

# Spring

**Spring boot** é uma **ferramenta/facilitador** que padroniza todas as configurações pra gente e já evita para não precisarmos fazer as configurações manualmente, então ele abstrai muita coisa das configurações iniciais que precisaríamos de fazer em projetos **Spring**, ou seja o **framework** que utilizamos não é **Spring Boot** e sim o **Spring**, **Spring Boot** é apenas a **ferramenta/facilitador** que usamos para criar nossos projetos.

Antes em projetos **WEB** gerávamos arquivos **WAR** ou um **EAR** se for projeto mais **enterprise**, e a gente precisando fazer um **deploy** desse arquivo **WAR** ou **EAR** em um **application container** (**servidor de aplicação com suporte Java**) temos então o **Tomcat**, **JBoss**, **weblogic**, etc. E o **Spring Boot** no momento que estava ficando popular a parte de micro serviços ele implementou o **Tomcat** e é por isso que hoje as aplicações do **Spring** a gente consegue rodar num **Docker**, pois acaba gerando um arquivo **JAR** como pacote final que vai para a produção e esse arquivo **JAR** só precisa do **Java** para ser executado e é o que facilitou a adoção de micro serviços no mundo Java.

## **Application.properties:**

Foi a forma de padronizar a forma que declaramos as propriedades em um projeto Spring.

## **Spring Core:**

Parte principal do **Spring** que é onde temos a parte de **injeção de dependências**, **bootstrap**.

## **Spring Web/Webmvc:**

Anotações **@Controller**, **@Services** que temos.

É todo baseado na especificação **Servlet** do **Java**, então quando criamos um **@Controller** no **Spring**, por trás é um **Servlet** do **Java** que a gente tem.

## **@ResponseBody:**

O **Spring** trabalha muito com aquele modelo do **MVC** ou **Model View**, geralmente quando retornávamos algo o **Spring** esperava qual que nosso **jsp** que iríamos retornar/fazer o redirecionamento, no caso o **@ResponseBody** ele vai transformar por exemplo uma **String** ele vai retornar apenas uma **String**, se for um **Json/objeto** vai retornar isso no formato **Json** ou formato **XML**.

## **@RestController:**

É apenas uma conveniência da gente poder utilizar **@Controller** e **@ResponseBody** e não precisar digitar os 2.

Que indica para o Spring que a classe ela contém um **end-point** (url) que nós vamos poder acessar a nossa API.

### @RequestMapping:

Diz aonde é o **end-point** de determinada **classe**.

Exemplo:

```
@RequestMapping("/api/courses")
public class CourseController {

    // Essa classe então fica com o end-point acima e
    // tudo nela será renderizado quando o end-point for acessado.
}
```

### LOMBOK:

**Lombok** é uma dependência que nos ajuda a ter "atalhos" na nossa aplicação, como por exemplo para gerar os **getters and setters** precisaríamos ter aquele monte de coisa escrita e gerar eles causaria "poluição" na nossa classe já que eles ocupam muitas linhas, com o **Lombok** nós temos as anotações **@Getter**, **@Setter**, **@Data** = cria os **getters and setters** e tem outras **anotações** também que ao colocado em cima ele gera sem precisar da poluição visual.

Exemplo:

```
@Data
@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(length = 60, nullable = false)
    private String name;

    @Column(length = 12, nullable = false)
    private String category;
}
```

Vemos então na imagem que não temos os **getters and setters** poluindo a classe visualmente, porém temos eles e podemos utiliza-los por causa do **@Data** do **Lombok**.

### @GeneratedValue:

Informa como esse valor deve ser gerado, isso depende bastante do banco de dados, porém o mais comum **MYSQL** utiliza o **AUTO** (cria o dado automaticamente ele mesmo).

**length:** Informa o máximo de caracteres que é aceita na coluna.

**nullable:** Informa se a coluna aceita valores nulos ou não.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

### @Component:

Se você quer que o Spring crie uma instância e automaticamente gerencie o ciclo de vida dessa instância você coloca essa anotação.

Existem os **components** especiais como o **@RestController** que é um **component** que vamos expor uma **API/end-point**, **@Repository** que é um **component** especial que está falando para o **Spring** que é uma conexão (vai fazer o acesso ao banco de dados), **@Service** que é um **component** especial que é geralmente aonde colocamos a nossa lógica de negócios e também conseguimos fazer controlar as transações com o banco de dados.

### @Repository:

Para termos acesso aos métodos do banco de dados a gente declara um **repository** como **interface** para podermos estender (**extends**) as **interfaces** que nós temos do próprio **JPA** no **Spring Data** que é um outro módulo que a gente adicionou no nosso **POM.xml** que possui interfaces para poder facilitar o acesso ao banco de dados, ao invés de fazer tudo manualmente com o **ORM** do **Spring** e fazer as conexões com o **hibernate** tudo manualmente, então o **Spring** criou essa outra camada para facilitar.

E ao fazer isso temos então acesso ao **JpaRepository** que iremos usar e nela temos que usar o tipo **generics** (**<>**) que temos que informar qual que é a nossa **entidade** (**Entity**) e qual é a **chave primária** dessa entidade, quando fazemos isso o **Spring** vai criar uma implementação dessa **interface** que já tem os métodos automaticamente para gente poder acessar (para ver isso podemos clicar com o **ctrl+mouse1** no **JpaRepository**) que ele mostra os métodos.

Podemos também declarar métodos, por exemplo **findByName** o **Spring Data** vai criar então um método para podermos acessar fazendo um **SELECT \* FROM (tabela) Where name**.

```
@Repository
public interface CoursesRepository extends JpaRepository<Course, Long> {
```

### @Bean:

Estamos com ele falando para o Spring que queremos que o Spring gerencie todo ciclo de vida.

### @JsonProperty:

Quando o `jxon` tiver fazendo a transformação de `JSON` para `OBJETO` ou `OBJETO` para `JSON` vai transformar o `id` em `_id` nesse caso.

```
@JsonProperty("_id")
private Long id;
```

### @JsonIgnore:

Ignora alguma propriedade na hora de criar o `JSON`.

### @RequestBody:

Se precisarmos pegar algo do corpo vindo do site (front-end) como por exemplo um `JSON` usamos essa anotação para não precisarmos manusear o `JSON` manualmente e sim de maneira automática.

//  
//

Quando temos um atributo que é obrigatório eu ter essa instância para que meus métodos funcionem, ou seja não vai funcionar se eu não tiver a `instância` dessa propriedade, por exemplo um `repository`, a gente considera isso como uma propriedade um atributo obrigatório, quando isso acontece a gente da preferência de fazer a `injeção via constructor`, porque quando o `Spring` for instanciar, o `Spring` vai falar "`Essa classe aqui precisa dessa instância para poder funcionar`", então no momento da criação (`instância`) (`new CourseController`) é que vamos passar essa `instância`, se fizermos isso via atributo (`@Autowired`) ou via `setter` a gente informa que precisamos disso em um 2º momento, então iremos sempre gerar o `constructor`.

### CORS:

Quando vamos `linkar` nossa `API` com o `front-end` existe um conceito chamado `CORS` que são chamadas entre domínios diferentes e isso existe para segurança das `APIs`, pois sem o mesmo qualquer aplicação poderia usar outro site sem problema algum, sem autorização, sem senha, só colocar o link, e por isso foi criado o `CORS`.

Para exemplificar isso imagine: nosso `front` ([meuprojeto.com](http://meuprojeto.com)) fazendo uma chamada para a `API` ([minhapi.com](http://minhapi.com)), ou seja são domínios completamente diferentes.

Então o **CORS** permite que você acesse outro **domínio** (API no caso) só que tem de ser configurado na aplicação.

Mas geralmente não configuramos ele, pois quando a aplicação for para **produção** depois não precisamos desse **CORS**, e ele é muito "chato" para configurar, pois tem de ser configurado qual domínio você irá permitir, se terá senha e usuário para poder acessar, como na maioria dos casos quando formos para **produção** não iremos precisar disso, pois não queremos que outras **APIs** acessem nossa **API** (neste caso, pode haver casos que precisaríamos dele como por exemplo se tivéssemos uma API pública) e sim apenas o nosso **front-end** acesse-a, então na hora que for para produção vamos configurar de forma apropriada, só é bom ter o **CORS** quando na produção precisássemos utiliza-lo.

### ResponseEntity:

Usamos ele como retorno nos métodos **HTTP** para mandar a resposta que desejamos e também especifica o status, geralmente usamos para definir por exemplos os códigos **HTTP** retornados, como exemplo o **201 (CREATED)**, e nele especificamos o **body** com a informação que temos de fazer \*nesse caso de salvar\*.

```
@PostMapping
public ResponseEntity<Course> salvar(@RequestBody Course curso) {
    return ResponseEntity.status(HttpStatus.CREATED)
        .body(coursesRepository.save(curso));
}
```

Porém temos a anotação **@ResponseStatus** que pode simplificar esse código do **ResponseEntity**, porém o diferencial é que o **ResponseEntity** nos permite manusear e alterar os dados da nossa **resposta** (**Response**)

```
@PostMapping
@ResponseStatus(code = HttpStatus.CREATED)
public Course salvar(@RequestBody Course curso) {
    return coursesRepository.save(curso);
}
```

### Location:

Classe do **Angular** que podemos usar para pegar a localização da página e dentro dela temos vários métodos para usarmos, como por exemplo para voltar a página anterior usamos o **back()**, podemos redirecionar o usuário para outro **end-point** com o **go()**.

```
onCancel() {
    this.location.back(); // Está fazendo com que ao ser acionado o método onCancel() ele volte a página para a anterior.
}
```

### @GetMapping:

Quando fazemos um método **GET** usamos essa anotação, porém podemos falar que o método que estamos criando ele vai aceitar uma variável de **URI**, ou seja uma variável que vai vir como parte de **URL** da nossa **API**, passamos então o valor da nossa variável o mesmo do nosso retorno entre chaves e aspas duplas.

```
@GetMapping("/{id}")
public ResponseEntity<Course> buscaId(@PathVariable Long id) {
```

### @PathVariable:

Para informar que nossa resposta está vindo como parte da **URL** adicionamos essa anotação (**@PathVariable**), ou seja uma variável que está no caminho da nossa **API** que no caso é **/api/courses/\*id\*** é parte da **URL**.

Se nossa variável tiver nome diferente podemos informar o nome como parâmetro dela para o **@GetMapping** poder identificar corretamente.

```
@GetMapping("/{id}")
public ResponseEntity<Course> buscaId(@PathVariable("id") Long id_diferente) {
```

### Update():

Para criar um método de atualizar podemos usar este método direto na base, porém não é o recomendado, pois estamos passando qual **identificador** que nós estamos tentando achar e também a parte do corpo com a informação atualizada, mas tem casos de o usuário passar um **id** que não existe e precisamos tratar este caso, então antes de fazer o **update** iremos verificar se o registro existe para evitar erros.

```
@PutMapping("/{id}")
public ResponseEntity<Course> update(@PathVariable Long id, @RequestBody Course curso) {
    return coursesRepository.findById(id) // Está verificando se o curso existe buscando por id.
        .map(recordFound -> { // Se o curso existir ele pega o curso faz o map e seta o nome do curso com o curso atualizado e a categoria também.
            recordFound.setName(curso.getName());
            recordFound.setCategory(curso.getCategory());
            Course updated = coursesRepository.save(recordFound); // Variável criada para conter um objeto do tipo Course que irá salvar a informação do curso atualizado.
            return ResponseEntity.ok().body(updated); // Retorna para o código ok e no corpo o curso já atualizado.
        })
        .orElse(ResponseEntity.notFound().build()); // Se não encontrar iremos fazer o retorno de 404 dizendo que não foi encontrado o registro.
}
```

Podemos observar que na verificação nós não atualizamos o **id** somente o **name** e **category**, isso acontece por não ser necessário já que vem **populado da base de dados** e também por ser **extremamente perigoso por gerar muitas brechas de falha**.

Com isso vemos que o **update()** não é muito usado e sim tratamos a informação para verificar se realmente existe e usamos o **save()** quando já tratada.

### Delete():

No delete usamos a anotação **@DeleteMapping** para indicar que é um método de **DELETE** para nossa **API**.

No método para retornarmos algo que foi removido, quando tratamos de **DELETE** agente retorna `noContent()` ou seja **nada**, mas isso tem um tipo que é diferente de `Void` que no caso é `object` por conta do `build()` que está retornando para a página o `noContent()`, então forçamos o retorno de `Void` que vai transformar (**casting**) o objeto do `noContent()` que é nada para o tipo `Void`.

```
@DeleteMapping("/{id}")
public ResponseEntity<Void> delete(@PathVariable Long id) {
    return coursesRepository.findById(id) // Está verificando se o curso existe buscando por id.
        .map(recordFound -> { // Se o curso existir ele pega o curso faz o map e exclui o curso que tem o id passado na url.
            coursesRepository.deleteById(recordFound.getId());
            return ResponseEntity.noContent().<Void>build(); // Retorna o noContent() que é o nada no corpo da requisição.
        })
        .orElse(ResponseEntity.notFound().build()); // Se não encontrar iremos fazer o retorno de 404 dizendo que não foi encontrado o registro.
}
```