CAPACITAÇÕES

JAVA PARA A PDPJ-Br

JAVA AVANÇADO

E-BOOK 3

Ronaldo Pinheiro Gonçalves Junior

Unifor | JUSTIÇA DO TRABALHO Conselho Superior da Justiça do Trabalho | Tribunal Superior Eleitoral | STJ SUPERIOR TRIBUNAL DE JUSTIÇA | JUSTIÇA 40 | PNUD | JUSTIÇA FEDERAL Conselho da Justiça Federal | CNJ CONSELHO NACIONAL DE JUSTIÇA

INTRODUÇÃO

OBJETIVO DA TRILHA

Utilizar o Sistema de Gerência de Bancos de Dados (SGBD) PostgreSQL para criação e manipulação de bancos de dados; realizar a conexão com uma camada de persistência de dados, utilizando o Spring Data JPA; e introduzir o conceito de microsserviços e filas de mensagens do RabbitMQ.

CONTEÚDOS DA TRILHA

7. Sistema de Gerência de Bancos de Dados (SGBD) PostgreSQL;
8. Acesso a bancos de dados e versionamento de bancos de dados, utilizando FlyWay;
9. Arquitetura de microsserviços e processamento de filas de mensagens, utilizando RabbitMQ.

INDICAÇÃO DE MATERIAL COMPLEMENTAR

INDICAÇÃO 1
Tipo: Tutorial
Título: PostgreSQL SELECT
Link: **https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-select/**

INDICAÇÃO 2
Tipo: Tutorial
Título: Tutorial RabbitMQ – Filas de mensagem
Link: **https://www.rabbitmq.com/tutorials/tutorial-two-java**

7 SISTEMA DE GERÊNCIA DE BANCOS DE DADOS (SGBD) PostgreSQL

7.1 Criação de databases e schemas

Ao utilizar um serviço web, um usuário pode fornecer diversos tipos de dados. Em cenários como esse, é comum esperar que uma parte significativa dos dados seja persistida para uso futuro. Por exemplo, os dados fornecidos pelo usuário sobre um caso judicial: chamamos o local onde os dados estão armazenados de banco de dados, ou BD, componente responsável por manter estruturas de diferentes tipos que podem estar relacionadas entre si.

A manutenção de um banco de dados possui um certo grau de complexidade, pois envolve não apenas a tarefa de criação das estruturas necessárias para comportar os dados provenientes do sistema de *software*, mas também as tarefas de garantir a consistência dos dados, lidar com concorrência, fornecer segurança e assim por diante. No contexto de desenvolvimento de *software*, uma das soluções mais populares, se não a mais popular, é utilizar um Sistema de Gerenciamento de Banco de Dados (SGBD), como o PostgreSQL.

O primeiro passo para realizar a persistência de dados é a criação de um banco de dados, também conhecido em inglês como *database* ou DB. Em seguida, podemos organizar nosso *database* por meio da criação de um ou mais *schemas*, coleções lógicas de objetos dentro do nosso *database*. Os principais elementos de um *schema* são as tabelas. Em bancos de dados relacionais, como o PostgreSQL, uma tabela representa uma entidade do mundo real, com um nome descritivo e uma lista de atributos representados em colunas. Já as linhas de uma tabela são os dados do mundo real. Além disso, podemos representar as relações entre diferentes entidades através do relacionamento entre tabelas. Veja, a seguir, o exemplo de uma tabela denominada “CasosJudiciais”, que tem os atributos número, decisão e descrição:

	numero [PK] integer	decisao "char"	descricao character varying
1	123	A	Descrição do caso 1
2	456	B	Descrição do caso 2
3	789	C	Descrição do caso 3
4	101112	A	Descrição do caso 4

Elaborada pelo autor, 2024.

No PostgreSQL, podemos criar um *database* por meio da execução do comando “CREATE DATABASE”, seguido do nome do banco de dados. Nas próximas seções, teremos um exemplo desse comando. De forma semelhante, podemos criar um ou mais *schemas* através da execução do “CREATE SCHEMA”. Vamos aprender as principais linguagens do PostgreSQL, nas seções a seguir.

7.2 A linguagem SQL (Structured Query Language) do PostgreSQL

A linguagem de consulta estruturada, ou *Structured Query Language* (SQL), é uma linguagem-padrão utilizada para as mais diversas operações em um banco de dados, desde a criação de tabelas, até a remoção de dados de uma tabela específica. A especificação conta com uma linguagem de definição de dados, uma linguagem de consulta de dados, uma linguagem de manipulação de dados, entre outras. Essas linguagens visam servir como um padrão, mas cada solução possui sua própria implementação dessas linguagens. Em outras palavras, podem existir algumas diferenças entre as linguagens SQL de diferentes SGBDs. Neste material, estudaremos o SQL do PostgreSQL.

7.3 SQL DDL (Data Definition Language)

A linguagem de definição de dados, ou *Data Definition Language* (DDL), foi desenvolvida especificamente para estruturação de tabelas. Isto é, definir o que será, quais dados serão persistidos e seus tipos. O comando mais utilizado dessa linguagem é o “CREATE TABLE”.

Veja como podemos criar uma tabela, utilizando esse comando:

```
1  CREATE TABLE CasosJudiciais (
2      numero INT,
3      decisao CHAR(1),
4      descricao VARCHAR(255),
5      PRIMARY KEY (numero)
6  );
```

Elaborada pelo autor, 2024.

Outro comando que faz parte do SQL DDL é o “ALTER TABLE”, usado para fazer alterações em tabelas existentes. Dentre as possíveis modificações estão: adição de uma nova coluna, modificação de uma coluna existente, exclusão de uma coluna, renomeação da tabela etc.

Veja, a seguir, alguns exemplos do comando:

```
1 ALTER TABLE CasosJudiciais ADD dataAbertura DATE;
2 ALTER TABLE CasosJudiciais DROP dataAbertura;
3 ALTER TABLE CasosJudiciais RENAME TO CasoJudicial;
4 ALTER TABLE CasoJudicial RENAME TO CasosJudiciais;
```

Elaborada pelo autor, 2024.

O primeiro comando modifica a tabela para adicionar uma nova coluna “dataAbertura” do tipo “Date”, o segundo remove esta coluna recém-criada, o terceiro modifica o nome da tabela para “CasoJudicial” e o quarto e último comando retorna a tabela ao nome original. Existem ainda outros comandos SQL DDL que são executados com menor frequência, como o “DROP TABLE”, que remove uma tabela. Vale ressaltar, nesse momento, que devemos exercer cautela ao operar em um banco de dados, pois, uma vez que um comando de remoção de tabela seja executado, se não houver um *backup*, os dados serão perdidos.

7.4 SQL DQL (Data Query Language)

A linguagem SQL DQL é representada por um comando único principal – o “SELECT”. Este comando é utilizado para consultar os dados de uma tabela e demais informações do banco de dados. Ou seja, esta linguagem não é utilizada para modificar a estrutura dos dados. O objetivo do uso de um “SELECT” é realizar a leitura de dados.

Um comando “SELECT” geralmente é acompanhado de cláusulas, como: “FROM”, que define onde a consulta é feita; “WHERE”, que define condições que funcionam como um “filtro”; “ORDER BY”, que define a coluna pela qual os resultados devem ser ordenados; e outros, como “GROUP BY”, “COUNT” etc.

```
1 SELECT * FROM CasosJudiciais
2 WHERE decisao = 'I'
3 ORDER BY numero ASC;
```

Elaborada pelo autor, 2024.

No exemplo acima, a linha 1 possui uma cláusula “FROM”, que define o local onde a consulta deve ser feita – tabela “CasosJudiciais”. O símbolo “*” indica que queremos todas as informações, ou todas as colunas, como resultado. É possível substituir esse símbolo pelo nome de uma ou mais colunas, separadas por vírgula, para obter apenas as informações dessas colunas específicas. Na segunda linha, estamos adicionando uma condição, através da cláusula

“WHERE”, definindo que a resposta deve conter apenas os casos judiciais com “decisao” igual a ‘I’. Isto é, casos judiciais cuja decisão foi indeferida. Finalmente, na linha 3, definimos que queremos o resultado ordenado pela coluna “numero”, de forma crescente.

7.5 SQL DML (Data Manipulation Language)

Dentre o conjunto de linguagens SQL, além da DDL e da DQL, temos ainda uma última linguagem, que também é muito utilizada: a linguagem de manipulação dos dados, DML. Os principais comandos que utilizamos nessa linguagem são: “INSERT”, para inserção de dados; “UPDATE”, para atualizar dados existentes; e “DELETE”, para remoção de dados.

No exemplo a seguir, a primeira linha realiza a inserção de um novo caso judicial, na tabela “CasosJudiciais”. Para isso, definimos as colunas nas quais iremos escrever novas informações e, em seguida, na mesma ordem das colunas, informamos os valores respectivos para cada uma delas. Na segunda linha, temos um comando que remove todos os casos judiciais que possuem uma decisão ‘A’ (absolvido) e, na terceira linha, temos uma versão alternativa do comando “INSERT”, para inserção de vários valores ao mesmo tempo. Finalmente, na linha 11, temos um comando que atualiza todos os casos judiciais que têm a decisão ‘A’ (absolvido) para um novo valor ‘C’ (condenado).

```
1  INSERT INTO CasosJudiciais (numero, decisao, descricao) VALUES (1, 'A', 'Caso 1');
2
3  DELETE FROM CasosJudiciais WHERE decisao = 'A';
4
5  INSERT INTO CasosJudiciais (numero, decisao, descricao)
6  VALUES
7      (2, 'I', 'Descrição do caso 2'),
8      (3, 'C', 'Descrição do caso 3'),
9      (4, 'A', 'Descrição do caso 4');
10
11 UPDATE CasosJudiciais SET decisao = 'C' WHERE decisao = 'A';
```

Elaborada pelo autor, 2024.

No caderno de atividades, você vai encontrar uma atividade para instalação do PostgreSQL e execução de todos esses comandos. Vamos agora estudar como construir programas em Java, para acessar um banco de dados.

8 ACESSO A BANCOS DE DADOS E VERSIONAMENTO DE BANCOS DE DADOS, UTILIZANDO FlyWay

8.1 Criação de datasource e conexão a uma instância de SGBD

No contexto de desenvolvimento de *software*, utilizamos um Sistema de Gerenciamento de Banco de Dados (SGBD), para auxiliar as tarefas de criação e manutenção de um banco de dados. Todavia, é possível que um desenvolvedor tenha a necessidade de mudar os dados ou a estrutura de um banco de dados. Em uma equipe com muitos desenvolvedores, isso pode levar a problemas de conflito ou inconsistência dos dados.

Da mesma forma que um *software* está sujeito a constantes mudanças, é possível que um banco de dados possa passar por reestruturações, devido a tarefas de manutenção, surgimento de novos requisitos, entre outros motivos. Consequentemente, ao atualizar um banco de dados, durante o desenvolvimento de um *software*, é provável que o responsável pelas mudanças queira notificar os demais interessados sobre as mudanças e pedir que se adequem para evitar maiores problemas.

A ferramenta FlyWay, de maneira análoga ao Git para controle de versão de arquivos de código-fonte, provê uma solução para mudanças de versões em bancos de dados, através de migrações. Por meio da criação de um *datasource*, componente responsável pela conexão a um banco de dados, podemos realizar a conexão a um SGBD, como o PostgreSQL.

No caderno de atividades, temos os passos de instalação, configuração e execução da ferramenta, com um exemplo da atividade de controle de versão de banco de dados, através de migrações para o FlyWay e o SGBD PostgreSQL.

8.2 Persistência de dados, utilizando Spring Data JPA, Hibernate e JDBC

Durante o processo de desenvolvimento de um *software*, em determinado momento, precisamos criar um componente que irá interfacear com um banco de dados. Em Java, podemos criar uma classe que se conecta com um SGBD, como, por exemplo, o Postgre. Todavia, existem muitas formas de se concretizar essa conexão entre código-fonte e banco de dados.

A JDBC é uma API do Java que permite que sistemas de *software* se conectem a um banco de dados. Podemos escrever uma classe Java que terá os atributos de configuração de conexão e que exercerá o papel de *datasource*. Nesta classe, poderemos utilizar um objeto de conexão a um banco de dados, para realizar diversas operações SQL, como “SELECT”, “UPDATE”, “DELETE” e assim por diante. Dessa forma, podemos escrever comandos SQL, dentro do código Java. Veremos um exemplo dessa abordagem em breve, no método “findAll”, nas próximas seções.

Uma alternativa muito popular que também realiza a persistência de dados em Java é o Hibernate. Este *framework*, *Object Relational Mapping* (ORM), permite que desenvolvedores escrevam códigos de persistência de dados que utilizam objetos para comunicação com um banco de dados. Dessa forma, o desenvolvedor não precisa escrever comandos SQL. Em vez disso, utiliza os objetos que estão em sua disposição para inserir novos dados no banco de dados, atualizar dados existentes, remover dados etc.

Apesar de possuir menor grau de flexibilidade quanto aos comandos SQL executados no banco de dados, o Hibernate atua como uma camada entre o código orientado a objetos e o banco de dados, permitindo que o desenvolvedor se abstraia da camada de dados e possa focar na programação orientada a objetos.

Por fim, uma opção que é parte do ecossistema Spring é o Spring Data JPA. A Java Persistence API (JPA), que também é uma solução ORM, pode ser combinada ao Spring *framework*, para permitir que desenvolvedores criem repositórios que são extensões às interfaces do Spring Data, facilitando a implementação de operações comumente encontradas em sistemas de *software*. Em outras palavras, o uso do Spring Data JPA tende a simplificar significativamente o código produzido e melhorar a produtividade dos desenvolvedores de um modo geral, devido aos recursos fornecidos. Nas próximas seções, veremos, em mais detalhes, esses importantes conceitos e suas aplicações no contexto de casos judiciais.

8.3 Utilização do Repository

Para utilizar um repositório em Java, podemos criar uma interface que atua como ponto de extensão para o Spring Data JPA. Mais especificamente, podemos criar uma interface que utiliza a anotação @Repository e estende a interface JpaRepository ou CrudRepository. Veja o exemplo para casos judiciais:

```
1 @Repository
2 public interface CasoJudicialRepository extends JpaRepository<CasoJudicial, Long> {
3     CasoJudicial findByNumero(String numero);
4 }
```

Elaborada pelo autor, 2024.

Neste exemplo, através da criação dessa interface, temos acesso a todas as operações de criação, atualização, consulta e remoção de casos judiciais. Além disso, podemos criar métodos específicos, como ”findByNumero”, conforme exemplificado na linha 3.

O caderno de atividades traz uma prática para configuração de um projeto, para fazer uso dos recursos Spring Data JPA e PostgreSQL.

8.4 Entities e DTO (Data Transfer Objects), utilizando MapStruct

Dois conceitos importantes da programação orientada a objetos no contexto de persistência de dados são as *entities* e os *Data Transfer Objects* (DTO). Entidades (*entities*) são classes Java simples, que representam elementos do domínio do sistema – como CasoJudicial, ParteInteressada etc. – enquanto DTOs são classes baseadas em entidades, mas que focam apenas na transferência de dados pelos *controllers*, sem se preocupar com lógica de negócio.

```java
@Entity
@Table(name = "CasosJudiciais")
public class CasoJudicial {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private int numero;
  private char decisao;
  private String descricao;
  //...
}

public class CasoJudicialDTO {

  private int numero;
  private char decisao;
  private String descricao;
  //...
}
```

Elaborada pelo autor, 2024.

Em sistemas de *software* simples, podemos dizer que o uso de entidades é suficiente. Todavia, quanto maior for o sistema de *software*, mais exposta será a lógica de negócio enquanto entidades forem usadas diretamente. Em um cenário ideal, é do interesse dos desenvolvedores esconder essa complexidade através de DTOs. Em outras palavras, utilizamos objetos entidades, dentro do nosso serviço, mas, no momento de transferência de dados, como, por exemplo, no retorno de um método de um *controller*, optamos por utilizar um objeto DTO.

Como esses dois conceitos estão conectados, uma boa prática é realizar um mapeamento automático entre eles, utilizando, por exemplo, o MapStruct. A interface MapStruct define o mapeamento entre entidades e seus respectivos DTOs. Veja o exemplo da interface CasoJudicialMapper:

```java
@Mapper
public interface CasoJudicialMapper {

    CasoJudicialMapper INSTANCE = Mappers.getMapper(CasoJudicialMapper.class);

    @Mapping(source = "numero", target = "numero")
    @Mapping(source = "decisao", target = "decisao")
    @Mapping(source = "descricao", target = "descricao")
    CasoJudicialDTO casoJudicialToDTO(CasoJudicial casoJudicial);

    @Mapping(source = "numero", target = "numero")
    @Mapping(source = "decisao", target = "decisao")
    @Mapping(source = "descricao", target = "descricao")
    CasoJudicial dtoToCasoJudicial(CasoJudicialDTO casoJudicialDTO);
}
```

Elaborada pelo autor, 2024.

8.5 DAO (Data Access Object)

Da mesma forma que um DTO atua, como uma camada de abstração entre cliente e *controllers*, o *Data Access Object* (DAO) exerce a função de uma camada de abstração entre a camada de dados e a camada de lógica de negócios. O exemplo a seguir mostra a definição de uma interface para CasoJudicialDAO:

```java
public interface CasoJudicialDAO {
    CasoJudicial findById(int id);
    List<CasoJudicial> findAll();
    void save(CasoJudicial casoJudicial);
    void update(CasoJudicial casoJudicial);
    void partialUpdate(int id, CasoJudicial casoJudicial);
    void delete(int id);
}
```

Elaborada pelo autor, 2024.

Uma classe que implementa essa interface servirá de repositório e proverá os métodos concretos para a execução. Essa lista de métodos está relacionada aos métodos que utilizamos nos *controllers*, para requisições HTTP. Segue uma classe exemplo de implementação da interface DAO:

```
1  @Repository
2  public class CasoJudicialDAOImpl implements CasoJudicialDAO {
3
4      @PersistenceContext
5      private EntityManager entityManager;
6
7      //...
8  }
```

Elaborada pelo autor, 2024.

Como a classe CasoJudicialDAOImpl implementa a interface CasoJudicialDAO, teremos que implementar um método para recuperação de dados (GET), um para inserção de dados (POST), dois para atualizações (PUT e PATCH) e um para exclusão de dados (DELETE).

8.5.1 Recuperação de dados (GET)

O primeiro método que iremos implementar é o de recuperação de dados (GET). Para gerenciar nossas entidades, como o CasoJudicial, utilizaremos a API EntityManager. Essa API fornece os métodos básicos de que precisamos e um objeto entityManager é criado e injetado diretamente na nossa classe CasoJudicialDAOImpl.

```
@Override
public List <CasoJudicial> findAll() {
  return entityManager.createQuery("SELECT c FROM CasoJudicial c", CasoJudicial.class).getResultList();
}
```

Elaborada pelo autor, 2024.

Note que o método createQuery realiza uma simples busca na tabela CasoJudicial. Vale ressaltar que, quando criamos nosso banco de dados, colocamos o nome de “CasosJudiciais” para a tabela. Um simples comando de alteração de nome nos permite usar a tabela existente. Uma vez que o método findAll é executado, a nossa classe DAO acessará o banco de dados e trará uma lista de resultados. Dessa forma, nosso controlador pode construir uma resposta e um DTO, contendo os resultados dessa execução.

8.5.2 Utilização de Query Parameters

É possível utilizar parâmetro de consulta para filtrar ou restringir os itens de interesse. Por exemplo, podemos estar interessados em casos judiciais que tenham data de abertura após determinada data ou em casos que tenham uma decisão indeferida. Nesses casos, nosso controlador poderá receber um ou mais *query parameters* ou parâmetros de consulta.

```
@Override
public CasoJudicial findById(int id) {
  return entityManager.find(CasoJudicial.class, id);
}
```

Elaborada pelo autor, 2024.

Para cada uma dessas consultas especializadas, podemos definir um novo método na nossa interface CasoJudicialDAO. No exemplo acima, estamos implementando um método por meio do qual realizamos a recuperação de dados baseada no número do caso judicial.

8.5.3 Paginação de resultados

Além de um método para recuperar um dado específico por ID ou outro de seus atributos, podemos ter métodos adicionais para recuperação de dados. Um deles envolve a paginação de resultados. É provável que o sistema de *software* do tribunal, com o passar do tempo, mantenha um volume cada vez mais elevado de casos judiciais. Consequentemente, uma única chamada ao método de recuperação de dados "findAll" pode retornar uma quantidade de dados grande demais, que implicará em uma lentidão no processo de transmissão e processamento dos casos judiciais.

```
@Override
public List<CasoJudicial> findAll(int page, int pageSize) {
    int offset = (page - 1) * pageSize;
    return entityManager.createQuery("SELECT c FROM CasoJudicial c", CasoJudicial.class)
            .setFirstResult(offset)
            .setMaxResults(pageSize)
            .getResultList();
}
```

Elaborada pelo autor, 2024.

A criação de um novo método para paginação nos permite indicar quantos casos queremos ver, de uma só vez, em determinado momento. Dessa forma, existe um menor volume de informações sendo processadas e transmitidas. Caso o usuário final queira continuar sua

consulta, é possível “virar uma página” e requisitar um novo subconjunto de dados e assim por diante. O método apresentado é um exemplo de sobrecarga do método de recuperação original, mas com parâmetros para adicionar o número e o tamanho da página.

8.5.4 Ordenação de resultados

No contexto de paginação, é muito comum utilizar uma função de ordenação para os dados, como ordenar os casos judiciais: pelo número, de forma decrescente; pela descrição, de forma alfabética; e assim por diante. Podemos ajustar nosso exemplo de paginação, para incluir dois novos parâmetros de ordenação, um para determinar o atributo utilizado na ordenação e outro para determinar a ordem:

```
@Override
public List<CasoJudicial> findAll(int page, int pageSize, String sortBy, String sortOrder) {
    int offset = (page - 1) * pageSize;
    String queryString = "SELECT c FROM CasoJudicial c ORDER BY c." + sortBy + " " + sortOrder;
    return entityManager.createQuery(queryString, CasoJudicial.class)
            .setFirstResult(offset)
            .setMaxResults(pageSize)
            .getResultList();
}
```

Elaborada pelo autor, 2024.

Note que, em nosso novo método, o parâmetro “sortBy” deve conter um atributo de caso judicial e o “sortOrder” deve conter “ASC” ou “DESC”, para ascendente ou descendente, respectivamente.

8.5.5 Inserção de dados (POST)

Para realizar a persistência de um novo caso judicial, devemos implementar o método save da interface DAO. Nesse caso, uma vez que uma requisição do tipo POST seja enviada, com um novo caso judicial em seu corpo, o controlador irá repassar um objeto do tipo da nossa entidade CasoJudicial, para que seja salvo no banco de dados. O código abaixo mostra como seria a implementação do método de inserção de dados:

```
@Override
public void save(CasoJudicial casoJudicial) {
  entityManager.persist(casoJudicial);
}
```

Elaborada pelo autor, 2024.

8.5.6 Atualização de dados (PUT e PATCH)

Vimos também que as operações de atualização de dados podem ser realizadas de duas formas: atualização do objeto ou atualização parcial. Essas operações são implementadas, respectivamente, pelos métodos update e partialUpdate, no seguinte exemplo:

```
@Override
public void update(CasoJudicial casoJudicial) {
  entityManager.merge(casoJudicial);
}

@Override
public void partialUpdate(int id, CasoJudicial casoJudicial) {
  CasoJudicial existingCasoJudicial = findById(id);
  if (existingCasoJudicial != null) {
    if (casoJudicial.getDescricao() != null) {
      existingCasoJudicial.setDescricao(casoJudicial.getDescricao());
    }
    // ... (demais campos, como descrição e decisão)
    entityManager.merge(existingCasoJudicial);
  }
}
```

Elaborada pelo autor, 2024.

Note que recebemos um parâmetro identificador no caso de uma atualização parcial e fazemos o reuso da recuperação de dados parametrizada findById, para localizar o casoJudicial existente e atualizar cada um de seus campos. Apesar da abordagem mostrada pelo exemplo apresentado ser comum, a operação de atualização parcial pode ser feita também em maior granularidade, através da criação de um método para cada atributo.

8.5.7 Exclusão de dados (DELETE)

Finalmente, para a operação de exclusão de dados, implementaremos o método delete, que recebe o identificador de um caso judicial e tenta localizá-lo no banco de dados. Se esse caso judicial existir, iremos removê-lo do banco de dados.

```
@Override
public void delete(int id) {
  CasoJudicial casoJudicial = findById(id);
  if (casoJudicial != null) {
    entityManager.remove(casoJudicial);
  }
}
```

Elaborada pelo autor, 2024.

8.6 Gerência de transações de banco de dados

Ao realizar uma transação de banco de dados, que pode ser definida como uma sequência de operações de banco de dados, é necessário garantir a consistência dos dados. Em outras palavras, é necessário manter a integridade do banco de dados antes e depois da execução das operações. Além disso, as operações de uma transação só podem ser, de fato, executadas e persistidas se a última operação da transação obtiver sucesso, senão elas devem ser "desfeitas". Por exemplo, ao realizar o pagamento de uma taxa processual através de transferência bancária, não podemos realizar o débito se o depósito não é bem-sucedido.

Existem alguns outros requisitos ao realizar o gerenciamento de transações de banco de dados, como isolamento e durabilidade. Mas, em um projeto com Spring Data JPA, esse processo é simplificado, pois podemos utilizar a anotação @Transactional e aproveitar o suporte do *framework* para o gerenciamento. Ao escrever essa anotação de um método, como no "findAll", por exemplo, estamos indicando que este método deve ser executado dentro de uma transação. Dessa forma, o desenvolvedor não precisa gerenciar a transação e pode focar na implementação dos métodos.

8.7 Tratamento de exceções

Da mesma forma que realizamos o tratamento de exceções em Java para diversos cenários, devemos implementar um tratamento para cada um de nossos métodos DAO. O código a seguir traz o exemplo de uso de um bloco “try-catch”, caso ocorra um erro ao acessar os dados no método “findAll”:

```
@Override
@Transactional(readOnly = true)
public List<CasoJudicial> findAll() {
    try {
        return entityManager.createQuery("SELECT c FROM CasoJudicial c", CasoJudicial.class).getResultList();
    } catch (DataAccessException e) {
        throw new RuntimeException("Erro de acesso a dados ao buscar casos judiciais: " + e.getMessage(), e);
    }
}
```

Elaborada pelo autor, 2024.

8.8 Operação via linha de comando

Em nosso caderno de atividades, temos uma atividade para operações de acesso a banco de dados e versionamento, utilizando a ferramenta FlyWay Desktop, mas podemos realizar as mesmas operações via linha de comando[1].

Após completar a atividade, crie um novo projeto e efetue o versionamento do banco de dados via linha de comando:

- flyway baseline: para estabelecer uma versão inicial de um banco de dados existente;
- flyway info: para exibir os estados das migrações do banco de dados;
- flyway clean: para limpar o esquema do banco de dados;
- flyway create: para criar uma migração para receber comandos SQL;
- flyway undo: para desfazer a última migração;
- flyway migrate: para efetuar uma migração para a última versão.

[1] *Site* oficial com a lista de todos os comandos: https://documentation.red-gate.com/flyway/flyway-cli-and-api/commands

8.9 Operação via Apache Maven

Para realizar essas operações via Maven, primeiro precisamos adicionar a dependência ao projeto:

```
<dependency>
<groupId>org.flywaydb</groupId>
<artifactId>flyway-core</artifactId>
<version>10.10.0</version>
</dependency>
```

Devemos também adicionar o plugin, para que o Maven possa escanear e achar o local das nossas migrações FlyWay para o PostgreSQL:

```
<plugin>
<groupId>org.flywaydb</groupId>
<artifactId>flyway-maven-plugin</artifactId>
<version>10.10.0</version>
<configuration>
<url>jdbc:postgresql://localhost:5432/seu_banco_de_dados</url>
<user>postgresql</user>
<password>Seguro10!</password>
<locations>
<location>classpath:db/migration</location>
</locations>
</configuration>
</plugin>
```

> Lembre-se de substituir o campo senha, pelo valor configurado em seu banco de dados.

Em seguida, crie um diretório na raiz do projeto, para guardar os scripts SQL de migração. Para este exemplo, iremos utilizar “db/migration”. Ao criar arquivos de migração, utilize a convenção do FlyWay, para nomear os scripts: “Vx__y.sql”, substituindo “x” pelo número atual e “y” pela descrição, como “V2__Adicao Coluna dataAbertura Tabela CasosJudiciais.sql”.

Uma vez que esteja com o processo configurado, você pode utilizar os mesmos comandos da linha de comando via Maven, como “mvn flyway:migrate”, “mvn flyway:clean” etc.

9 ARQUITETURA DE MICROSSERVIÇOS E PROCESSAMENTO DE FILAS DE MENSAGENS, UTILIZANDO RabbitMQ

9.1 O que são microsserviços?

O desenvolvimento de *software* e suas disciplinas estão sob constante modernização. O advento de tecnologias inovadoras traz novas abordagens para o desenvolvimento de soluções tecnológicas. Muitos projetos de *software* tomam como premissa uma alta disponibilidade da Internet, em diferentes camadas da aplicação, desde a interação com os usuários finais até a comunicação entre os componentes internos do sistema de *software*. Neste contexto, a arquitetura de microsserviços se torna extremamente popular.

No estilo arquitetural monolítico, todos os componentes de *software* são desenvolvidos em um único local, utilizando uma única linguagem de programação, um único sistema gerenciador de banco de dados e assim por diante. Este projeto unificado fornece uma visão clara de como os componentes estão conectados e um maior controle devido à centralização das tecnologias. Todavia, caso a equipe de desenvolvimento deseje implementar uma nova funcionalidade ou alterar ou remover uma funcionalidade existente, o projeto, como um todo, pode ser impactado.

Em uma arquitetura de microsserviços, cada funcionalidade é desenvolvida como um serviço individual. Dessa forma, ao implementar uma nova funcionalidade, não há necessidade de modificar outras funcionalidades. Em outras palavras, podemos dizer que, em um nível arquitetural, nosso sistema de *software* possui um baixo nível de acoplamento e um elevado nível de modularidade e coesão.

9.2 Princípios fundamentais do projeto de microsserviços

Um princípio importante do projeto de microsserviços é que a comunicação entre serviços acontece via APIs, como as que aprendemos anteriormente. Dessa forma, um microsserviço não sabe os detalhes de outros microsserviços, apenas quais funcionalidades estão disponíveis e como acessá-las. Ou seja, cada serviço pode ser implementado, testado e implantado de forma independente. Devido ao baixo acoplamento e à alta modularidade dessa abordagem, existe uma maior manutenibilidade e extensibilidade quando comparada a outras abordagens, como a arquitetura monolítica.

Outro princípio importante do projeto de microsserviços está relacionado ao requisito de escalabilidade. Em uma arquitetura monolítica, se o servidor banco de dados, o servidor de aplicação ou outro componente estiverem sobrecarregados, a aplicação, como um todo, deve “crescer”, para acomodar a nova demanda. Em uma arquitetura microsserviços, se uma funcionalidade está sobrecarregada, apenas aquele serviço deve “crescer”, para atender ao novo volume de requisições.

Finalmente, como os microsserviços possuem um certo grau de isolamento, o sistema, como um todo, possui maior tolerância a falhas, pois defeitos em um microsserviço, em geral, não impactam os demais microsserviços do sistema.

9.3 Configuração

Para realizar a comunicação entre microsserviços, como implementaremos uma comunicação assíncrona, utilizaremos o RabbitMQ para processamento de filas de mensagens. O RabbitMQ é um sistema de mensageria que realiza a complexa tarefa de gerenciar múltiplas mensagens assíncronas entre microsserviços distintos.

Nosso projeto Spring Boot com Spring Data JPA pode ser configurado facilmente para incluir o RabbitMQ. Como estamos utilizando um projeto Maven, podemos adicionar a dependência diretamente no arquivo “pom.xml”. Além disso, como iremos implementar, nas seções seguintes, um exemplo de código que produz uma mensagem para a fila e um que consome, precisamos criar um arquivo de configuração Java para a fila de mensagens. No caderno de atividades, temos uma prática que realiza todos esses passos de configuração.

9.4 Filas Producer e Consumer

Em uma arquitetura de microsserviços com fila de mensagens, temos os conceitos de produtor e consumidor. O produtor, ou Producer, é aquele que envia uma mensagem. O código a seguir é um exemplo de Producer, para nosso projeto Spring Boot:

```
@Component
public class Producer {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private Queue queue;

    public void send(String order) {
        rabbitTemplate.convertAndSend(this.queue.getName(), order);
    }
}
```

Elaborada pelo autor, 2024.

Já o consumidor, ou Consumer, é aquele responsável por consumir as mensagens. Segue um exemplo simples de Consumer, para o nosso projeto Spring Boot, em que a mensagem recebida é impressa no terminal:

```
@Component
public class Consumer {

    @RabbitListener(queues = {QueueConfig.NOME_FILA})
    public void receive(@Payload String fileBody) {
        System.out.println("RECEBI ESSA MENSAGEM: " + fileBody);
    }

}
```

Elaborada pelo autor, 2024.

Tente executar esse código e veja a resposta no console. Vale lembrar que esses exemplos só poderão ser executados se você realizou a configuração do RabbitMQ, conforme a respectiva tarefa do caderno de atividades.

9.5 Reprocessamento de filas

Agora que temos uma fila de mensagens implementada, podemos aprimorar o código conforme necessário. Uma possível melhoria que já podemos aplicar é o reprocessamento de filas. É possível que algumas mensagens falhem, exigindo que elas sejam reenviadas. Todavia, podemos implementar, diretamente no nosso código de Consumer, uma solução para reprocessamento.

Uma forma muito simples de reprocessamento de filas, porém bastante utilizada, é o reencaminhamento manual. Nesse caso, ao detectar uma falha no consumidor, como durante a captura de uma exceção, a mensagem é registrada em um log para que seja enviada novamente de forma manual. Outra solução seria a de tentativas repetidas automáticas, quando configuramos, no RabbitMQ, um processo de reenvio no caso de falhas, com um valor para número máximo de vezes e intervalo entre tentativas. Finalmente, podemos fazer a configuração de uma fila exclusiva para mensagens falhas, ou até mesmo salvar mensagens falhas, em um banco de dados para reprocessamento.

Antes de seguir com o conteúdo, realize todas as tarefas do caderno de atividades, para que o projeto esteja completamente configurado para uma arquitetura microsserviços, com as tecnologias estudadas nesta trilha de aprendizagem. Na próxima trilha, veremos a execução de APIs REST, utilizando Docker Containers, documentação de APIs REST e versionamento de APIs REST; criação e manutenção de logs de execução, utilizando Apache Log4j2; e execução de APIs REST, utilizando Amazon Elastic Kubernetes Service (Amazon EKS).

REFERÊNCIAS

BRETET, A. **Spring MVC Cookbook**: Over 40 Recipes for Creating Cloud-ready Java Web Applications With Spring MVC. 1ª edição. Editora Packt Pub Ltd, 2016.

CARNELL, J. **Spring Microservices in Action**. 1ª edição. Editora Manning, 2017.

CIOLLI, G. **PostgreSQL 16 Administration Cookbook**: Solve real-world Database Administration challenges with 180+ practical recipes and best practices. 1ª edição. Editora Packt Publishing Ltd, 2023.

GUTIERREZ, F. **Pro Spring Boot 2**: An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices. 2ª edição. Editora Apress, 2018.

NEWMAN, S. **Building Microservices**. 1ª edição. Editora O'Reilly, 2015.

RICHARDSON, C. **Microservices Patterns**: With examples in Java. 1ª edição. Editora Manning, 2018.

ROYAL, P. **Building Modern Business Applications**: Reactive Cloud Architecture for Java, Spring, and PostgreSQL. 1ª edição. Editora Apress, 2022.

VIDELA, A.; WILLIAMS, J. **RabbitMQ in Action**: Distributed Messaging for Everyone. 1ª edição. Editora Manning, 2012.