**Anotações!!!**

**Sobre Camadas**

*Exemplificação!*

É gerada uma requisão (Get para buscar.... Post para enviar) através do front... essa **requisição** é feita para a **Camada de Controle** (Rest), que por sua vez aciona a **camada de serviço**.

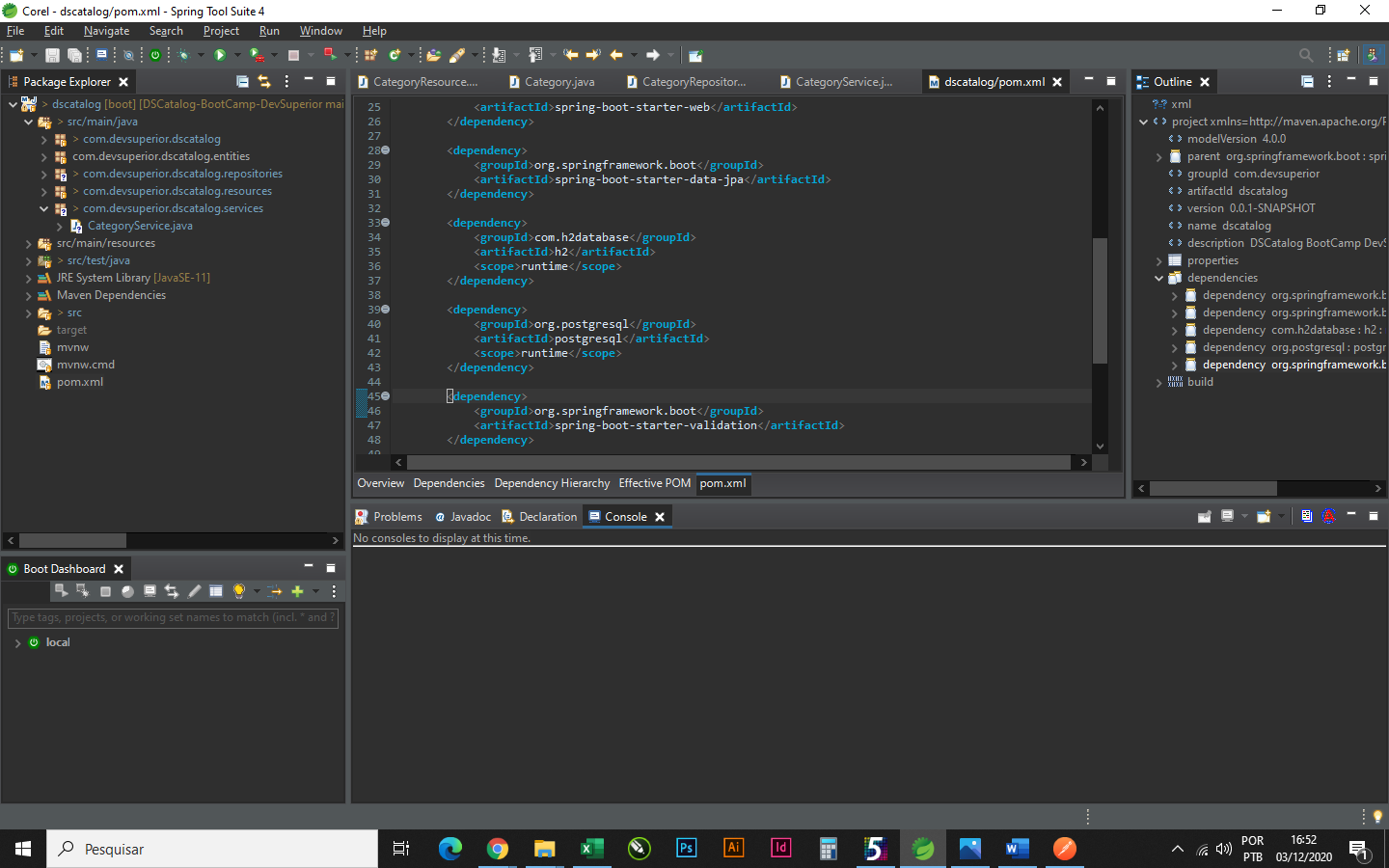
Na camada de serviços é acionado o a camada de acesso a dados, que irá retornar com o resultado da requisição para a camada de serviços, que por sua vez irá responder para a camada de Controle.

Requisição Controle(Rest/Api) Serviço Dados

**JPA**

É um recurso utilizado no java para fazer entre outras coisas o relacionamento da aplicação com o banco de dados, é responsável por “transformar” essas informações em objetos.

Necessário fazer a implementação das dependências no arquivo de configuração do maven, **pom.xml**



***Recursos a considerar do JPA:***

* Inserir as dependências no arquivo de configurações gerais do Spring Boot... POM.XML, buscar na web a versão mais atual do Maven JPA, e utilizar como dependência;
* Método: **ResponseEntity**; irá retornar um objeto com os dados.

**CLASSES RESOURCES**

**CAMADA DE CONTROLADORES**

TODA **CLASSE RESOURCE** SERÁ RESPONSÁVEL POR DISPONIBILIZAR **END POINT**!

**O Que é End-Point ?**

É o recurso que o front-web irá buscar na aplicação!

**Como preparar uma classe “End-Point”**

Acrescentar as annotatios na identificação da classe:

**@RestController**

*Essa annotation será responsável por fazer um pré-compilamento, informando que este será um recurso da classe.*

**@RequestMapping(value=””)**

*Esta annotation é responsável por indicar o caminho a qual o front-web irá chamar. Exemplo: (@RequestMappint(value=”/produtos”) no front-web será chamando pelo endereçoAplicação****/produtos****).*

***Portanto este será o end-point : /produtos***

**@GetMapping**

*Essa annotation informa ao STS que o método a qual ela está sobrescrita irá retornar um recurso para o end-point.*

**CLASSES CATEGORIES**

**CAMADA ENTIDADES**

É classe com os atributos referentes à Categoria (por exemplo).

No inicio da classe acrescenta-se a annotation:

***@Entity***

*Essa anotação vem do pacote* ***javax.persinstence.entity***, esse pacote vem da JPA.

Abaixo da annotation ***@Entity***, acrescente-se a annotation ***@Table***, ela será responsável por criar uma tabela no banco de dados, com o nome que for especificado.

***@Table (name = “tb\_category”)***

Serão utilizadas outras annotations para mostrar ao JPA qual será o campo referente a chave primária (***@Id***) e se ele será auto incrementável (**@GenerateValue(stategy=GenerationType.IDENTIFY**)

Exemplo:

*@Entity*

*@Table*(name= "tb\_category")

public class Category implements Serializable {

private static final long ***serialVersionUID*** = 1L;

*@Id*

*@GeneratedValue*(strategy=*GenerationType*.***IDENTITY***)

private Long id;

private String name;

**CLASSES REPOSITORIES**

**CAMADA ACESSO A DADOS**

Será a classe responsável por conectar os serviços ao banco de dados, irá utilizar recursos do JPA.

Uma classe/“**interface**” que irá **extender** a interface **JpaRepository**.

É um tipo genérico, necessitando identificar os atributos, nome da classe e o atributo do id.

***@Repository***

public interface CategoryRepository **extends JpaRepository<Category, Long**> {

}

É necessário identificar como um componente injetável, para isso utiliza a annotation

**@Repository**.(atenção para a semântica)

**@Repository**

*Irá especificar que a classe/interface é um* ***componente****.*

**CLASSES SERVICES**

**CAMADA DE SERVIÇOS**

A **Camada de serviço** é a responsável por fazer toda parte de serviços, e regras de negócio do sistema, ela faz a **ponte** entre os **Controladores**, e o **Acesso aos Dados**.

No ínicio da classe é adicio a annotatio:

**@Service**

*Ela vai registrar a classe como um componente que irá participar do sistema de injeção de dependências automatizadas do spring, quer dizer que quem vai gerenciar as dependecias será o maven.*

*Utiliza essa annotatio para manter a semântica em relação a camada da classe.*

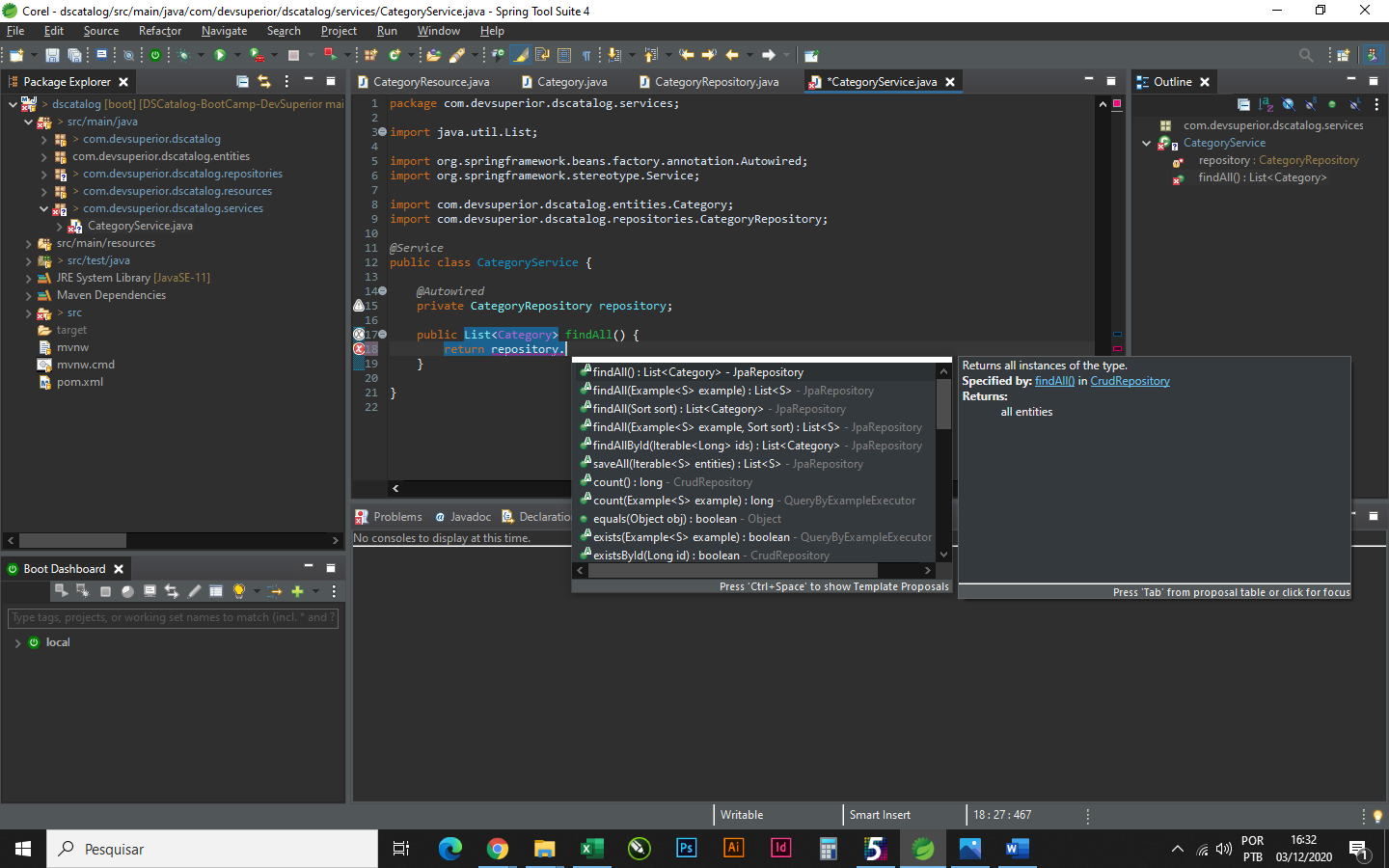
**@AutoWired**  
para que o servisse possa injetar uma instância gerenciada pelo Spring. Ele vai injetar uma dependência válida a qual ela foi instanciada.

Exemplo:

*@Autowired*

**private** CategoryRepository repository;

Com o uso da injeção de dependências, é possível utilizar vários métodos que já estão prontos, economizando escrita de códigos.



Para garantir que o método na camada de serviço, que irá tratar alguma relação com o banco de dados usa-se a annotation ***@Transactional****,* essa anotação irá garantir que toda a transação de dados seja efetuada, para garantir a integridade da transação. O próprio framework vai garantir para que a transação seja efetuada.

**@Transactionl é da biblioteca do spring!**

É possível passar instruções para a annotation, readOnly=true, vai evitar que o método trave o banco de dados. Evita o Lock! Degrada a performance, toda operação de leitura sempre colocar!

*@Transactional*(readOnly = true)

public List<Category> findAll() {

return repository.findAll();

}

**BANCO DE DADOS H2**

O **H2**, é um banco de dados sql que roda **em memória**, é utilizado para testes no **spring**, no desenvolvimento de uma aplicação.

Deve-ser acrescentar a **dependência** no **pom.xml**.

<**dependency**>

<**groupId**>com.h2database</**groupId**>

<**artifactId**>h2</**artifactId**>

<**scope**>runtime</**scope**>

</**dependency**>

Importante, deve-se acrescentar um arquivo de configuração, a pasta **src/main/resources**

*Application.properties*

spring.profiles.active=test

spring.jpa.open-in-view=false

(esta configuração irá encerrar todas as transações referentes a banco de dados, antes que chegue no controlador rest)

*Application-teste.properties*

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.username=sa

spring.datasource.password=

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

**SEEDING DE DADOS NO H2**

É utilizado para criar uma carga de dados no banco de dados h2, para desenvolvimento e testes da aplicação.

Cria-se um **arquivo.sql** na pasta scr/main/resources.

Insere as informações em sql no banco de dados, toda vez que a aplicação for startada, será feito a inserção dos dados.

INSERT INTO tb\_category (name) VALUES ('Books');

INSERT INTO tb\_category (name) VALUES ('Electronics');

INSERT INTO tb\_category (name) VALUES ('Computer');

toda vez que a aplicação for startada

**DTO**

Data Transfer Object

O objeto DTO será responsável por transferir os dados da entidade para a camada rest, ele é usado para se adequar ao modelo de arquitetura. **O DTO** é que será transmitido da camada de **serviço** para a **camada do controlador**. É UM OBJETO PARA CARREGAR OS DADOS

Algumas vantagens:

* Controlar quais dados serão jogados para o controlador que a API irá disponibilizar, pode-se projetar os dados, caso tenham muitos dados. Projetando apenas os dados necessários. Por exemplo, quer retornar os dados de um cliente cadastrado, porém não quer retornar todos os dados, fazendo um filtro.
* Pode-se customizar o que se quer entregar para a aplicação; por exemplo: entregar o cadastro de um cliente, junto com os pedidos que esse cliente tenha feito.

A classe DTO, é praticamente idêntica a classe referência da Entitidade (exemplo: classe Category e classe CategoryDTO).

A diferença entre as classes, é que é criado **também um construtor** recebendo a entidade, por comodidade.

Exemplo:

public CategoryDTO(Category entity) {

this.id = entity.getId();

this.name = entity.getName();

depois de criado a classe DTO, ela é que irá retornar na classe de serviço:

classe de serviços: CategoryService.java **(antes)**

*@Transactional*(readOnly = true)

public List<Category> findAll() {

return repository.findAll();

}

classe de serviços: CategoryService.java **(depois)**

*@Transactional*(readOnly = true)

public List<CategoryDTO> findAll() {

List<Category> list = repository.findAll();

List<CategoryDTO> listDTO = new ArrayList<>();

for (Category cat : list) {

listDTO.add(new CategoryDTO(cat));

}

return listDTO;

}

Pode ser substituído também por uma expressão lambda, deixando o código mais curto.

**.stream()** vai converter a coleção para uma stream (que é um recurso que permite trabalhar com funções de alta ordem, programação funcional)

**.map()** transforma cada elemento original em outra coisa, aplicando uma função a cada elemento.

.map(x para cada elemento x eu vou levar para outro elemento)

.map(x -> new categoryDTO(x)

.**collect()¸**é uma função que irá transformar a stream par ao objeto original.

.collect(Collectors.toList()) vai transformar o stream em uma lista.

Com expressão Lâmbda, o método para retornar o findAll(), fica assim:

*@Transactional*(readOnly = true)

public List<CategoryDTO> findAll() {

List<Category> list = repository.findAll();

**List<CategoryDTO> listDTO = list.stream().map(x -> new**

**CategoryDTO(x)).collect(Collectors.*toList*());**

return listDTO;

}

**TRATAMENTO DE EXCESSÕES**

Classe Resource

Utiliza algumas práticas e técnicas diferentes ao uso do **Try /Catch**, com a intenção de deixar o código mais enxuto, uma boa prática deixar o código no controlador o mais enxuto possível.

***@ControllerAdvice*** *-> responsável por capturar a excessão*

***HttpServletRequest*** *-> objeto que tem as informações da requisição*

***@ExceptionHandler(nomeDaClasse.class)*** *-> vai acima do método que irá retornar ao controlador, no argumento tem que especificar o tipo de exceção que ele deve interceptar. (nomeDaClasse onde foi tratada a exceção.)*

**INSERÇÃO DE NOVO REGISTRO**

Classe Resource

Utiliza algumas práticas e técnicas diferentes ao uso do **Try /Catch**, com a intenção de deixar o código mais enxuto, uma boa prática