

Desenvolvimento Java com

Jdbc

Jpa e Hibernate

Spring Data

Alex Fernando Egidio
Desenvolvedor Java Sênior
<https://www.javaavancado.com>

Sumário

Introdução ao JDBC.....	4
Introdução ao SQL.....	4
Linguagem DDL.....	5
Linguagem DML.....	6
Criando banco de dados.....	7
Criando tabela de dados.....	7
Componentes JDBC.....	8
Classe de conexão com banco.....	9
Interface PreparedStatement.....	10
Por que usar o PreparedStatement?.....	10
A classe de modelo de dados.....	10
Padrão DAO.....	11
Inserindo dados no banco de dados.....	12
Realizando consulta de todos os dados da tabela.....	12
Realizando buscar por ID.....	13
Realizando Update de dados.....	14
Deletando registro no banco de dados.....	14
Considerações finais sobre o JDBC.....	15
O que é JPA?.....	16
História da Especificação JPA.....	16
JPA 2.0 (Java Persistence API).....	17
Configurando Maven com JPA.....	18
Arquivo de configuração do JPA.....	19
Classe de conexão do JPA.....	20
Introdução ao EntityManager.....	20
DaoGeneric JPA.....	21
Anotação @Entity.....	21
Anotação @Id.....	22
Anotação @OneToMany e @ManyToOne.....	22
Anotação @OneToOne.....	23
Anotação @ManyToMany.....	23
Anotação @NaturalId.....	25
Anotação @Table.....	25
Anotação @UniqueConstraint.....	26
Anotação @Version.....	26
Anotação @Colum.....	27
Anotação @GeneratedValue e @SequenceGenerator.....	27
Anotação @GeneratedValue.....	28
Anotação @Transient.....	28
Anotação @Lob.....	29
Retornando a primary key da entidade.....	29
Salvando com JPA (Persist).....	30
Save or Update com JPA (Merge).....	30
Pesquisar um registro no banco de dados.....	31
Deletando registros.....	31
Trazendo lista de dados do banco de dados.....	31

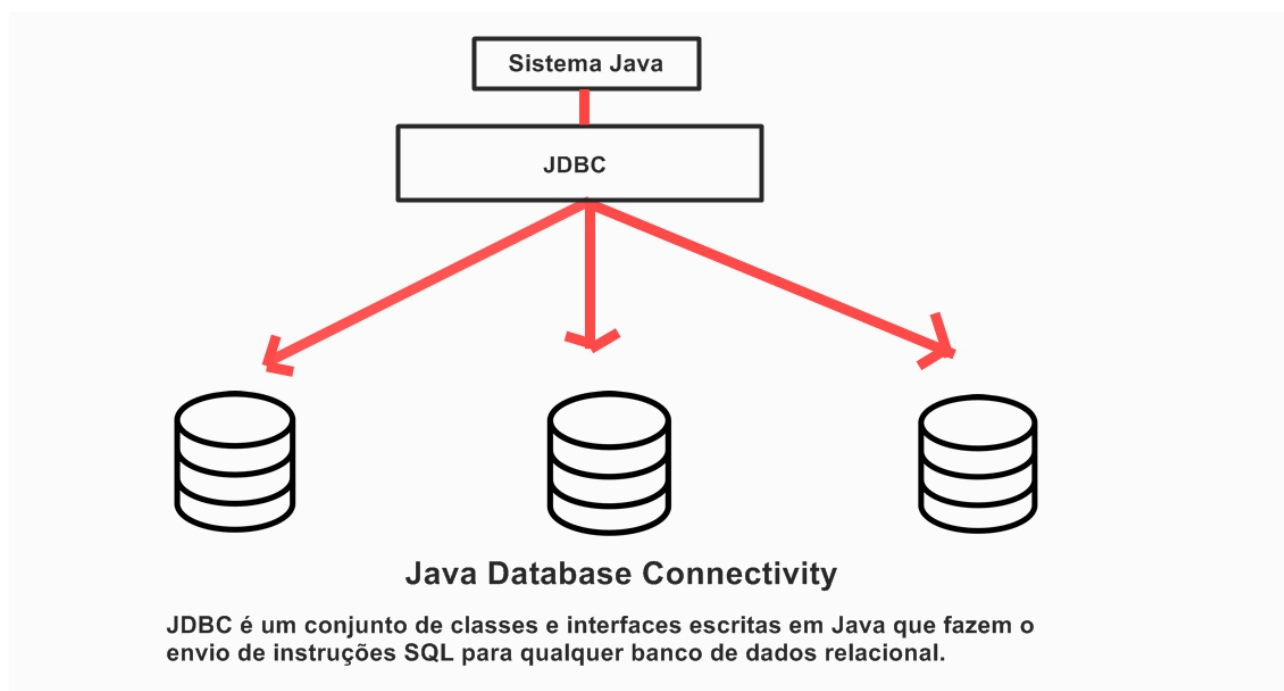
Gravando objeto no banco com JPA.....	32
Consultando objetos no banco de dados.....	32
Atualizando objetos no banco de dados.....	33
Deletando objetos no banco de dados.....	33
Consultando lista de dados.....	34
Carregando uma lista condicional.....	34
Limitando máximo de resultados.....	35
Usando parâmetros dinâmicos.....	36
Operações de média de uma coluna.....	36
Anotação @NamedQuery e @NamedQueries.....	37
Chamando um query nomeada.....	37
JPA - API de critérios.....	38
Bean Validation.....	40
Hibernate Search.....	41
Hibernate Envers.....	42
Considerações finais sobre o JPA.....	42
Spring Data.....	43
Configuração do Maven e Spring Data.....	43
Configurando o Persistence para integrar com Spring.....	44
Configurando o arquivo Spring-Config.xml.....	44
Ativando a auto configuração do Spring.....	44
Configurando a conexão com o banco de dados.....	45
Configurando o JPA integrando com o Spring e o Hibernate.....	45
Controle Transacional e Repository.....	45
Interface CrudRepository.....	46
Anotação @Repository.....	47
Anotação @Transactional.....	47
Anotação @Query.....	48
Anotação @Modifying.....	48
Sobrescrevendo métodos de interface.....	49
Junit com Spring Data.....	49
Anotação @Autowired.....	50
Teste unitário @Test.....	51
Criando o teste de inserir.....	52
Criando o teste de consulta.....	52
Criando o teste de consulta por lista.....	53
Criando o teste de update.....	54
Criando o teste de delete.....	54
Consulta assíncrona.....	55
Usando Sort.....	55
Auditoria.....	56
Metadados de auditoria baseados em anotação.....	57
Considerações finais sobre o Spring Data.....	57
Spring Web MVC.....	58
Spring RESTful.....	58
Spring Boot.....	60
Spring Batch.....	61
Spring Batch Architecture.....	63
Conclusão.....	64
Cursos para você ser profissional em programação.....	64
Referências.....	65

Introdução ao JDBC

Java Database Connectivity ou JDBC é um conjunto de classes e interfaces (API) escritas em Java que fazem o envio de instruções SQL para qualquer banco de dados relacional;

- Api de baixo nível e base para api's de alto nível;
- Amplia o que você pode fazer com Java;
- Possibilita o uso de bancos de dados já instalados;

Para cada banco de dados há um driver JDBC específico que no Java é sempre uma lib/jar que deve ser adicionando ao projeto de acordo com o banco que será usado.



Introdução ao SQL

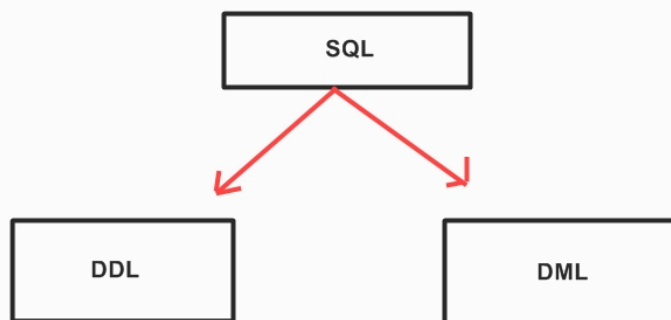
Structured Query Language, ou Linguagem de Consulta Estruturada ou SQL, é a linguagem de pesquisa declarativa padrão para banco de dados relacional (base de dados relacional).

Muitas das características originais do SQL foram inspiradas na álgebra relacional.

O SQL foi desenvolvido originalmente no início dos anos 70 nos laboratórios da IBM em San Jose, dentro do projeto System R, que tinha por objetivo demonstrar a viabilidade da implementação do modelo relacional proposto por E. F. Codd.

SQL

Structured Query Language, ou Linguagem de Consulta Estruturada ou SQL, é a linguagem de pesquisa declarativa padrão para banco de dados relacional. Muitas das características originais do SQL foram inspiradas na álgebra relacional.



Linguagem DDL

Linguagem de definição de dados (LDD ou DDL, do Inglês Data Definition Language) é uma linguagem de computador usada para a definição de estruturas de dados.

O termo foi inicialmente introduzido em relação ao modelo de banco de dados Codasyl, onde o esquema de banco de dados era escrito em uma Linguagem de Definição de Dados descrevendo os registros, campos e "conjuntos" que constituíam o Modelo de dados do usuário.

Inicialmente referia-se a um subconjunto da SQL, mas hoje é usada em um sentido genérico para referir-se a qualquer linguagem formal para descrição de estruturas de dados ou informação, assim como esquemas.

Um exemplo é o CREATE DATA BASE ou então o CREATE TABLE e assim por diante, sempre destinado a definir a estrutura do banco de dados.

Linguagem de definição de dados (DDL)

Linguagem de definição de dados é uma linguagem de computador usada para a definição de estruturas de dados.

Exemplo - Criando banco de dados

```
CREATE DATABASE posjava
WITH OWNER = postgres
ENCODING = 'UTF8'
TABLESPACE = pg_default
CONNECTION LIMIT = -1;
```

Exemplo - Criando Tabela

```
CREATE TABLE posjava (
  id bigint NOT NULL,
  nome character varying(255),
  email character varying(255),
  CONSTRAINT aluno_pkey PRIMARY KEY (id))
```

Responsável por criar a estrutura de banco, tabelas, index e etc do banco de dados.

Linguagem DML

Linguagem de Manipulação de Dados (ou DML, de Data Manipulation Language) é uma família de linguagens de computador utilizada para a recuperação, inclusão, remoção e modificação de informações em bancos de dados.

Pode ser procedural, que especifica como os dados devem ser obtidos do banco; pode também ser declarativa (não procedural), em que os usuários não necessitam especificar o caminho de acesso, isto é, como os dados serão obtidos.

O padrão SQL é não procedural. DMLs foram utilizadas inicialmente apenas por programas de computador, porém (com o surgimento da SQL) também têm sido utilizadas por pessoas.

Dentro desse contexto o mais importante são as cláusulas SELECT, UPDATE, DELETE e INSERT e a mais usada é SELECT que é usada para extrair dados do banco de dados.

Linguagem de manipulação de dados (DML)

Linguagem de Manipulação de Dados é uma família de linguagens de computador utilizada para a recuperação, inclusão, remoção e modificação de informações em bancos de dados.

Exemplo - Insert na Tabela

```
INSERT INTO aluno(  
    id,  
    nome,  
    email)  
VALUES (1,  
    'Alex',  
    'alex.fernando.egidio@gmail.com');
```

Exemplo - Update na tabela

```
UPDATE aluno  
    SET nome= 'Alex Fernando'  
    WHERE id = 1;
```

Responsável por manipular dados em um banco de dados o famoso CRUD Create, Read, Update e Delete que são respectivamente Insert, Select, Atualizar e Delete dos dados do sistema no banco de dados.

Criando banco de dados

Usando a linguagem SQL mais precisamente a DDL que é a linguagem de definição de dados o primeiro passo é a criação do banco de dados.

```
CREATE DATABASE posjava  
    WITH OWNER = postgres  
    ENCODING = 'UTF8'  
    TABLESPACE = pg_default  
    LC_COLLATE = 'Portuguese_Brazil.1252'  
    LC_CTYPE = 'Portuguese_Brazil.1252'  
    CONNECTION LIMIT = -1;
```

Criando tabela de dados

Usando a linguagem SQL mais precisamente a DDL que é a linguagem de definição de dados o segundo passo é a criação da tabela de dados.

```
CREATE TABLE userposjava  
(  
    id bigint,  
    nome character varying,  
    email character varying  
);
```

Componentes JDBC

DriverManager: esta classe gerencia uma lista de drives de banco de dados. Corresponde às solicitações de conexão do aplicativo Java com o driver de banco de dados adequado usando o subprotocolo de comunicação. O primeiro driver que reconhece um determinado subprotocolo no JDBC será usado para estabelecer uma conexão com o banco de dados.

Driver: Essa interface lida com as comunicações com o servidor de banco de dados. Você irá interagir diretamente com objetos Driver muito raramente. Em vez disso, você usa objetos DriverManager, que gerencia objetos desse tipo. Ele também abstrai os detalhes associados ao trabalho com objetos Driver.

Connection: Essa interface com todos os métodos para entrar em contato com um banco de dados. O objeto de conexão representa o contexto de comunicação, ou seja, toda a comunicação com o banco de dados é feita apenas por meio do objeto de conexão.

Statement: Você usa objetos criados a partir dessa interface para enviar as instruções SQL para o banco de dados. Algumas interfaces derivadas aceitam parâmetros além de executar procedimentos armazenados.

ResultSet: Esses objetos armazenam dados recuperados de um banco de dados depois que você executa uma consulta SQL usando objetos Statement. Ele age como um iterador para permitir que você se mova através de seus dados.

SQLException: Esta classe manipula todos os erros que ocorrem em um aplicativo de banco de dados.

Classe de conexão com banco

Sempre precisamos de uma classe que estabelece a conexão com o banco de dados e quando falar de JDBC é usado uma classe Java com padrão Singleton para conectar e oferecer apenas uma conexão ativa e funcional para nosso projeto de sistemas.

```
6 public class SingleConnection {
7
8     //Url do banco de dados
9     private static String url = "jdbc:postgresql://localhost:5432/posjava";
10
11    //Senha do banco de dados
12    private static String password = "admin";
13
14    //User do banco de dados
15    private static String user = "postgres";
16
17    // Classe de conexão com o banco de dados
18    private static Connection connection = null;
19
```

Dentro desta mesma classe criamos o método responsável por executar a conexão e retorna a mesma quando necessitamos.

```
28 private static void conectar(){
29     try {
30
31         // Verifica se já existe a conexão
32         if (connection == null){
33
34             /// Registra o driver do banco de dados
35             Class.forName("org.postgresql.Driver");
36
37             // Faz a conexão com o banco de dados
38             connection = DriverManager.getConnection(url, user, password);
39
40             // Configuração para não commitar automaticamente os dados no banco de dados
41             connection.setAutoCommit(false);
42
43             System.out.println("Conectou com sucesso");
44         }
45
```

Pra finalizar o padrão Singleton teremos sempre ao final um método estático que retorna o objeto de conexão com o banco de dados.

```
51 public static Connection getConnection(){  
52     return connection;  
53 }  
54
```

Interface PreparedStatement

A interface PreparedStatement é uma subinterface do Statement. É usado para executar consultas parametrizadas.

Por que usar o PreparedStatement?

Melhora o desempenho : O desempenho do aplicativo será mais rápido se você usar a interface PreparedStatement porque a consulta é compilada apenas uma vez.

A classe de modelo de dados

Como Java é orientado a objetos nada mais correto do que ter um objeto que representa nosso negócio e também representa a tabela no banco de dados e para isso já podemos seguir um padrão mais básico e fácil que existe no desenvolvimento de sistemas que é criar uma classe que tráfega os dados em todas as camadas do sistema até chegar ao banco de dados e do banco para as camadas do sistema, então, em uma tabela que possui id, nome e e-mail a classe de modelo ficaria como a imagem abaixo:

```
1 package model;
2
3 public class Userposjava {
4
5     private Long id;
6     private String nome;
7     private String email;
8
9
10    public Long getId() {
11        return id;
12    }
13
14    public void setId(Long id) {
15        this.id = id;
16    }
17
18    public String getNome() {
19        return nome;
20    }
```

Padrão DAO

Objeto de acesso a dados (ou simplesmente **DAO**, acrônimo de Data Access Object), é um **padrão** para persistência de dados que permite separar regras de negócio das regras de acesso a banco de dados.

Neste caso que vou exemplificar para nosso estudo de caso. Dentro do DAO já definimos o objeto Connection e dentro do construtor já chamamos a nossa classe que faz a conexão e retorna a conexão pronta para efetuar as operações de CRUD.

```
1 package dao;
2
3 import java.sql.Connection;
4
12
13 public class UserPosDAO {
14
15     private Connection connection;
16
17    public UserPosDAO() {
18        connection = SingleConnection.getConnection();
19    }
20
```

Inserindo dados no banco de dados

Neste passo já vamos aprender como fazer o método salvar em nossa estrutura DAO. O salvar já recebe o objeto com os dados, escrevemos o SQL e os parâmetros são representados pelas interrogações (?, ?) e são setados na mesma ordem que são escritos na String do SQL;

```
21 public void salvar(Userposjava userposjava) {
22     try {
23         String sql = "insert into userposjava (nome, email) values (?,?)"; // String do SQL
24         // Retorna o objeto de instrução
25         PreparedStatement insert = connection.prepareStatement(sql);
26         insert.setString(1, userposjava.getNome()); // Parâmetro sendo adicionados
27         insert.setString(2, userposjava.getEmail());
28         insert.execute(); // SQL sendo executado no banco de dados
29         connection.commit(); // salva no banco
30
31     } catch (Exception e) {
32         try {
33             connection.rollback(); // reverte operação caso tenha erros
34         } catch (SQLException e1) {
35             e1.printStackTrace();
36         }
37         e.printStackTrace();
38     }
39 }
40 }
```

Realizando consulta de todos os dados da tabela

Depois que criamos a rotina de insert que é a gravação de dados no banco de dados já podemos testar e fazer outra rotina muito importante que é a consulta ao banco de dados.

Para isso usamos o a instrução d DML que é o select, com ela podemos estabelecer a consulta a todos os registros da tabela ou apenas a alguns valores específicos de algumas colunas e o mais importante é que precisamos de um objeto que nos retorna esse tipo de consulta e o JDBC tem o ResultSet que é o responsável por armazenar os dados trazidos do banco de dados e com isso usando a API Jdbc com Java podemos recuperar os dados facilmente.

```

41
42 public List<Userposjava> listar() throws Exception {
43     List<Userposjava> list = new ArrayList<Userposjava>(); // Lista de retorno do método
44
45     String sql = "select * from userposjava"; // Instrução SQL
46
47     PreparedStatement statement = connection.prepareStatement(sql); // Objeto de instrução
48     ResultSet resultado = statement.executeQuery(); // Executa a consulta ao banco de dados
49
50     while (resultado.next()) { // Iteramos percorrendo o objeto ResultSet que tem os dados
51         Userposjava userposjava = new Userposjava(); // Criamos um novo objeto para cada linha
52         userposjava.setId(resultado.getLong("id"));
53         userposjava.setNome(resultado.getString("nome")); // Setamos os valores para o objeto
54         userposjava.setEmail(resultado.getString("email"));
55
56         list.add(userposjava); // Para cada objetos adicionamos ele na lista de retorno
57     }
58
59     return list;
60 }
61

```

Realizando buscar por ID

A busca por ID específico no banco de dados é superfácil, anteriormente criamos a consulta onde carregamos toda a lista de dados de uma única tabela e agora vamos aprender a carregar apenas um objeto passando por parâmetro o identificador de primary key.

```

62 public Userposjava buscar(Long id) throws Exception {
63     Userposjava retorno = new Userposjava();
64     String sql = "select * from userposjava where id = " + id; // Sql recebendo o parâmetro
65
66     PreparedStatement statement = connection.prepareStatement(sql); // Instrução compilada
67     ResultSet resultado = statement.executeQuery(); // Consulta sendo executada
68
69     while (resultado.next()) { // retorna apenas um ou nenhum
70
71         retorno.setId(resultado.getLong("id")); // Capturando os dados e jogando no objeto
72         retorno.setNome(resultado.getString("nome"));
73         retorno.setEmail(resultado.getString("email"));
74
75     }
76     return retorno;
77 }
78

```


Realizando Update de dados

Um dos principais pontos de um sistema é a atualização de dados que estão no banco de dados e que já foram cadastrados em outras etapas. Isso é possível fazer usando o DML com a cláusula update onde especificamos os parâmetros que serão alterados e informamos alguma condição para filtrar dados para serem alterados ou simplesmente alterar toda a tabela de dados.

```
79 public void atualizar(Userposjava userposjava) {  
80     try {  
81         // Sql usando o SET para informa o nome valor  
82         String sql = "update userposjava set nome = ? where id = " + userposjava.getId();  
83  
84         PreparedStatement statement = connection.prepareStatement(sql); // Comilando o SQL  
85         statement.setString(1, userposjava.getNome()); // Passando o parâmetro para update  
86  
87         statement.execute(); // Executando a atualização  
88         connection.commit(); // Comitando/Gravando no banco de dados  
89  
90     } catch (Exception e) {  
91         try {  
92             connection.rollback(); // Reverte caso dê algum erro  
93         } catch (SQLException e1) {  
94             e1.printStackTrace();  
95         }  
96         e.printStackTrace();  
97     }  
98 }
```

Deletando registro no banco de dados

Um dos principais pontos de um sistema é a remoção de dados que estão no banco de dados e que já foram cadastrados em outras etapas. Isso é possível fazer usando o DML com a cláusula delete onde especificamos sempre o ID que identifica a chave única do registro ou uma determina condição que pode remover vários registro de uma vez.

```
100 public void deletar(Long id) {  
101     try {  
102  
103         String sql = "delete from userposjava where id = " + id; // SQL para delete  
104         PreparedStatement preparedStatement = connection.prepareStatement(sql); // Compilando  
105         preparedStatement.execute(); // Executando no banco de dados  
106         connection.commit(); // Efetuando o commit/gravando no banco de dados  
107  
108     } catch (Exception e) {  
109         try {  
110             connection.rollback();  
111         } catch (SQLException e1) {  
112             e1.printStackTrace();  
113         }  
114         e.printStackTrace();  
115     }  
116 }  
117 }
```

Considerações finais sobre o JDBC

Usar apenas JDBC hoje em dia é inviável e completamente improdutivo para a equipe e gera muitos bugs e manutenção para a equipe, mas, ainda assim, é muito importante para aprender a usar os frameworks que usam todo esse conceito por baixo dos panos e quando se fala de processamento rápido onde cada segundo é importante o SQL e JDBC puro ainda é o que temos de mais rápido

O que é JPA?

JPA é um framework leve, baseado em POJOS (Plain Old Java Objects) para persistir objetos Java. A Java Persistence API, diferente do que muitos imaginam, não é apenas um framework para Mapeamento Objeto-Relacional (ORM - Object-Relational Mapping), ela também oferece diversas funcionalidades essenciais em qualquer aplicação corporativa.

Atualmente temos que praticamente todas as aplicações de grande porte utilizam JPA para persistir objetos Java.

JPA provê diversas funcionalidades para os programadores, como será mais detalhadamente visto nas próximas seções. Inicialmente será visto a história por trás da JPA, a qual passou por algumas versões até chegar na sua versão atual.

História da Especificação JPA

Após diversos anos de reclamações sobre a complexidade na construção de aplicações com Java, a especificação Java EE 5 teve como principal objetivo a facilidade para desenvolver aplicações JEE 5. O EJB 3 foi o grande precursor para essa mudança fazendo os Enterprise JavaBeans mais fáceis e mais produtivos de usar.

No caso dos Session Beans e Message-Driven Beans, a solução para questões de usabilidade foram alcançadas simplesmente removendo alguns dos mais onerosos requisitos de implementação e permitindo que os componentes sejam como Plain Java Objects ou POJOS.

Já os Entity Beans eram um problema muito mais sério. A solução foi começar do zero. Deixou-se os Entity Beans sozinhos e introduziu-se um novo modelo de persistência.

A versão atual da JPA nasceu através das necessidades dos profissionais da área e das soluções proprietárias que já existiam para resolver os problemas com persistência. Com a ajuda dos desenvolvedores e de profissionais experientes que criaram outras ferramentas de persistência, chegou a uma versão muito melhor que é a que os desenvolvedores Java conhecem atualmente.

Dessa forma os líderes das soluções de mapeamento objetos-relacionais deram um passo adiante e padronizaram também os seus produtos. Hibernate e TopLink foram os primeiros a firmar com os fornecedores EJB.

O resultado final da especificação EJB finalizou com três documentos separados, sendo que o terceiro era o Java Persistence API. Essa especificação descrevia o modelo de persistência em ambos os ambientes Java SE e Java EE.

JPA 2.0 (Java Persistence API)

No momento em que a primeira versão do JPA foi iniciada, outros modelos de persistência ORM já haviam evoluído. Mesmo assim muitas características foram adicionadas nesta versão e outras foram deixadas para uma próxima versão.

A versão JPA 2.0 incluiu um grande número de características que não estavam na primeira versão, especialmente as mais requisitadas pelos usuários, entre elas a capacidade adicional de mapeamento, expansões para a Java Persistence Query Language (JPQL), a API Criteria para criação de consultas dinâmicas, entre outras características.

Entre as principais inclusões na JPA destacam-se:

POJOS Persistentes: Talvez o aspecto mais importante da JPA seja o fato que os objetos são POJOs (Plain Old Java Object ou Velho e Simples Objeto Java), significando que os objetos possuem design simples que não dependem da herança de interfaces ou classes de frameworks externos.

Qualquer objeto com um construtor default pode ser feito persistente sem nenhuma alteração numa linha de código. Mapeamento Objeto-Relacional com JPA é inteiramente dirigido a metadados. Isto pode ser feito através de anotações no código ou através de um XML definido externamente.

Consultas em Objetos: As consultas podem ser realizadas através da Java Persistence Query Language (JPQL), uma linguagem de consulta que é derivada do EJB QL e transformada depois para SQL. As consultas usam um esquema abstraído que é baseado no modelo de entidade como oposto às colunas na qual a entidade é armazenada.

Configurações simples: Existe um grande número de características de persistência que a especificação oferece, todas são configuráveis através de

anotações, XML ou uma combinação das duas. Anotações são simples de usar, convenientes para escrever e fácil de ler. Além disso, JPA oferece diversos valores defaults, portanto para já sair usando JPA é simples, bastando algumas anotações.

Integração e Teste: Atualmente as aplicações normalmente rodam num Servidor de aplicação, sendo um padrão do mercado hoje. Testes em servidores de aplicação são um grande desafio e normalmente impraticáveis.

Efetuar teste de unidade e teste caixa branca em servidores de aplicação não é uma tarefa tão trivial. Porém, isto é resolvido com uma API que trabalha fora do servidor de aplicação.

Isto permite que a JPA possa ser utilizada sem a existência de um servidor de aplicação. Dessa forma, testes unitários podem ser executados mais facilmente.

Configurando Maven com JPA





















Abaixo estão as dependências para serem baixadas apenas do hibernate/jpa, adicionando ela serão baixar uma lista enorme de bibliotecas que são obrigatórias para o uso do frameworks.

```
<!-- HIBERNATE -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.2.6.Final</version>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>5.2.6.Final</version>
</dependency>
```

Abaixo podemos ver a lista enorme de bibliotecas que são baixadas quando adicionamos as tags xml no pom.xml do Maven, esta parte pode ser vista no projeto.

📦 Maven Dependencies

- ▷  junit-4.12.jar - C:\Users\alex\.m2\repository\junit\junit-4.12.jar
- ▷  hamcrest-core-1.3.jar - C:\Users\alex\.m2\repository\hamcrest\hamcrest-core-1.3.jar
- ▷  postgresql-9.2-1003-jdbc4.jar - C:\Users\alex\.m2\repository\org.postgresql\postgresql-9.2-1003-jdbc4.jar
- ▷  hibernate-core-5.2.6.Final.jar - C:\Users\alex\.m2\repository\org.hibernate\hibernate-core-5.2.6.Final.jar
- ▷  jboss-logging-3.3.0.Final.jar - C:\Users\alex\.m2\repository\org.jboss.logging\jboss-logging-3.3.0.Final.jar
- ▷  hibernate-jpa-2.1-api-1.0.0.Final.jar - C:\Users\alex\.m2\repository\org.hibernate.javax.persistence\hibernate-jpa-2.1-api-1.0.0.Final.jar
- ▷  javassist-3.20.0-GA.jar - C:\Users\alex\.m2\repository\org.javassist\javassist-3.20.0-GA.jar
- ▷  antlr-2.7.7.jar - C:\Users\alex\.m2\repository\antlr\antlr-2.7.7.jar
- ▷  geronimo-jta_1.1_spec-1.1.1.jar - C:\Users\alex\.m2\repository\org.jboss.jta\geronimo-jta_1.1_spec-1.1.1.jar
- ▷  jandex-2.0.3.Final.jar - C:\Users\alex\.m2\repository\org.jboss.jandex\jandex-2.0.3.Final.jar
- ▷  classmate-1.3.0.jar - C:\Users\alex\.m2\repository\com.fasterxml.classmate\classmate-1.3.0.jar
- ▷  dom4j-1.6.1.jar - C:\Users\alex\.m2\repository\org.dom4j\dom4j-1.6.1.jar
- ▷  hibernate-commons-annotations-5.0.1.Final.jar - C:\Users\alex\.m2\repository\org.hibernate.common\hibernate-commons-annotations-5.0.1.Final.jar
- ▷  cdi-api-1.1.jar - C:\Users\alex\.m2\repository\javax.enterprise\cdi-api-1.1.jar
- ▷  el-api-2.2.jar - C:\Users\alex\.m2\repository\javax.el\el-api-2.2.jar
- ▷  jboss-interceptors-api_1.1_spec-1.0.0.Beta1.jar - C:\Users\alex\.m2\repository\org.jboss.interceptors\jboss-interceptors-api_1.1_spec-1.0.0.Beta1.jar
- ▷  jsr250-api-1.0.jar - C:\Users\alex\.m2\repository\javax.annotation\jsr250-api-1.0.jar
- ▷  javax.inject-1.jar - C:\Users\alex\.m2\repository\javax.inject\javax.inject-1.jar
- ▷  hibernate-entitymanager-5.2.6.Final.jar - C:\Users\alex\.m2\repository\org.hibernate\hibernate-entitymanager-5.2.6.Final.jar
- ▷  byte-buddy-1.5.4.jar - C:\Users\alex\.m2\repository\net.bytebuddy\byte-buddy-1.5.4.jar

Arquivo de configuração do JPA

Quando usamos JDBC tudo era configurado direto em uma classe Java criada do zero, usando JPA isso é feito de forma mais organizada usando um arquivo de xml onde são configurados os mesmos parâmetros como url do banco de dados, usuário do banco, senha do banco de dados e ainda temos muitos outros recursos como permitir a geração automática das tabelas no banco de dados.

```
<persistence-unit name="pos-java-maven-hibernate">
  <class>model.UsuarioPessoa</class>
  <class>model.TelefoneUser</class>
  <properties>
    <!-- Dados de conexão com o banco -->
    <property name="hibernate.connection.driver_class" value="org.postgresql.Driver" />
    <property name="hibernate.connection.url"
      value="jdbc:postgresql://localhost:5432/posjavahibernate" />
    <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
    <property name="hibernate.connection.username" value="postgres" />
    <property name="hibernate.connection.password" value="admin" />
    <property name="hibernate.format_sql" value="false" />
    <property name="hibernate.hbm2ddl.auto" value="update" />
  </properties>
</persistence-unit>
```

Classe de conexão do JPA

Agora como temos todos os dados escritos no arquivo de xml de configuração precisamos apenas de uma classe para dar o start inicial na conexão do banco de dados, sendo um frameworks extremamente completo o JPA nos dá uma classe que faz exatamente a leitura do arquivo de configuração e já realiza todas as funções configuradas no arquivos e já disponibiliza um objeto EntityManagerFactory que é o responsável por ter os métodos de operações com o banco de dados.

```
15 private static void init(){
16
17     try {
18
19         if (factory == null){
20             factory = Persistence.createEntityManagerFactory("pos-java-maven-hibernate");
21         }
22     } catch (Exception e) {
23         e.printStackTrace();
24     }
25 }
```

Introdução ao EntityManager

Na nova Java Persistence Specification, o **EntityManager** é o serviço central para todas as ações de persistência. Entidades são objetos de Java claros que são alocados como qualquer outro objeto Java. O **EntityManager** administra o O/R que o mapea entre uma classe de entidade e uma fonte de dados subjacente.

```
27 public static EntityManager geEntityManager(){
28     return factory.createEntityManager(); /*Prove a parte de persistencia*/
29 }
30
```

DaoGeneric JPA

Quando trabalhamos com JPA a escrita de SQL é quase zero, o frameworks além de ter sua própria linguagem que é o HQL ainda possui inúmeros métodos que são comuns a todo sistema que é criado e então nós programadores apenas precisamos saber qual método invocar e com apenas uma chamada fazer muito mesmo no banco de dados, ao invés de escrever linhas e mais linhas de SQL puro apenas chamamos no caso do JPA o método PERSIST que os dados são gravados corretamente no banco de dados, o melhor de tudo que o DAO genérico agora fica muito mais simples e fácil de entender. Ao invés da conexão direta nós invocamos o EntityManager dentro do DAO.

```

10 @SuppressWarnings("unchecked")
11 public class DaoGeneric<E> {
12
13     private EntityManager entityManager = HibernateUtil.geEntityManager();
14

```

Anotação @Entity

A anotação @Entity é utilizada para informar que uma classe também é uma entidade. A partir disso, a JPA estabelecerá a ligação entre a entidade e uma tabela de mesmo nome no banco de dados, onde os dados de objetos desse tipo poderão ser persistidos.

Uma entidade representa, na Orientação a Objetos, uma tabela do banco de dados, e cada instância dessa entidade representa uma linha dessa tabela.

Caso a tabela possua um nome diferente, podemos estabelecer esse mapeamento com a anotação @Table, a qual será explorada em outra documentação.

```

1 package model;
2
3 import java.util.List;
11
12 @Entity
13 public class UsuarioPessoa {
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.AUTO)
17     private Long id;
18
19     private String nome;
20     private String sobrenome;
21     private String email;

```


Anotação @Id

A anotação @Id é utilizada para informar ao JPA qual campo/atributo de uma entidade estará relacionado à chave primária da respectiva tabela no banco de dados. Essa é uma anotação obrigatória e um erro será gerado em tempo de execução caso ela não esteja presente.

```
L4  
L5 @Id  
L6 @GeneratedValue(strategy = GenerationType.AUTO)  
L7 private Long id;  
L8
```

Anotação @OneToMany e @ManyToOne

O relacionamento OneToMany é bem usado, e são poucas vezes que de fato não precisamos te-lo, então é aquele lance se não precisou até hoje, espere mais um pouco que essa necessidade vai nascer.

Vamos tirar a vantagem dos annotations e veremos @OneToMany e @ManyToOne ao invés dos .hbm. Como exemplo há vários cenários para exemplificar este tipo de relacionamento, tais como: um time de futebol tem vários jogadores, uma infra-estrutura tem vários servidores, porém um jogador só pode jogar em um time(ou não depende da regra de negócio, aqui está o pulo do gato), e um servidor está em uma infra-estrutura.

```
25  
26 @OneToMany(mappedBy = "usuarioPessoa", fetch = FetchType.EAGER)  
27 private List<TelefoneUser> telefoneUsers;  
28
```

```
24 @ManyToOne(optional = false, fetch = FetchType.EAGER)  
25 private UsuarioPessoa usuarioPessoa;  
26
```

Anotação @OneToOne

A anotação One-to-One é utilizada para associar duas entidades onde uma não é componente da outra, ao contrário da definição acima.

Numa associação One-to-One também podemos ter um relacionamento bidirecional. Nesse caso, um dos lados precisará ser o dono do relacionamento e ser responsável por atualizar uma coluna com uma chave estrangeira.

```
public class Pessoa {
    @OneToOne(fetch = FetchType.LAZY, mappedBy="pessoa", optional =
    true)
    private Endereco endereco;
}

public class Endereco {
    @OneToOne
    private Pessoa pessoa;
}
```

Anotação @ManyToMany

Define uma associação de muitos valores com multiplicidade de muitos para muitos.

Toda associação muitos-para-muitos tem dois lados, o lado proprietário e o lado não-proprietário ou inverso. A tabela de junção é especificada no lado proprietário. Se a associação for bidirecional, qualquer um dos lados pode ser designado como o lado proprietário. Se o relacionamento for bidirecional, o lado não proprietário deverá usar o mappedBy elemento da ManyToMany anotação para especificar o campo ou propriedade de relacionamento do lado proprietário.

A tabela de junção para o relacionamento, se não for padronizada, é especificada no lado proprietário.

A ManyToMany anotação pode ser usada em uma classe incorporável contida em uma classe de entidade para especificar um relacionamento com uma coleção de entidades. Se o relacionamento for bidirecional e a entidade que contém a classe incorporável for o proprietário do relacionamento, o lado não proprietário deverá usar o mappedBy elemento da ManyToMany anotação para especificar o campo ou a propriedade de relacionamento da classe incorporável.

```
@Entity
public class Projeto implements Serializable {
    private static final long serialVersionUID = 1081869386060246794L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nome;
    @ManyToMany(mappedBy="projetos", cascade = CascadeType.ALL)
    private List<Funcionario> desenvolvedores;

    public List<Funcionario> getDesenvolvedores() { return desenvolvedores; }
    public void setDesenvolvedores(List<Funcionario> desenvolvedores) {
        this.desenvolvedores = desenvolvedores;
    }
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }
}

@Entity
public class Funcionario implements Serializable {
    private static final long serialVersionUID = -9109414221418128481L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nome;
    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name="PROJETO_FUNCIONARIO",
        joinColumns=@JoinColumn(name="PROJETO_ID"),
        inverseJoinColumns=@JoinColumn(name="FUNCIONARIO_ID"))
    private List<Projeto> projetos;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public List<Projeto> getProjetos() {
        return projetos;
    }
    public void setProjetos(List<Projeto> projetos) {
        this.projetos = projetos;
    }
}
```


Anotação @NaturalId

Embora não seja usada como propriedade do identificador, algumas propriedades (do grupo) representam o identificador natural de uma entidade. Isso é especialmente verdadeiro quando o esquema usa a abordagem recomendada de uso de chave primária substituta, mesmo que exista uma chave de negócios natural.

O Hibernate permite mapear essas propriedades naturais e reutilizá-las em uma consulta Criteria. O identificador natural é composto de todas as propriedades marcadas como @NaturalId.

Para exemplificar melhor podemos ter a entidade usuario ou pessoa e além do ID que é primary key podemos ter um chave natural por exemplo o CPF.

```
32 @NaturalId
33 private String cpf;
34
```

Anotação @Table

@Tabela é definida no nível da turma; Ele permite que você defina os nomes de tabelas, catálogos e esquemas para o mapeamento de entidades. Se nenhuma @Table for definida, os valores padrão serão usados: o nome de classe não qualificado da entidade.

O elemento @Table contém um esquema e atributos de catálogo, se precisarem ser definidos. Você também pode definir restrições exclusivas para a tabela usando a anotação @UniqueConstraint em conjunto com @Table.

```
--
18 @Entity
19 @Table(name = "usuariobanco")
20 public class UsuarioSpringData {
21
```

Anotação @UniqueConstraint

É possível e normal termos restrições em um banco de dados como por exemplo não poder cadastrar a mesma pessoa com login e senha igual, bem sabemos que cadastrar com o mesmo login já é um bloqueio normal e mais comum de se fazer em um banco de dados e também é super importante não deixar gravar com a mesma senha também e isso podemos fazer de um forma bem simples com anotações no JPA usando a anotação @UniqueConstraint.

```
18 @Entity
19 @Table(name = "usuariobanco", uniqueConstraints
20       = {@UniqueConstraint(columnNames={"login", "senha", "cpf"})})
21 public class UsuarioSpringData {
22
```

Anotação @Version

Você pode adicionar capacidade de bloqueio otimista a uma entidade usando a anotação @Version.

A propriedade version será mapeada para a coluna OPTLOCK e o gerenciador de entidades a utilizará para detectar atualizações conflitantes (evitando atualizações perdidas que você poderia ver com a estratégia last-commit-wins).

A coluna da versão pode ser um numérico (a solução recomendada) ou um registro de data e hora. O Hibernate suporta qualquer tipo de tipo desde que você defina e implemente o UserVersionType apropriado.

O aplicativo não deve alterar o número da versão configurado pelo Hibernate de qualquer forma. Para aumentar artificialmente o número da versão, verifique na documentação de referência do Hibernate Entity Manager LockModeType.OPTIMISTIC_FORCE_INCREMENT ou LockModeType.PESSIMISTIC_FORCE_INCREMENT.

```
37 @Version
38 @Column(name="OPTLOCK")
39 private int version;
40
```

Anotação @Column

É usado para especificar a coluna mapeada para uma propriedade ou campo persistente.

Se nenhuma Column anotação for especificada, os valores padrão serão aplicados.

As colunas usadas para um mapeamento de propriedades podem ser definidas usando a anotação @Column. Use-o para substituir os valores padrão. Você pode usar essa anotação no nível da propriedade para propriedades que são.

Essa anotação possibilita varios recursos importantes:

- name : nome da coluna diferente do atributo
- unique : define se o valor é unico
- nullable: define se posse ou não ser obrigatorios
- insertable: se a coluna será ou não parte da instrução insert
- updatable: se a coluna será ou não parte da instrução de atualização
- columnDefinition: define o tipo da coluna no banco da dos.
- table :define a tabela de destino (tabela primária padrão)
- length: comprimento da coluna
- precision: precisão decimal da coluna
- scale: coluna decimal da coluna se for útil

Vendo todas as opções que temos para as colunas podemos fazer e demonstrar uma regra bem bacana, usando a coluna de cpf como exemplo podemos definir as regras que o cpf é um atributo natural e que sua coluna do banco tem um nome diferente do atributo e o campo será único no banco de dados e depois de gravado não poderá ser modificado.

```
11 @NaturalId
12 @Column(name = "cpg_pessoa", unique = true, updatable = false)
13 private String cpf;
```

Anotação @GeneratedValue e @SequenceGenerator

O JPA define cinco tipos de estratégias de geração de identificadores:

- AUTO - coluna, sequência ou tabela de identidade, dependendo do banco de dados subjacente

- TABELA - tabela segurando o id
- IDENTIDADE - coluna de identidade
- SEQUÊNCIA – sequência cópia de identidade - a identidade é copiada de outra entidade

```

25 @SequenceGenerator(initialValue = 1, name = "usuario_sequence")
26 public class UsuarioSpringData {
27
28     @Id
29     @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="usuario_sequence")
30     private Long id;
31

```

Então criando a sequencia e dando nome a ela nós podemos definir que o ID ou sendo a primary key usara esse nosso sequenciador.

Anotação @GeneratedValue

A anotação @GeneratedValue é utilizada para informar que a geração do valor do identificador único da entidade será gerenciada pelo provedor de persistência. Essa anotação deve ser adicionada logo após a anotação @Id. Quando não anotamos o campo com essa opção, significa que a responsabilidade de gerar e gerenciar as chaves primárias será da aplicação, em outras palavras, do nosso código, como vemos no exemplo a seguir:

```

14
15     @Id
16     @GeneratedValue(strategy = GenerationType.AUTO)
17     private Long id;
18

```

Anotação @Transient

Em muito casos precisamos carregar um valor apenas em tempo de execução até terminar um determinado processo e esse valor não deve ser gravado no banco de dados, o JPA possui a anotação @Transient que não torna o atributo da entidade persistente.

```
30 @Transient
31 private String nomeAuxiliar;
32
```

Anotação @Lob

@Lob indica que a propriedade deve ser mantida em um Blob ou Clob, dependendo do tipo de propriedade: java.sql.Clob, Character [], char [] e java.lang.String serão persistidos em um Clob. java.sql.Blob, Byte [], byte [] e tipo serializável serão persistidos em um Blob.

Esse tipo de campo normalmente é usado para gravar imagem, arquivos como pdf e textos enormes.

```
31 @Lob
32 private byte[] foto;
33
34
```

Retornando a primary key da entidade

O próprio JPA possui um método dentro do EntityManagerFactory que é responsável por retornar o valor da primary key da entidade, como sabemos esse valor se encontra no atributos anotado com @ID e isso é útil para criar rotinas genéricas dentro do nosso sistema. Abaixo temos o código onde pode ser passado a entidade e é retornado a PK da entidade.

```
31 public static Object getPrimaryKey(Object entity){ // Retorna a primary key
32     return factory.getPersistenceUnitUtil().getIdentifier(entity);
33 }
```

Salvando com JPA (Persist)

Usando JPA toda aquela complexidade se ficar escrevendo SQL na mão e tendo que a todo momento ficar adicionando novos atributos e gerando manutenção na parte de persistencia tendo que ir manualmente e adicionar mais SQL a todo momento, isso não existe com JPA, toda essa complexidade é abstraída e para salvar basta chamar o método PERSIST.

```
15 public void salvar(E entidade) {  
16     EntityTransaction transaction = entityManager.getTransaction(); //Obeito de transação  
17     transaction.begin(); // Inicia uma transação  
18     entityManager.persist(entidade); // Salva no banco de dados  
19     transaction.commit(); // Grava no banco de dados  
20 }  
21
```

Save or Update com JPA (Merge)

Bem traduzindo ao pé da letra o que acontece é que é muito a gente precisar salvar os dados no banco de dados e consulta pra poder ter o estado persistente do objeto e também para poder obter a chave única que foi gerada, estamos falando da PK ou primary key e como um frameworks que se preze deve facilitar nosso trabalho o JPA tem um método chamado de MERGE, ele atualiza dados existentes ou então insere caso seja um novo objeto que está sendo inserido no banco de dados e o melhor de tudo ele retorna o objeto pra nós igual está no banco de dados.

```
24 public E updateMerge(E entidade) { // Salva ou Atualiza  
25     EntityTransaction transaction = entityManager.getTransaction();  
26     transaction.begin();  
27     E entidadeSalva = entityManager.merge(entidade); //Salva, atualiza e retorna o objeto  
28     transaction.commit();  
29  
30     return entidadeSalva;  
31 }
```


Pesquisar um registro no banco de dados

Para realizar a pesquisa o JPA também tem um método pronto e nós apenas precisamos passar a classe pra ele saber qual tabela buscar no banco e o ID do objeto que no caso é o valor da chave única do registro que precisamos trazer.

```

40 public E pesquisar(Long id, Class<E> entidade) {
41     E e = (E) entityManager.find(entidade, id); // ID = a PK e o ENTIDADE e a Class do Objeto
42     return e;
43 }
44
45

```

Deletando registros

Para deletar registro a forma mais prática para evitar problemas de objetos em memória é fazer o JPA executar um SQL de delete, mas antes disso precisamos saber o ID do objeto que desejamos remover, podemos passar direto como parâmetro ou então fazer de forma bem genérica igual eu criei no método abaixo.

```

46 public void deletarPorId(E entidade) {
47     Object id = HibernateUtil.getPrimaryKey(entidade); // Obtem o ID do objeto PK
48     EntityTransaction transaction = entityManager.getTransaction(); // Objeto de transação
49     transaction.begin(); // Começa uma Transação no banco de dados
50
51     entityManager
52         .createNativeQuery(
53             "delete from " + entidade.getClass(). // Monta a Query para delete
54             getSimpleName().toLowerCase() + " where id =" + id)
55         .executeUpdate(); // Executa o delete no banco de dados
56     transaction.commit(); // Grava alteração no banco
57 }
58

```

Trazendo lista de dados do banco de dados

Para recuperar uma lista de dados e registros do banco de dados precisamos saber classe queremos carregar para que o JPA saiba qual tabela no banco de dados buscar e assim montamos a nossa query usando o createQuery e logo após a montagem dela pedimos o retorno em forma de lista de objetos chamando o método getResultList.

```
60 public List<E> listar(Class<E> entidade) {
61     EntityTransaction transaction = entityManager.getTransaction();
62     transaction.begin();
63     List<E> lista = entityManager.
64         createQuery("from " + entidade.getName())//Cria a queru de consulta
65         .getResultList();// Retorna a lista de objetos consultados
66     transaction.commit();
67
68     return lista;
69 }
70
```

Gravando objeto no banco com JPA

Depois que temos toda a parte de persistência criada, JPA configurado, entidades criadas e mapeadas e o DAO genérico criado e funcionando corretamente. Agora é a hora de comer a gravar dados. Primeiramente precisamos chamar nosso DAO logo em seguida instanciar um objeto e setar os dados do seu atributo e pra finalizar gravar no banco de dados.

```
--
13 @Test
14 public void testeHibernateUtil() {
15
16     DaoGeneric<UsuarioPessoa> daoGeneric =
17         new DaoGeneric<UsuarioPessoa>();// Instância o DAO genérico
18     UsuarioPessoa pessoa = new UsuarioPessoa(); // Cria o obeito para ser salvo
19
20     // Seta todas as propriedades do objeto
21     pessoa.setIdade(45);
22     pessoa.setLogin("teste");
23     pessoa.setNome("Paulo");
24     pessoa.setSenha("123");
25     pessoa.setSobrenome("Egidio");
26     pessoa.setEmail("javaavancado@javaavancado.com");
27
28     daoGeneric.salvar(pessoa);// Chama o salvar para gravar no banco de dados.
29
30 }
31
```

Consultando objetos no banco de dados

Para consultar algum registro especifico precisamos apenas do ID ou sendo a PK do registro no banco de dados e com isso passando para o método de pesquisa o objeto é retornado.


```

32 @Test
33 public void testeBuscar() {
34     DaoGeneric<UsuarioPessoa> daoGeneric =
35         new DaoGeneric<UsuarioPessoa>(); // Inicia o DAO
36     UsuarioPessoa pessoa = new UsuarioPessoa(); // Inicia o Objeto
37     pessoa.setId(2L); // Precisamos apenas do ID
38
39     pessoa = daoGeneric.pesquisar(pessoa); // Envia para pesquisa
40
41     System.out.println(pessoa); // Imprime objeto para conferir
42
43 }

```

Atualizando objetos no banco de dados

Para atualizar algum registro específico precisamos apenas do ID ou sendo a PK do registro no banco de dados e com isso passando para o método de pesquisa o objeto é retornado. Após o retorno a gente somente seta os valores dos novos atributos e invocamos o método de MERGE onde ele irá atualizar os dados no banco de dados e retornar o objeto em seu novo estado persistente.

```

55 @Test
56 public void testeUpdate() {
57     DaoGeneric<UsuarioPessoa> daoGeneric = new DaoGeneric<UsuarioPessoa>();
58
59     UsuarioPessoa pessoa = daoGeneric.
60         pesquisar(1L, UsuarioPessoa.class); // Carrega Objeto para editar
61
62     pessoa.setIdade(99); // Atualiza os atributos
63     pessoa.setNome("Nome atualizado Hibernate");
64     pessoa.setSenha("sd4s5d4s4d");
65
66     pessoa = daoGeneric.updateMerge(pessoa); // Atualiza no banco de dados.
67
68     System.out.println(pessoa);
69
70 }
71

```

Deletando objetos no banco de dados

Para deletar algum registro específico precisamos apenas do ID ou sendo a PK do registro no banco de dados e com isso passando para o método de pesquisa o

objeto é retornado. Após o retorno passamos nosso objeto para nosso método que faz o delete/remoção desse registro na base de dados.

```
@Test
public void testeDelete() {
    DaoGeneric<UsuarioPessoa> daoGeneric = new
        DaoGeneric<UsuarioPessoa>(); // Instancia o DAO

    UsuarioPessoa pessoa = daoGeneric.
        pesquisar(3L, UsuarioPessoa.class); // Consulta o objeto antes de remover

    daoGeneric.deletarPoId(pessoa); // Invoca o método de remoção do banco de dados
}
```

Consultando lista de dados

Já temos construído a estrutura persistencia e agora queremos trazer uma lista de dados e imprimir no console para termos a certeza que nosso método no listar no DAO está correto.

```
4 @Test
5 public void testeConsultar() {
6     DaoGeneric<UsuarioPessoa> daoGeneric =
7         new DaoGeneric<UsuarioPessoa>(); // Instanciar DAO
8
9     List<UsuarioPessoa> list = daoGeneric.
10        listar(UsuarioPessoa.class); // Invocar o método de lista passando a classe
11
12    for (UsuarioPessoa usuarioPessoa : list) { // Percorrer a lista verificando se está correto
13        System.out.println(usuarioPessoa);
14        System.out.println("-----");
15    }
16 }
17 }
18 }
```

Carregando uma lista condicional

A linguagem HQL nada mais é do que um SQL Orientado a Objetos, quando estamos dentro do banco de dados nós pensamos em tabelas e suas relações e quando estamos escrevendo para o JPA nós pensamos em objetos.

No método que estou criando a lista é trazida do banco de dados quando os dados satisfazem a pesquisa e sua condição, para exemplificar estou passando o meu nome dentro do HQL dizendo para retornar apenas os dados onde o nome seja exatamente igual a 'Egidio'.

```

99 @Test
100 public void testeQueryList() {
101     DaoGeneric<UsuarioPessoa> daoGeneric = new DaoGeneric<UsuarioPessoa>();
102     List<UsuarioPessoa> list = daoGeneric.getEntityManager().
103         .createQuery(" from UsuarioPessoa where nome = 'Egidio'").getResultList();
104
105     for (UsuarioPessoa usuarioPessoa : list) {
106         System.out.println(usuarioPessoa);
107     }
108
109 }

```

Limitando máximo de resultados

Vamos supor que tenhamos a seguinte regra de negócio: Trazer apenas a 5 primeira pessoas ordenadas por id ou que foram cadastradas primeiro, ou melhor ainda trazer a 5 pessoas que foram cadastradas inicialmente, usando o createQuery e após a instrução setarmos o setMaxResults(5) estamos dizendo que nossa consulta ao banco somente irá trazer os 5 primeiro e a ordenação eu coloquei no próprio HQL.

```

113 @Test
114 public void testeQueryListMaxResult() {
115     DaoGeneric<UsuarioPessoa> daoGeneric = new DaoGeneric<UsuarioPessoa>();
116     List<UsuarioPessoa> list = daoGeneric.getEntityManager().
117         .createQuery(" from UsuarioPessoa order by id")
118         .setMaxResults(5).getResultList();
119
120     for (UsuarioPessoa usuarioPessoa : list) {
121         System.out.println(usuarioPessoa);
122     }
123
124 }
125

```

Usando parâmetros dinâmicos

O HQL possui uma sintaxe de passagem de parâmetros dinâmicos isso facilita muito a criação de métodos dinâmicos em nosso sistema em nosso caso estou passando dois parâmetros onde pode corresponder ao nome OU ao sobrenome com isso será retornado dados quando tiver registros com o nome ou com o sobrenome passados como parâmetros.

```

128 @Test
129 public void testeQueryListParameter() {
130     DaoGeneric<UsuarioPessoa> daoGeneric = new DaoGeneric<UsuarioPessoa>();
131
132     List<UsuarioPessoa> list = daoGeneric.getEntityManager()
133         .createQuery("from UsuarioPessoa where nome = :nome or sobrenome = :sobrenome")
134         .setParameter("nome", "sdsd").
135         setParameter("sobrenome", "Alex")
136         .getResultList();
137
138     for (UsuarioPessoa usuarioPessoa : list) {
139         System.out.println(usuarioPessoa);
140     }
141 }
142

```

Operações de média de uma coluna

Operações matemáticas são realizadas facilmente usando SQL e podemos usar esse poder junto com o JPA e unir os dois para termos um resultado final por exemplo média da idades das pessoas cadastradas, no exemplo abaixo fazemos exatamente isso, devemos perceber que o resultado vai ser um número e então pra isso no final usamos o `getSingleResult()` para obter o resultado numérico.

```

13 @Test
14 public void testeQuerySomaIdade() {
15
16     DaoGeneric<UsuarioPessoa> daoGeneric = new DaoGeneric<UsuarioPessoa>();
17
18     Double somaIdade = (Double) daoGeneric.getEntityManager()
19         .createQuery("select avg(u.idade) from UsuarioPessoa u ")
20         .getSingleResult();
21
22     System.out.println("Soma de todas as idades é --> " + somaIdade);
23 }
24
25

```


Anotação @NamedQuery e @NamedQueries

Uma consulta nomeada é uma consulta definida estaticamente com uma string de consulta imutável predefinida. O uso de consultas nomeadas em vez de consultas dinâmicas pode melhorar a organização do código separando as cadeias de consulta JPQL do código Java. Ele também reforça o uso de parâmetros de consulta em vez de embutir literais dinamicamente na cadeia de consulta e resulta em consultas mais eficientes.

A anotação @NamedQuery contém quatro elementos - dois dos quais são obrigatórios e dois são opcionais. Os dois elementos obrigatórios, nome e consulta, definem o nome da consulta e a própria string de consulta e são demonstrados acima. Os dois elementos opcionais, LockMode e dicas, fornecer substituição estática para os setLockMode e setHint métodos.

Anexar várias consultas nomeadas à mesma classe de entidade requer envolvê-las em uma anotação @NamedQueries

```

14 @Entity
15 @NamedQueries({
16
17     @NamedQuery(name = "UsuarioPessoa.todos",
18                 query = "select u from UsuarioPessoa u"),
19     @NamedQuery(name = "UsuarioPessoa.buscaPorNome",
20                 query = "select u from UsuarioPessoa u where u.nome = :nome")
21 })
22 public class UsuarioPessoa {
23
24     @Id
25     @GeneratedValue(strategy = GenerationType.AUTO)
26     private Long id;
27

```

Chamando um query nomeada

Anteriormente colocar as queries nomeadas e anotadas em cima da nossa classe persistente e agora é a hora de aprender como invocar essa query e executar ela.

Para isso usaremos o método `createNamedQuery` passando o nome da query e apenas mandamos executar e processar o retorno.

```

159  @Test
160  public void testeNameQuery1() {
161      DaoGeneric<UsuarioPessoa> daoGeneric = new DaoGeneric<UsuarioPessoa>();
162
163      List<UsuarioPessoa> list = daoGeneric.getEntityManager().
164          createNamedQuery("UsuarioPessoa.todos")// Query escrita na entidade
165          .getResultList();
166
167      for (UsuarioPessoa usuarioPessoa : list) {
168          System.out.println(usuarioPessoa);
169      }
170
171  }

```

Caso a query tenha parâmetros isso não muda nada no que já aprendemos, apenas chamamos a query pelo nome e passamos os parâmetros.

```

175  @Test
176  public void testeNameQuery2() {
177      DaoGeneric<UsuarioPessoa> daoGeneric = new DaoGeneric<UsuarioPessoa>();
178
179      List<UsuarioPessoa> list = daoGeneric.getEntityManager().
180          createNamedQuery("UsuarioPessoa.buscaPorNome")
181          .setParameter("nome", "Paulo")
182          .getResultList();
183
184      for (UsuarioPessoa usuarioPessoa : list) {
185          System.out.println(usuarioPessoa);
186      }
187
188  }

```

JPA - API de critérios

A API Criteria é uma API predefinida usada para definir consultas para entidades. É a maneira alternativa de definir uma consulta JPQL. Essas consultas são seguras, portáteis e fáceis de modificar, alterando a sintaxe.

Semelhante ao JPQL segue esquema abstrato (fácil de editar esquema) e objetos incorporados. A API de metadados é mesclada com a API de critérios para modelar entidades persistentes para consultas de critérios.

A principal vantagem da API de critérios é que os erros podem ser detectados mais cedo durante o tempo de compilação. Consultas JPQL baseadas em strings e consultas baseadas em critérios JPA são iguais em desempenho e eficiência.

A API Criteria e o JPQL estão intimamente relacionados e têm permissão para projetar usando operadores semelhantes em suas consultas.

Ele segue o pacote `javax.persistence.criteria` para projetar uma consulta. A estrutura de consulta significa a consulta de critérios de sintaxe.

A consulta de critérios simples a seguir retorna todas as instâncias da classe de entidade na fonte de dados.

```
230     @Test
231     public void testeCriteria1(){
232         DaoGeneric daoGeneric = new DaoGeneric();
233
234         CriteriaBuilder criteriaBuilder = daoGeneric.
235             getEntityManager().getCriteriaBuilder();
236
237         CriteriaQuery<UsuarioPessoa> criteriaQuery = criteriaBuilder.
238             createQuery(UsuarioPessoa.class);
239
240         Root<UsuarioPessoa> root = criteriaQuery.from(UsuarioPessoa.class);
241
242         criteriaQuery.select(criteriaQuery.from(UsuarioPessoa.class));
243
244         TypedQuery<UsuarioPessoa> query = daoGeneric.getEntityManager()
245             .createQuery(criteriaQuery);
246
247         List<UsuarioPessoa> list = query.getResultList();
248
249         for (UsuarioPessoa usuarioPessoa : list) {
250             System.out.println(usuarioPessoa);
251         }
252
253     }
```

A consulta demonstra as etapas básicas para criar um critério.

- EntityManager instância é usada para criar um objeto CriteriaBuilder .
- A instância CriteriaQuery é usada para criar um objeto de consulta. Os atributos desse objeto de consulta serão modificados com os detalhes da consulta.
- O método CriteriaQuery.from é chamado para definir a raiz da consulta.

- CriteriaQuery.select é chamado para definir o tipo de lista de resultados.
- A instância TypedQuery <T> é usada para preparar uma consulta para execução e especificar o tipo do resultado da consulta.
- Método getResultList no objeto TypedQuery <T> para executar uma consulta. Esta consulta retorna uma coleção de entidades, o resultado é armazenado em uma lista.

Bean Validation

O Bean Validation padroniza como definir e declarar restrições de nível de modelo de domínio. Você pode, por exemplo, expressar que uma propriedade nunca deve ser nula, que o saldo da conta deve ser estritamente positivo, etc.

Essas restrições de modelo de domínio são declaradas no próprio bean anotando suas propriedades. O Bean Validation pode então lê-los e verificar se há violações de restrição. O mecanismo de validação pode ser executado em diferentes camadas em seu aplicativo sem ter que duplicar nenhuma dessas regras (camada de apresentação, camada de acesso a dados). Seguindo o princípio DRY, Bean Validation e sua implementação de referência, o Hibernate Validator foi desenvolvido para essa finalidade.

A integração entre o Hibernate e o Bean Validation funciona em dois níveis. Primeiro, é capaz de verificar instâncias na memória de uma classe para violações de restrição. Segundo, ele pode aplicar as restrições ao metamodelo do Hibernate e incorporá-las ao esquema do banco de dados gerado.

Cada anotação de restrição é associada a uma implementação de validador responsável por verificar a restrição na instância da entidade. Um validador também pode (opcionalmente) aplicar a restrição ao metamodelo do Hibernate, permitindo que o Hibernate gere DDL que expressa a restrição. Com o ouvinte de eventos apropriado, você pode executar a operação de verificação em inserções, atualizações e exclusões feitas pelo Hibernate.

Ao verificar instâncias em tempo de execução, o Hibernate Validator retorna informações sobre violações de restrições em um conjunto de ConstraintViolations.

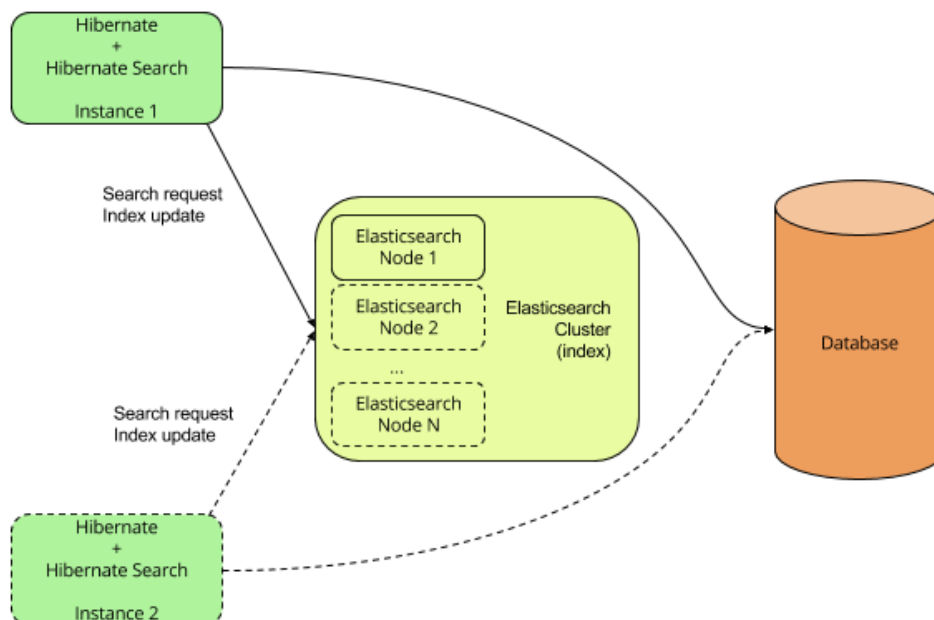
Entre outras informações, o `ConstraintViolation` contém uma mensagem de descrição de erro que pode incorporar o valor de parâmetro do pacote com a anotação (por exemplo, limite de tamanho), e seqüências de mensagens que podem ser externalizadas para um `ResourceBundle`.

Hibernate Search

Mecanismos de pesquisa de texto completo, como o Apache Lucene™, são uma tecnologia muito poderosa para fornecer consultas gratuitas de texto / eficiência aos aplicativos.

Se sofre várias incompatibilidades ao lidar com um modelo de domínio de objeto (mantendo o índice atualizado, incompatibilidade entre a estrutura de índice e o modelo de domínio, consultando incompatibilidade ...) .

Hibernate Search indexa seu modelo de domínio graças a algumas anotações, cuida de a sincronização de banco de dados / índice e traz de volta objetos gerenciados regulares a partir de consultas de texto livre.



Hibernate Envers

O projeto Envers visa permitir uma auditoria fácil de classes persistentes. Tudo o que você precisa fazer é anotar sua classe persistente ou algumas de suas propriedades, que você deseja auditar, com `@Audited`. Para cada entidade auditada, será criada uma tabela que conterá o histórico de alterações feitas na entidade. Você pode recuperar e consultar dados históricos sem muito esforço.

Similarmente ao Subversion, a biblioteca tem um conceito de revisões. Basicamente, uma transação é uma revisão (a menos que a transação não modifique nenhuma entidade auditada). Como as revisões são globais, com um número de revisão, você pode consultar várias entidades nessa revisão, recuperando uma visão (parcial) do banco de dados nessa revisão. Você pode encontrar um número de revisão com uma data e, ao contrário, você pode obter a data em que uma revisão foi confirmada.

A biblioteca funciona com o Hibernate e requer anotações do Hibernate ou gerenciador de entidades. Para que a auditoria funcione corretamente, as entidades devem ter identificadores exclusivos imutáveis (chaves primárias). Você pode usar o Envers onde quer que o Hibernate funcione: independente, dentro do JBoss AS, com o JBoss Seam ou Spring.

Considerações finais sobre o JPA

O JPA é o frameworks mais usado no mundo quando se fala de persistência de dados em Java, vimos como é poderoso e quanto facilita nosso trabalho não precisando escrever tantas linha de código e ficar atento a tantos detalhes porque usando um frameworks bugs, erro e manutenção diminuem muito mesmo em um sistema.

Spring Data

O Spring Data JPA é um framework que nasceu para facilitar a criação dos nossos repositórios.

Ele faz isso nos liberando de ter que implementar as interfaces referentes aos nossos repositórios (ou DAOs), e também já deixando pré-implementado algumas funcionalidades como, por exemplo, de ordenação das consultas e de paginação de registros.

Ele (o Spring Data JPA) é, na verdade, um projeto dentro de um outro maior que é o Spring Data. O Spring Data tem por objetivo facilitar nosso trabalho com persistência de dados de uma forma geral. E além do Spring Data JPA, ele possui vários outros projetos:

Configuração do Maven e Spring Data

Antes de tudo precisamos adicionar as lib/jar ao nosso projeto e para não precisarmos passar o dia inteiro baixando na internet um a um usaremos o Maven para baixar de uma vez tudo pra gente, então abaixo está as configurações do pom.xml.

```
50      <!-- https://mvnrepository.com/artifact/org.springframework.data/spring-data-jpa -->
51      <dependency>
52          <groupId>org.springframework.data</groupId>
53          <artifactId>spring-data-jpa</artifactId>
54          <version>2.0.10.RELEASE</version>
55      </dependency>
56
57      <!-- https://mvnrepository.com/artifact/org.springframework/spring-test -->
58      <dependency>
59          <groupId>org.springframework</groupId>
60          <artifactId>spring-test</artifactId>
61          <version>5.1.0.RELEASE</version>
62          <scope>test</scope>
63      </dependency>
```

Configurando o Persistence para integrar com Spring

Agora integrando com Spring Frameworks nós apenas vamos configurar o nome para o persistence-unit do JPA para que seja possível ativar os recursos em nosso projeto.

A única declaração que teremos que manter nesse arquivo é a declaração para as entidades persistentes.

```
8 <persistence-unit name="projeto-spring-data-aula">
9   <class>projeto.spring.data.aula.model.UsuarioSpringData</class>
10  <class>projeto.spring.data.aula.model.Telefone</class>
11 </persistence-unit>
```

Configurando o arquivo Spring-Config.xml

Esse arquivo em formato XML é lido pelo Spring quando o projeto é iniciado e suas configurações adicionadas ao contexto do sistemas rodando no servidor. Neste arquivo vamos configurar nossos repositórios, nossa injeção de dependência, nossas configurações com o banco de dados e ativar a integração com JPA + Hibernate + Spring.

Ativando a auto configuração do Spring

Nesta parte iremos falar para o Spring fazer as configurações lendo as anotações em nossas classes Java.

```
14 <!-- Ativa recursos automáticos no Spring como Injeção de Dependência por anotações -->
15 <context:annotation-config />
16 <context:component-scan base-package="projeto.spring.data.aula.model" />
```

Configurando a conexão com o banco de dados

Nesta parte iremos configurar a conexão com nosso banco de dados, é igual configurar com jdbc porém aqui quem vai cuidar dos processos mais complexos é o Spring.

```

19      <!-- Define o DataSource e a conexão com o banco de dados -->
20      <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
21          <property name="driverClassName" value="org.postgresql.Driver"/>
22          <property name="url" value="jdbc:postgresql://localhost:5432/spring-data-teste-aula"/>
23          <property name="username" value="postgres"/>
24          <property name="password" value="admin"/>
25      </bean>

```

Configurando o JPA integrando com o Spring e o Hibernate

Nesta parte vamos configurar a conexão criada com DataSource a ligar com o JPA do Spring Frameworks. Neste ponto vamos configurar nosso persistence unit e também onde são configurados as propriedades para o hibernate

```

28      <!-- Define as configurações do JPA -->
29      <bean id="entityManagerFactory"
30          class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean" scope="singleton">
31          <property name="dataSource" ref="dataSource"/>
32          <property name="jpaVendorAdapter">
33              <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
34                  <property name="database" value="POSTGRESQL"/>
35                  <property name="generateDdl" value="true"/> <!-- Gerar as tabelas no banco -->
36              </bean>
37          </property>
38          <property name="persistenceUnitName" value="projeto-spring-data-aula"/>
39      </bean>
40

```

Controle Transacional e Repository

Neste momento estamos ativando a parte de controle transacional para o Spring controlar as transações/operações do nosso sistemas com o banco de dados e

também a configuração de nossos repositories que serão usados para escrever nossos métodos de operações de persistência.

```

41      <!-- Ativa o controle transacional -->
42      <bean id="transactionManager"
43          class="org.springframework.orm.jpa.JpaTransactionManager" scope="singleton">
44          <property name="entityManagerFactory" ref="entityManagerFactory" />
45      </bean>
46
47      <!-- Ativa os recursos para os Repository -->
48      <jpa:repositories base-package="projeto.spring.data.aula.dao" />

```

Interface CrudRepository

A interface central na abstração do repositório Spring Data é Repository (provavelmente não é uma grande surpresa).

Leva a classe de domínio para gerenciar, bem como o tipo de id da classe de domínio como argumentos de tipo. Essa interface age principalmente como uma interface de marcação para capturar os tipos com os quais trabalhar e para ajudá-lo a descobrir as interfaces que a estendem.

O CrudRepository fornece funcionalidade CRUD sofisticada para a classe de entidade que está sendo gerenciada.

Normalmente, a sua interface de repositório vai estender Repository, CrudRepository ou PagingAndSortingRepository.

Alternativamente, se você não deseja estender interfaces Spring Data, você também pode anotar sua interface de repositório com @RepositoryDefinition.

Estendendo CrudRepository expõe um conjunto completo de métodos para manipular suas entidades.

Se você preferir ser seletivo sobre os métodos que estão sendo expostos, basta copiar os que você deseja expor CrudRepository para o seu repositório de domínio.

```

22      @Transactional(readOnly = true)
23      @Query(value = "select p from UsuarioSpringData p where p.nome like %?1%")
24      public List<UsuarioSpringData> buscaPorNome (String nome);
25

```


Anotação @Repository

@Repository anota classes na camada de persistência, que atuará como um repositório de banco de dados, toda camada do sistema que possui, classes normal ou mesmo interfaces anotadas com @Repository identifica que são responsáveis por processos no banco de dados e a chamada cama de persistência.

```
17 @Repository
18 public interface InterfaceSpringDataUser
19 extends CrudRepository<UsuarioSpringData, Long> {
20
```

Quando se trabalha com Spring Data usamos a interface CrudRepository que nos dispões de todo o poder de persistencia com métodos prontos e fáceis de usar e também torna super fácil a implementação de novas queries para nosso sistemas.

Anotação @Transactional

Além da abordagem declarativa baseada em XML para configuração de transação, você pode usar uma abordagem baseada em anotação.

Declarar semântica de transação diretamente no código-fonte Java coloca as declarações muito mais próximas do código afetado.

Não há muito perigo de acoplamento indevido, porque o código que se destina a ser usado transacionalmente é quase sempre implantado dessa maneira de qualquer maneira.

```
22 @Transactional(readOnly = true)
23 @Query(value = "select p from UsuarioSpringData p where p.nome like %?1%")
24 public List<UsuarioSpringData> buscaPorNome (String nome);
25
```

A facilidade de uso proporcionada pelo uso da `@Transactional` anotação é melhor ilustrada com um exemplo, que é explicado no texto a seguir. Considere a seguinte definição de classe.

Anotação @Query

Usar consultas nomeadas para declarar consultas para entidades é uma abordagem válida e funciona bem para um pequeno número de consultas.

Como as consultas em si estão vinculadas ao método Java que as executa, você pode vinculá-las diretamente usando a `@Query` anotação Spring Data JPA, em vez de anotá-las na classe de domínio.

Isso liberará a classe de domínio de informações específicas de persistência e colocará a consulta na interface do repositório.

```
22 @Transactional(readOnly = true)
23 @Query(value = "select p from UsuarioSpringData p where p.nome like ?1%")
24 public List<UsuarioSpringData> buscaPorNome (String nome);
25
```

Anotação @Modifying

Mas se os métodos não são implementados pelo desenvolvedor, e sim apenas a assinatura deles, como criar métodos de update ou delete, específicos para uma dada situação? Bem, para isso, o Spring-Data fornece a anotação `@Modifying`

A anotação `@Modifying` deve ser utilizada em conjunto com a anotação `@Query`.

Na anotação `@Query` se adiciona a JPQL em questão, referente a operação de update ou delete.

E a anotação `@Modifying` será usada apenas para informar ao Spring-Data que a `@Query` não possui um método de consulta, mas sim, um método de escrita.

Outro ponto importante que deve ser levado em conta é o uso da anotação `@Transactional` com o atributo `readOnly` setado como `false`.

Isto porque, todas as interfaces que herdam `CrudRepository` são por padrão do tipo `readOnly = true`, ou seja, todos os métodos que você assinar na interface, serão do tipo leitura e não de escrita.

Então, para evitar uma exceção na transação dos métodos anotados com `@Modifying`, é preciso inserir a `@Transactional(readOnly = false)`, para dizer que esta transação não será para leitura e sim de escrita.

```
38 @Modifying
39 @Transactional
40 @Query("delete from UsuarioSpringData u where u.nome = ?1")
41 public void deletePorNome(String nome);
42
43 @Modifying
44 @Transactional
45 @Query("update UsuarioSpringData u set u.email = ?1 where u.nome = ?2")
46 public void updateEmailPorNome(String email, String nome);
```

Sobrescrevendo métodos de interface

É possível sobrescrever métodos de interfaces no Java e na parte de persistência é muito comum acontecer verificações antes de salvar algum registro, caso seja necessário criar um método específico dentro do Repository é possível fazer usando truques da linguagem Java.

```
32
33 default <S extends UsuarioSpringData> S saveAtual(S entity) {
34     // Processa outros métodos e regar antes de salvar
35     return save(entity);
36 }
37
```

Junit com Spring Data

Junit é um dos frameworks mais famosos no mundo Java para a criação de testes unitários, como nesses momentos não temos telas do sistema para testar a camada de persistência do sistema é super útil implementar testes para verificar se tudo está funcionando corretamente em nossos sistemas.

O que temos que fazer é quando algum método que temos que testar antes de tudo o arquivo de configuração do Spring tem que ser lido para podermos ter todos os recursos acessíveis.

Essa configuração é feita com anotações e nela passamos o caminho do nosso arquivo para que seja lido e configurado automaticamente.

```
17 @RunWith(SpringJUnit4ClassRunner.class)
18 @ContextConfiguration(locations = { "classpath:META-INF/spring-config.xml" })
19 public class AppSpringDataTest {
20
```

Anotação @Autowired

Injeção de dependências (ou Dependency Injection – DI) é um tipo de inversão de controle (ou Inversion of Control – IoC) que dá nome ao processo de prover instâncias de classes que um objeto precisa para funcionar.

Marca um construtor, um campo, um método setter ou um método de configuração para ser autoperfurado pelos recursos de injeção de dependência do Spring.

Depois que a injeção de anotação é ativada, o autowiring pode ser usado em propriedades, setters e construtores.

Apenas um construtor (no máximo) de qualquer classe de bean pode levar esta anotação, indicando que o construtor será ativado quando usado como um bean Spring. Tal construtor não precisa ser público.

Os campos são injetados logo após a construção de um bean, antes que qualquer método de configuração seja invocado. Esse campo de configuração não precisa ser público.

Os métodos de configuração podem ter um nome arbitrário e qualquer número de argumentos; Cada um desses argumentos será autowired com um bean correspondente no contêiner Spring. Os métodos de definição de propriedade do bean são efetivamente apenas um caso especial de um método de configuração geral. Esses métodos de configuração não precisam ser públicos.

No caso de um construtor ou método multi-arg, o parâmetro 'required' é aplicável a todos os argumentos. Parâmetros individuais podem ser declarados como estilo Java-8 Optional ou, como no Spring Framework 5.0, também como @Nullable ou um tipo de parâmetro não-nulo em Kotlin, substituindo a semântica necessária básica.

No caso de um tipo Collection ou Map dependência, o contêiner retira todos os beans correspondentes ao tipo de valor declarado. Para tais propósitos, as chaves do mapa devem ser declaradas como tipo String, o qual será resolvido para os nomes de beans correspondentes. Tal coleta fornecida por contêiner será ordenada, levando em conta Ordered/ Order valores dos componentes de destino, caso contrário, seguindo sua ordem de registro no contêiner.

Como alternativa, um único bean de destino correspondente também pode ser geralmente digitado Collection ou em Map si, sendo injetado como tal.

```
21 @Autowired
22 private InterfaceSpringDataUser interfaceSpringDataUser;
23
```

Teste unitário @Test

O conceito de Desenvolvimento Guiado por Testes define que antes de criarmos um código novo (classe), devemos escrever um teste (classe de test case) para ele. Essa prática traz vários benefícios às equipes de desenvolvimento e inclusive estes testes serão usados como métrica em todo o tempo de vida do projeto.

Imagine por exemplo, se um avião só fosse testado após a conclusão de sua construção, com certeza isso seria um verdadeiro desastre, é nesse ponto que a engenharia aeronáutica é uma boa referência em processos de construções de projetos de software, principalmente em sistemas de missão crítica, pois durante a construção e montagem de um avião todos os seus componentes são testados isoladamente até a exaustão, e depois cada etapa de integração também é devidamente testada e homologada.

O teste unitário, de certa forma se baseia nessa ideia, pois é uma modalidade de testes que se concentra na verificação da menor unidade do projeto de software. É realizado o teste de uma unidade lógica, com uso de dados suficientes para se testar apenas a lógica da unidade em questão.

Em sistemas construídos com uso de linguagens orientadas a objetos, essa unidade pode ser identificada como um método, uma classe ou mesmo um objeto.

Criando o teste de inserir

Anotando nosso método com `@Test` ele será executado automaticamente e nossos métodos serão rodados simulando 100% como se fosse nosso sistema rodando em produção.

Então para cadastrar nosso objetos no banco de dados precisamos instanciar esse objeto e setar os atributos que serão cadastrados no banco de dados e pra finalizar apenas chamamos nosso método de salvar.

```
27 @Test
28 public void testeInsert() {
29
30     UsuarioSpringData usuarioSpringData = new UsuarioSpringData();
31
32     usuarioSpringData.setEmail("javaavancado@javaavancado.com");
33     usuarioSpringData.setIdade(31);
34     usuarioSpringData.setLogin("teste 123");
35     usuarioSpringData.setSenha("123");
36     usuarioSpringData.setNome("Egidio Alex");
37
38     interfaceSpringDataUser.save(usuarioSpringData);
39
40     System.out.println("Usuario cadastrado -> " + interfaceSpringDataUser.count());
41
42 }
```

Criando o teste de consulta

Anotando nosso método com `@Test` ele será executado automaticamente e nossos métodos serão rodados simulando 100% como se fosse nosso sistema rodando em produção.

Então para consultar nosso objetos no banco de dados precisamos apenas chamar nossa interface de persistencia e invocar o método de consulta que nosso caso é o `findById`.

O Spring Data quando efetuamos consulta a um objeto ele é retornado dentro de um objeto chamado `Optional` que possui alguns métodos que ajudam em verificações e o mais importante é o método `GET` que retorna o objeto em si que

precisamos da nossa camada de modelo e então para isso nós resgatamos ele da seguinte forma → `usuarioSpringData.get()`;

```

44 @test
45 public void testeConsulta() {
46
47     Optional<UsuarioSpringData> usuarioSpringData = interfaceSpringDataUser.findById(1L);
48
49     System.out.println(usuarioSpringData.get().getIdade());
50     System.out.println(usuarioSpringData.get().getEmail());
51     System.out.println(usuarioSpringData.get().getIdade());
52     System.out.println(usuarioSpringData.get().getLogin());
53     System.out.println(usuarioSpringData.get().getNome());
54     System.out.println(usuarioSpringData.get().getSenha());
55     System.out.println(usuarioSpringData.get().getId());
56
57     for (Telefone telefone : usuarioSpringData.get().getTelefones()){
58         System.out.println(telefone.getNumero());
59         System.out.println(telefone.getTipo());
60         System.out.println(telefone.getId());
61         System.out.println(telefone.getUsuarioSpringData().getNome());
62         System.out.println("-----");
63     }
64 }

```

Criando o teste de consulta por lista

Anotando nosso método com `@Test` ele será executado automaticamente e nossos métodos serão rodados simulando 100% como se fosse nosso sistema rodando em produção.

Então para consultar todos os dados de nossos objetos no banco de dados precisamos apenas chamar nossa interface de persistencia e invocar o método de consulta que nosso caso é o `findAll`.

O Spring Data quando efetuamos consulta por lista é retornado diretamente a lista de objetos já pronta para nós então para trabalhar esse lista nós precisamos percorrer ela e processar da forma que queremos em nosso sistema.

```

66 @Test
67 public void testeConsultaTodos() {
68     Iterable<UsuarioSpringData> lista = interfaceSpringDataUser.findAll();
69
70     for (UsuarioSpringData usuarioSpringData : lista) {
71
72         System.out.println(usuarioSpringData.getEmail());
73         System.out.println(usuarioSpringData.getIdade());
74         System.out.println(usuarioSpringData.getLogin());
75         System.out.println(usuarioSpringData.getNome());
76         System.out.println(usuarioSpringData.getSenha());
77         System.out.println(usuarioSpringData.getId());
78         System.out.println("-----");
79     }
80 }
81

```

Criando o teste de update

Para efetuarmos o update/atualização de dados em nosso banco de dados nós precisamos resgatar esse objeto do banco de dados e após isso como ele em mãos setar os atributos com novos valores e invocar novamente o método de salvar para gravar as alterações no banco de dados.

```

82 @Test
83 public void testeUpdate() {
84
85     Optional<UsuarioSpringData> usuarioSpringData = interfaceSpringDataUser.findById(3L);
86
87     UsuarioSpringData data = usuarioSpringData.get();
88
89     data.setNome("Alex Egidio Update Spring Data");
90     data.setIdade(25);
91
92     interfaceSpringDataUser.save(data);
93 }

```

Criando o teste de delete

Para efetuarmos o delete/remoção de um registro no banco de dados nós podemos buscar no banco de dados e passar para nosso método de delete ou também podemos passar um delete direto para o banco, mas aqui nesse exemplo estou buscando os dados no banco e passando para nossa camada de persistencia deletar para ter a certeza que está funcionando a rotina que foi criada.

```

35 @Test
36 public void testeDelete() {
37     Optional<UsuarioSpringData> usuarioSpringData = interfaceSpringDataUser.findById(3L);
38
39     interfaceSpringDataUser.delete(usuarioSpringData.get());
40 }
41

```

Consulta assíncrona

As consultas de repositório podem ser executadas de forma assíncrona usando o recurso de execução de método assíncrono do Spring .

Isso significa que o método retorna imediatamente após a chamada enquanto a execução real da consulta ocorre em uma tarefa que foi enviada para um Spring TaskExecutor.

A execução de consulta assíncrona é diferente da execução de consulta reativa e não deve ser mesclada.

Em poucas palavras isso significa que uma consulta muito demorada fica rodando por baixo dos panos até ficar concluída e isso é feito com `@Async`.

```

22 @Async
23 @Transactional(readOnly = true)
24 @Query(value = "select p from UsuarioSpringData p where p.nome like %?1%")
25 public List<UsuarioSpringData> buscaPorNomeAsync (String nome);
26

```

Usando Sort

A classificação pode ser feita fornecendo um PageRequest usando Sort diretamente. As propriedades realmente usadas nas Order instâncias de Sort necessidade correspondem ao seu modelo de domínio, o que significa que elas precisam ser resolvidas para uma propriedade ou um alias usado na consulta. O JPQL define isso como uma expressão de caminho de campo de estado.

No entanto, o uso em Sort conjunto `@Query` permite que você se esgueire em Order instâncias não verificadas por caminho contendo funções dentro da ORDER

BYcláusula. Isso é possível porque o Orderé anexado à string de consulta fornecida.

Por padrão, o Spring Data JPA rejeita qualquer Order instância que contenha chamadas de função, mas você pode usar JpaSort.unsafe para adicionar pedidos potencialmente inseguros.

Então para criarmos nossa consulta com ordenação nós criamos um método que recebe o objeto Sort e quando formos chamar a rotina nós passar o Sort e a propriedade do objeto que é para usar na nossa ordenação.

Método criado em nosso repositório.

```

34 @Transactional(readOnly = true)
35 @Query(value = "select p from UsuarioSpringData p where p.nome like %?1%")
36 public List<UsuarioSpringData> buscaPorNomeSort (String nome, Sort sort);
27

```

E na chamada podemos passar um ou vários atributos para ordenação.

```

L60 @Test
L61 public void testeConsultaNomeParamSort() {
L62
L63     List<UsuarioSpringData> list = interfaceSpringDataUser.
L64         buscaPorNomeSort("Egidio", Sort.by("id"));
L65
L66     for (UsuarioSpringData usuarioSpringData : list) {
L67
L68         System.out.println(usuarioSpringData.getEmail());
L69         System.out.println(usuarioSpringData.getIdade());
L70         System.out.println(usuarioSpringData.getLogin());
L71         System.out.println(usuarioSpringData.getNome());
L72         System.out.println(usuarioSpringData.getSenha());
L73         System.out.println(usuarioSpringData.getId());
L74         System.out.println("-----");
L75     }
L76
L77 }
L78
L79 }

```

Auditoria

O Spring Data fornece suporte sofisticado para controlar de forma transparente quem criou ou mudou uma entidade e quando a mudança aconteceu. Para se beneficiar dessa funcionalidade, você precisa equipar suas classes de entidade com metadados de auditoria que podem ser definidos usando anotações ou implementando uma interface.

Metadados de auditoria baseados em anotação

Nós fornecemos `@CreatedBy` `@LastModifiedBy` para capturar o usuário que criou ou modificou a entidade, bem como `@CreatedDate` `@LastModifiedDate` para capturar quando a mudança aconteceu.

Como você pode ver, as anotações podem ser aplicadas seletivamente, dependendo de quais informações você deseja capturar.

Caso você não queira usar anotações para definir metadados de auditoria, você pode permitir que sua classe de domínio implemente a `Auditable` interface. Ele expõe métodos setter para todas as propriedades de auditoria.

Há também uma classe base de conveniência `AbstractAuditable`, que você pode estender para evitar a necessidade de implementar manualmente os métodos de interface.

Isso aumenta o acoplamento de suas classes de domínio ao Spring Data, que pode ser algo que você deseja evitar. Normalmente, a maneira baseada em anotação de definir metadados de auditoria é preferida, pois é menos invasiva e mais flexível.

Caso você use uma `@CreatedBy` ou outra `@LastModifiedBy`, a infraestrutura de auditoria precisa, de alguma forma, tomar conhecimento do principal atual. Para isso, fornecemos uma `AuditorAware<T>` interface SPI que você precisa implementar para informar a infraestrutura que o usuário ou sistema atual está interagindo com o aplicativo.

O tipo genérico `T` define com que tipo as propriedades são anotadas `@CreatedBy` `@LastModifiedBy` precisam ser.

Considerações finais sobre o Spring Data

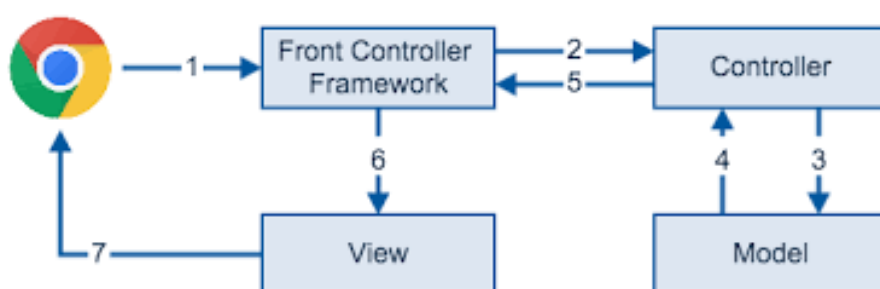
Vimos a facilidade que um frameworks de persistência trás pra gente, como operações complexas se tornam muito fáceis com uso de anotações e como o código fica cada vez mais simples e eficiente estudando e aplicando técnicas oferecidas pelo frameworks ao nosso projeto.

Spring Web MVC

O Spring Web MVC é o framework web original criado na API Servlet e foi incluído no Spring Framework desde o início. O nome formal, “Spring Web MVC”, vem do nome do seu módulo de origem (spring-webmvc), mas é mais comumente conhecido como “Spring MVC”.

O Spring MVC, como muitos outros frameworks web, é projetado em torno do padrão de front controller, onde a central Servlet, the DispatcherServlet, fornece um algoritmo compartilhado para o processamento de solicitações, enquanto o trabalho real é executado por componentes delegados configuráveis. Este modelo é flexível e suporta diversos fluxos de trabalho.

O DispatcherServlet, como qualquer Servlet, precisa ser declarado e mapeado de acordo com a especificação Servlet usando a configuração Java ou em web.xml. Por sua vez, o DispatcherServlet usa a configuração Spring para descobrir os componentes delegados necessários para o mapeamento de solicitações, a resolução da exibição, o tratamento de exceções e muito mais .



Spring RESTful

O REST tornou-se rapidamente o padrão de fato para a criação de serviços da Web na Web, porque eles são fáceis de construir e fáceis de consumir.

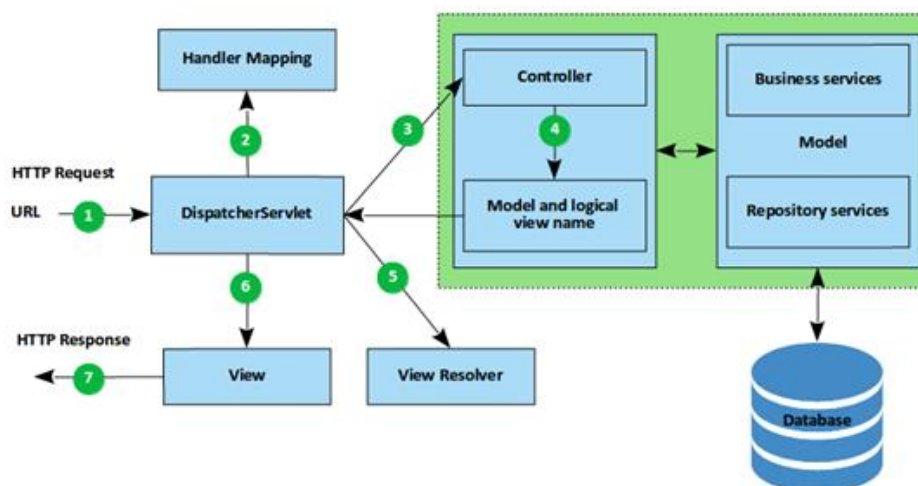
Há uma discussão muito maior sobre como o REST se encaixa no mundo dos microsserviços, mas - para este tutorial - vamos apenas olhar para a construção de serviços RESTful.

Por que REST? REST abraça os preceitos da web, incluindo sua arquitetura, benefícios e tudo mais. Isso não é surpresa, já que seu autor, Roy Fielding, esteve

envolvido em provavelmente uma dúzia de especificações que governam como a web opera.

Quais benefícios? A web e seu protocolo principal, HTTP, fornecem uma pilha de recursos:

Ações adequadas (GET, POST, PUT, DELETE, ...)



Cache

Redirecionamento e encaminhamento

Segurança (criptografia e autenticação)

Todos esses são fatores críticos na construção de serviços resilientes. Mas isso não é tudo. A web é construída a partir de muitas especificações minúsculas, por isso, tem sido capaz de evoluir facilmente, sem se atolar em "guerras de padrões".

Os desenvolvedores são capazes de utilizar ferramentas de terceiros que implementam essas diversas especificações e, instantaneamente, têm a tecnologia de cliente e servidor na ponta dos dedos.

Assim, construindo sobre HTTP, as APIs REST fornecem os meios para construir APIs flexíveis que podem:

Suportar compatibilidade com versões anteriores

- APIs evolutivas
- Serviços escalonáveis
- Serviços seguráveis

- Um espectro de serviços sem estado a stateful

O que é importante perceber é que o REST, por mais onipresente que seja, não é um padrão, por si só, mas uma abordagem, um estilo, um conjunto de restrições em sua arquitetura que pode ajudá-lo a construir sistemas em escala da web. Neste tutorial, usaremos o portfólio do Spring para criar um serviço RESTful, aproveitando os recursos sem pilha do REST.

Spring Boot

Se você está começando com o Spring Boot, ou “Spring” em geral, comece lendo esta seção. Ele responde às questões básicas “o quê?”, “Como?” E “por quê?”. Inclui uma introdução ao Spring Boot, juntamente com instruções de instalação. Nós, em seguida, orientá-lo através da construção de sua primeira aplicação Spring Boot, discutindo alguns princípios fundamentais como nós vamos.

O Spring Boot facilita a criação de aplicativos baseados em Spring autônomos e de produção que você pode executar. Adotamos uma visão opinativa da plataforma Spring e de bibliotecas de terceiros, para que você possa começar com o mínimo de barulho. A maioria dos aplicativos Spring Boot precisa de uma configuração de Spring muito pequena.

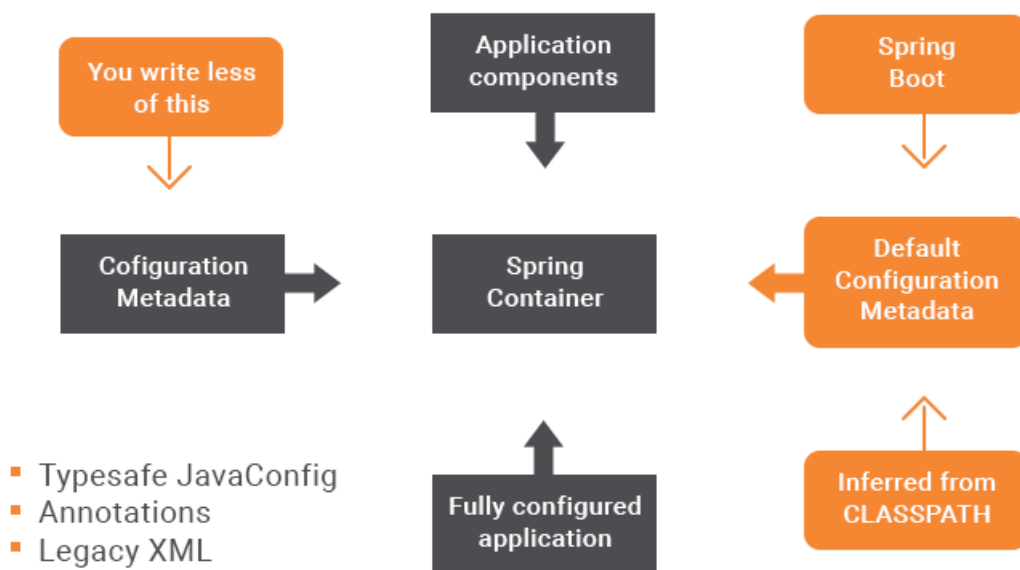
Você pode usar o Spring Boot para criar aplicativos Java que podem ser iniciados usando `java -jar` ou mais implementações tradicionais de guerra.

Nossos principais objetivos são:

- Forneça uma experiência de introdução radicalmente mais rápida e amplamente acessível para todo o desenvolvimento do Spring.
- Seja opinativo fora da caixa, mas saia do caminho rapidamente, pois os requisitos começam a divergir dos padrões.

- Forneça vários recursos não funcionais que são comuns a grandes classes de projetos (como servidores incorporados, segurança, métricas, verificações de integridade e configuração externalizada).
- Absolutamente nenhuma geração de código e nenhum requisito para configuração XML.

Spring Boot Simplifies Configuration



Spring Batch

Muitos aplicativos dentro do domínio corporativo exigem processamento em massa para realizar negócios operações em ambientes de missão crítica. Essas operações de negócios incluem:

- Processamento automatizado e complexo de grandes volumes de informação que é mais eficiente processado sem interação do usuário. Essas operações geralmente incluem eventos baseados em tempo cálculos, avisos ou correspondências do final do mês).
- Aplicação periódica de regras de negócios complexas processadas repetidamente em conjuntos de dados muito grandes (por exemplo, determinação de benefícios de seguro ou ajustes de taxa).

- Integração de informações recebidas de sistemas internos e externos que normalmente requer formatação, validação e processamento de forma transacional para o sistema de registro. O processamento em lote é usado para processar bilhões de transações todos os dias para empresas.

O Spring Batch é uma estrutura em lote leve e abrangente projetada para permitir o desenvolvimento de aplicações em lote robustas, vitais para as operações diárias dos sistemas corporativos.

Constrói sobre as características do Spring Framework que as pessoas esperam (produtividade, Abordagem de desenvolvimento baseada em POJO e facilidade geral de uso), facilitando ao acesso e alavancar serviços empresariais mais avançados quando necessário.

Spring Batch não é um estrutura de agendamento. Existem muitos planejadores corporativos bons (como Quartz, Tivoli, ControlM, etc.) disponível nos espaços comerciais e de código aberto.

Pretende-se trabalhar em conjunção com um agendador, não substitua um agendador.

O Spring Batch fornece funções reutilizáveis que são essenciais no processamento de grandes volumes de registros. incluindo registro / rastreamento, gerenciamento de transações, estatísticas de processamento de tarefas, reinicialização de tarefas, ignorar e gestão de recursos.

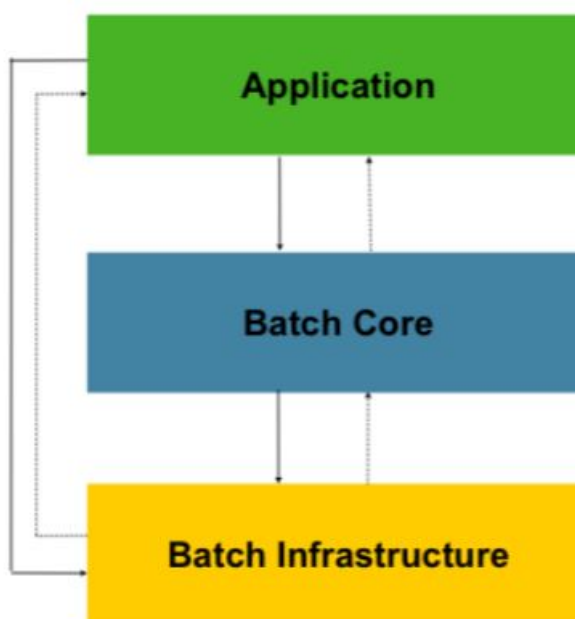
Ele também fornece serviços técnicos e recursos mais avançados que permitem trabalhos em lote de volume extremamente alto e alto desempenho por meio de otimização e particionamento técnicas.

O Spring Batch pode ser usado em ambos os casos de uso simples (como ler um arquivo em um banco de dados ou executando um procedimento armazenado), bem como casos de uso complexos e de alto volume (como volumes de dados entre bancos de dados, transformando-os e assim por diante). Trabalhos em lote de alto volume aproveite a estrutura de maneira altamente escalonável para processar volumes significativos de informações.

Spring Batch Architecture

O Spring Batch é projetado com extensibilidade e um grupo diversificado de usuários finais em mente.

A figura abaixo mostra a arquitetura em camadas que suporta a extensibilidade e a facilidade de uso para o usuário final desenvolvedores.



Essa arquitetura em camadas destaca três principais componentes de alto nível: Application, Core e a infraestrutura. O aplicativo contém todos os trabalhos em lote e código personalizado escritos por desenvolvedores usando Lote em Spring. O Núcleo de Lote contém as classes principais de tempo de execução necessárias para iniciar e controlar trabalho em lote. Inclui implementações para JobLauncher, Job e Step. Aplicativo e Núcleo são

construído em cima de uma infra-estrutura comum. Esta infra-estrutura contém leitores e escritores comuns e serviços (como o RetryTemplate), que são usados tanto por desenvolvedores de aplicativos (leitores e escritores, como ItemReader e ItemWriter) e o próprio framework principal (tente novamente, que é própria biblioteca).

Conclusão

Vimos várias forma de persistência com banco de dados, vimos a evolução dos frameworks em Java e como estão se tornando poderosos ajudando toda a comunidade de programadores pelo mundo.

Todo o conteúdo desde livro é obrigatório para que qualquer desenvolver Java tenha sucesso e seja um ótimo profissional.

Grande abraços e até mais.

Cursos para você ser profissional em programação



JAVA WEB

FORMAÇÃO COMPLETA

Full-Stack Web Java EE

Mais de 600 aulas em PrimeFaces, JSF, Spring, Hibernate, JPA, Jreport, CDI e muitos mais. Iremos do básico ao avançado.

2.985 alunos ⌚ 150h 🏆 certificado

[CONHECER O CURSO](#)



PACOTE FULLSTACK

PHP Expert 4 CURSOS COMPLETOS

[INSCREVA-SE AGORA!](#)

Full-Stack Web PHP

Domine as principais tecnologias do mercado e se torne Desenvolvedor Full-Stack, com salários na faixa dos R\$5.000,00.

1.600 alunos ⌚ 150h 🏆 certificado

[CONHECER O CURSO](#)



CURSO INGLÊS PARA PROGRAMADORES

100% FOCADO NO QUE É IMPORTANTE

Inglês para Programador

Saber inglês o aprendizado de programação extremamente fácil. Aprenda inglês de uma forma divertida e 100% prática.

798 alunos ⌚ 38h 🏆 certificado

[CONHECER O CURSO](#)



CURSO DESENVOLVIMENTO DE GAMES COMPLETO JAVA

Games com JAVA

São 7 jogos desenvolvidos no curso e com uma própria game engine, desenvolvida também no curso completo.

648 alunos ⌚ 40h 🏆 certificado

[CONHECER O CURSO](#)

Referências

<https://www.javaavancado.com/formacao-java-web-profissional/index.html>

<https://www.objectdb.com/api/java/jpa>

<https://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/entity.html>

http://docs.jboss.org/hibernate/orm/5.3/quickstart/html_single/

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

<https://docs.spring.io/spring-data/data-jpa/docs/1.4.x/reference/htmlsingle/>

<https://www.tutorialspoint.com/jdbc/>

<https://www.tutorialspoint.com/hibernate/>

<https://www.tutorialspoint.com/spring/>

<https://www.javaavancado.com>