# Passo a Passo no SpringBoot com JPA

Objetivo do SpringBoot - Fornecer mecanismos (bibliotecas) para construirmos uma API conectada com banco de dados que é acessada pela Web (através de URLs)

### Fundamental: Criar nosso projeto no Spring Initializer (<https://start.spring.io>)

Precisamos informar os seguintes itens:

* projeto Maven
* Linguagem Java
* Versão do SpringBoot (Sempre a que está recomendada)
* grupo - nome do pacote-base da empresa
* artefato - nome do projeto (pacote base é a junção do grupo e artefato)
* name - nome efetivo do projeto
* Descrição - sem firula
* Packaging - JAR
* Versão do Java - compatível com a versão que temos instalado na máquina (lembrando que o SpringBoot a partir da versão 3.0 exige no mínimo Java 17)
* Dependências
  + Spring Web - para efetivamente nos dar a infraestrutura Web para conseguirmos atender solicitações via URL
  + DevTools - para a cada alteração de arquivos ele reiniciar a API (poupando trabalho nosso)
  + JPA - para fazer a integração com o BAnco
  + MySQL Driver - para conectar com MySQL

### Importar para o Eclipse

* descompactar o arquivo ZIP gerado no workspace
* no Eclipse: File → Import → Maven → Existing Maven Project → Browse (Seleciona a pasta) → Finish
* Aguarde o download dos arquivos de dependências

### 

### 

### Configurar a conexão com o Banco (lembrando: o arquivo NÃO DEVE ter acentos)

#### No arquivo **application.properties (src/java/resources)**, vamos inserir 4 parâmetros obrigatórios e 1 opcional

#1 - obrigatorio - usuario do banco

**spring.datasource.username = root**

#2 - obrigatorio - senha do banco

**spring.datasource.password = mysql**

#3 - url do banco

**spring.datasource.url = jdbc:mysql://localhost:3306/nomedabase**

#4 - dialeto do banco

**spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect**

#5 - opcional (gerar log dos comandos SQL)

**spring.jpa.show\_sql = true**

### Agora vamos configurar nosso projeto

precisamos criar, a partir do pacote base, outros 4 pacotes (pelo menos)

.**model** = para armazenar as classes que serão mapeadas para as tabelas do banco

**.repo** = para armazenar as interfaces que farão o acesso ao banco

**.controller** = para armazenar as classes que irão atender às URLs

**.services** = para armazenar classes e interfaces de serviços (usados pelo Controller)

#### 

#### Como fica uma classe MODEL?

* Basicamente uma classe JAVA composta de atributos, getters e setters com as seguintes annotations (@)

| **Anotação** | **Uso** |
| --- | --- |
| @Entity | Indica na **classe** que ela será mapeada para o banco |
| @Table(name = "nome da tabela") | Para mapear a **classe** ao nome da tabela |
| @Id | Para indicar um atributo que é chave-primária |
| @GeneratedValue(strategy = GenerationType.IDENTITY) | Para indicar que o atributo é um valor auto\_incrementável cuja decisão de qual estratégia usar é do banco de dados |
| @Column(name = "xx", length = xxx ..) | Define o mapeamento do atributo para a coluna da tabela. Quanto mais informação além do nome da coluna da tabela, mais precisa a definição (tamanho da coluna, se é nulo ou não, se é única ou não, etc) |
| @ManyToOne | Para indicar objetos que são fruto de relacionamento entre tabelas (neste caso para relações 1:N) |
| @JoinColumn(name = "nome da coluna") | Para especificar com precisão o nome da coluna que representa a chave estrangeira (neste caso sempre vem junto com a anotação @ManyToOne) |

Uma vez que temos nosso Model já definido, podemos pensar na interface REPO que oferece o conjunto de métodos para manipular os dados na tabela (o famoso CRUD - Create, Read, Update, Delete)

#### Pacote REPO

No pacote REPO, criamos uma interface que deve herdar da interface do JPA Chamada CrudRepository. Esta interface é parametrizada com 2 tipos de dados

* A classe do model que vamos manipular
* Uma classe que representa o tipo de dado da chave primária (ex comum: Integer)

public interface ProdutoRepo extends

CrudRepository<Produto, Integer>{

}

#### Pacote SERVICES

Por que criar serviços? Existem algumas explicações

* Evitar que o controller fique sobrecarregado de lógica (o que dificulta a manutenção e compreensão do código). Dessa forma, delegamos uma camada de tratamento
* Encapsular uma regra de negócios que pode ser chamada por qualquer outro componente do sistema, que não seja o controller (ex: um serviço chamando outro serviço). Isso na Engenharia de Software é bastante comum

Como proceder? Cada serviço é (via de regra) definido como

* uma interface que especifica quais são os métodos daquele serviço
* uma classe que implementa estes métodos

Por que fazer isso? Porque podemos ter mais de uma classe implementando a mesma interface, nos proporcionando a possibilidade de trocar a lógica interna do serviço sem mudar a maneira como o serviço é invocado.

Elementos técnicos:

* geralmente a implementação é quem vai fazer acesso ao repositório. Como isso é declarado? Injetando a dependência
  + **@Autowired**
* Qual a diferença entre Instanciar e Injetar? Quando instanciamos, nós somos responsáveis pelo "new". Na injeção, delegamos o "new" para o SpringBoot
* A classe que implementa precisa ter a anotação **@Component** para que ela seja também uma classe gerenciável pelo SpringBoot (ou seja, uma classe injetável em outras classes)

#### Pacote Controller

O Pacote controller terá as classes que irão atender às requisições dos Endpoints

Endpoint = URL + Método HTTP (GET, POST, PUT e DELETE)

ou seja, uma mesma URL (ex: /produtos) pode ser mapeada para GET, POST, PUT e DELETE e mesmo assim serão considerados Endpoints diferentes

Como tudo funciona

A classe do controller tem que ter a anotação @RestController

Os métodos são anotados de acordo com o método HTTP que querem atender. Exemplo:

@GetMapping("/produto") indicando que vamos atender uma requisição via browser através da URL /produto (e vamos retornar alguns dados em formato JSON)

@PostMapping("/produto") indicando que mesmo sendo o mesmo caminho (URL) a operação representada pelo "/produto" deve receber dados do usuário (seja através de um formulário ou mesmo através de uma ferramenta de simulação de requisições tipo Postman, Insomnia ou Thunder Client)

**Detalhe:** quando fazemos uso de @PostMapping obrigatoriamente nosso método deve declarar o objeto a ser recebido como parâmetro juntamente com a anotação @RequestBody

O que um método como estes deve retornar?

Alternativa 1 - os objetos puramente simples (caso o objeto não exista ou algum erro seja identificado, simplesmente retorna vazio)

Alternativa 2 (mais usual, mais elegante e mais poderosa) - Retorna um objeto completo de resposta, contendo tanto o código de status HTTP e o respectivo conteúdo. Este objeto é o **ResponseEntity**

**Status HTTP**

200 - Operação bem sucedida (método ok())

201 - O registro foi incluído com sucesso no banco de dados

400 - O objeto que eu recebi no endpoint está errado ou incompleto (método badRequest())

403 - meu usuário não tem direito de acessar esse endpoint (não tem permissão)

404 - O Objeto que eu quis consultar não existe (método notFound())

Qualquer outro status diferente destes, podemos usar o método status(codigo) passando por parâmetro o código que queremos

Como fica o caminho da requisição:

* Caminho da requisição
  + Controller → service → repo → model → Banco de dados

O controller aciona o service que chama um determinado método do repositório para poder consultar ou armazenar um objeto no banco de dados.

Métodos GET, fazem tudo isso e esperam receber do Service um objeto ou uma lista de objetos para retornarem (em formato JSON) para o Usuário

Métodos POSt fazem o mesmo caminho, porém na invocação passam o objeto que receberam via Corpo da Requisição.

No fundo, bem lá no fundo um Controller é uma classe que executa uma determinada operação de acordo com a URL recebida.

Internamente o Spring tem uma classe chamada DispatcherServlet (que fica em execução no Tomcat). Esse dispatcher é acionado quando qualquer requisição ao Tomcat é feita. A partir daí, ele intercepta esta requisição, olha para todas as classes que foram anotadas com @RestController e verifica quais métodos correspondem a quais Endpoints. Uma vez identificado o endpoint, o DispatcherServlet executa o método da respectiva classe. Algo parecido com isso aqui:

1. Requisição HTTP
2. DispatcherServlet decodifica a requisição
3. consulta em suas tabelas internas qual URL e qual método HTTP essa requisição
4. Determina qual classe anotada com @RestController tem o método que corresponde à Requisição
5. Se necessário instancia este objeto e executa o método