ESTUDO DE SPRING BOOT

O spring, assim como a maioria dos frameworks utiliza a arquitetura MVC para desenvolvimento;

MODEL

faz jus às tabelas do banco de dados, que podem ser escritas como:

```
@Entity
@Table(name = "nomeTabela")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@ToString
public class Tabela{
      @Id
      @GenratedValue(strategy = GenerationType.IDENTITY)
      private int IDTabela;

      @Column(name = "nomeColuna1")
      private String nomeGenerico;

      @Column(name = "nomeColuna2")
      private int valorGenerico

}
```

"E no caso de relações entre tabelas? 1:1; 1:N; N;N"

- No caso a anotação OneToMany é feita do lado de quem "envia" a chave estrangeira para a outra tabela;
- Também pode ser usado de forma bidirecional `@ManyToOne` do lado da tabela que recebe a chave, de modo a manter consistência nas consultas para o JPA
- OneToOne segue a mesma lógica, mas é utilizado apenas [private Tabela tabela] para referenciar, e não private List<Tabela> tabela;

- Observemos o **`nullable=false`** dentro da anotação **`@JoinColumn`** para deixar claro que a chave estrangeira é não-nula,reforçando a obrigatoriedade do relacionamento.

Caso one-to-many

Suponhamos que um aluno possa cursar **N** cursos, mas um curso só pode ser frequentado por **UM** aluno (relacionamento **obrigatório**)

```
@Entity
@Table(name = "estudante")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class Estudante{
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      @Column(name="idAluno")
      private Long idAluno;

      @Column(name="nomeAluno", nullable=false)
      private String nomeAluno;

      @OneToMany(cascade = CascadeType.ALL, mappedBy = “aluno”)
      private List<Curso> curso;

}

@Entity
@Table(name="curso")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class Curso{
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      @Column(name="idCurso")
      private Long idCurso;

      @Column(nullable=false, unique=true)
      private String nomeCurso;
```

```
	@ManyToOne
	@JoinColumn(name = “id_aluno”, nullable=false)
	private Aluno aluno;

	//Perceba que também incluímos @ManyToOne para manter a consistência
}
```

Caso one-to-one

Imaginemos uma situação de um funcionário tem apenas **UM** projeto e no projeto só pode trabalhar **UM** funcionário:

```
@Entity
@Table(name = “funcionario”)
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class Funcionario{
	@Id
	private String CpfFuncionario;

	private String nomeFunc;

	private Float salarioFunc;

	@OneToOne
	@JoinColumn(name = “projeto_id”)
	private Projeto projeto;
}
```

```
//classe do tipo enum em outro pacote, mas que é referenciado por projeto
public enum DificuldadeProjeto{
	FACIL, INTERMEDIARIO, DIFICIL;
}
```

```
@Entity
@Table(name = “projeto”)
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class Projeto{
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      private Long idProjeto;

      private String nomeProjeto;

      @Enumerated(EnumType.STRING)
      private DificuldadeProjeto dificuldadeProjeto;

      private Date dataSubmitProjeto;

      @OneToOne
      @JoinColumn(name = “func_id”)
      private Funcionario func;
}
```

Caso many-to-many

Agora imaginemos que um artigo possa ser escrito por **N** pesquisadores, e um autor possa redigir **N** artigos

```
@Entity
@Table(name = “autor_tb”)
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Autor{
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      private Long idAutor;
      private String nomeAutor;
      private String instutuicaoAutor;
      private String grauAcadAutor;

      @ManyToMany(mappedBy = “autores”)
      private List<Artigo> artigos;
```

```
}

@Entity
@Table(name = “artigo_tb”)
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class Artigo{
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      private Long idArtigo;
      private String tituloArtigo;

      @Lob
      private String descricaoArtigo;
      private Integer numPagsArtigo;
      private String generoArtigo;

      @ManyToMany
      @JoinTable(
            joinColumns = @JoinColumn(name = “artigo_id”)
            inverseJoinColumns = @JoinColumn(name = “autor_id”)
      )
      private List<Autor> autores;
}
```

Caso many-to-many (com tabela associativa que apresenta atributos)

Agora imaginemos que um artigo possa ser escrito por **N** pesquisadores, e um autor (pesquisador) possa redigir **N** artigos, de maneira similar a apresentada. Contudo, na tabela associativa do relacionamento há o atributo data_de_submissão, que faz jus à data em que o artigo foi enviado à revisão acadêmica (e não é a data de publicação final do artigo). Como resolver isso?

Isso pode ser resolvido **criando uma nova classe (tabela) no pacote do model**, que terá relacionamento de **Many-to-one** com as tabelas das quais recebe as chaves estrangeiras. Além disso, é utilizado um **Embedded ID** para juntar as 2 chaves estrangeiras como **uma única chave primária composta**, sobrescrevendo na nova tabela seu hashCode() e seu método equals() para fins de comparação.

Importância da anotação `@Valid` junto ao `@RequestBody` e como adicionar restrições extras ao Model

Para **verificar** se as **restrições** de campo no meu Model foram cumpridas, podemos adicionar a anotação `@Valid` após o `@RequestBody`. Deste modo o Spring analisa se as restrições impostas no Model estão sendo respeitadas e decide se o método do controller deve prosseguir e se o JPA pode registrá-los no BD.

Exemplo:

```
@Entity
@Table(name = “tb_produto”)
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class Produto{

	@Id
	@GeneratedValue(strategy = GenerationType.UUID)
	@Column(name = “id_produto”)
	private UUID idProduto;

	@Column(name = “descricao_produto”, nullable=false)
	@Lob
	@Size(min = 50, max = 500)
	@NotBlank
	private String descricaoProduto;

	@Column(name = “nome_produto”, nullable=false, unique=true)
	@NotNull
	@NotBlank
	@Size(min = 1, max = 40)
	private String nomeProduto;

	@Column(name=”preco_produto”, nullable=false)
	@NotNull
	@DecimalMin(value = 10.0)
	@DecimalMax(value = 15000.0)
	private Double precoProduto;

	@Column(name=”secao_produto”, nullable=true)
```

```
	@Size(max = 30)
	private String secaoProduto;

	@Column(name=”peso_produto”, nullable=false)
	@DecimalMin(value = 100)
	@NotNull
	private Double pesoProduto;
}

public ResponseEntity<Produto> salvaNovaProduto(@RequestBody @Valid
Produto novoProduto){
	try{
		Produto objProduto = produtoRepository.save(novoProduto);
		return new ResponseEntity<>(objProduto, HttpStatus.OK);
	}
	catch(Exception e){
		return new
ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
	}
}
```

No exemplo acima o **`@Valid`** verifica se o que foi passado pelo **`@RequestBody`** **cumpre** os requisitos de **restrições (@Size, @Max, @DecimalMax, @NotNull)** de tamanho e não-nulo etc.

Se houver algum campo que viole pelo menos uma das restrições, o **Spring lança uma exceção** do tipo **MethodArgumentNotValidException** que automaticamente **retorna um BAD REQUEST 400** para a API e evita que o método do controller seja executado e, por conseguinte, que o JPA persista os dados, o que assegura a integridade do BD.

É recomendado que se **adicione a anotação** **`@Valid`** também nos métodos do **Service** e inclua o **`@Validated`** logo abaixo da anotação **`@Service`** para melhor funcionamento da validação.

CAMADAS INTERMEDIÁRIAS

Podemos ter camadas intermediárias entre **Model** e **View**, **Model** e **controller**, **View** e **controller**. As camadas mais utilizadas são:

CAMADA SERVICE

Classe que extende o repositor para a criação de métodos personalizados de acordo com as lógicas de negócio do projeto, para não sobrecarregar o repositório;

```
@Service
public class tabelaService{

	@Autowired
	private TabelaRepo tabelaRepo;

	@Autowired
	private TabelaMapper tabelaMapper; //opcional

	public List<TabelaDTO> retornarTodasTuplas(){
		List<Tabela> tuplasBd = tabelaRepo.findall();
		list<TabelaDTO>  dto  =  tuplasBd.stream().map(v  ->  new
TabelaDTO(v)).toList();
		return dto;

	}

	//Usando o mapper (BEM MAIS PRÁTICO!)
	public List<tabelDTO> retornarTodasTuplasDTO(){
		Lista<Tabela> tuplasBd = tabelaRepo.findAll();
		return tabelaMapper.toTabelaDTOList(tuplasBd);

	}
}
```

CAMADA DTO

Classe que serve como **cópia** e "**camufla**" o Model de modo que podemos, por exemplo, dizer quais atributos devem ser mostrados pela API, **protegendo** assim os **dados sensíveis** de brechas e não mexendo diretamente na camada Model.

```
@Getter
@NoArgsConstructor
public class tabelaDTO{
	private String atr1;
	private Integer atr2;
	private Double atr3;

	//Quando se utiliza mapper não é necessário ter esse construtor, pois o mapper já faz o mapeamento!
	public tabelaDTO(Tabela entidade){
		atr1 = entidade.getAtr1();
		atr2 = entidade.getAtr2();
		atr3 = entidade.getAtr3();

	}
}
```

MAS TAMBÉM É PRECISO FAZER A CONVERSÃO DO MODEL PARA DTO!

MAPPER

Uma boa prática é usar `@Mapper(componentModel = "spring")`, pois ele mapeia de forma automática e auto atualiza-se se há mudanças tanto na camada DTO quanto na camada Model

```
@Mapper(componentModel = "spring")
public interface EntidadeMapper{
	TabelaDTO toTabelaDTO(Tabela tabela); //método de mapeamento para único registro | Tabela => TabelaDTO
	List<TabelaSTO> toTabelaDTOList(List<Tabela> tabela); //método para mapeamento de listas | List<Tabela> => List<TabelaDTO>
	Tabela toTabela(TabelaDTO tabelaDTO); //TabelaDTO => Tabela
	List<Tabela>	toTabelaList(List<TabelaDTO>	tabelaDTO); //List<TabelaDTO> => List<Tabela>

}
```

Como usar?

1. primeiro injetamos a interface na camada `Service` com `@Autowired`, assim como é feito com o `Repository` injetado no `Service`

2. Depois escrevemos os métodos utilizando os **métodos padrões do repositório**, mas transformamos o retorno dos **objetos para objetos do tipo `List<TabelaDTO>` ou `TabelaDTO`**, de modo que tais métodos recebam como tipo `List<TabelaDTO>` ou `TabelaDTO` também.

- para optional<>: `Optional<Tabela> tabela = tabelaRepo.findById(id) ==> TabelaDTO tabDTO = tabela.map(tabelaMapper::toTabelaDTO);`

- para List<>: `List<tabela> entidade = tabelaRepo.findAll() ==> List<TabelaDTO> listTabDTO = tabelaMapper.toTabelaDTOList(entidade);`

- para tipo Tabela: `Tabela entity1 ==> tabelaMapper.toTabelaDTO(entity1);`

CONTROLLER

Parte do código que **controla os Endpoints** e as **ações** que devem ser executadas:

Perceba que o **repositório** e **projection** ficam em **classes diferentes** do controller, mas ele é necessário para fazer o **intermédio** entre as **tabelas** do BD e os **métodos** do controller;

```
@Repository
public interface TabelaRepo extends JpaRepository<Tabela, Integer>{

	//Tabela é o nome da tabela, ou nome da classe. No caso poderia ser Estudante;
	//Integer faz referência ao tipo do ID utilizado na tabela, neste caso, como foi usado private int IDTabela, usamos Integer. Se fosse private Long IDTabela, teríamos JpaRepository<Tabela, Long>.

	//No repo é possível criar métodos personalizados usando SQL ou JPQL, o que pode ser útil em casos específicos de consulta
```

```
	//Note que usamos uma interface de projeção que pega apenas os campos que precisamos na consulta e agiliza o processo;

	@Query("SELECT * FROM table1 WHERE value>= :valorVariavel AND name= :nomeVariavel", nativeQuery=true)
	public TabelaProjectionInterface encontraTuplaValorNome(@Param("valorVariavel") Double valorVariavel, @Param("nomeVariavel") String nomeVariavel);
}
```

OBS: A interface projections deve estar num pacote separado, assim como repository, service, dto, controller têm seus próprios pacotes

```
public interface TabelaProjectionInterface{
	//Deve ser condizente com os tipos e getters do DTO
	TipoVar1 getVar1();
	TipoVar2 getVar2();
	TipoVar3 getVar3();
}

@RestController
public class TabelaController{

	//repo autowired para ser usado no controller
	@Autowired
	private TabelaRepo tabelaRepo;

	//recuperar todas as tuplas de uma tabela
	@GetMapping("/rotaGenerica")
	public ResponseEntity<List<Tabela>> todosValoresTabela(){

		//guarda em uma lista todos os valores de tuplas
		List<Tabela> listaTabela = tabelaRepo.findAll();

		//verifica se a lista não está vazia
		if(listaTabela.isEmpty()){
			return new ResponseEntity<>(HttpStatus.NO_CONTENT);
		}

		return new ResponseEntity<>(listaTabela, HttpStatus.OK);

	}
```

```
	//recuperar pelo id;
	//É importante notar o @PathVariable que serve como suporte
para o framework
	//entender como recuperar uma variável passada pelo URL.
	@GetMapping("/rotaGenerica2/{id}")
	public ResponseEntity<Tabela> recuperaTuplaPorId(@PathVariable
Integer id){

		//note que usamos o Optional<> para evitar caso o id não
exista no BD
		//com Optional<> é possível verificar se há ocorrência e
então apresentá-la
		Optional<Tabela> dadoTabela = tabelaRepo.findById(id);

		if(dadoTabela.isPresent()){
			return new ResponseEntity<>(dadoTabela.get(),
HttpStatus.OK);
		}

		return new ResponseEntity<>(HttpStatus.NOT_FOUND);
	}

	//adicionar tupla no BD;
	//NOTEMOS o @RequestBody utilizado no parâmetro que sinaliza
como o framework
	//deve lidar com o tipo do model, no caso o tipo Tabela, para
assim transformar
	//em objetos java e salvar no BD.
	@PostMapping("/addTupla")
	public ResponseEntity<Tabela> adicionaTupla(@RequestBody
Tabela tabela){

		//método save() do tabela repo retorna um objeto que é
guardado por
		//uma variável do tipo tabela
		Tabela objTabela = tabelaRepo.save(tabela);
		return new ResponseEntity<>(objTabela, HttpStatus.OK);

	}

	@PutMapping("/mudaTuplaPorID/{ID}")
	public ResponseEntity<Tabela> atualizaTupla(@PathVariable
Integer ID, @RequestBody Tabela tuplaNova){
```

```
            Optional<Tabela> tuplaAntiga = tabelaRepo.findById(ID);
            if(tuplaAntiga.isPresent()){
                  Tabela tuplaAtualizada = tuplaAntiga.get();


tuplaAtualizada.setAtributo1(tuplaNova.getAtributo1());

tuplaAtualizada.setAtributo2(tuplaNova.getAtributo2());

                  Tabela objTab = tabelaRepo.save(tuplaAtualizada);

                  return new ResponseEntity<>(objTab, HttpStatus.OK);
            }

            return new ResponseEntity<>(HttpStatus.NOT_FOUND);

      }

}

      @DeleteMapping("/deletarTupla/{identificador}")
      public ResponseEntity<Void> deletaTuplsPorId(@PathVariable
Integer identificador){

            try{
                  Optional<Tabela> tabelaBD =
tabelaRepo.findById(identificador);
                  if(tabelaBD.isPresent()){
                        Tabela retTabela = objTabela.get();
                        Tabela objTabela =
tabelaRepo.delete(retTabela);
                        return new ResponseEntity<>(HttpStatus.OK);


                  }

                  return new ResponseEntity<>(HttpStatus.NOT_FOUND);
            }
            catch(Exception ex){
                  return new
ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
            }
```

```
}
```

EXPLICAÇÃO DE CADA MÉTODO POR PASSOS

@GetMappping("/rota1")
public ResponseEntity<List<Tabela>> todosOsValoresTabela() = retorna todas as tuplas de uma tabela
Como é feito? Utilizando o método findAll() do repositório, e em uma lista List<Tabela> tb guardamos esses valores e os retornamos à API com ResponseEntity<>(tb, ...)

@GetMapping("/rota2/{Id}")
public ResponseEntity<Tabela> recuperaTuplaPorId(@PathVariable Long Ident) = retorna apenas uma ocorrência correspondente ao Id passado na URL.
Como é feito? Utilizamos novamente o repositório, mas nesse caso é utilizado o método findById(Ident), lembrando que o método retorna um objeto Optional<>, por isso deve-se usar Optional<Tabela> para guardar valor e passar para o ResponseEntity como ResponseEntity<>(dadoTab.get(), HttpStatus.OK);

@PostMapping("/rota3")
public ResponseEntity<Tabela> salvaNovaTupla(@RequestBody Tabela tab)
Como é feito? utilizamos o método save(tab) e guardando num objeto Tabela para retornar com Response Entity, retornando o objeto Tabela obj = tabelaRepo.save(tab) como ResponseEntity<>(obj, HttpStatus.OK)

@PutMapping("/rota4/{identificador}")
public ResponseEntity<Tabela> atualizaTupla(@RequestBody Tabela tb, @PathVariable Long identificador)
como é feito? primeiro recuperamos o registro lá da api usando o identificador e salvando isso num Optional<Tabela> varObj; logo em seguida pegamos o corpo da tabela armazenando em outra variável usando .get() [Tabela retTabela = varObj.get()] e finalmente, usando o que foi enviado e tratado pelo @RequestBody, tb neste caso, utilizamos setters em retTabela com os getters do Tabela tb retornado pelo @RequestBody. Após isso, utilizamos o método .save(retTabela) armazenando em uma outra variável e depois retornando com ResponseEntity

OBS:
é possível usar menos variáveis nos métodos para não tornar cansativa a leitura do código, mas também não utilizar variáveis de menos a ponto de ser ilegível

CONEXÃO COM MYSQL

```
spring.application.name=springtest

spring.datasource.url=jdbc:mysql://localhost:3306/db_test_spri
ng1
spring.datasource.username=root
spring.datasource.password=

spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl.auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.
MySQLDialect
```

CONFIGURAÇÃO DE PLUGIN NO *pom.xml* PARA FUNCIONAMENTO DO MAPSTRUCT

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.8.1</version>
   <configuration>
      <source>1.8</source>
      <target>1.8</target>
      <annotationProcessorPaths>
         <!--Project Lombok compile preprocessor-->
         <path>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>${lombok.version}</version>
         </path>
         <!--Maps Struct compile preprocessor-->
         <path>
            <groupId>org.mapstruct</groupId>
            <artifactId>mapstruct-processor</artifactId>
```

```
            <version>${org.mapstruct.version}</version>
        </path>
    </annotationProcessorPaths>
  </configuration>
</plugin>
```

Como fazer @Override no equals()
[Overriding equals method in Java - GeeksforGeeks]