

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:

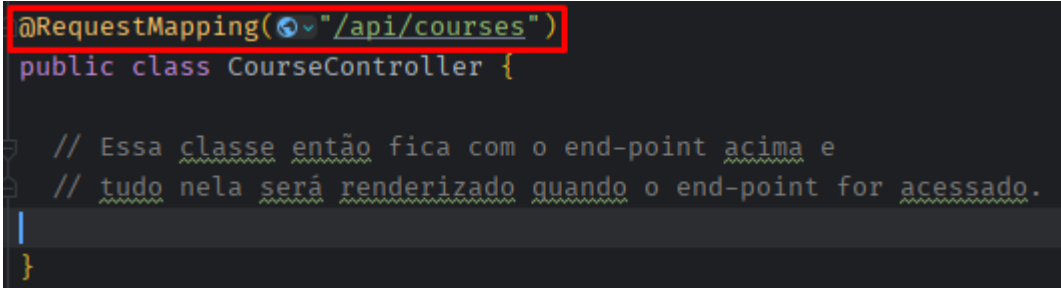
É apenas uma conveniência da gente poder utilizar @Controller e @ResponseBody e não precisar digitar os 2.

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**.

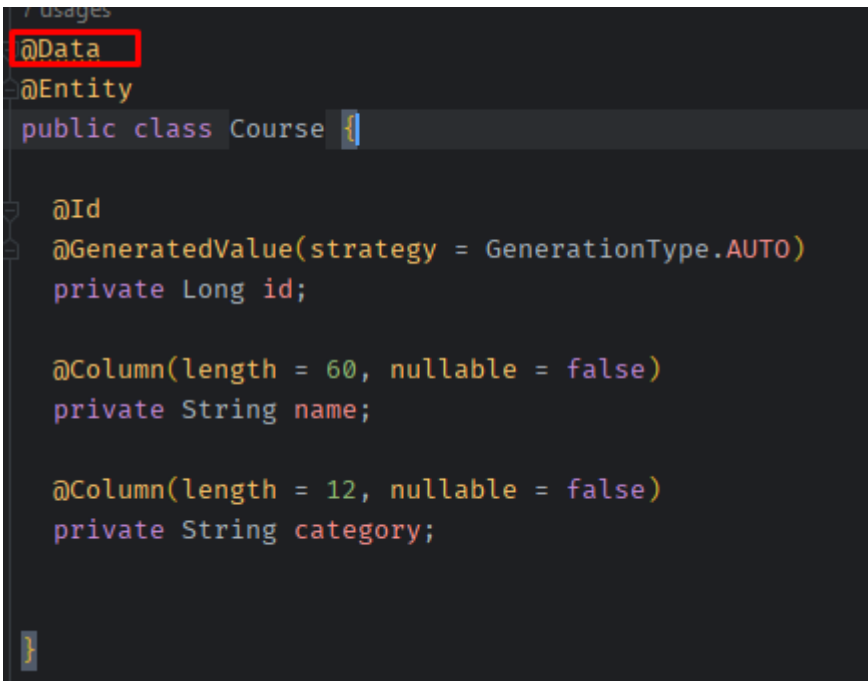
Exemplo:

- A screenshot of a code editor showing a Java class named CourseController. The first line is @RequestMapping("/api/courses"), which is highlighted with a red box. Below it is the class definition: public class CourseController {. There are two comments: // Essa classe então fica com o end-point acima e and // tudo nela será renderizado quando o end-point for acessado. The class ends with a closing brace }.

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:

- A screenshot of a code editor showing a Java class named Course. The first line is @Data, which is highlighted with a red box. Below it is @Entity. The class definition is: public class Course {. There are three fields: @Id @GeneratedValue(strategy = GenerationType.AUTO) private Long id;, @Column(length = 60, nullable = false) private String name;, and @Column(length = 12, nullable = false) private String category;. The class ends with a closing brace }.

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 metodo 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.

- ```
@PostMapping("/{id}")
public ResponseEntity<Course> update(@PathVariable Long id, @RequestBody Course curso) {
    return coursesRepository.findById(id) // Esta 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.
}
```

## Validação:

### Java Bean Validation:

Para o nosso código podemos fazer com **IFs** a validação, por exemplo de tamanho que seria `if (course.getName() > 100)` retornaria uma **exception**, porém isso fica muito inviável e com muito código, ainda mais se for vários campos para se verificar.

E o próprio **Java** tem uma biblioteca que cuida disso para nós que se chama **Bean Validation**, como estamos usando o **Spring** e ele é baseado na **API Jsp** e **Servlet**, na **API Servlet** que também faz parte do **Jakarta**, a gente já tem essa validação disponível para nós.

### @NotNull:

Não deixa ser nulo nem vazio.

### @NotBlank:

Verifica se tem pelo menos um caractere que não seja espaço.

### @Pattern:

É usado para definirmos um padrão, esse pattern tem uma opção chamada de regexp (expressão regular) que com ele podemos definir os valores que aceitamos naquele campo.

- ```
@Pattern(regexp = "Back-end|Front-end")
```

Após adicionarmos no **model** do **Course** todas as validações desejadas, para funcionar realmente precisamos de ir no **controller** e adicionar o **@Valid** nos parâmetros que chamam o **Course** de todos os métodos, ele então quando receber aquela requisição vai validar se o nosso **JSON** que foi transformado para uma instância da variável **Course** contém informações válidas de acordo com as validações que colocamos no **MODEL**.

- ```

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

```

Podemos também usar essas anotações diretamente na classe ou no controller, como faremos no id por exemplo.

- ```

@GetMapping("/{id}")
public ResponseEntity<Course> buscaId(@PathVariable @NotNull @Positive Long id) {
    return coursesRepository.findById(id)
        .map(recordFound -> ResponseEntity.ok().body(recordFound)) //Se nosso optional
        .orElse(ResponseEntity.notFound().build()); // Se não encontrar iremos fazer o
}

```

Mas como estamos declarando ali no parâmetro do método precisamos adicionar uma outra validação no **RestController** que é o **@Validated**, com isso nosso **Controller** vai validar todas as validações do **Java Bean** ou então do **Hibernate validators** desde que estejam declaradas nos parâmetros dos método no caso o **@NotNull** e o **@Positive**.

Soft Delete (Remoção lógica): A remoção normal ela exclui do banco de dados o dado sem deixar informações sobre essa exclusão e sem ser possível recuperar este dado, porém para termos o **Soft Delete** que é um método de fazer o **delete** que irá armazenar a informação de quem excluiu, a informação ao invés de ser deletada ela passará a ter outro estado, etc. Para isso ao invés de fazermos um **delete** a gente coloca a informação em **outro status** (coluna), então ao invés de fazermos um **delete** estamos fazendo um **PUT** (atualizando) do registro naquela coluna no banco de dados.

Para fazer o **Soft Delete** podemos fazer de **2 maneiras**, modificando o código de **delete** para **UPDATE** e filtrando os cursos no **GET** por **status = ativo** ou podemos colocar no **Model** a anotação **@SqlDelete** que irá passar o comando **SQL** que o **hibernate** irá executar toda vez que chamarmos o método **DELETE** do nosso **repository**.

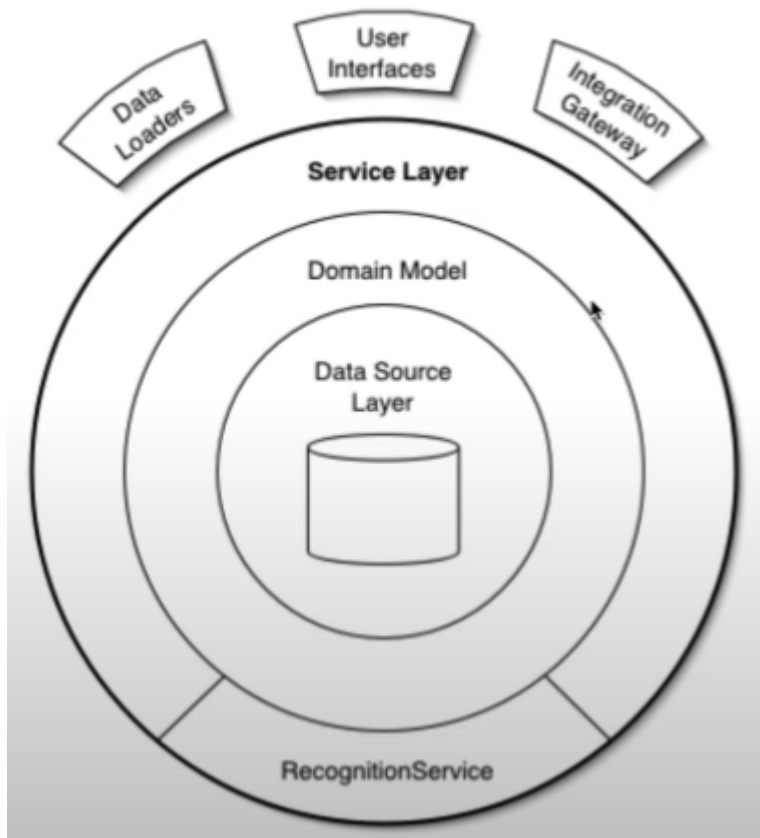
- ```
@Data
@Entity
@SQLDelete(sql = "UPDATE Course SET status = 'Inativo' WHERE id = ?")
public class Course {

 no usages
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 @JsonProperty("_id") //Quando o jxon tiver fazendo a transformação d
 private Long id;
```

Assim quando um **DELETE** for feito o status passa a ser inativo, porém quando dermos um **GET** ainda mostrará esse curso com a categoria inativo, resolvemos isso com a anotação **@Where** que toda vez que formos fazer um **SELECT** no nosso banco de dados o **Hibernate** automaticamente vai adicionar esse filtro na clausula **WHERE**, podemos ter outras clausulas **WHEREs** também, pois ele irá concatena-los.

- ```
@Data
@Entity
@SQLDelete(sql = "UPDATE Course SET status = 'Inativo' WHERE id = ?")
@Where(clause = "status = 'Ativo'") // toda vez que formos fazer um
public class Course {
```

Camada de Serviço: Algumas empresas tem usado da camada serviço que é a camada que oferece as operações e coordena as repostas da aplicação em alguma operação



Conforme vemos na imagem temos os **Data Loaders**, **User interfaces**, **Integration Gateway** e a camada de serviço está protegendo a camada de domínio (modelo/entidade), e também temos o repositório que está fazendo a interface diretamente com a camada de dados que é parte do banco de dados, outro aspecto positivo porém pessoal para usar a camada de serviço é que em aplicações quando começamos a aumentar a complexidade, ter muita **lógica de negócio** as vezes tenhamos que ter vários acessos ao banco de dados de uma vez ou escrever no banco de dados como por exemplo temos de fazer o **update** em várias tabelas diferentes como parte de uma única transação, equando temos uma camada de serviço fica mais fácil fazer isso e também de fazer a manutenção, além de ficar melhor de entender o código.

ALGO BOM DE SE FAZER É DEIXAR AS VALIDAÇÕES TAMBÉM NA CLASSE DE SERVIÇO, POIS PODEMOS TER VÁRIOS CONTROLLERS USANDO APENAS UM SERVICE E ASSIM PODEMOS TER VALIDAÇÕES DIFERENTES PARA TAIS APENAS ESPECIFICANDO AS DIFERENÇAS DE VALIDAÇÕES NO PRÓPRIO CONTROLLER.

Muitas pessoas fazem a camada de serviço criando uma interface e uma classe que implementa ela, mas isso é só indicado em casos em que usamos o **Spring AOP** que é a **programação orientada a aspectos**.

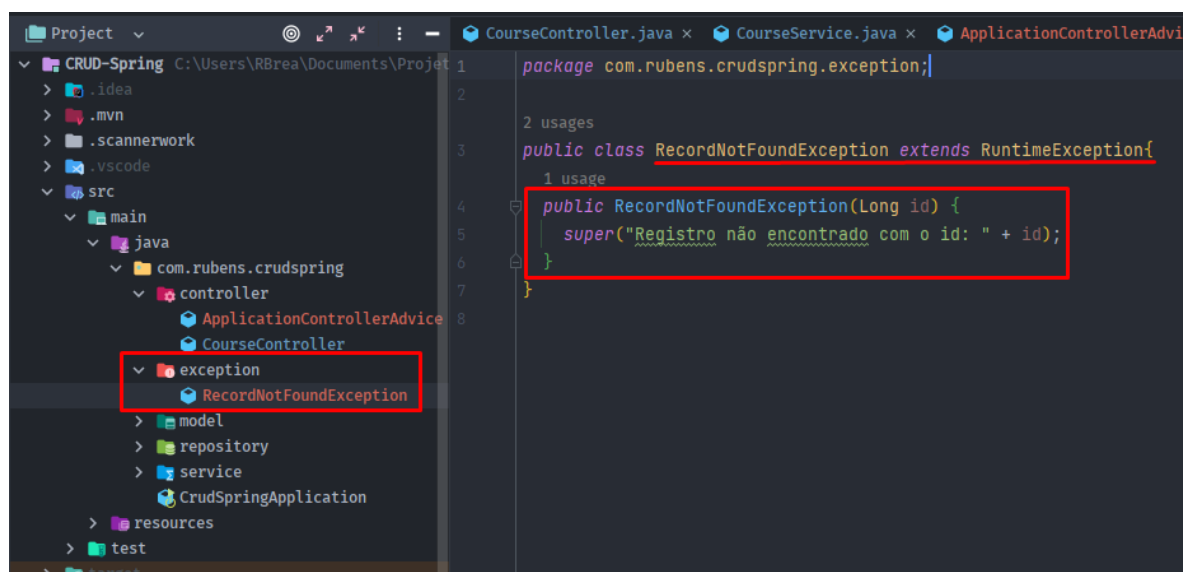
@Service no **Spring** temos essa anotação para indicar ao **Spring** que a classe que a possui é uma classe de **Service** e ela é derivada da anotação **@Component** e ela

permite que o **Spring** detecte essa classe e crie a instância automaticamente para podermos utilizar no controle de independências.

TRATANDO ERROS:

Para podermos tratar as exceções da nossa aplicação podemos tirar do código o map que temos e criar um package exception e nele termos uma classe para cada erro de requisição que deriva de **RuntimeException** e assim criamos por exemplo o

RecordNotFoundException com um construtor que entrega a mensagem do erro.



Para devolvermos para o usuário o erro **HTTP** correto criamos dentro do **Controller** uma classe que irá capturar e vai recomendar e o que fazer com as exceções que forem lançadas por **quaisquer Controllers** que a gente tem no nosso projeto e nessa classe iremos usar nela uma **anotação especial** que temos no **Spring** que é justamente para os **Controllers Advice** que é um **@RestControllerAdvice**, ou seja é a classe que vai dizer para todos os **Rest Controllers** o que fazer com as exceções.

Iremos fazer de exemplo um método dentro do **Controller Advice** que irá tratar de erros **Not Found HTTP**, criamos então um método que recebe a tratativa do erro **Not Found** da classe que criamos em **Exception** e retorna a mensagem que possui nessa classe e além disso devemos colocar uma anotação em cima do método indicando qual a exceção que esse método vai lidar que é a **@ExceptionHandler** e passamos o parâmetro com a nossa classe do **Not Found** para indicar qual o erro que tratamos no método.

```

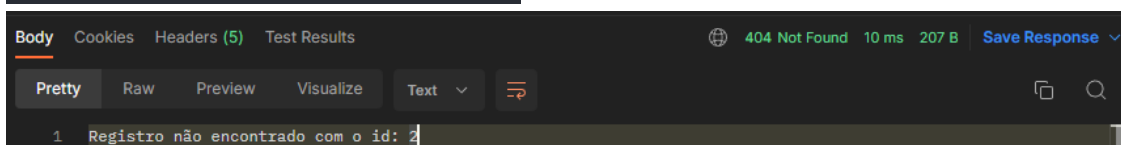
no usages
@RestControllerAdvice
public class ApplicationControllerAdvice {

    no usages
    @ExceptionHandler({RecordNotFoundException.class})
    public String handleNotFoundException(RecordNotFoundException ex) {
        return ex.getMessage();
    }
}

```

Porém dessa maneira ele exibe a mensagem do erro, mas o status do **HTTP** é retornado o **200 (OK)** e não o **404 (Not Found)**, para resolvermos isso iremos adicionar mais uma anotação que é a **@ResponseStatus** e passamos qual que é o tipo do status que queremos que o **Spring** retorne automaticamente no **ResponseEntity** quando capturar essa exceção.

```
@ResponseStatus(HttpStatus.NOT_FOUND)
```



PADRÃO DTO (Data Transfer Object):

O indicado pelas boas práticas é que não usemos a nossa **entidade** no **controller**, pois essa abordagem trás alguns problemas para nós como: **Expor as informações do jeito que elas existem no banco de dados na nossa API.**

O **DTO** nada mais é que uma classe que vai representar nossa **informação/requisição** tanto a que está vindo quanto as repostas que estamos enviando.

Esse padrão também foi criado para que economizasse o número de requisições para o servidor e para que simplificasse a requisição, quando temos um objeto na nossa requisição que é um pouco mais complexo com vários campos, com objetos aninhados fica mais fácil criarmos um **DTO** para recebermos toda a informação de uma vez e fazer uso dessa informação e aplicar a lógica necessária para poder ler e processar toda a informação que está vinda no **DTO**.

Abordagem "antiga" para criar um **DTO**:

Para criarmos uma classe **DTO** é bem simples, basta criar um **package** de **DTO** e a classe **DTO** da **entidade** que estamos fazendo esse padrão (DTO) e copiar tudo que tem dentro da classe e tirar as "coisas" relacionadas ao **banco de dados**. Em seguida iríamos precisar somente refatorar nosso código não retornasse nem enviasse a entidade em si na nossa **API**, só que como estamos utilizando o Spring 6 com o Spring Boot 3 e também

estamos usando Java 19 no nosso pom.xml agente pode utilizar uma abordagem mais nova e mais moderna do Java que são as Records (em projetos da versão 13 ou inferior temos que seguir o padrão de classe, pois Records não estão disponíveis).

Abordagem atualizada para criar um DTO (**RECORD**):

Criamos o `package de dto` porém ao invés de criamos a classe normalmente criamos ela como record e não classe.

- ```
public record CourseDTO() {
 }
}
```

Uma **Record** é basicamente uma classe **Java** e nessa classe **Java** nós não temos construtor vazio nós apenas temos o **construtor com todos os campos** e nós também **não temos métodos Setters**, então a única forma de termos informações (dados nas nossas propriedades) é através do construtor, uma vez que criou o construtor e instanciou a classe não conseguimos mais modificar a informação, pelo motivo de não conseguirmos mais modificar a informação que está na classe a **Record** é uma classe imutável do **Java**, outra diferença é que os métodos **Getters** não temos mais o prefixo **GET** é somente o nome da propriedade.

Uma **Record** é apenas uma classe, e essa classe estende um tipo especial do **Java** que é a classe **Record** e é por esse motivo também que ela não consegue estender outras classes, porém podemos implementar quantas interfaces forem necessárias.

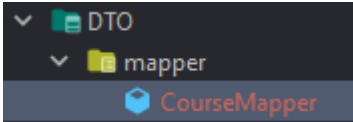
**Extras:** Podemos criar um **DTO** de resposta e o de requisição (request) isso vai por escolha.

Não conseguimos utilizar **Records** como entidade **AINDA** por conta das regras do **JPA**, o **JPA** fala que **para termos uma classe como entidade** essa classe tem que ter um construtor vazio e ter **Setters**, pois quando o **JPA/Hibernate** vai instanciar, buscar informações na base ele por trás dos panos vai instanciar a classe e depois vai utilizar os métodos **Setters** para poder configurar e **setar** o valor em cada propriedade. Porém com o **JDBC** fazendo as **queries** na mão conseguimos mapear com as Records.

**A gente usa o DTO sempre na camada de serviço, pois é ela que "protege" as camadas de domínio, se ao invés de usarmos no service a gente usar no controller, acaba que se tivermos outro controller que pegue o model em questão iremos que repetir tudo. Então é o serviço que via pegar o dto e transformar em entidade e vai fazer os acessos ao banco de dados e vai retornar as respostas para nosso controller.**

**Padrão Mapper:** É um padrão que faz o mapeamento de DTO para entidade e entidade para DTO quase que automaticamente, temos vários frameworks que podemos adicionar no nosso projeto que irá fazer isso, mas como estamos em um projeto para estudos e pequeno iremos fazer o nosso próprio.

Criamos um package chamado **mapper** e a classe do **mapper** dentro do package de **DTO**.

- 

Existem **anotações** do **Spring** que cria instâncias automáticas dessa classe para nós e para podermos também usar a instância dessa classe em outras classes, como por exemplo na nossa classe de serviço, e é a anotação mais básica do **Spring** que é a **@Component**, nessa classe iremos criar dois métodos **umtoDTO** que vamos transformar uma instância de uma entidade em uma instância de um **DTO** e nós vamos retornar um **DTO**, e vamos também fazer o **toEntity** que nesse caso vai receber uma instância de **DTO** e transformar em uma instância de entidade e retornaremos uma entidade.

- ```
public CourseDTO toDTO(Course course) {  
    return new CourseDTO(course.getId(), course.getName(), course.getCategory());  
}
```

No nosso **Service** iremos usar de **programação reativa**, no exemplo iremos refatorar o comando abaixo.

- ```
public List<CourseDTO> list() {
 return courseRepository.findAll();
}
```

Esse **findAll()** é uma lista e essa lista podemos retornar um **streaming** de informações, ou seja **para cada objeto que estiver nessa lista podemos fazer alguma coisa**.

- ```
public List<CourseDTO> list() {  
    return coursesRepository.findAll().stream().map(courseMapper::toDTO).collect(  
    ArrayList::new, ArrayList::add, ArrayList::addAll);  
}
```

Primeiro estamos pegando todos os registros que foram retornados para nós, depois transformamos em um **stream** para que possamos fazer nossa **programação reativa**, e os **streamings** no **Java** também tem um operador chamado **map** que **nele estamos pegando nosso Mapper e chamando toDTO** (nesse caso é uma lambda introduzida no **Java 8** para simplificar o comando) sem essa lambda seria assim o comando:

- ```
.map(course -> courseMapper.toDTO(course))
```

E como nesse `map` iremos retornar vários objetos do tipo `DTO` a gente precisa colocar esses objetos dentro de uma lista novamente, para isso usamos o `collect` e instanciamos o `ArrayList` com `ArrayList::new`, e depois adicionamos as informações com o `ArrayList::add`, e damos um `ArrayList::addAll` para adicionar tudo, ou então podemos utilizar outra forma de fazer isso que é através do `Collectors.toList()` que assim vamos pegar todo objeto que foi retornado pelo `map` que é do tipo `DTO` e vamos colocar `nolist` já que estamos retornando uma `List`, \*o `Collectors` pertence ao pacote do `Java Util`.\*

- `.collect(Collectors.toList());`

Já o método `toEntity` fazemos assim como mostra a imagem abaixo:

- ```
public Course toEntity(CourseDTO courseDTO) {
    if (courseDTO == null) {
        return null;
    }

    Course course = new Course();
    if (courseDTO.id() != null) {
        course.setId(courseDTO.id());
    }

    course.setName(courseDTO.name());
    course.setCategory(courseDTO.category());

    return course;
}
```

Isso fazemos por ser um projeto pequeno, mas se for um projeto de uma organização (enterprise) a gente pode aplicar um padrão chamado `Builder`, que cuida quando temos uma classe com muitos campos, mas no caso não iremos utiliza-lo, pois o `Course` tem apenas 4 campos.

Enums:

São utilizados quando temos opções esperadas no banco de dados como por exemplo um campo que espera os valores "Ativo" ou "Inativo", que se usarmos `Enums` fica bem mais fácil de tratarmos e verificarmos, além de deixar nosso banco de dados muito mais limpo.

Para começar a usar um `Enum` iremos criar um `package enum` e com a classe do mesmo nome do campo cujo iremos transformar em um `Enum`, dentro dessa nova classe de `enum` iremos colocar os valores que esperamos receber nele.

- ```
public enum Category {
 BACKEND, FRONTEND
}
```

E o nosso campo iremos colocar o tipo dele do nosso **Enum** (que no caso é Category).

- ```
private Category category;
```

Se estivermos com nosso código sem o uso de Enums então precisaríamos refatorar principalmente o mapper, a parte de mapeamento com DTOS faz com que tenhamos que alterar tudo isso.

Como antes a gente colocava uma **String** de "Back-end" ou "Front-end" e agora estamos colocando um **enumerador** ele vai ter um valor diferente no banco de dados e existem 2 maneiras de fazermos isso, a 1ª maneira é que quando nós temos esse **enumerador** a gente precisa falar pro **Hibernate/JPA** qual que é o tipo de dados que nós vamos salvar, afinal isso é um **Enum** e um **Enum** não é exatamente um tipo de coluna que existe no banco de dados, no banco de dados temos o tipo **String**, **Varchar**, **integer**, **data**, etc. Então usamos um por exemplo

@Enumerated(EnumType.ORDINAL) que nós então estamos falando para o **JPA** e pro **Hibernate** que esse campo é um campo do tipo **enumerador** e a gente precisa salvar essa informação no banco de dados e a forma que nós vamos salvar no banco de dados é do tipo ordinal (Número).

- ```
SELECT * FROM COURSE;
```

| ID | CATEGORY | NAME               | STATUS |
|----|----------|--------------------|--------|
| 1  | 1        | Angular com Spring | Ativo  |
| 2  | 0        | Pintinhos          | Ativo  |

(2 rows, 0 ms)

Quando salvamos com **Hibernate** do tipo **ordinal** nós estamos salvando o **index** do **enumerador**, então o "Back-end" como é o primeiro fica com valor 0 e o "Front-end" como é o segundo fica com valor de 1 é como se fosse um **Array**, mas essa abordagem tem um problema que é se em algum momento dermos manutenção no nosso código e trocarmos os valores de ordem na classe **enum** iremos ter problemas com todos os dados que já estão salvos no nosso banco de dados, então esse é um problema de salvarmos do tipo **ordinal**.

A 2ª forma que temos de fazer isso é mapeando a **String** que ao invés de colocarmos **@Enumerated(EnumType.ORDINAL)** iremos colocar **@Enumerated(EnumType.STRING)** e ficaria com o mesmo nome do nosso **Enum**.



- ```
SELECT * FROM COURSE;
```

ID	CATEGORY	NAME	STATUS
1	FRONTEND	Angular com Spring	Ativo
2	BACKEND	Pintinhos	Ativo

(2 rows, 0 ms)

Mas se por algum motivo decidirmos renomear as informações do **Enum**, por exemplo:

- ```
public enum Category {
 BACK_END, FRONT_END
}
```

Sempre que renomearmos o valor do nosso **Enum** iremos ter aquele mesmo problema do **Ordinal**, vamos ter problema com os valores antigos no banco de dados e iríamos ter que dar um **update** neles, então esses são os pros e contras das duas abordagens.

**O pro:** É a abordagem mais simples e direta possível de fazermos isso.

**Contra:** Qualquer mudança que fizermos no nosso **enumerador** se tivermos informações salvas no nosso banco de dados iremos ter de atualizar essas informações, então precisa ter muito cuidado quando formos fazer qualquer tipo de refatoração nos nossos **Enums**.

**3º abordagem** e a mais preferível: No lugar de colocar apenas o valor do Enum [e termos Enums com valores, ou seja essas informações:

- ```
BACKEND, FRONTEND
```

elas irão se tornar construtores da seguinte forma:

- ```
1 usage
BACKEND(value: "Back-end"), FRONTEND(value: "Front-end");

2 usages
private String value;

2 usages new *
private Category(String value) {
 this.value = value;
}
```

O **BACKEND** com o value de "Back-end" e **FRONTEND** com value de "Front-end", declaramos também uma variável do tipo **String** com nome de value para pegar o valor desses 2 campos, o construtor colocamos como **private**, pois não queremos instanciar nada.

Depois iremos colocar o método `getValue()` para pegar o valor que vamos salvar no banco de dados do nosso **enumerador**, enfim criaremos um `toString()` já que o nosso tipo é `Enum` então temos esse método para transforma-lo em `String` quando chamarmos esse método (`toString()`).

**E assim ficará nossa classe do enumerador:**

- ```
8 usages  🧑 RubensSsN *
public enum Category {

    1 usage
    BACKEND( value: "Back-end"), FRONTEND( value: "Front-end");

    3 usages
    private String value;

    2 usages  new *
    private Category(String value) {
        this.value = value;
    }

    no usages  new *
    public String getValue() {
        return value;
    }

    new *
    @Override
    public String toString() {
        return value;
    }
}
```

Mas se formos salvar desta maneira o banco de dados não estará com os valores que queremos (value) e sim com os nomes dos `Enums` (BACKEND, FRONTEND).

Para fazermos o nosso banco de dados salvar o value e não o nome do **enumerador** a gente vai precisar criar um conversor para que o `Spring` saiba o que exatamente queremos salvar no nosso banco de dados, lembrando que no nosso **enumerador** poderíamos ter mais de um atributo e poderíamos ter mais de um valor no nosso construtor, por isso o `Spring` precisa que você fale como é que queremos que este **enumerador** seja persistido no banco de dados.

Para fazer isso precisamos criar uma classe nova que iremos mapear no nosso `model Course`, essa classe nova será um conversor.

Dentro do `package enums` criamos um novo `package` chamado `converters` que é aonde ficarão os conversores, para cada `enumerador` que tivermos no nosso projeto a gente vai criar um conversor também.

Iremos então na classe do Conversor implementar uma interface que existe no `JPA` que se chama `Attribute Converter` (Conversor de atributo que é o que estamos fazendo), nessa implementação através do operador `<>` precisamos passar qual que é o nosso `enumerador` e o tipo que queremos salvar no banco de dados, implementamos então os métodos que ele vai sugerir para parar o erro de compilação, com esses métodos toda vez que tentarmos salvar essa informação no banco de dados no 1º método ele vai pegar qual que é o `enumerador` e usar a lógica que colocamos para dizer qual o valor que vamos salvar no banco de dados, no 2º método da mesma forma quando o `JPA` e o `Spring` tentar ler essa informação do banco de dados no banco de dados teremos uma `String` independentemente do valor da `String` a gente precisa falar para o `Spring`, `JPA`, `Hibernate` como é que vamos transformar essa `String` em um valor do nosso `enumerador` que é a lógica que colocaremos no 2º método.

- ```
@Converter(autoApply = true)
public class CategoryConverter implements AttributeConverter<Category, String> {

 @Override
 public String convertToDatabaseColumn(Category category) { // Pega o Enum e converte em uma String com o mesmo valor (keep).
 if (category == null) {
 return null;
 }
 return category.getValue();
 }

 @Override
 public Category convertToEntityAttribute(String value) {
 if (value == null) {
 return null;
 }

 return Stream.of(Category.values()) // A classe utilitária Stream do Java tem o operador of que consegue transformar qualquer lista de informações (no caso temos aqui um Array de informações) em um streaming.
 .filter(category -> category.getValue().equals(value)) // Com o Streaming em mãos conseguimos usar esse operador (filter) que nele conseguimos colocar um filtro que iremos usar que no caso estamos pegando a category do
 // enumerador e usando o equals para poder compararmos com value
 .findFirst() // Ao fazermos o filter pode ser que retorne muitos valores, não é o nosso caso aqui, mas é assim que o Java funciona para isso temos o findFirst que de todos os valores retornados pega somente o primeiro.
 .orElseThrow(new IllegalArgumentException()); // Caso não encontre o valor acima a gente vai lançar um IllegalArgumentException que é uma exceção em tempo de execução que irá falar que o argumento não é válido
 }
}
```

Se não quisermos chamar essa conversão na mão podemos usar a anotação `@Converter(autoApply = true)` que irá pedir ao `JPA` aplicar essa conversão sempre que for necessário.

- ```
@Converter(autoApply = true)
public class CategoryConverter implements AttributeConverter<Category, String>
```

Após isso só colocarmos no `model` a anotação no nosso campo desejado como `@Convert(converter = CategoryConverter.class)` para informar qual é o conversor daquele campo.

- ```
@NotNull
@Column(length = 10, nullable = false)
@Convert(converter = CategoryConverter.class)
private Category category;
```

Dessa maneira teremos o resultado como esperado só que com **Enums**:

- ```
SELECT * FROM COURSE;
```

ID	CATEGORY	NAME	STATUS
1	Front-end	Angular com Spring	Ativo
2	Back-end	Pintinhos	Ativo

(2 rows, 1 ms)

O **DTO** está retornando uma **String**, para fazê-lo funcionar corretamente precisamos obter qual o valor do enumerador, podemos fazer de duas maneiras: Copiar o código do **streaming** que está abaixo.

```
return Stream.of( ... values: Status.values())
    .filter(predicate: c → c.getValue().equals(anObject: value))
    .findFirst()
    .orElseThrow(exceptionSupplier: IllegalArgumentException::new);
```

Ou fazer um método novo para podermos calcular qual o valor do enumerador sem utilizar o código acima, para isso iremos utilizar uma expressão **switch**, nós temos o bloco de comando que é o **switch case** só que o **Java** agora também tem a expressão **switch** que nela vamos pegar qual o valor do nosso **value** e colocar as ocasiões.

Caso no **switch** o **value** tenha o valor de "**Back-end**" por exemplo ele vai chamar o **Category.BACKEND**, como é um valor que pode ser retornado, no final das chaves colocamos o ponto e vírgula (;) e adicionamos o **return** atrás do **switch** que assim ele retorna o valor normalmente do nosso enumerador.

```
public Category convertCategoryValue(String value) {
    if (value == null) {
        return null;
    }
    return switch (value) {
        case "Front-end" → Category.FRONTEND;
        case "Back-end" → Category.BACKEND;
        default → throw new IllegalArgumentException("Categoria inválida: " + value);
    };
}
```

Então no nosso **Mapper** também no método **toEntity()** iremos colocar no **setCategory** a chamada para este método passando como valor a categoria que está vindo com o **DTO**.

```

public Course toEntity(CourseDTO courseDTO) { // Método que transforma um CourseDTO em uma entidade Course.
    if (courseDTO == null) {
        return null;
    }

    Course course = new Course();
    if (courseDTO.id() != null) {
        course.setId(courseDTO.id());
    }

    course.setName(courseDTO.name());
    course.setCategory(convertCategoryValue(courseDTO.category()));

    return course;
}

```

No **Service** iremos fazer a mesma chamada só que ao invés de **DTO** é curso.

```

public CourseDTO update(@NotNull @Positive Long id, @Valid @NotNull CourseDTO curso) {
    return coursesRepository.findById(id) // Está verificando se o curso existe buscando
        .map(recordFound -> { // Se o curso existir ele pega o curso faz o map e seta o no
            recordFound.setName(curso.name());
            recordFound.setCategory(courseMapper.convertCategoryValue(curso.category()));
            return courseMapper.toDTO(coursesRepository.save(recordFound)); // Está transform
        }).orElseThrow(() -> new RecordNotFoundException(id));
}

```

RELACIONAMENTOS:

OneToMany: Um para muitos, é um relacionamento que temos no Spring que uma tabela do banco de dados pode ter vários links de outras tabelas, ex: Um Course pode ter várias Aulas. Essa anotação que também pertence ao pacote do **jakarta persistences** ou **javax persistences** caso você esteja utilizando uma versão mais antiga do Spring.

Artigo "[A melhor maneira de mapear uma anotação @OneToMany com JPA e Hibernate](#)".

No **OneToMany** podemos adicionar algumas informações e temos algumas opções que a gente pode adicionar nele, uma delas que é muito comum de utilizarmos pe o **cascade**.




No nosso caso iremos defini-lo com um valor de **CascadeType.ALL** com esse valor sempre que fizermos qualquer mudança na **entidade** (Course) a gente também vai passar e vai verificar se essas mudanças também precisam ser passadas para as **classes filhas** que no caso é a classe Lesson, então se removermos um curso não faz sentido termos as aulas dele para isso usaremos o **orphanRemoval** para que eles remova os registros que fiquem órfãos, então se fizermos um **Delete** de algum curso ele também irá remover as lições.

```

Estamos na classe Course
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
private List<Lesson> lessons = new ArrayList<>();








```

Mas se deixarmos somente dessa maneira acaba que o mapeamento ficará como de **muitos para muitos**(**ManyToMany**), fazendo com que tenhamos 3 tabelas, uma de **Course** outra de **Lesson** e a outra de **Course_Lessons**, esse tipo de mapeamento ocorre quando temos um **Muito para muitos** aonde a gente precisa de uma terceira tabela para poder ajudar no nosso mapeamento, então quando usamos o comando apenas daquela maneira no **Hibernate** a gente vai ter esse relacionamento equivoco de **muitos para muitos**.

-  **COURSE**
 **COURSE_LESSONS**
 **LESSON**

Porém existe uma outra anotação quando queremos que a **Lesson** tenha uma **coluna** nova que seria a **chave primária** do **Course** (**id**), essa anotação seria **@JoinColumn** que nele conseguimos dizer exatamente qual que é o nome da coluna aonde queremos fazer esse **Join**, isso seria como se a **coluna de id do Course** existisse na **tabela de Lessons** já, **por colocar isso não precisamos também declarar necessariamente que essa tabela existe em Lesson**.

- ```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@JoinColumn(name = "course_id")
private List<Lesson> lessons = new ArrayList<>();
```

-  jdbc:h2:mem:devdb  
 **COURSE**  
 **LESSON**  
 INFORMATION\_SCHEMA  
 Sequences  
 Users  
 H2 2.1.214 (2022-06-13)

Run Run Selected Auto complete Clear SQL statement:

SELECT \* FROM LESSON|

SELECT \* FROM LESSON;  

| ID | NAME       | YOUTUBE_URL | COURSE_ID |
|----|------------|-------------|-----------|
| 1  | Astronomia | Acadovski   | 1         |

(1 row, 1 ms)

Edit

Se a gente tivesse fazendo o design do nosso código aonde **Curso** e **Lesson** fossem entidade totalmente separadas e a gente fosse gerenciar esse relacionamento manualmente aí sim teríamos que ter a **coluna de id** declarada em **Lesson** como mostramos abaixo:

- ```

@Data
@Entity
public class Lesson {

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

    @Column(length = 100, nullable = false)
    private String name;

    @Column(length = 11, nullable = false)
    private String youtubeUrl;

    private String course_id;
}

```

Porém no nosso caso estamos utilizando a anotação **@OneToMany** a gente não precisa fazer isso.

Mas essa não é a melhor maneira de fazermos isso é a mais simples porém não é a alternativa que tem a melhor performance, abaixo irei falar sobre a melhor maneira de fazermos o relacionamento **OneToMany**.

Temos que nos preocupar além do código com o nosso banco de dados, em como é que essas queries estão sendo executadas no banco de dados. Olhando o nosso **LOG** vemos que estamos inserindo na tabela de **Curso** e de **Lesson** e acabamos criando elas e depois dando um **update** na tabela de **Lesson** só pra setar o **Curso_id**.

- ```

Hibernate: insert into course (category, name, status, id) values (?, ?, ?, ?)
Hibernate: insert into lesson (name, youtube_url, id) values (?, ?, ?)
Hibernate: update lesson set course_id=? where id=?

```

Isso ocorre por causa que estamos usando o **@JoinColumn** que acabou fazendo o **update** em **Lesson** do registro **Course\_id**, quando usamos isso em produção no nosso caso pode ser que não tenhamos uma perda de performance muito grande, pois não iremos trabalhar com muitos registros, se formos considerar apenas o caso de uso que esse back end seria para um projeto pequeno não precisaríamos nos preocupar com isso, mas **se for**



um projeto que tenha muitos dados e é um projeto enorme com toda certeza iríamos notar a perda de performance.

Existe um problema bem comum na computação que quando começamos a treinar algoritmos para competições de programação o primeiro problema que resolvemos é o **N+1**, ele é um problema computacional que também está presente na parte de **persistência** do **Java** não apenas no **Hibernate** mas em muitas ferramentas de **ORM** que é mapear um **objeto relacional** como é o caso de nossos objetos que estão representando tabelas no **banco de dados**. Esse problema acontece muito quando temos um relacionamento **unidirecional** do **One to many** que no caso o relacionamento aqui do nosso exemplo é apenas de curso para **Lesson**, não conseguimos através de uma **Lesson** obter a outra parte do relacionamento, ou seja através de uma **Lesson** obter o curso, no artigo do **N+1** destacado aqui ele fala como fazemos um relacionamento **bidirecional** utilizando **One to many**, no caso é bem semelhante ao método antes feito como veremos abaixo.

A principal diferença é que não teremos mais a anotação **@JoinColumn** e tentamos ao máximo não usa-la.

- ```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
// @JoinColumn(name = "course_id")
private List<Lesson> lessons = new ArrayList<>();
```

E falaremos que esse relacionamento na verdade é mapeado também pela classe **Course** para indicar isso colocamos no relacionamento de **Course** a propriedade **mappedby**.

- ```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, mappedBy = "course")
private List<Lesson> lessons = new ArrayList<>();
```

Ou seja a parte que é dona desse relacionamento é a classe **Course** e na classe de **Lesson** mapearemos o **Course** também **declarando uma variável de Course dentro de Lesson**.

- ```
private Course course;
```

E como estamos fazendo relacionamento de **One to Many** (Um para muitos) Em **Lesson** na variável de **Course** o relacionamento vai ser de **Many to One** (Muitos para um), ou seja temos várias aulas que pertence somente a um **course**, também adicionaremos algumas opções: **Fetch** que usando ela falamos para o **Hibernate** como a gente quer que esse relacionamento seja feito no nosso caso colocaremos como **Lazy** que faz com que somente quando chamarmos o **.getCourse()** dessa **Lesson** é que vai carregar esse

mapeamento, a segunda opção é a **optional** que colocaremos com valor de falso que faz com que essa coluna de **Course_id** é uma coluna obrigatória (sempre vai ter que estar populada).

- ```
@ManyToOne(fetch = FetchType.LAZY, optional = false)
private Course course;
```

Outra coisa importante de colocarmos é o **@JoinColumn** que vai estar agora em **Lesson** ao invés de estar na classe de **Course**, pois como temos aqui um objeto esse objeto vai ser traduzido em uma coluna do banco de dados, então daremos um nome para essa coluna (**course\_id** seguindo os padrões de boas práticas que colocamos a entidade ou tabela falando depois do "\_" falando o atributo) e não aceitando que pode ser vazia.

- ```
@ManyToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "course_id", nullable = false)
private Course course;
```

Se rodarmos o projeto assim ele dará um erro, pois nosso id de **Course** está nulo, então daremos valor a ele, mas como é o objeto inteiro passaremos não somente o **id** e sim o **Course** todo para o método **setCourse()** de **Lesson**.

- ```
l.setCourse(c);
```

**Mas podemos estar nos perguntando como passamos o objeto inteiro como pegaremos o id?** é justamente por isso que setamos o objeto, no **LOG** mostra o **SQL** que foi executado quando rodamos o projeto e nele vemos que foi inserido o **Course** no banco de dados e depois **Lesson** (Sempre começa com o objeto principal do relacionamento no caso o **Course**).

```
Hibernate: insert into course (category, name, status, id) values (?, ?, ?, ?)
Hibernate: insert into lesson (name, youtube_url, id) values (?, ?, ?)
```

Depois que o **Course** é inserido no banco de dados teremos o **identificador** desse **Course** então o objeto passado no **setCourse()** de **Lesson** tem um **id** já, lembrando que estamos trabalhando com objeto e o mesmo é uma referência, ou seja, quando instanciamos um **Course** ele seta um **id** e o mesmo terá ele quando formos passar para **Lesson**, então quando o **Hibernate** ou **JPA** tenta fazer o **insert** já vai fazer como **id** de **Course**, com isso temos apenas **2 inserts**, ou seja 1 vez a menos acessando o banco de dados comparado a antes e com isso ganhamos muita performance dependendo do nível da nossa aplicação, pois tudo feito no banco de dados tem um custo computacional.

- jdbc:h2:mem:devdb
  - + COURSE
  - + LESSON
  - + INFORMATION\_SCHEMA
  - + Sequences
  - + Users
  - i H2 2.1.214 (2022-06-13)

Run
 Run Selected
 Auto complete
 Clear
 SQL statement:

SELECT \* FROM LESSON|

SELECT \* FROM LESSON;
 

| ID | NAME       | YOUTUBE_URL | COURSE_ID |
|----|------------|-------------|-----------|
| 1  | Astronomia | Acadovski   | 1         |

 (1 row, 2 ms)

Edit