parametros pelo arquivo application.yml

```
spring:
  datasource:
    url: "jdbc:postgresql://localhost:5432/projeto"
    username: postgres
    password: postgres
    driver-class-name: org.postgresql.Driver
```

Neste exemplo, configuramos um DataSource para PostgreSQL mas poderia ser para qualquer outro banco de dados com suporte por exemplo: SQL Server, Oracle, H2, etc.

Por padrão, cada DataSource é configurado usando as propriedades definidas em application.properties ou application.yml para cada banco de dados, podendo ser configurado múltiplas datasources. Nos próximos tópicos, exploraremos como configurar o projeto para obter os parâmetros de conexão do banco de dados do Config Server. Isso nos permitirá centralizar e gerenciar de forma mais eficiente as configurações do banco de dados em um ambiente distribuído.

6. Banco de dados não relacional (NoSQL)

O termo "NoSQL" se refere a uma categoria de bancos de dados que diferem dos tradicionais bancos de dados relacionais (SQL). Enquanto os bancos de dados relacionais são baseados no modelo relacional e usam SQL como sua linguagem de consulta padrão, os bancos de dados NoSQL são projetados para lidar com volumes massivos de dados não estruturados ou semiestruturados de forma mais eficiente e escalável.

Principais características dos bancos de dados NoSQL:

Modelo de Dados Flexível: Os bancos de dados NoSQL permitem que os dados sejam armazenados de forma flexível, sem a necessidade de uma estrutura rígida de tabelas. Isso significa que você pode armazenar dados heterogêneos e não estruturados em um banco de dados NoSQL.

Escalabilidade Horizontal: Os bancos de dados NoSQL são projetados para escalabilidade horizontal, o que significa que podem lidar com um grande volume de dados distribuindo-os por vários servidores ou nós. Isso permite que eles dimensionem facilmente para atender às demandas de aplicativos de alto desempenho.

Desempenho Elevado: Devido à sua arquitetura distribuída e ao uso eficiente de índices e estruturas de dados otimizadas para leitura e gravação rápida, os bancos de dados NoSQL geralmente oferecem um desempenho muito alto para operações de leitura e gravação em comparação com os bancos de dados relacionais.

Consistência Eventual: Alguns bancos de dados NoSQL adotam o modelo de consistência eventual, onde as atualizações são propagadas assincronamente para garantir a disponibilidade e o desempenho, em detrimento da consistência imediata dos dados. Isso é adequado para aplicativos que podem tolerar uma pequena janela de tempo onde os dados podem estar em um estado não consistente.

Diversos Tipos de Dados: Os bancos de dados NoSQL suportam uma ampla variedade de tipos de dados, incluindo documentos, pares chave-valor, grafos e colunas. Isso permite que eles sejam usados em uma variedade de aplicativos, desde sistemas de gerenciamento de conteúdo até análises de big data.

Alguns exemplos populares de bancos de dados NoSQL incluem MongoDB, Cassandra, Redis e Neo4j. Esses bancos de dados são amplamente utilizados em aplicativos da web, IoT (Internet das Coisas), análise de big data, jogos online e muito mais, devido à sua capacidade de lidar com grandes volumes de dados e sua flexibilidade em relação ao modelo de dados.

6.1. MongoDB

O MongoDB é um banco de dados NoSQL de código aberto e orientado a documentos, desenvolvido para lidar com grandes volumes de dados de forma eficiente e escalável. Ele se destaca por sua flexibilidade, desempenho e capacidade de lidar com dados não estruturados ou semiestruturados.

Principais características do MongoDB:

Modelo de Dados Orientado a Documentos: No MongoDB, os dados são armazenados em documentos no formato JSON (JavaScript Object Notation) ou BSON (Binary JSON). Cada documento é uma unidade independente que pode conter diferentes tipos de dados e estruturas complexas, tornando-o ideal para representar objetos complexos ou hierárquicos.

Flexibilidade: O MongoDB oferece uma flexibilidade excepcional no esquema dos dados. Não há necessidade de seguir uma estrutura de tabela rígida, como nos bancos de dados relacionais. Isso permite que os desenvolvedores adicionem ou removam campos dos documentos conforme necessário, sem afetar o esquema global do banco de dados.

Escalabilidade Horizontal: O MongoDB é projetado para escalabilidade horizontal, o que significa que pode distribuir os dados em vários servidores ou nós para lidar com cargas de trabalho crescentes. Isso permite que o MongoDB escale facilmente para atender às demandas de aplicativos de alto desempenho e de grande escala.

Desempenho Elevado: Graças à sua arquitetura distribuída e ao uso eficiente de índices, o MongoDB oferece um desempenho excepcional para operações de leitura e gravação, mesmo em ambientes de alto volume de dados e tráfego.

Consultas Avançadas: O MongoDB suporta consultas avançadas, incluindo consultas ad hoc, consultas por campo, consultas geoespaciais e consultas de agregação. Ele também suporta índices secundários e índices geoespaciais para melhorar o desempenho das consultas.

Recursos de Alta Disponibilidade e Tolerância a Falhas: O MongoDB oferece recursos avançados de replicação e fragmentação para garantir a alta disponibilidade e a tolerância a falhas do sistema. Ele suporta replicação assíncrona, failover automático e recuperação automática de falhas.

Comunidade Ativa e Ecossistema Rico: O MongoDB possui uma comunidade ativa de desenvolvedores e uma ampla variedade de ferramentas e recursos disponíveis. Ele é suportado por uma vasta gama de plataformas e integrações, facilitando a sua adoção e uso em uma variedade de cenários de aplicativos.

Em resumo, o MongoDB é uma poderosa e flexível solução de banco de dados NoSQL, adequada para uma ampla variedade de aplicativos, desde aplicativos da web e móveis até análises de big data e IoT. Sua capacidade de lidar com grandes volumes de dados e sua flexibilidade no esquema dos dados o tornam uma escolha popular entre os desenvolvedores.

7. Config Server

No ambiente Java com Spring Framework, configurar um Config Server para usar o MongoDB como repositório de configurações é simples.

Dependências Maven

Certifique-se de incluir a dependência do Spring Cloud Config Server e a dependência do MongoDB no arquivo pom.xml do seu projeto:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

Configuração do Config Server usando um arquivo application.yml:

```
spring:
  application:
    name: config-server
  profiles:
    active: mongodb
  cloud:
    config:
      server:
        mongodb:
          uri: "mongodb://localhost:27017/config-database"
          collection: configurations
  profiles:
    active: mongodb
server:
  port: "8888"
```

Nome da aplicação Config Server: config-server
Porta em que o Config Server estará escutando: 8888
URI de conexão com o MongoDB: mongodb://localhost:27017/config-database
Nome do banco de dados: config-database
Nome da coleção no MongoDB onde as configurações serão armazenadas: configurations

7.1. Configuração das Informações de Conexão com o PostgreSQL dentro do MongoDB

Conforme comentado anteriormente, podemos armazenar as informações de conexão com o banco de dados relacional em um documento dentro do MongoDB. Aqui está um exemplo de como você pode fazer isso:

Exemplo JSON para usar no mongo para configuração da conexão com banco de dados PostgreSQL:

```
{
  "spring": {
    "datasource": {
      "url": "jdbc:postgresql://localhost:5432/projeto",
      "username": "postgres",
      "password": "postgres",
      "driverClassName": "org.postgresql.Driver"
    }
  }
}
```

Os demais serviços podem acessar essas configurações usando as ferramentas do Spring Cloud Config. Eles podem então usar as informações fornecidas para se conectar ao banco de dados PostgreSQL durante a inicialização.

Exemplo configuração no arquivo application.yml de uma API para se conectar ao Config Server:

```
spring:
  cloud:
    config:
      uri: "http://localhost:8888"
```

8. Mensageria

A mensageria é um padrão de comunicação entre diferentes sistemas ou componentes de um sistema distribuído, onde as mensagens são usadas para transmitir informações e instruções de forma assíncrona. Isso significa que os sistemas ou componentes não precisam estar ativamente conectados o tempo todo; em vez disso, eles podem enviar mensagens uns aos outros e continuar com suas tarefas sem esperar por uma resposta imediata.

8.1. Principais Conceitos e Componentes

Mensagens: São os pacotes de dados que contêm informações transmitidas entre os sistemas ou componentes. As mensagens podem conter qualquer tipo de informação, desde simples strings até estruturas de dados complexas.

Canais de Comunicação: São os canais ou canais de transporte através dos quais as mensagens são enviadas e recebidas. Esses canais podem ser baseados em diferentes tecnologias, como TCP/IP, HTTP, AMQP (Advanced Message Queuing Protocol), MQTT (Message Queuing Telemetry Transport), entre outros.

Ponto a Ponto vs. Publicação/Assinatura: Na mensageria ponto a ponto, uma mensagem é enviada de um remetente para um destinatário específico. Já na publicação/assinatura, uma mensagem é publicada em um tópico ou canal e pode ser consumida por um ou mais assinantes interessados nesse tópico.

Filas e Tópicos: São estruturas de dados usadas para armazenar mensagens temporariamente até que sejam processadas pelos destinatários. Em uma fila, as mensagens são consumidas por um único destinatário, enquanto em um tópico, as mensagens são distribuídas para todos os assinantes interessados.

8.2. Importância da Mensageria em Arquiteturas de Microsserviços

Desacoplamento: A mensageria permite que os diferentes serviços em uma arquitetura de microsserviços se comuniquem sem conhecer os detalhes de implementação um do outro, promovendo o desacoplamento e a independência entre eles.

Resiliência: Como as mensagens são enviadas de forma assíncrona, os serviços podem continuar funcionando mesmo se outros serviços estiverem temporariamente indisponíveis. Isso aumenta a resiliência do sistema como um todo.

Escala: A mensageria é altamente escalável e pode lidar com grandes volumes de mensagens e picos de tráfego de forma eficiente, permitindo que os sistemas cresçam conforme a demanda.

Flexibilidade: A mensageria oferece flexibilidade na integração de diferentes tecnologias e sistemas heterogêneos, facilitando a comunicação entre eles.

8.3. AMQP

O AMQP, ou Advanced Message Queuing Protocol, é um protocolo de comunicação de mensagens padrão, aberto e altamente escalável, projetado para sistemas de mensageria assíncrona.
Produtores e Consumidores: São componentes responsáveis por enviar e receber mensagens em um sistema de mensageria baseado em AMQP.

Filas e Trocas: Filas são estruturas de dados onde as mensagens são temporariamente armazenadas até serem processadas pelos consumidores. As trocas são componentes que recebem mensagens dos produtores e decidem para qual fila ou filas encaminhá-las.

Roteamento: É o processo de encaminhamento de mensagens das trocas para as filas com base em regras de roteamento predefinidas.

Tópicos: São semelhantes às filas, mas oferecem maior flexibilidade para o roteamento de mensagens com base em padrões de roteamento definidos.

8.4. RabbitMQ

RabbitMQ é um sistema de mensageria de código aberto que implementa o protocolo AMQP (Advanced Message Queuing Protocol). Ele atua como um intermediário entre os produtores de mensagens e os consumidores, permitindo a comunicação assíncrona entre diferentes partes de um sistema distribuído.

Características e Uso do RabbitMQ:

Filas e Trocas: O RabbitMQ usa o conceito de filas e trocas para rotear mensagens dos produtores para os consumidores. Ele oferece flexibilidade na definição de padrões de roteamento, como ponto a ponto, publicação/assinatura e roteamento de tópicos.

Confiabilidade: Oferece garantias de entrega confiável das mensagens, incluindo confirmações de entrega e persistência das mensagens em caso de falha do sistema.

Escalabilidade: É altamente escalável e pode lidar com grandes volumes de mensagens e picos de tráfego de forma eficiente, permitindo que os sistemas cresçam conforme a demanda.

8.5. Integração RabbitMQ com Spring Framework

O RabbitMQ é integrado ao Spring Framework por meio do projeto Spring AMQP, que fornece abstrações e APIs para facilitar o desenvolvimento de aplicativos baseados em RabbitMQ em um ambiente Java com Spring. Isso inclui recursos como templates de envio de mensagens, listeners de recebimento de mensagens e gerenciamento de conexões com o RabbitMQ.

Dependências do RabbitMQ no arquivo pom.xml do maven:

```
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Configuração da conexão com RabbitMQ no arquivo application.yml:

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

Exemplo de Produtor de Mensagens:

```
@Configuration
public class RabbitMQConfig {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Bean
    public Queue queue() {
        return new Queue("minha-fila");
    }

    public void enviarMensagem(String mensagem) {
        rabbitTemplate.convertAndSend("minha-fila", mensagem);
    }
}
```

Exemplo de Consumidor de Mensagens:

```
@Component
public class MensagemReceiver {

    @RabbitListener(queues = "minha-fila")
    public void receberMensagem(String mensagem) {
        System.out.println("Mensagem recebida: " + mensagem);
    }
}
```

9. Keycloak

Keycloak é um sistema de código aberto para gerenciamento de identidade e acesso. Ele oferece recursos de autenticação, autorização e gerenciamento de identidade para aplicativos e serviços, permitindo que você proteja suas aplicações com segurança robusta e padrões modernos de autenticação.

Funcionalidades do Keycloak:

Autenticação: O Keycloak suporta uma variedade de métodos de autenticação, incluindo login com usuário/senha, autenticação de dois fatores, autenticação baseada em token, autenticação social (como Google, Facebook, etc.) e muito mais.

Autorização: Ele fornece um sistema flexível de controle de acesso baseado em políticas, permitindo que você defina quem tem acesso a quais recursos em sua aplicação com base em regras e permissões configuráveis.

Integração com Padrões de Segurança: O Keycloak suporta padrões de segurança modernos, como OAuth 2.0, OpenID Connect, SAML (Security Assertion Markup Language) e JWT (JSON Web Tokens), garantindo interoperabilidade e segurança em suas aplicações.

Gerenciamento de Identidade: Ele oferece recursos abrangentes de gerenciamento de usuários, grupos e papéis, permitindo que você gerencie facilmente a identidade de seus usuários em uma interface administrativa intuitiva.

Single Sign-On (SSO): O Keycloak suporta SSO, permitindo que os usuários façam login uma vez e acesse vários aplicativos e serviços protegidos sem a necessidade de autenticar novamente.

9.1. Utilizando Keycloak com Spring Framework

Com o Spring Framework, é possível integrar o Keycloak para adicionar funcionalidades de autenticação e autorização aos seus aplicativos Java. O Spring Security oferece suporte para integração com o Keycloak por meio do adaptador Keycloak Spring Security, que simplifica a configuração de autenticação e autorização baseadas em Keycloak em aplicativos Spring.

Exemplo de Integração com Spring Security:

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)
public class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
        KeycloakAuthenticationProvider keycloakAuthenticationProvider =
```

```
keycloakAuthenticationProvider();
        auth.authenticationProvider(keycloakAuthenticationProvider);
    }

    @Bean
    public KeycloakConfigResolver keycloakConfigResolver() {
        return new KeycloakSpringBootConfigResolver();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        super.configure(http);
        http.authorizeRequests()
            .antMatchers("/public/**").permitAll()
            .antMatchers("/secured/**").hasRole("USER")
            .anyRequest().authenticated();
    }
}
```

10. Session API

Com o Spring Framework podemos criar uma API de sessão que permite armazenar dados de sessão de forma segura e eficiente. A Session API representada no diagrama pode ser usada para gerenciar sessões de usuário e armazenar informações relevantes, como detalhes do perfil do usuário ou configurações de aplicativos.

Abaixo está um exemplo simples de como você pode usar a Session API em conjunto com o Keycloak para armazenar e recuperar dados de sessão:

```
@Controller
@SessionAttributes("userProfile")
public class UserController {

    @GetMapping("/profile")
    @ResponseBody
    public String getUserProfile() {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        UserProfile userProfile = (UserProfile)
authentication.getPrincipal();
        return userProfile.toString();
    }

    @PostMapping("/updateProfile")
    public String updateProfile(@RequestBody UserProfile userProfile) {
```

```
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        SecurityContextHolder.getContext().setAuthentication(new
UserProfileToken(userProfile, authentication.getAuthorities()));
        return "redirect:/profile";
    }
}
```

Neste exemplo, estamos usando a Session API para armazenar informações do perfil do usuário na sessão e acessá-las em diferentes solicitações.

Com essas configurações, você pode integrar o Keycloak com a Session API em aplicativos Java com o Spring Framework para gerenciar sessões de usuário de forma eficaz e segura.

11. Variáveis de ambientes (Extra 1)

O uso de variáveis de ambiente é fundamental para garantir a flexibilidade, segurança e portabilidade dos aplicativos. Aqui estão algumas razões pelas quais o uso de variáveis de ambiente é importante:

Configuração Dinâmica: As variáveis de ambiente permitem configurar dinamicamente o comportamento do aplicativo sem a necessidade de alterar o código fonte. Isso facilita a configuração do aplicativo em diferentes ambientes, como desenvolvimento, teste e produção, sem a necessidade de recompilação ou redistribuição do código.

Segurança: O uso de variáveis de ambiente para armazenar informações sensíveis, como senhas e chaves de API, ajuda a proteger essas informações de acessos não autorizados. As variáveis de ambiente são geralmente mais seguras do que armazenar informações sensíveis diretamente no código fonte ou em arquivos de configuração.

Portabilidade: Ao utilizar variáveis de ambiente, o aplicativo se torna mais portátil, pois as configurações podem ser facilmente modificadas para se adequar a diferentes ambientes de implantação sem a necessidade de alterações no código. Isso simplifica o processo de implantação e migração do aplicativo entre diferentes ambientes de hospedagem.

Conformidade com Padrões de Implantação: Muitas plataformas e serviços de hospedagem de aplicativos suportam o uso de variáveis de ambiente para configurar aplicativos. Ao seguir as práticas recomendadas de uso de variáveis de ambiente, os aplicativos se tornam mais compatíveis e interoperáveis com uma variedade de ambientes de implantação.

Facilidade de Manutenção: Separar a configuração do aplicativo em variáveis de ambiente torna mais fácil entender, manter e atualizar a configuração do aplicativo. Isso reduz a complexidade do código fonte e facilita a colaboração entre membros da equipe de desenvolvimento.

Em resumo, o uso de variáveis de ambiente é uma prática recomendada para configurar aplicativos de forma flexível, segura e portátil, proporcionando uma melhor experiência de desenvolvimento e implantação.

A seguir, vamos ilustrar como alguns dos exemplos mencionados neste documento podem ser aprimorados para utilizar variáveis de ambiente. É importante notar que, mesmo ao empregar variáveis de ambiente, podemos fornecer um valor padrão para o caso de a variável não ser definida, o qual pode ser especificado imediatamente depois dos dois pontos (:) à frente do nome da variável. Exemplo: “${DATABASE_USERNAME:postgres}”,

Exemplo JSON para usar no mongo para configuração da conexão com banco de dados PostgreSQL:

```
{
  "spring": {
    "datasource": {
```

```
      "url":
"${SPRING_DATASOURCE_URL:jdbc:postgresql://localhost:5432/projeto}",
      "username": "${SPRING_DATASOURCE_USERNAME:postgres}",
      "password": "${SPRING_DATASOURCE_PASSWORD:postgres}",
      "driverClassName":
"${SPRING_DATASOURCE_DRIVER_CLASS_NAME:org.postgresql.Driver}"
    }
  }
}
```

Configuração do RabbitMQ no arquivo application.yml do Spring Boot:

```
spring:
  rabbitmq:
    host: ${RABBITMQ_HOST:localhost}
    port: ${RABBITMQ_PORT:5672}
    username: ${RABBITMQ_USERNAME:guest}
    password: ${RABBITMQ_PASSWORD:guest}
```

12. Docker (Extra 2)

O Docker pode ser usado em todos os tópicos citados neste documento para criar, distribuir e executar aplicativos de forma consistente em diferentes ambientes. O Docker é uma ferramenta poderosa para o empacotamento de aplicativos e todas as suas dependências em contêineres, garantindo a portabilidade e a escalabilidade dos aplicativos.

O uso do Docker oferece uma série de benefícios para o desenvolvimento e implantação de aplicativos, incluindo portabilidade, consistência e escalabilidade. Ao utilizar o Docker, você pode criar e gerenciar seus aplicativos de forma mais eficiente e confiável em diferentes ambientes.

12.1. Utilização do Docker

Criação de Imagens: O Docker permite a definição de arquivos de configuração chamados Dockerfiles, nos quais você especifica como o seu aplicativo deve ser empacotado em uma imagem Docker. Isso inclui a definição das dependências, configurações de ambiente e comandos de inicialização do aplicativo.

Distribuição de Imagens: Uma vez que você tenha criado uma imagem Docker para o seu aplicativo, você pode distribuí-la para outros ambientes facilmente. As imagens Docker são armazenadas em repositórios Docker, como o Docker Hub, onde podem ser compartilhadas e baixadas por outros desenvolvedores.

Execução em Contêineres: As imagens Docker podem ser executadas em contêineres, que são instâncias isoladas do aplicativo e de suas dependências. Os contêineres oferecem consistência no ambiente de execução, garantindo que o aplicativo se comporte da mesma forma em diferentes sistemas operacionais e ambientes de hospedagem.

Orquestração de Contêineres: O Docker também oferece ferramentas para orquestração de contêineres, como o Docker Swarm e o Kubernetes. Estas ferramentas permitem gerenciar e escalar automaticamente os contêineres em um ambiente de produção, garantindo alta disponibilidade e escalabilidade do aplicativo.

Ambientes de Desenvolvimento Consistentes: Utilizando o Docker, você pode criar ambientes de desenvolvimento consistentes, onde todos os desenvolvedores trabalham com as mesmas versões das dependências e configurações de ambiente. Isso reduz as diferenças entre os ambientes de desenvolvimento e produção, minimizando os problemas de compatibilidade.

Implantação Simplificada: Com o Docker, a implantação de novas versões do aplicativo torna-se mais simples e confiável. Você pode construir novas imagens Docker com as atualizações do aplicativo e distribuí-las facilmente para os ambientes de produção, garantindo uma implantação consistente e sem interrupções.