



JPA e Spring Boot

O que é o JPA - Java Persistence API - uma biblioteca que armazena e recupera objetos que são armazenados em bancos de dados. Ele é responsável por fazer todas as instruções SQL sem que precisemos escrever uma única linha em nosso código (tudo é gerado nos bastidores).

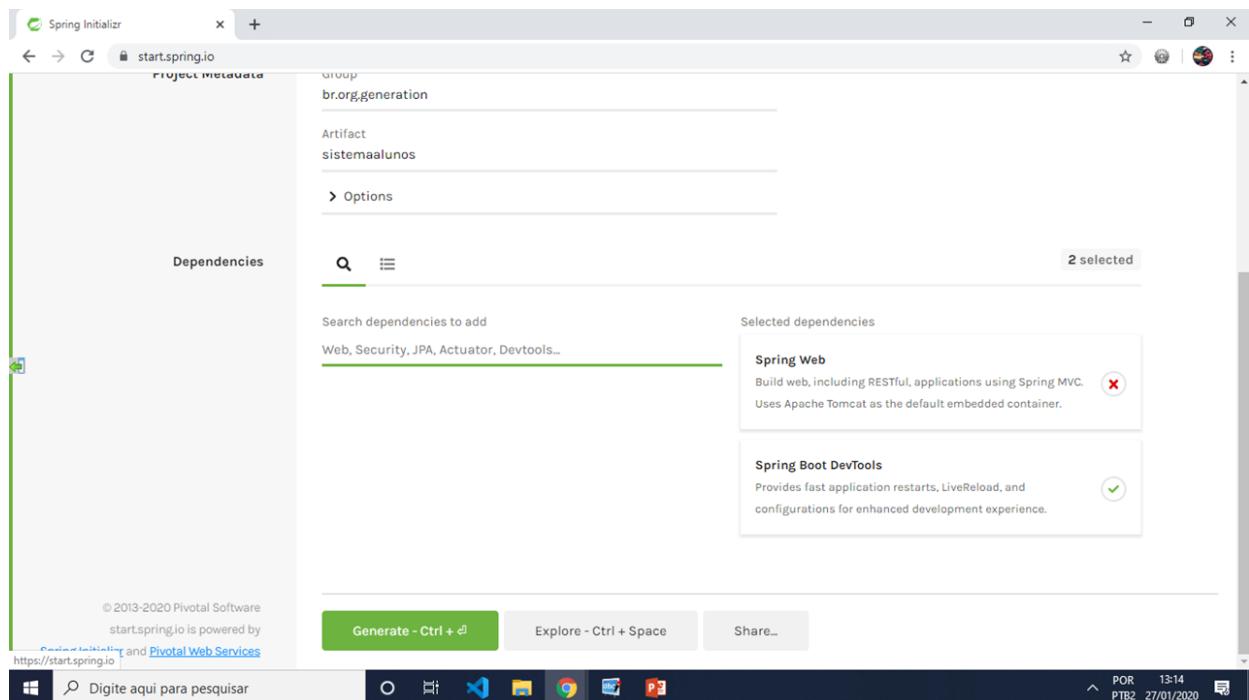
Setup do projeto

Vamos criar nosso projeto no mesmo modelo de antes, conforme os passos

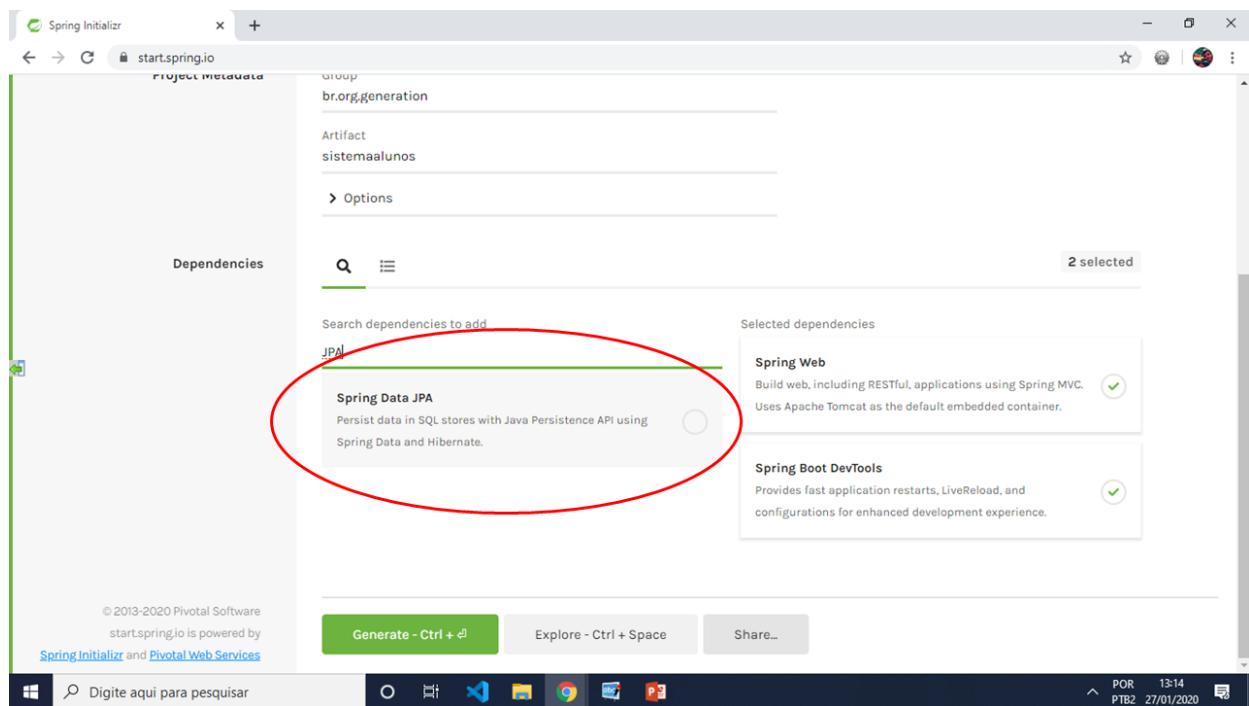
Passo 1 - Defina seu grupo e artefato

The screenshot shows the Spring Initializr web application at start.spring.io. The configuration is set for a Maven Project in Java, using Spring Boot version 2.2.4. In the 'Project Metadata' section, the Group is set to 'br.org.generation' and the Artifact is set to 'sistemaalunos'. Two blue arrows point from the text 'Não se esqueça do grupo' and 'Não se esqueça do artefato' to the respective fields. At the bottom, there are buttons for 'Generate - Ctrl + d!' and 'Explore - Ctrl + Space', along with a 'Share...' button.

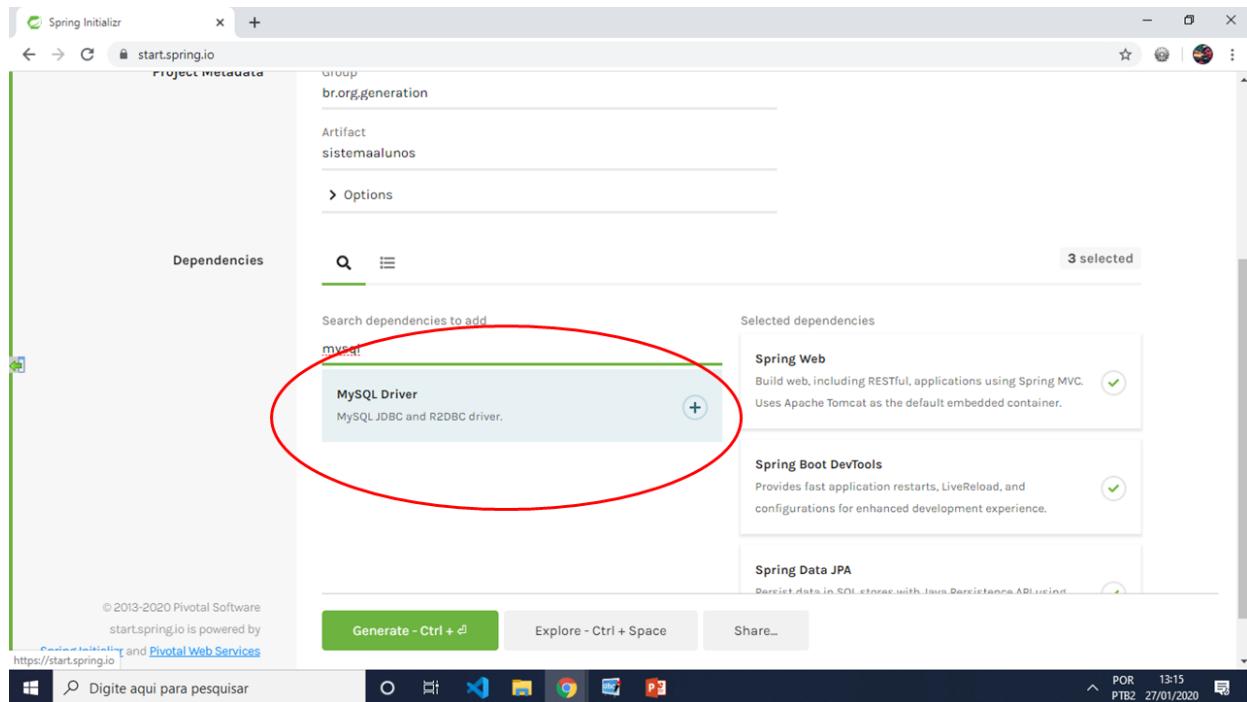
Passo 2 - Importe as bibliotecas padrão (Spring Web e DevTools)



Passo 3 - Agora vamos precisar da biblioteca do JPA

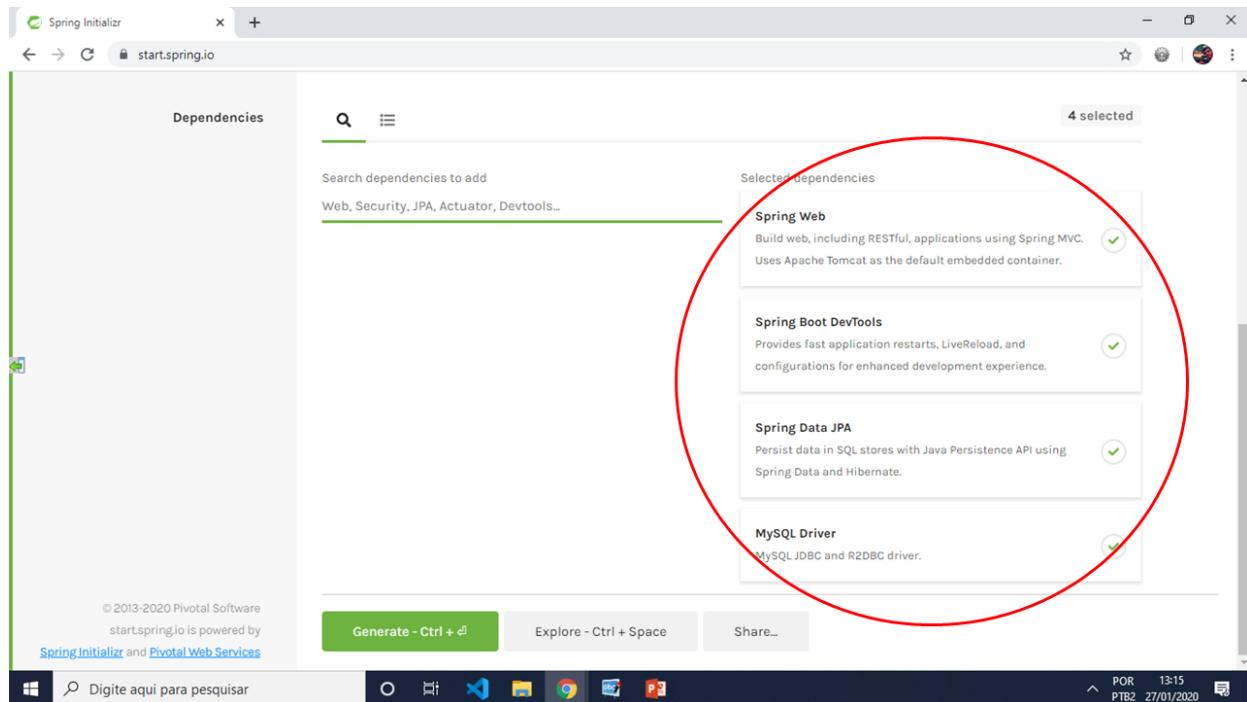


Passo 4 - E como vamos trabalhar com MySQL, vamos precisar também do Driver



Finalmente...

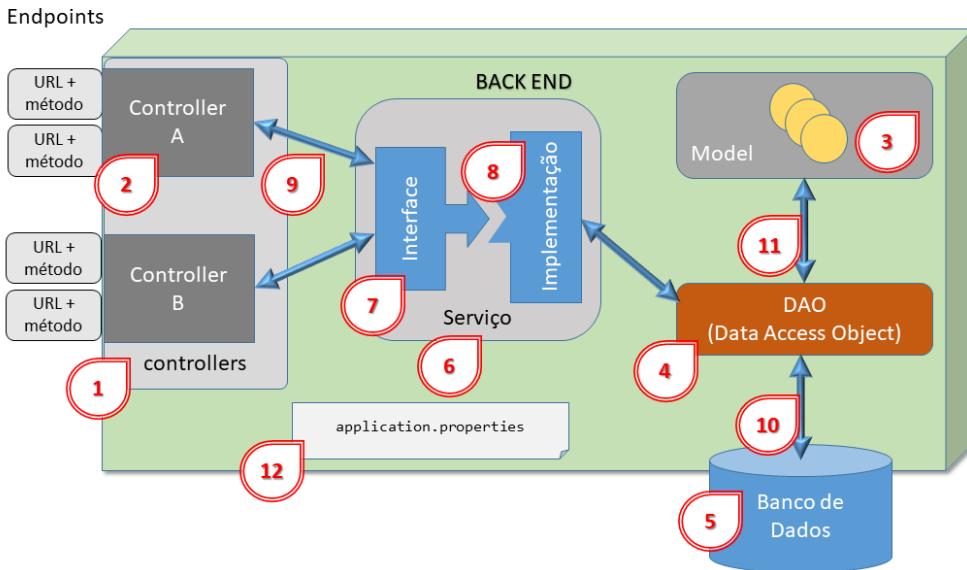
Nosso projeto tem que ficar com as seguintes dependências



Agora é importar para o Eclipse e começar a trabalhar.

Entendendo a Arquitetura do Back End

ARQUITETURA

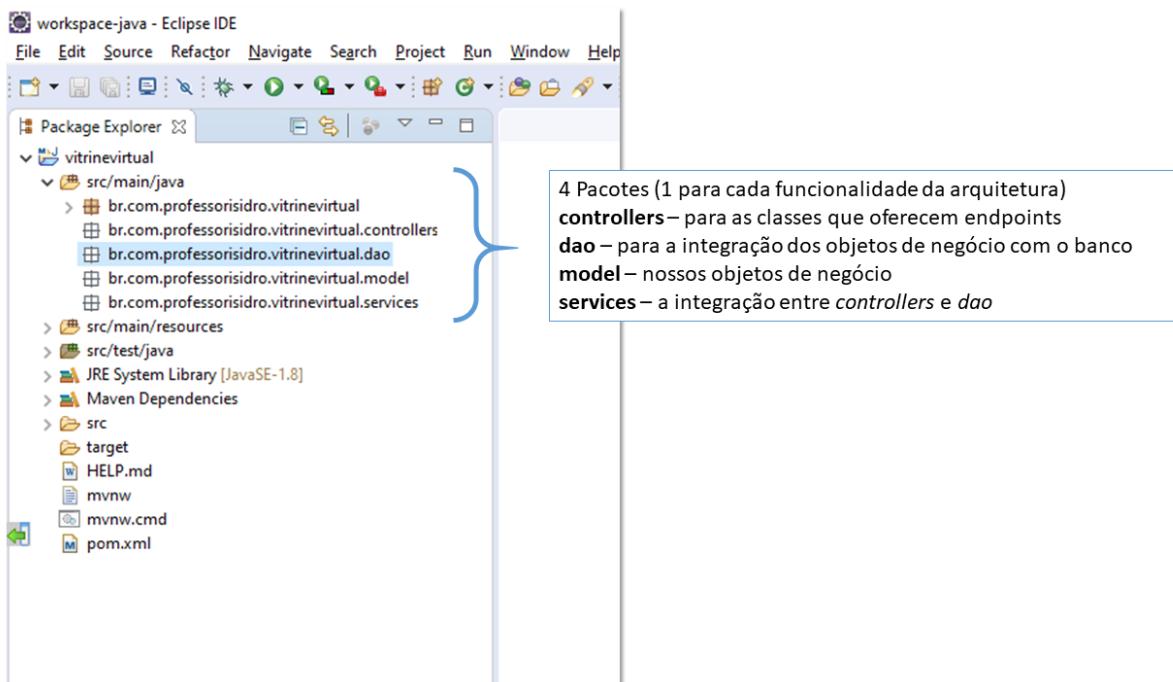


1. Precisamos de um pacote de Controllers para oferecer os endpoints para o Front
2. Cada Controller é uma classe específica anotada com `@RestController` composta de métodos que implementam os mapeamentos dos métodos (ex: `GetMapping`, `PostMapping`, `PutMapping`, `DeleteMapping`)
3. Precisamos dos nossos objetos de negócio. Eles que efetivamente armazenarão as informações sobre o que queremos do nosso sistema.
4. Para armazenar estes objetos de negócio, precisamos implementar uma camada que chamamos de DAO (*Data Access Object*) para que o JPA armazene em tabelas os valores dos atributos dos objetos
5. Obviamente que nosso banco de dados precisa estar criado. Neste caso, existem 2 alternativas para a implementação das tabelas. **Alternativa 1:** Deixar a responsabilidade para o JPA através da configuração `spring.jpa.hibernate.ddl-auto = update` (no arquivo de configuração do projeto - passo 12). **Alternativa 2:** não deixar essa responsabilidade para o JPA, porém o programador deve criar as tabelas manualmente.
6. Os Controlles não podem acessar diretamente os objetos do DAO (não é boa prática). Para isso, criamos um novo pacote com classes que implementam essa “conversa”. A esse pacote chamamos de **servico**.
7. Todo serviço também tem um modo de ser declarado. Sempre declaramos a interface do serviço com os possíveis métodos que farão essa conversão (ex: invocar a gravação de um objeto no banco, recuperar um único objeto, recuperar vários objetos, etc).
8. Não basta apenas a interface, precisamos da lógica destes métodos. Para isso, criamos objetos que implementam estas interfaces e anotamo-os com `@Component` para indicar que serão injetáveis.

9. A chamada dos serviços é declarada nos controllers através da referência à interface do serviço (anotada com `@Autowired`). O SpringBoot vai sozinho encontrar o objeto que implementa isso. Como ele faz? Buscando os objetos que estão anotados com `@Component`.
10. A comunicação entre DAO e Banco de Dados ocorre através da criação de comandos SQL, porém isso é transparente para o programador (significa que não precisamos escrever nenhuma instrução SQL).
11. Tanto na inserção quanto na recuperação, quem preenche ou consulta os valores dos atributos dos objetos é o próprio JPA.
12. Tudo só é possível, porque existe um arquivo chamado `application.properties` que configura nosso projeto (desde a porta que o BackEnd vai atender, até os parâmetros de conexão com o banco de dados).

Criando nosso projeto JPA Spring Boot

Uma vez importado nosso projeto para o Eclipse, teremos:



Requisitos do nosso sistema:

Nossa Vitrine Virtual terá Produtos e Departamentos.

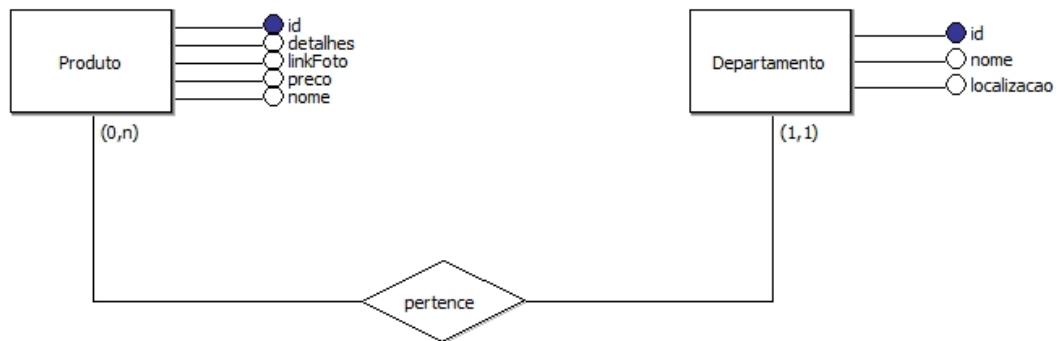
Produto

```
id: inteiro  
nome: string  
detalhes: string  
preco: float  
linkFoto: string
```

Departamento

```
id: inteiro  
nome: string  
localizacao: string
```

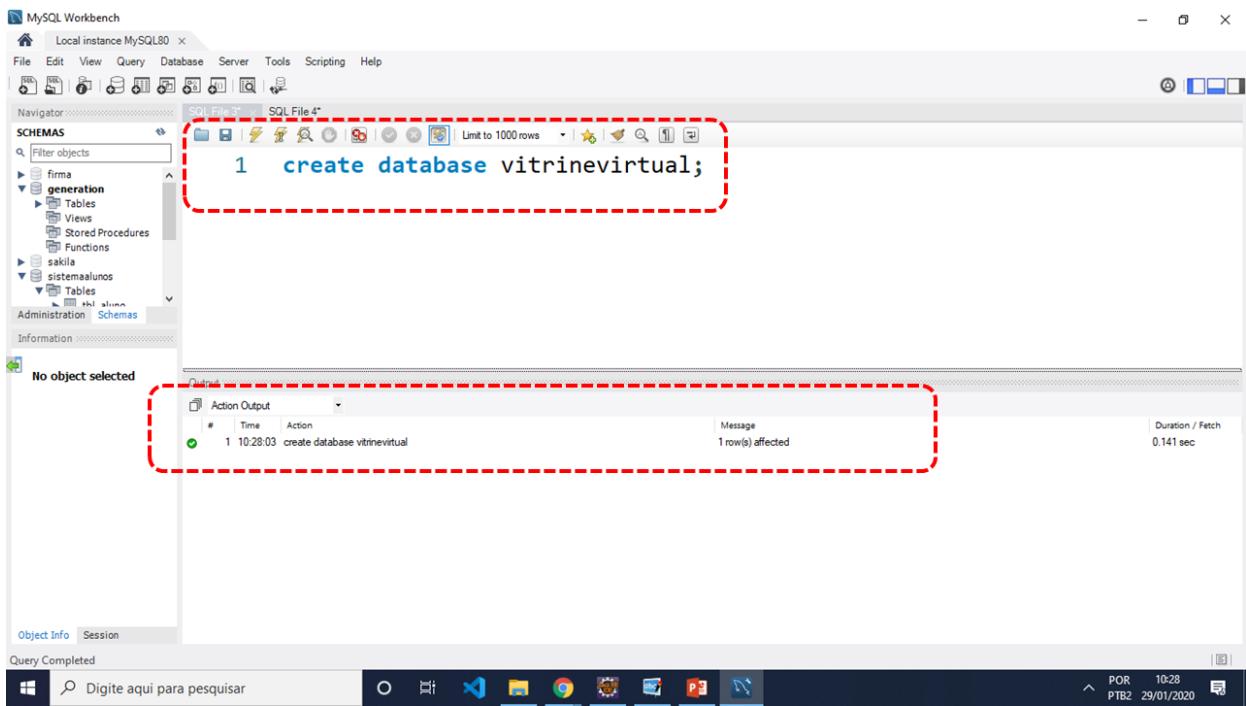
A relação entre eles segue o diagrama



Configurar parâmetros de conexão do nosso projeto com o banco de dados

Pre-requisito: Criar uma base de dados do nosso sistema.

Nome da base: `vitrinevirtual`



sistemaalunos`

Precisamos editar nosso arquivo de configuração do projeto. O arquivo `application.properties` com o seguinte conteúdo **obrigatório** a cada linha.

```
spring.datasource.url = jdbc:mysql://localhost:3306/vitrinevirtualsistemaal
spring.datasource.username = root
spring.datasource.password = admin
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Diale
```

- A primeira linha define a URL do banco que vamos utilizar. Neste caso, utilizamos o protocolo `jdbc:mysql` na máquina local `localhost`, na porta padrão do serviço `mysql` (`3306`) com a base que criamos (`sistemaalunos`)
- Na segunda linha, temos o nome do usuário do banco (`root`).
- Na terceira linha, temos a senha deste usuário (neste caso `admin`)
- Na quarta linha, definimos o dialeto que o nosso sistema irá conversar com o banco. Por ser um banco MySQL, precisamos utilizar o dialeto correspondente. Dialetos são variações sintáticas do padrão ANSI SQL implementados em cada banco de dados.

Opcionalmente podemos incluir mais 3 linhas no arquivo `application.properties`, que são:

```
spring.jpa.show_sql = true
spring.datasource.testWhileIdle = true
spring.datasource.validationQuery = SELECT 1
```

- na primeira linha, temos um tipo de “DEBUG”, onde o SpringBoot mostra na tela de console todas as instruções SQL que são geradas durante as chamadas ao banco.
- a 2a linha indica que o springboot vai ficar monitorando se a conexão permanece ativa e para isso vai precisar de um comando simples (visto na próxima linha)
- finalmente a 3a linha indica qual será o comando que o springboot (JPA) vai executar para verificar se a conexão com o banco está ativa ou não

Além disso há uma linha que, se for decidido atribuir a criação das tabelas por responsabilidade do JPA, deve ser incluída da seguinte forma:

```
spring.jpa.hibernate.ddl-auto = update
```

Criando nossos objetos Model

A idéia é criarmos nossos objetos de forma simples e depois relacioná-los. Vamos entender como funciona.

Inicialmente vamos criar a classe **Departamento**. Preste muita atenção nas importações (tem que ser `javax.persistence`).

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity // Departamento é uma entidade
@Table(name="tbldepartamento") // cujo nome da tabela foi definido
public class Departamento {

    @Id // id é chave primária
    @GeneratedValue(strategy=GenerationType.IDENTITY) // gerado pelo banco
    @Column(name="id") // nome da coluna
    private int id;

    @Column(name="nome", length=100) // aqui definimos nome e tamanho da c
    private String nome;

    @Column(name="localizacao", length=30)
    private String localizacao;

    /* aqui seguem getters e setters */
}
```

Agora vamos criar e anotar a classe **Produto**. Claro que não vamos comentar, pois seguem as mesmas anotações.

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="tblproduto")
public class Produto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="nome", length = 100)
    private String nome;

    @Column(name="detalhes", length = 500)
    private String detalhes;

    @Column(name="preco")
    private float preco;

    @Column(name="linkfoto", length = 200)
    private String linkfoto;

    /* aqui seguem getters e setters */
}
```

E as relações?

De acordo com nosso modelo ER, a Relação entre Departamento e Produto é 1:N. Dessa forma, vamos precisar complementar as nossas classes.

Complementando a classe **Produto**.

```
...
public class Produto{
    ...
    @ManyToOne
```

```
    private Departamento depto;  
    ...  
}
```

Neste caso, a anotação `@ManyToOne` indica que muitos produtos podem estar vinculados a um único departamento, o que obedece a nossa modelagem

Agora vamos complementar a nossa class **Departamento**

```
...  
public class Departamento {  
    ...  
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "depto")  
    private List<Produto> produtos;  
    ...  
}
```

Neste caso, a classe Departamento tem como atributo uma Lista de Produtos (pensando de forma intuitiva, toda vez que recuperamos um departamento, é desejável saber sua lista de produtos). Por isso anotamos com `OneToMany`.

Entretanto a anotação tem 2 parâmetros:

- `cascade = CascadeType.ALL` que significa que qualquer alteração na chave primária de Departamento irá afetar os registros da tabela Produto.
 - **Situação 1:** Apaguei 1 registro do tipo departamento. Todos os demais produtos vinculados a este departamento serão também removidos.
 - **Situação 2:** Alterei a chave primária de um departamento. As chaves estrangeiras armazenadas na tabela produto também sofrerão a atualização, para manter a consistência dos dados.
- `mappedBy="depto"` indica o nome do atributo do tipo Departamento que foi declarado na classe Produto. Isso é o equivalente ao `references` do SQL no momento da criação da tabela e será utilizado para os INNER JOIN.

Se tudo isso estiver correto, as tabelas com as respectivas relações serão criadas no nosso banco de dados. Isso pode ser verificado através das imagens abaixo.

Consultando as tabelas criadas

The screenshot shows the MySQL Workbench interface with the 'Schemas' tree on the left containing 'firma', 'generation', 'sakila', and 'sistemaalunos'. A red dashed box highlights the 'Tables' section under 'sistemaalunos', which lists 'Tables_in_vitrinevirtual', 'tbldepartamento', and 'tblproduto'. The 'Result Grid' pane displays the output of the 'show tables' command:

Tables_in_vitrinevirtual
tbldepartamento
tblproduto

The 'Action Output' pane shows the history of database operations:

#	Time	Action	Message	Duration / Fetch
1	10:28:03	create database vitrinevirtual	1 row(s) affected	0.141 sec
2	11:32:21	use vitrinevirtual	0 row(s) affected	0.016 sec
3	11:33:18	show tables	2 row(s) returned	0.000 sec / 0.000 sec

Verificando a estrutura da tabela Departamento (observe a chave primária com auto-incremento)

The screenshot shows the MySQL Workbench interface with the 'Schemas' tree on the left containing 'firma', 'generation', 'sakila', and 'sistemaalunos'. A red dashed box highlights the 'Tables' section under 'sistemaalunos', which lists 'Tables_in_vitrinevirtual', 'tbldepartamento', and 'tblproduto'. The 'Result Grid' pane displays the output of the 'desc tbldepartamento' command:

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI		auto_increment
localizacao	varchar(30)	YES			
nome	varchar(100)	YES			

The 'Action Output' pane shows the history of database operations:

#	Time	Action	Message	Duration / Fetch
1	10:28:03	create database vitrinevirtual	1 row(s) affected	0.141 sec
2	11:32:21	use vitrinevirtual	0 row(s) affected	0.016 sec
3	11:33:18	show tables	2 row(s) returned	0.000 sec / 0.000 sec
4	11:33:39	desc departamento	Error Code: 1146. Table 'vitrinevirtual.departamento' doesn't exist	0.094 sec
5	11:33:45	desc tbldepartamento	3 row(s) returned	0.031 sec / 0.000 sec

Verificando a estrutura da tabela Produto (observe a chave primária também com auto-incremento e a chave estrangeira que é o id do departamento).

MySQL Workbench Schemas: Local instance MySQL80

SQL File 3* SQL File 4*

```
1 desc tblproduto;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI		auto_increment
detalhes	varchar(500)	YES			
linkfoto	varchar(200)	YES			
nome	varchar(100)	YES			
preco	float	YES			
depto_id	int(11)	YES	MUL		

Output:

#	Time	Action	Message	Duration / Fetch
2	11:32:21	use vitrinevirtual	0 row(s) affected	0.016 sec
3	11:33:18	show tables	2 row(s) returned	0.000 sec / 0.000 sec
4	11:33:39	desc departamento	Error Code: 1146. Table 'vitrinevirtual.departamento' doesn't exist	0.094 sec
5	11:33:45	desc departamento	3 rows(e)s returned	0.031 sec / 0.000 sec

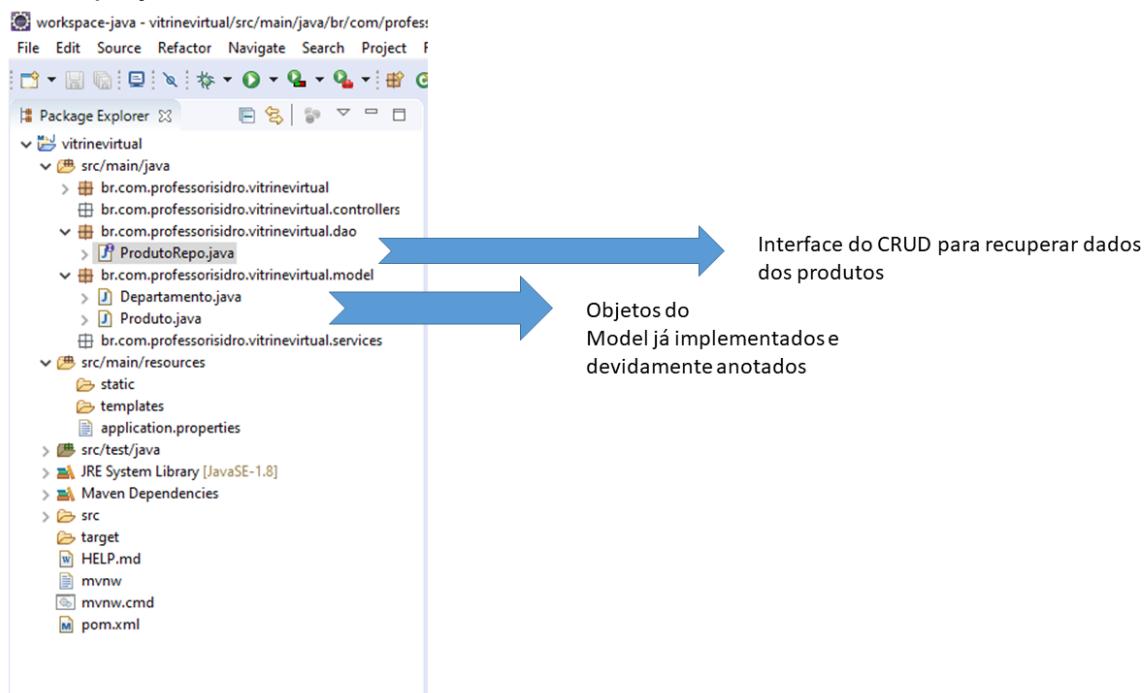
Object Info Session Query Completed

Criando nosso DAO

Basicamente vamos precisar, para este exemplo, de apenas 1 serviço DAO para recuperar nossos produtos.

Vamos criar uma interface `ProdutoRepo` que herda da interface padrão `CrudRepository`

Nosso projeto vai ficar com essa estrutura:



A interface `ProdutoRepo` fica simples assim:

```
import org.springframework.data.repository.CrudRepository;
import br.com.professorisidro.vitrinevirtual.model.Produto;

public interface ProdutoRepo extends CrudRepository<Produto, Integer> {
    // aqui podem vir outros métodos
}
```

A interface herda do `CrudRepository<T, I>` com 2 parâmetros relativos a tipos de dados.

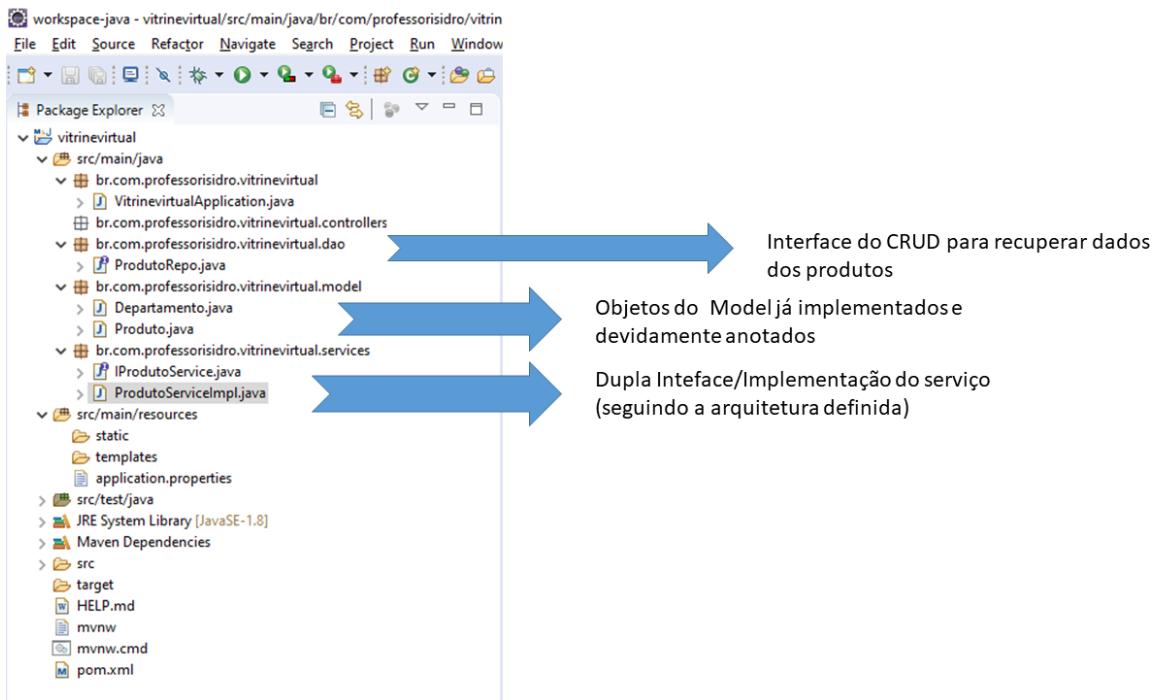
- T corresponde ao tipo de dado que será armazenado/recuperado do banco. No nosso caso, a classe `Produto`
- I corresponde ao tipo de dado que identifica a chave primária do nosso objeto. Como o atributo `id` de `Produto` é `int`, usamos seu *wrapper* `Integer`.

`CrudRepository` é uma interface nativa do JPA que já implementa alguns métodos para manipulação dos objetos no banco de dados, tais como:

- `save(T)` - armazena/atualiza o objeto no banco
- `deleteById(Id)` - remove o registro da tabela
- `findAll()` - recupera todos os registros
- `findById(Id)` - recupera um objeto pela sua chave primária
- `count()` - retorna o número de registros na tabela

Criar o serviço que “conversa” com o DAO (Repository)

Vamos sempre manter a estrutura da nossa arquitetura. Sempre um serviço tem que ser definido como interface e implementação (o SpringBoot exige isso). Para tanto, vamos criar 2 arquivos: Interface `IProdutoService` e classe `ProdutoServiceImpl`. Dessa forma, nosso projeto fica assim:



A interface terá apenas 2 métodos: um para buscar todos os produtos e outro para buscar os detalhes do produto.

IProdutoService

```
import java.util.List;
import br.com.professorisidro.vitrinevirtual.model.Produto;

public interface IProdutoService {
    public List<Produto> recuperarTodos();
    public Produto recuperarPorId(int id);
}
```

Agora, na implementação, temos:

```
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import br.com.professorisidro.vitrinevirtual.dao.ProdutoRepo;
import br.com.professorisidro.vitrinevirtual.model.Produto;

@Component
public class ProdutoServiceImpl implements IProdutoService {

    @Autowired // declaramos a interface do DAO que criamos
    private ProdutoRepo repo; // e a anotação AutoWired busca uma implementação

    @Override
    public List<Produto> recuperarTodos() {
```

```

        // isso corresponde a um
        //      select * from tblproduto
        //      inner join tbldepartamento
        //      on tblproduto.depto_id = departamento.id
        return (List<Produto>)repo.findAll();
    }

    @Override
    public Produto recuperarPorId(int id) {
        return repo.findById(id).get();
    }
}

```

Algumas considerações sobre esta classe:

- A anotação `@Component` no cabeçalho da classe indica explicitamente ao `SpringBoot` que esta classe é uma implementação injetável de um objeto (faremos uso dele nos controllers)
- Nossa classe usa um objeto do tipo `ProdutoRepo` que é uma interface. Onde está essa implementação? Na verdade, delegamos a responsabilidade de econtrar a implementação ao `SpringBoot`. Como fazemos isso? Através da anotação `@Autowired`. Dessa forma, a declaração do repositório fica como descrito.

```

@Autowired
private ProdutoRepo repo;

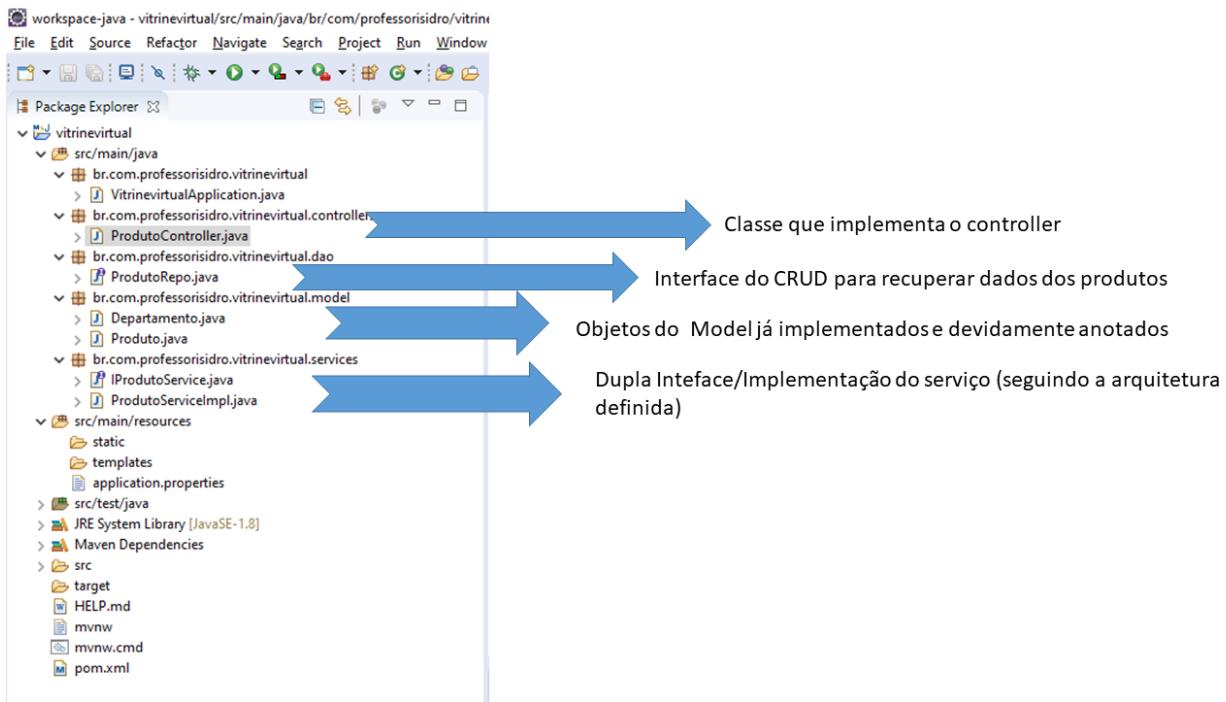
```

Dentro dos métodos temos o uso das funcionalidades que já vem com o `CrudRepository` (`findAll` e `findById`). No caso do método `recuperarTodos`, fazemos apenas uma conversão (*casting*) do resultado do `findAll` para uma lista de produtos (`List<Produto>`)

Finalmente criando o Controller

Vamos ter apenas 1 único controller que pode, tanto recuperar todos os produtos, quanto recuperar detalhes de 1 único produto.

Teremos a classe `ProdutoController` e nosso projeto fica com a seguinte estrutura.



Nosso controller será basicamente a invocação dos métodos do serviço e retornando de acordo com os códigos de status do protocolo HTTP

```

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import br.com.professorisidro.vitrinevirtual.model.Produto;
import br.com.professorisidro.vitrinevirtual.services.IProdutoService;
@RestController
@CrossOrigin("*")
public class ProdutoController {

    @Autowired
    private IProdutoService servico; // declaramos sempre a interface

    @GetMapping("/produto")
    public ResponseEntity<List<Produto>> mostrarTodos(){
        return ResponseEntity.ok(servico.recuperarTodos());
    }

    @GetMapping("/produto/{id}")
    public ResponseEntity<Produto> mostrarPeloId(@PathVariable int id){
        Produto p = servico.recuperarPorId(id);
        if (p != null) {
            return ResponseEntity.ok(p);
        }
    }
}

```

```

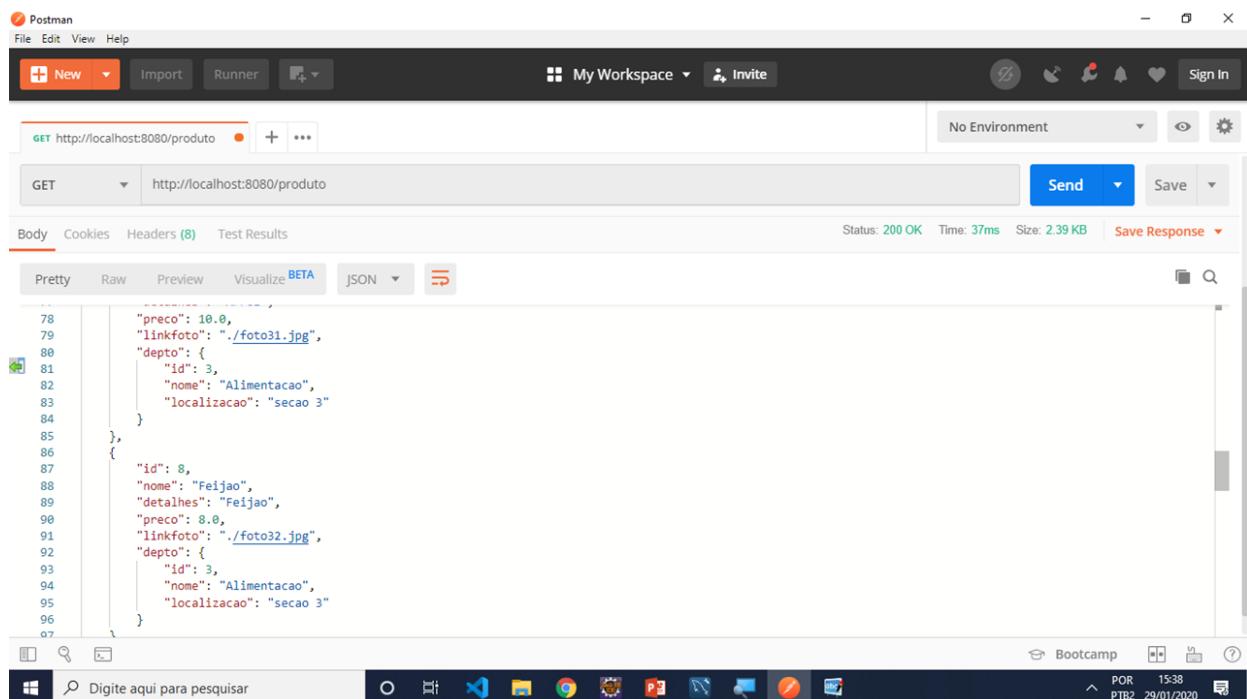
        return ResponseEntity.notFound().build();
    }
}

```

Apenas reforçando que usamos os retornos de Status do protocolo HTTP para retornar. Isso é explícito no método `mostrarPorId` que recebe um código que pode não existir no banco. Por conta disso, temos que considerar um produto inexistente e retornar o erro 404.

Testando

Basta ir ao Postman e digitar `http://localhost:8080/produto` no método GET



The screenshot shows the Postman interface with a successful GET request to `http://localhost:8080/produto`. The response body is displayed in JSON format, showing two products:

```

[{"id": 78, "name": "Pao de Queijo", "price": 5.0, "department": {"id": 1, "name": "Alimentacao", "localizacao": "secao 1"}, "details": "Pao de queijo com recheio de queijo ralado."}, {"id": 79, "name": "Bacon", "price": 8.0, "department": {"id": 2, "name": "Alimentacao", "localizacao": "secao 2"}, "details": "Bacon fumado com sal e especiarias."}]

```

Detalhe importante.

Em função da relação de Produto com Departamento, podemos correr um risco alto de uma referência cíclica infinita (produto tem departamento que tem lista de produtos, que tem departamentos, e assim por diante).

Para evitar esse tipo de loop infinito, temos que colocar 2 anotações (uma em cada classe do pacote model). Usaremos a anotação `@JsonIgnoreProperties` da seguinte forma:

Classe Produto

```
@ManyToOne  
@JsonIgnoreProperties("produtos")  
private Departamento depto;
```

Isso impede, toda vez que o JPA encontrar um produto de buscar a lista de produtos que estão relacionada ao departamento que ele pertence.

Classe Departamento

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "depto")  
@JsonIgnoreProperties("depto")  
private List<Produto> produtos;
```

Isso impede que, ao recuperar um departamento, o JPA não busque quais são os departamentos que estão na lista de produtos.

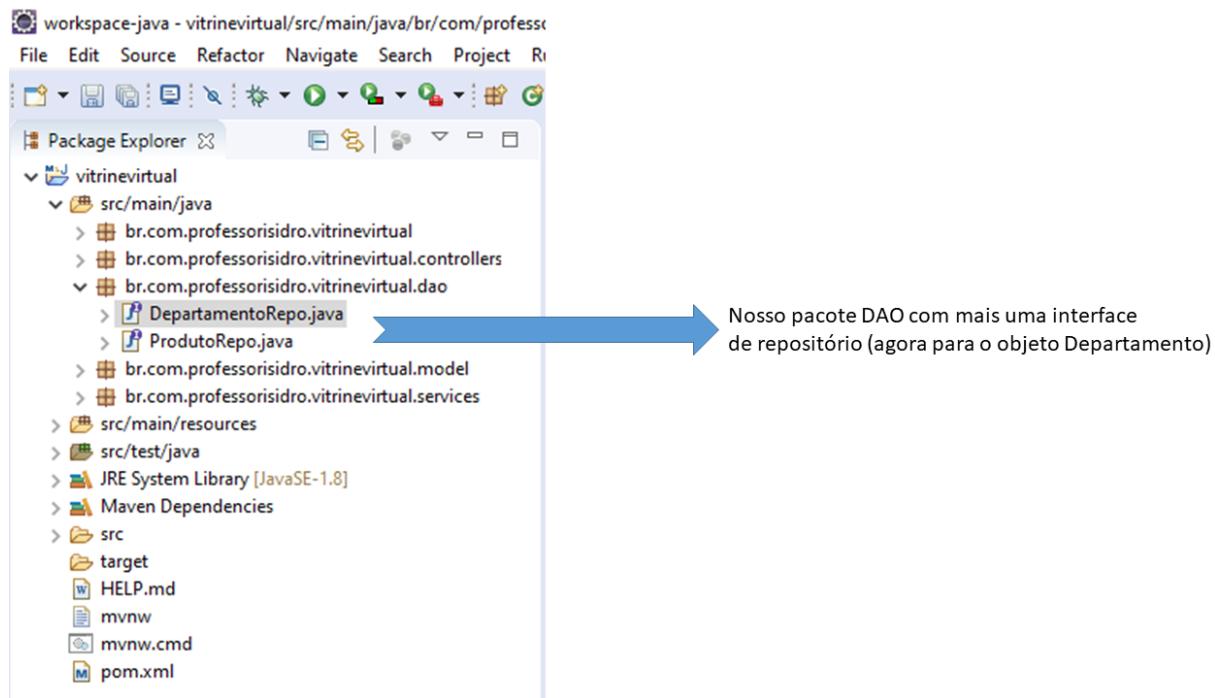
Inserindo dados

Vamos criar mais 2 funcionalidades no nosso sistema de vitrine virtual. Vamos gerar as funcionalidades de inserir novos departamentos e também novos produtos.

Considerando que já temos objetos, interfaces e serviços que manipulam os produtos, precisamos, agora, criar mais algumas funcionalidades e complementar outras.

Vamos nos concentrar em funcionalidades referentes ao Departamento

Precisamos, agora, criar nossa interface para as operações CRUD do departamento. Criamos esta interface no pacote `dao`.



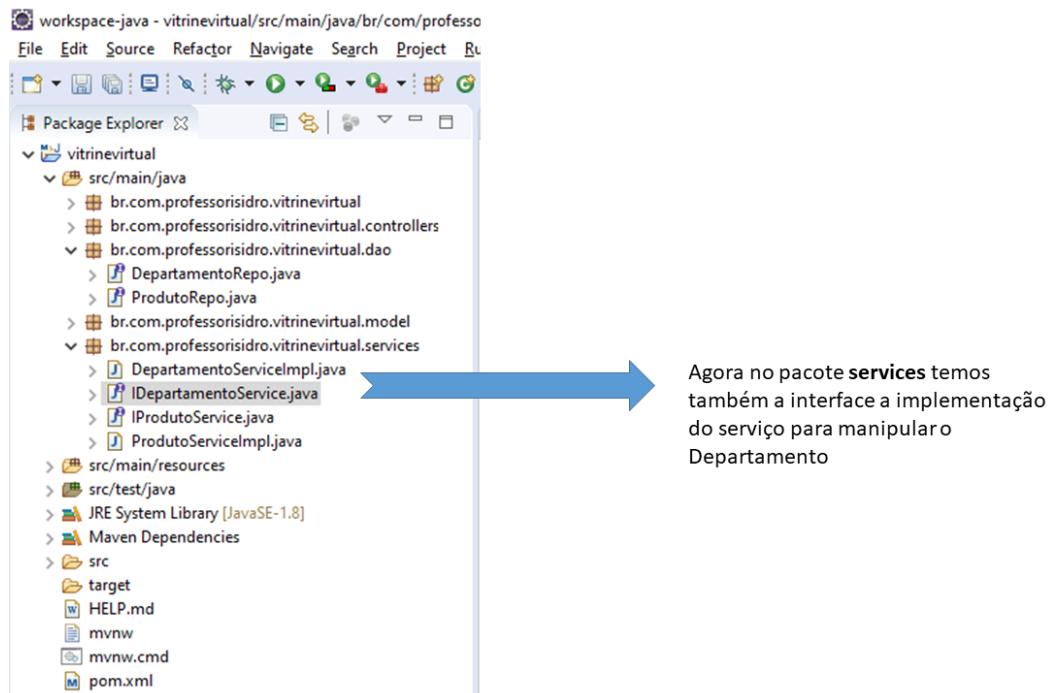
O conteúdo da interface `DepartamentoRepo` segue o mesmo padrão.

```
import org.springframework.data.repository.CrudRepository;
import br.com.professorisidro.vitrinevirtual.model.Departamento;

public interface DepartamentoRepo extends CrudRepository<Departamento, Inte
}
```

O próximo passo é justamente criar a dupla interface/implementação do serviço que irá manipular o departamento. Quais funcionalidades podemos colocar neste serviço?

Vamos começar com uma que lista todos os departamentos, pega detalhes de um departamento específico e inclui novo departamento.



Temos a descrição da interface IDepartamentoService a seguir:

```
import java.util.List;
import br.com.professorisidro.vitrinevirtual.model.Departamento;

public interface IDepartamentoService {

    public void adicionarNovoDepartamento(Departamento depto);
    public Departamento recuperarDetalhes(int id);
    public List<Departamento> recuperarTodos();
}
```

A implementação da classe segue o mesmo padrão também:

- A classe precisa ser anotada como @Component
- Injeta uma dependência do DepartamentoRepo através da anotação @Autowired

```
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import br.com.professorisidro.vitrinevirtual.dao.DepartamentoRepo;
import br.com.professorisidro.vitrinevirtual.model.Departamento;

@Component
public class DepartamentoServiceImpl implements IDepartamentoService{
    @Autowired
    private DepartamentoRepo repo;

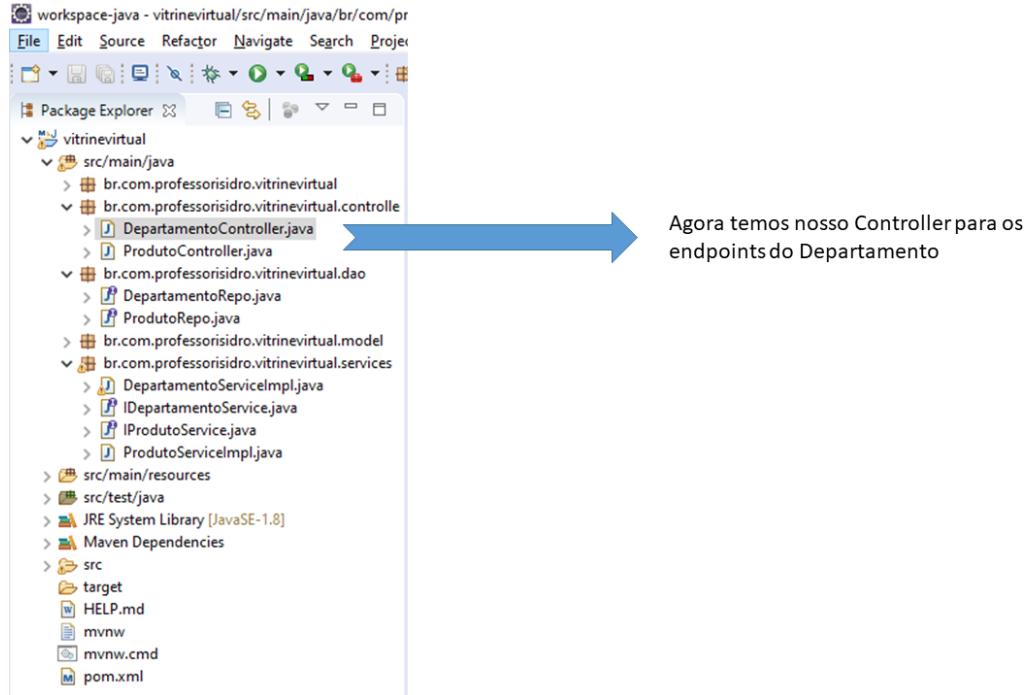
    @Override
    public void adicionarNovoDepartamento(Departamento depto) {
        repo.save(depto);
    }

    @Override
    public Departamento recuperarDetalhes(int id) {
        return repo.findById(id).get();
    }

    @Override
    public List<Departamento> recuperarTodos() {
        return (List<Departamento>)repo.findAll();
    }
}
```

Finalmente precisamos criar nosso Controller, que chamaremos de `DepartamentoController`, com os seguintes endpoints

- `/departamento/novo` (POST)
- `/departamenot/todos` (GET)
- `/departamento/{id}` (GET)



A seguir a implementação (lembrando novamente que o controller é declarado com a anotação `@RestController`, permite acesso através do `@CrossOrigin("*")` e injeta uma dependência do serviço através do `@Autowired`.

```
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import br.com.professorisidro.vitrinevirtual.model.Departamento;
import br.com.professorisidro.vitrinevirtual.services.IDepartamentoService;

@RestController
@CrossOrigin("*")
public class DepartamentoController {

    @Autowired
    private IDepartamentoService servico;
```

```

    @PostMapping("/departamento/novo")
    public ResponseEntity<Departamento> incluirNovo(@RequestBody Departamen
        servico.adicionarNovoDepartamento(depto);
        return ResponseEntity.ok(depto);
    }

    @GetMapping("/departamento/todos")
    public ResponseEntity<List<Departamento>> listarTodos(){
        return ResponseEntity.ok(servico.recuperarTodos());
    }

    @GetMapping("/departamento/{id}")
    public ResponseEntity<Departamento> listarDetalhes(@PathVariable int id
        Departamento depto = servico.recuperarDetalhes(id);
        if (depto != null) {
            return ResponseEntity.ok(depto);
        }
        return ResponseEntity.notFound().build();
    }
}

```

Vamos testar!

Teste 1: No Postman, vamos gerar uma requisição GET para recuperar todos os Departamentos, através da URL `http://localhost:8080/departamento/todos`

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/departamento/todos`. The response body is a JSON array of department objects:

```

1  [
2   {
3     "id": 1,
4     "nome": "Cama/Mesa/Banho",
5     "localizacao": "secao 1",
6     "produtos": [
7       {
8         "id": 1,
9         "nome": "Almofada",
10        "detalhes": "Almofada",
11        "preco": 10.0,
12        "linkfoto": "./foto11.jpg"
13      },
14      {
15        "id": 2,
16        "nome": "Colcha",
17        "detalhes": "Colcha",
18        "preco": 20.0,
19        "linkfoto": "./foto12.jpg"
20      },
21      {
22        "id": 3,
23        "nome": "Mesa de Jantar",
24        "detalhes": "Mesa de Jantar com 4 cadeiras",
25        "preco": 300.0,
26        "linkfoto": "./foto13.jpg"
27      }
28    ]
29  ]

```

Teste 2: No mesmo Postman vamos gerar uma requisição POST para a URL <http://localhost:8080/departamento/novo> enviando no corpo da mensagem um JSON com os dados do novo departamento (exceto o `id`). Exemplo do que seria um Request Body para inserir novo departamento:

```
{  
    "nome" : "Novo departamento",  
    "localizacao" : "Nova Localizacao"  
}
```

The screenshot shows the Postman application interface. At the top, there are tabs for 'My Workspace' and 'Invite'. Below the tabs, there are two requests listed: a 'GET' request to 'http://localhost:8080/departamento...' and a 'POST' request to 'http://localhost:8080/departamento/novo'. The 'POST' request is selected. In the 'Body' tab, the JSON payload is defined:

```
1+ {  
2   "nome": "Eletronicos",  
3   "localizacao": "secao 5"  
4 }
```

Below the body, the response is shown in 'Pretty' format:

```
1 {  
2   "id": 6,  
3   "nome": "Eletronicos",  
4   "localizacao": "secao 5",  
5   "produtos": null  
6 }
```

The status bar at the bottom indicates 'Status: 200 OK'.

Inserindo agora um Novo Produto.

Não são muitas alterações, basicamente precisamos de:

- Mais um método no `ProdutoService` (e obviamente na interface) para cadastrar Novo produto
- Mais um método no Controller do Produto (para oferecer uma URL via método POST)

Interface IProdutoService

```
public interface IProdutoService {  
    public List<Produto> recuperarTodos();  
    public Produto recuperarPorId(int id);
```

```
    public void novoProduto(Produto produto); // novo metodo
}
```

Classe ProdutoServiceImpl

```
...
public class ProdutoServiceImpl implements IProdutoService {
...
    @Override
    public void novoProduto(Produto produto) {
        repo.save(produto);
    }
...
}
```

E, finalmente, o ProdutoController.

```
...
public class ProdutoController {
...

    @PostMapping("/produto/novo")
    public ResponseEntity<Produto> inserirProduto(@RequestBody Produto prod
        servico.novoProduto(produto);
        return ResponseEntity.ok(produto);
    }
...
}
```

Para testar, precisamos ter um **cuidado** ao montar o objeto. Vamos usar o Postman como ferramenta, indicando o método e a URL, porém precisamos montar o objeto da maneira correta:

```
{
    "nome": "Radio Relogio",
    "detalhes": "Radio Relogio que toca musica e mostra hora",
    "preco": 49.90,
    "linkfoto": "./foto50.jpg",
    "depto": {
        "id": 6
    }
}
```

Notem que mesmo para informar apenas o **id** do departamento (que é a chave estrangeira na tabela **produto**), precisamos estruturá-lo no padrão JSON seguindo à risca a estrutura definida em nosso model. Por isso que criamos um JSON com os atributos **nome**, **detalhes**, **preco**, **linkfoto** e um JSON interno para o atributo **depto**, cujo conteúdo é apenas o **id** deste departamento (claro que tem que ser um id válido).

The screenshot shows the Postman interface with a POST request to `http://localhost:8080/produto/novo`. The JSON body is:

```

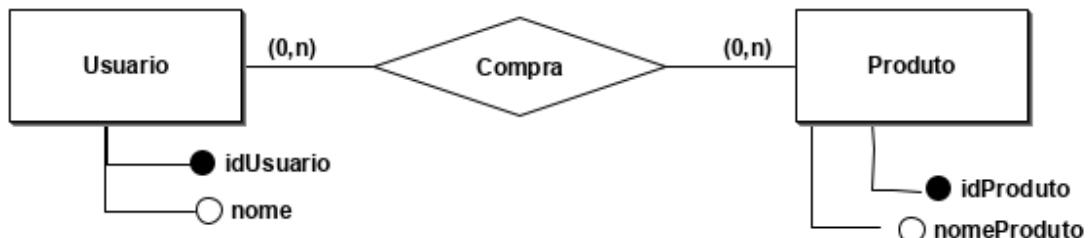
1  {
2     "nome": "Radio Relogio",
3     "detalhes": "Radio Relogio que toca musica e mostra hora",
4     "preco": 49.90,
5     "linkfoto": "./foto50.jpg",
6
7     "depto": {
8         "id": 6
9     }
10 }

```

The response status is 200 OK, time 82ms, size 429 B.

E Relações Muitos-Para-Muitos?

Vamos melhorar nosso modelo? E se nossos produtos tivessem Usuários/Cientes interessados?



Neste caso, vamos explorar este exemplo de uma forma bem simples. A interpretação será: Um usuário pode comprar vários produtos e um produto pode ser comprado por

vários usuários. Neste caso não teremos o usuário comprando várias vezes o mesmo produto.

Como ficaríamos em relação às classes? Criaremos nossa classe `Usuario` da seguinte maneira:

```
@Entity
@Table(name="tblusuario")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="idusuario")
    private int id;

    @Column(name="nome", length = 100)
    private String nome;

    @Column(name="email", length = 100)
    private String email;

    @Column(name="senha", length = 50)
    private String senha;

    @ManyToMany
    @JoinTable(name="tblcompra",
        joinColumns = @JoinColumn(name="idusuario"),
        inverseJoinColumns = @JoinColumn(name="idproduto")
    )
    @JsonIgnoreProperties("compradores")
    private List<Produto> compras;

    /* aqu seguem getters e setters */
}
```

Neste caso, a definição segue o mesmo padrão anterior, com exceção do último atributo (que chamamos de `compras`). Este atributo está anotado com `@ManyToMany`. Como sabemos que as relações *Muitos-para-Muitos* precisam de uma tabela de ligação, ela é definida através da anotação `@JoinTable`.

Na anotação `@JoinTable` temos 3 atributos:

- `name` : o nome da tabela que será criada para ligação. neste caso `tblcompra`
- `joinColumns` : as colunas da entidade atual (neste caso `Usuario`) que farão parte da tabela de ligação como chaves estrangeiras. Aqui utilizamos `joinColumns = @JoinColumn(name="idusuario")`

- `inverseJoinColumns` : as colunas da entidade relacionada (neste caso a entidade `Produto`) que fará parte da tabela de ligação como chave estrangeira. Aqui utilizamos `inverseJoinColumns = @JoinColumn(name="idproduto")`

Aqui também utilizamos a anotação `@JsonIgnoreProperties` para evitarmos *loops* nas buscas, devido às referências cíclicas.

Agora, também precisamos modificar a classe `Produto` para que ela também reconheça a relação *muitos-para-muitos*. Ela fica da seguinte forma:

```
@Entity
@Table(name="tblproduto")
public class Produto {

    ...
    @ManyToMany(mappedBy = "compras", cascade = CascadeType.ALL)
    @JsonIgnoreProperties("compras")
    private List<Usuario> compradores;
    ...
}
```

A classe `Produto` também terá um atributo (que neste caso chamamos de `compradores` para recuperar todos os usuários que realizaram compras daquele produto. Entretanto a anotação é bem mais simples. Não precisamos das tabelas de ligação. Na anotação apenas mapeamos qual o nome do atributo na classe `Usuario` que implementa a relação (neste caso o atributo `compras`, através do parâmetro `mappedBy`).

Também utilizamos a estratégia `Cascade` para que uma exclusão de um produto também exclua as compras efetuadas nele.

E, por fim, também utilizamos a anotação `@JsonIgnoreProperties` para evitar os *loops* devidos às referências cíclicas.

Se tudo estiver ok, o próprio JPA cria a tabela de ligação de forma correta, como na figura:

Field	Type	Null	Key	Default	Extra
idusuario	int(11)	NO	MUL		
idproduto	int(11)	NO	MUL		

Pronto! Agora, ao testarmos, é recuperado, para um único usuário, toda lista de produtos que ele comprou.

```

1 {
2   "id": 1,
3   "nome": "Pro Isidro",
4   "email": "isidro@isi.com",
5   "senha": "12345",
6   "compras": [
7     {
8       "id": 4,
9       "nome": "Computador",
10      "detalhes": "Computador",
11      "preco": 1000.0,
12      "linkfoto": "./foto21.jpg",
13      "depto": {
14        "id": 2,
15        "nome": "Informatica",
16        "localizacao": "secao 2"
17      }
18    },
19    {
20      "id": 5,
21    }
22  ]
23}
  
```

Algumas outras consultas bem bacanas

JPA tem um módulo chamado `QueryByMethodName` que automaticamente analisa como os métodos de consulta foram definidos e gera Strings SQL a partir desses nomes de

métodos. Os detalhes de como tudo funciona estão na documentação do JPA do Spring, cujo link está aqui [Spring JPA Query Methods](#)

Situação 1: Pense no login do seu usuário. Ele deve recuperar as informações do Usuário de acordo com Email e Senha. Como proceder neste caso?

Resposta: Simplesmente definindo um método que busque por Email e Senha na interface do Repositório do Usuário.

```
public interface UsuarioRepo extends CrudRepository<Usuario, Integer>{  
    public Usuario findByEmailAndSenha(String email, String senha);  
}
```

O próprio nome **findByEmailAndSenha** busca (por estar na interface do repositório **Usuario**), objetos que coincidam os valores dos atributos **email** e **senha** com os parâmetros informados.

Vale lembrar que os elementos que constam dos métodos devem obedecer a nomenclatura da classe

A String SQL gerada com esse método é a seguinte (isso pode ser visto na saída do console do seu projeto):

```
select usuario0_.idusuario as idusuario1_2_,  
       usuario0_.email as email2_2_,  
       usuario0_.nome as nome3_2_,  
       usuario0_.senha as senha4_2_  
  
  from tblusuario usuario0_  
  
 where usuario0_.email=? and usuario0_.senha=?
```

Situação 2: Pense em buscar produtos por palavras-chave. Como proceder? Existem alguns *predicados* que o JPA entende e que utiliza na geração das consultas. Alguns deles: `startsWith` (indicando que uma String inicia com uma determinada palavra-chave), `endsWith` (indicando que uma String termina com uma palavra-chave) e `contains`, indicando que a palavra chave pode estar em qualquer posição na String.

Exemplo:

```
public interface ProdutoRepo extends CrudRepository<Produto, Integer> {  
    public List<Produto> findAllByNomeContains(String texto);  
}
```

Neste caso, buscamos todos os produtos (`findAll`) cujo nome (`ByName`) contenha (`Contains`) o que está no parâmetro do método.

A String SQL gerada com isso é a seguinte:

```
select produto0_.id as id1_1_,  
       produto0_.depto_id as depto_id6_1_,  
       produto0_.detalhes as detalhes2_1_,  
       produto0_.linkfoto as linkfoto3_1_,  
       produto0_.nome as nome4_1_,  
       produto0_.preco as preco5_1_  
  
  from tblproduto produto0_  
  
 where produto0_.nome like ? escape ?
```