

5.1. Implementando consultas JPQL em repositórios

terça-feira, 14 de fevereiro de 2023

11:49

```
@Override
public List<Cozinha> consultarPorNome(String nome) {
    return entityManager.createQuery(qlString: "from Cozinha where nome =:nome", Cozinha.class)
        .setParameter( name: "nome", nome).getResultList();
}
```

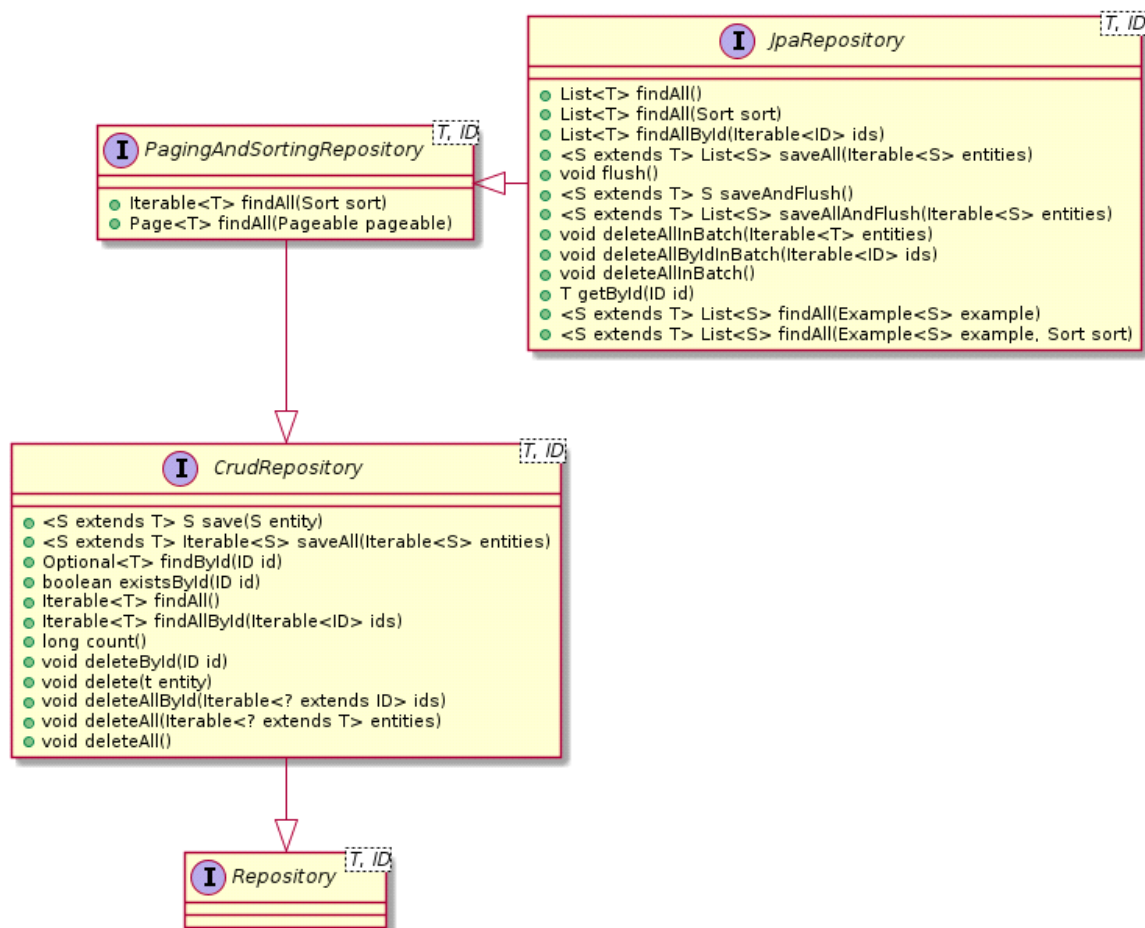
5.2. Conhecendo o projeto Spring Data JPA (SDJ)

terça-feira, 14 de fevereiro de 2023 14:29

O Spring Data JPA é uma framework que faz parte do conjunto de projetos do Spring Data que tem como finalidade tornar a integração de aplicações Spring com a JPA (Java Persistence API), uma de suas principais vantagens é a capacidade que o mesmo possui para criar a camada de acesso aos dados sem a necessidade de termos que implementar manualmente as famosas classes de DAO (Data Access Object).

Assim como o projeto Spring Data possui interfaces que são utilizadas ao longo de todos os seus subprojetos, o Spring Data JPA também possui sua principal interface que geralmente iremos utilizar durante a criação da camada de persistência da nossa aplicação, essa interface é a `JpaRepository`.

Veja no diagrama abaixo quais os métodos definidos pela interface `JpaRepository` e como ela se relaciona com as interfaces do Spring Data:



Cada um deles define sua própria funcionalidade:

- **CrudRepository** fornece funções CRUD
- **PagingAndSortingRepository** fornece métodos para paginação e classificação de registros
- **JpaRepository** fornece métodos relacionados ao JPA, como liberar o contexto de persistência e excluir registros em um lote

dependência via maven:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

Lembrando que não precisamos informar a versão dos starters oficiais do Spring Boot uma vez que o próprio Spring Boot define as versões a serem utilizadas. Outro ponto importante é a inclusão do driver de conexão com o banco de dados, neste exemplo estou utilizando o MySQL.

application.properties:

Configuração do DataSource

```
spring.datasource.url=jdbc:mysql://localhost:3306/conhecendo_spring_data_jpa
spring.datasource.username=root
spring.datasource.password=root
```

Configuração do Hibernate

```
spring.jpa.hibernate.ddl-auto=update
```

Configuração da JPA

```
spring.jpa.show-sql=true
```

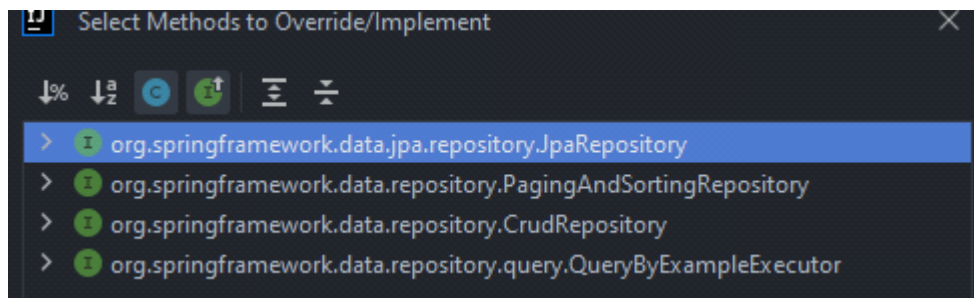
5.3. Criando um repositório com Spring Data JPA (SDJ)

terça-feira, 14 de fevereiro de 2023 14:34

```
@Repository
public interface CozinhaRepository extends JpaRepository<Cozinha, Long> {
    List<Cozinha> consultarPorNome(String nome);
}
```

É necessário anotar a interface que usaremos com `@Repository` e estender a interface `JpaRepository` do `SpringDataJpa`, com os argumentos de tipo `<Entidade, Identificador da entidade>`

o Spring irá criar uma instância da interface em tempo de execução e disponibilizará os métodos de `JpaRepository` e sua hierarquia.



5.4. Refatorando o código do projeto para usar o repositório do SDJ

terça-feira, 14 de fevereiro de 2023 15:53

```
@GetMapping("/{cozinhaId}")
public ResponseEntity<Cozinha> buscar(@PathVariable Long cozinhaId) {
    final Optional<Cozinha> cozinha = cozinhaRepository.findById(cozinhaId);

    return cozinha.
        map(cozinhaFound -> ResponseEntity.ok(cozinhaFound))
        .orElseGet(() -> ResponseEntity.notFound().build());
}
```

Ao buscar uma entidade com `findById`, a instância retorna com um `Optional` encapsulando, podemos utilizar a função `map` para percorrer a única instância dentro do `Optional` cozinha e usar uma classe anônima para consumir o supplier que espera ser recebido no argumento de `map`, a lógica é:
a primeira instância a ser iterada pelo `map` é `cozinhaFound` e logo ela é retornada após a arrow caso realmente exista, caso não exista, a chamada no método `.orElseGet` acontece e retorna uma `ResponseEntity`.

5.5. Desafio refatorando todos os repositórios para usar SDJ

terça-feira, 14 de fevereiro de 2023 20:06

5.6. Criando consultas com query methods

quarta-feira, 15 de fevereiro de 2023 10:58

Podemos declarar alguns métodos para o Spring Data Jpa implementar em tempo de execução, como:

```
@Repository
public interface CozinhaRepository extends JpaRepository<Cozinha, Long> {

    Optional<Cozinha> findByName(String nome);
    List<Cozinha> findTodasByName(String nome);

}
```

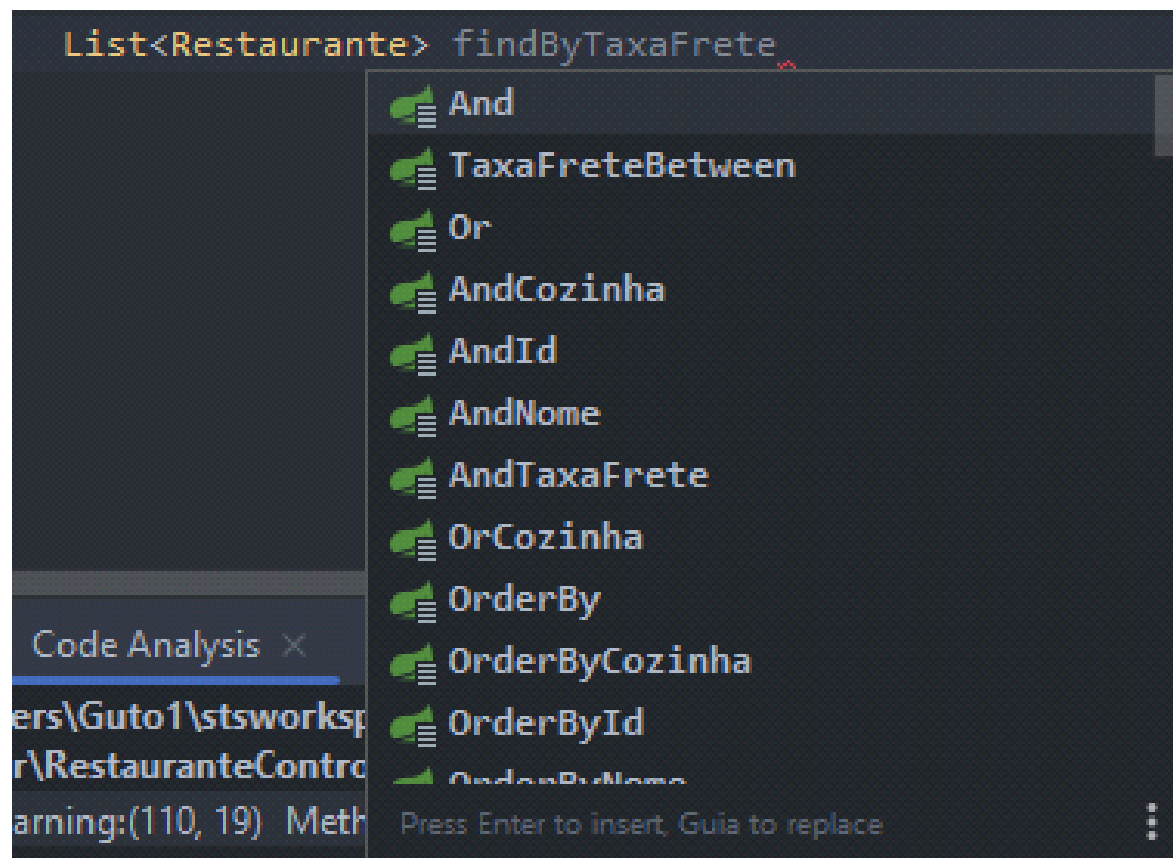
O Spring Data Jpa utiliza o prefixo `findBy` para criar métodos somente pela definição de assinaturas de método, baseado no nome da propriedade que contém na instância tipada de `JpaRepository`

Nesse método, retornam instância por nome exato, pois essa é a forma da implementação.

5.7. Usando as keywords para definir critérios de query methods

quarta-feira, 15 de fevereiro de 2023

11:10



```
@Repository
public interface RestauranteRepository extends JpaRepository<Restaurante, Long> {
    List<Restaurante> findByTaxaFreteBetween(BigDecimal taxaInicial, BigDecimal taxaFinal);
    List<Restaurante> findByNomeAndCozinhaId(String nome, Long cozinha);
}
```

Criação de query methods usando as keywords fornecidas pelo SDJ, o SpringDJ também consegue acessar propriedades dos objetos

5.8. Conhecendo os prefixos de query methods

quarta-feira, 15 de fevereiro de 2023 13:07

```
List<Restaurante> queryByTaxaFreteBetween(BigDecimal taxaInicial, BigDecimal taxaFinal);  
List<Restaurante> findByNomeContainingAndCozinhaId(String nome, Long cozinha);  
Optional<Restaurante> findFirstRestauranteByNomeContaining(String nome);  
List<Restaurante> findTop2ByNomeContaining(String nome);  
int countByCozinhaId(Long cozinha);
```

5.9. Usando queries JPQL customizadas com @Query

quarta-feira, 15 de fevereiro de 2023 13:14

```
@Repository
public interface RestauranteRepository extends JpaRepository<Restaurante, Long> {
    List<Restaurante> findByTaxaFreteBetween(BigDecimal taxaInicial, BigDecimal taxaFinal);
    List<Restaurante> findByNomeAndCozinhaId(String nome, Long cozinha);

    @Query("from Restaurante where nome like %:nome% and Cozinha.id = :id")
    List<Restaurante> consultarPorNome(String nome, @Param("id") Long cozinha);

    Optional<Restaurante> findFirstRestauranteByNomeContaining(String nome);

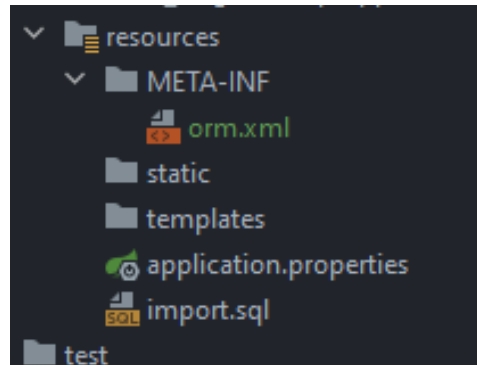
    List<Restaurante> findTop2ByNomeContaining(String nome);

    int countByCozinhaId(Long cozinha);
}
```

Criando consultas personalizadas utilizando @Query

5.10. Externalizando consultas JPQL para um arquivo XML

quarta-feira, 15 de fevereiro de 2023 14:48



```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
  xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/orm_2_2.xsd"
  version="2.2">
  <named-query name="Restaurante.consultarPorNome">
    <query>
      from Restaurante
      where nome
        like concat('%',:nome, '%')
        and cozinha.id = :id</query>
    </named-query>
  </entity-mappings>
```

```
//@Query("from Restaurante where nome like %:nome% and Cozinha.id = :id")
List<Restaurante> consultarPorNome(String nome,@Param("id") Long cozinha);
```

Consultas externas

5.11. Implementando um repositório SDJ customizado

quarta-feira, 15 de fevereiro de 2023 15:31

Para criar um repositório com as próprias consultas JPQL para ser utilizada pelo projeto Spring Data JPA com outras consultas, é necessário criar uma classe com métodos próprios e persistir com o entity manager, além de declarar a assinatura do método na interface implementada por JpaRepository. A classe com os métodos precisa conter o prefixo **Impl** :

```
@Repository
public class RestauranteRepositoryImpl {

    1 usage
    @PersistenceContext
    private EntityManager manager;

    public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {

        var jpql = "from Restaurante where nome like :nome "
            + "and taxaFrete between :taxaInicial and :taxaFinal";

        return manager.createQuery(jpql, Restaurante.class)
            .setParameter( name: "nome", value: "%" + nome + "%")
            .setParameter( name: "taxaInicial", taxaFreteInicial)
            .setParameter( name: "taxaFinal", taxaFreteFinal)
            .getResultList();
    }
}
```

```
0 usages
@Repository
public interface RestauranteRepository extends JpaRepository<Restaurante, Long> {

    List<Restaurante> findByTaxaFreteBetween(BigDecimal taxaInicial, BigDecimal taxaFinal);
    List<Restaurante> findByNomeAndCozinhaId(String nome, Long cozinha);

    // @Query("from Restaurante where nome like %:nome% and Cozinha.id = :id")
    List<Restaurante> consultarPorNome(String nome, @Param("id") Long cozinha);

    Optional<Restaurante> findFirstRestauranteByNomeContaining(String nome);

    List<Restaurante> findTop2ByNomeContaining(String nome);

    int countByCozinhaId(Long cozinha);

    1 usage
    List<Restaurante> find(String nome,
        BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal);
}
```

A assinatura do método declarada no repositório da interface

O método sendo usado

```
@GetMapping("/restaurantes/por-nome-e-frete")
public List<Restaurante> restaurantesPorNomeFrete(String nome,
                                                    BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {
    return restauranteRepository.find(nome, taxaFreteInicial, taxaFreteFinal);
}
```

Para deixar as classes com as assinaturas com um vínculo mais forte e evitar a quebra de código em tempo de execução, podemos criar uma interface com as assinaturas dos métodos do repositório customizado.

```
@Repository
public class RestauranteRepositoryImpl implements RestauranteRepositoryQueries {

    1 usage
    @PersistenceContext
    private EntityManager manager;

    @Override
    public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {
```

Podemos definir uma interface a partir de um método através da própria IDE. Basta selecionar o método e extrair uma interface do método, automaticamente a IDE implementa a interface.

```
@Repository
public interface RestauranteRepository extends JpaRepository<Restaurante, Long>, RestauranteRepositoryQueries {
    List<Restaurante> findByTaxaFreteBetween(BigDecimal taxaInicial, BigDecimal taxaFinal);
    List<Restaurante> findByNomeAndCozinhaId(String nome, Long cozinha);

    // @Query("from Restaurante where nome like %:nome% and Cozinha.id = :id")
    List<Restaurante> consultarPorNome(String nome, @Param("id") Long cozinha);

    Optional<Restaurante> findFirstRestauranteByNomeContaining(String nome);

    List<Restaurante> findTop2ByNomeContaining(String nome);

    int countByCozinhaId(Long cozinha);

    1 usage
    List<Restaurante> find(String nome,
                          BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal);
}
```

Agora o repositório também estende a interface.

5.12. Implementando uma consulta dinâmica com JPQL

quarta-feira, 15 de fevereiro de 2023 15:59

Como a classe que implementa um repositório contém código java, podemos deixar a lógica mais dinâmica. Exemplo:

Formar uma consulta de acordo com os parâmetros da requisição, como, informar ou não nomes, preços, taxa de frete etc.

```
@Repository
public class RestauranteRepositoryImpl implements RestauranteRepositoryQueries {

    1 usage
    @PersistenceContext
    private EntityManager manager;

    1 usage
    @Override
    public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {

        var jpql = "from Restaurante where nome like :nome "
            + "and taxaFrete between :taxaInicial and :taxaFinal";

        return manager.createQuery(jpql, Restaurante.class)
            .setParameter( name: "nome", value: "%" + nome + "%")
            .setParameter( name: "taxaInicial", taxaFreteInicial)
            .setParameter( name: "taxaFinal", taxaFreteFinal)
            .getResultList();
    }
}
```

Query :

```
var jpql = "from Restaurante where nome like :nome "
    + "and taxaFrete between :taxaInicial and :taxaFinal";
```

O **nome** poderá vir nulo, como **taxaInicial** e **taxaFinal** também, podemos fazer concatenação de String para formar uma query dinâmica de acordo com os valores que podem ser nulos ou não, uma dica é utilizar o `StringBuilder` para concatenação de String, pois facilita a legibilidade do código.

```
@Override
public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal)

    var jpql2 = new StringBuilder();
    jpql2.append("From Restaurante ");

    if(nome != null){
        jpql2.append(("where like = :nome"));
    }

    if(taxaFreteInicial != null){
        jpql2.append("taxaFrete >= :taxaFreteInicial");
    }
```

O código contém um erro de lógica, pois a query é composta assim `from Restaurante where like % nome% and taxaFrete >= taxaFreteInicial` e caso na requisição o nome vir nulo, a cláusula `where` não vai ser adicionada na query. Um jeito de burlar isso é sempre deixado `true` a cláusula `where` na query

com `0 = 0` (zero igual a zero sempre é true)

```
@Override
public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {

    var jpql2 = new StringBuilder();
    jpql2.append("from Restaurante where 0 = 0 ");

    if(nome != null){
        jpql2.append(("and like :nome "));
    }

    if(taxaFreteInicial != null){
        jpql2.append("and taxaFrete >= :taxaFreteInicial");
    }
}
```

No momento a query está assim `from Restaurante where 0 = 0 and like %nome% and taxaFrete >= taxaFreteInicial` listar todos os restaurantes onde zero é igual a zero e que contenha nome (informada pelo user ex: Japonesa) no meio dos nomes e que a taxa de frete especificada seja maior que a taxaFreteInicial (informada pelo user ex: 10, 00 reais)

Se analisarmos, tem um erro de lógica pois o nome pode vir não informado, diferente de nulo, o mesmo se aplica a taxaFreteInicial ou Final. Podemos utilizar a classe utilitária que o SpringFramework fornece `StringUtils.hasLength` do pacote `org.springframework.util.StringUtils`

```
@Override
public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {

    StringBuilder jpql = new StringBuilder();
    jpql.append("from Restaurante where 0 = 0 ");

    if(StringUtils.hasLength(nome)){
        jpql.append(("and nome like :nome "));
    }

    if(taxaFreteInicial != null){
        jpql.append("and taxaFrete >= :taxaFreteInicial ");
    }

    if(taxaFreteFinal != null){
        jpql.append("and taxaFrete <= :taxaFreteFinal");
    }

    return manager.createQuery(jpql.toString(), Restaurante.class)
        .setParameter(name: "nome", value: "%" + nome + "%")
        .setParameter(name: "taxaFreteInicial", taxaFreteInicial)
        .setParameter(name: "taxaFreteFinal", taxaFreteFinal)
        .getResultList();
}
```

Chamando recurso de listar por parâmetro na requisição

GET <http://localhost:8080/teste/restaurantes/por-nome-e-frete?nome=a&taxaFreteInicial=0&taxaFreteFinal=11> Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> nome	a	
<input checked="" type="checkbox"/> taxaFreteInicial	0	
<input checked="" type="checkbox"/> taxaFreteFinal	11	

Body Cookies Headers (5) Test Results Status: 200 OK Time: 224 ms Size: 341 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   {
3     "id": 1,
4     "nome": "Thai Gourmet",
5     "taxaFrete": 10.00,
6     "cozinha": {
7       "id": 1,
8       "nome": "Tailandesa"
9     }
10  },
11  {
12    "id": 2,
13    "nome": "Thai Delivery",
14    "taxaFrete": 9.50,
15    "cozinha": {
16      "id": 1,
17      "nome": "Tailandesa"
18    }
19  }
20 }
```

Apesar das consultas criadas dinamicamente, os parâmetros da query não são dinâmicos, eles estão estáticos

```
return manager.createQuery(jpql.toString(), Restaurante.class)
    .setParameter( name: "nome", value: "%" + nome + "%")
    .setParameter( name: "taxaFreteInicial", taxaFreteInicial)
    .setParameter( name: "taxaFreteFinal", taxaFreteFinal)
    .getResultList();
```

mesmo filtrando valores nulos ou vazios, os parâmetros estão sendo setados, uma forma de contornar isso é setando os parâmetros da query de forma dinâmica de acordo com os parâmetros adicionados na query. Exemplo:


```

@Override
public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {

    StringBuilder jpql = new StringBuilder();
    jpql.append("from Restaurante where 0 = 0 ");

    final HashMap<String, Object> parametros = new HashMap<>();

    if(StringUtils.hasLength(nome)){
        jpql.append("and nome like :nome ");
        parametros.put("nome", "%" + nome + "%");
    }

    if(taxaFreteInicial != null){
        jpql.append("and taxaFrete >= :taxaFreteInicial ");
        parametros.put("taxaFreteInicial", taxaFreteInicial);
    }

    if(taxaFreteFinal != null){
        jpql.append("and taxaFrete <= :taxaFreteFinal");
        parametros.put("taxaFreteFinal", taxaFreteFinal);
    }

    TypedQuery<Restaurante> query = manager.createQuery(jpql.toString(), Restaurante.class);
    parametros.forEach(query::setParameter);
    return query.getResultList();
}

```

Foi feito um map com as propriedades e valores de cada parâmetro adicionado na query e depois setado percorrido por um `forEach` utilizando expressão lambda e method reference, funciona pois a interface funcional esperada é um Bi-consumer que aceita 2 argumentos, e um method reference aceita quaisquer parâmetros se a classe anônima esperada receber a quantidade equivalente de argumento.

O map foi de String-Object pois o parâmetro da query tem que ser equivalente ao tipo da propriedade ex: BigDecimal, Integer ou String, todos são Object .

GET

▼

http://localhost:8080/teste/restaurantes/por-nome-e-frete?taxaFretelInicial=5

Params ●

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	taxaFretelInicial	5	
	Key	Value	Description

Body

Cookies

Headers (5)

Test Results

🌐

Status: 200 OK

Time: 3

Pretty

Raw

Preview

Visualize

JSON ▼

↺

7

"id": 1,

8

"nome": "Tailandesa"

9

}

10

},

11

{

12

"id": 2,

13

"nome": "Thai Delivery",

14

"taxaFrete": 9.50,

15

"cozinha": {

16

"id": 1,

17

"nome": "Tailandesa"

18

}

19

},

20

{

21

"id": 3,

22

"nome": "Tuk Tuk Comida Indiana",

23

"taxaFrete": 15.00,

Página 18 de 5 - Super poderes do Spring Data JPA

5.13. Implementando uma consulta simples com Criteria API

quinta-feira, 16 de fevereiro de 2023 15:38

É ideal para consultas complexas e dinâmicas pois facilita o trabalho se fosse utilizar JPQL, essa API permite montar uma query com código java e seja executada como um sql no banco de dados.

```
23      @Override
24      public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {
25
26          //constroi elementos para criar a consulta
27          CriteriaBuilder criteriaBuilder = manager.getCriteriaBuilder();
28
29          //instância de criteria de restaurante para criar uma consulta de restaurante
30          CriteriaQuery<Restaurante> criteria = criteriaBuilder.createQuery(Restaurante.class);
31
32          //constroi a consulta
33          criteria.from(Restaurante.class);
34
35          final TypedQuery<Restaurante> query = manager.createQuery(criteria);
36
37          return query.getResultList();
38      }
```

linha 27: Temos que criar uma instância de um CriteriaBuilder através de um entityManager anotado com @PersistenceContext

linha 30: através da instância de um CriteriaBuilder, conseguimos criar uma instância de um CriteriaQuery tipada

linha 33: uma criteria consegue criar consultas e depois passa por argumento em uma instância do entityManager através do método sobrecarregado createQuery que recebe um CriteriaQuery<T>

5.14. Adicionando restrições na cláusula where com Criteria API

quinta-feira, 16 de fevereiro de 2023 16:16

```
@Override
public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {

    //instância de CriteriaBuilder que constrói elementos para criar a consulta
    CriteriaBuilder criteriaBuilder = manager.getCriteriaBuilder();

    //instância de criteria de restaurante para criar uma consulta de restaurante
    CriteriaQuery<Restaurante> criteria = criteriaBuilder.createQuery(Restaurante.class);

    //equivalente a "from Restaurante" é o root da consulta
    final Root<Restaurante> root = criteria.from(Restaurante.class);

    //predicados da consulta
    Predicate nomePredicate = criteriaBuilder.like(root.get("nome"), pattern: "%" + nome + "%");
    Predicate taxaFreteInicialPredicate = criteriaBuilder.greaterThanOrEqualTo(root.get("taxaFrete"), taxaFreteInicial);
    Predicate taxaFreteFinalPredicate = criteriaBuilder.lessThanOrEqualTo(root.get("taxaFrete"), taxaFreteFinal);

    //criado a consulta "from Restaurante where nome like :nome and taxaFrete >= :taxaFreteInicial and taxaFrete >= :taxaFreteFinal"
    criteria.where(nomePredicate, taxaFreteInicialPredicate, taxaFreteFinalPredicate);

    //constrói a query e retorna a query do banco
    final TypedQuery<Restaurante> query = manager.createQuery(criteria);

    return query.getResultList();
}
```

5.15. Tornando a consulta com Criteria API com filtros dinâmicos

quinta-feira, 16 de fevereiro de 2023

17:57

```
public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {
    List<Predicate> predicates = new ArrayList<>();

    //instância de CriteriaBuilder que constroi elementos para criar a consulta
    CriteriaBuilder criteriaBuilder = manager.getCriteriaBuilder();

    //instância de criteria de restaurante para criar uma consulta de restaurante
    CriteriaQuery<Restaurante> criteria = criteriaBuilder.createQuery(Restaurante.class);

    //equivalente a "from Restaurante" é o root da consulta
    final Root<Restaurante> root = criteria.from(Restaurante.class);

    //predicados da consulta

    if(StringUtils.hasText(nome)){
        predicates.add(criteriaBuilder.like(root.get("nome"), pattern: "%"+nome+"%"));
    }

    if(taxaFreteInicial != null){
        predicates.add(criteriaBuilder.greaterThanOrEqualTo(root.get("taxaFrete"), taxaFreteInicial));
    }

    if(taxaFreteFinal != null){
        criteriaBuilder.lessThanOrEqualTo(root.get("taxaFrete"), taxaFreteFinal);
    }

    //criado a consulta "from Restaurante where nome like :nome and taxaFrete >= :taxaFreteInicial and taxaFrete >= taxaFreteFinal"
    criteria.where(predicates.toArray(new Predicate[0]));

    //constroi a query e retorna a query do banco
    final TypedQuery<Restaurante> query = manager.createQuery(criteria);

    return query.getResultList();
}
```

5.16. Conhecendo o uso do padrão Specifications (DDD) com SDJ

quinta-feira, 16 de fevereiro de 2023 19:47

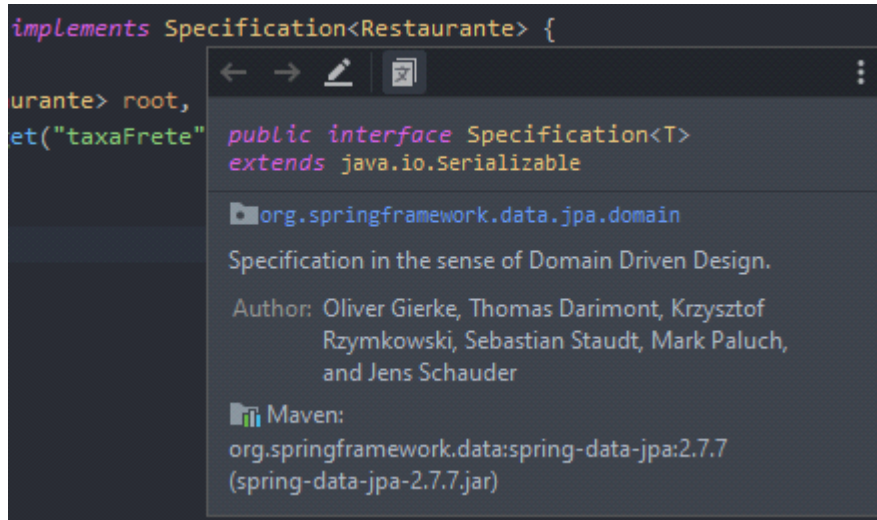
O padrão Specification , é utilizado quando as regras de negócios podem ser recombinadas encadeando as regras de negócios usando a lógica booleana . O padrão é freqüentemente usado no contexto de design orientado a domínio. O padrão define que o princípio do aberto/fechado estabelece que "entidades de software (classes, módulos, funções, etc.) devem ser abertas para extensão, mas fechadas para modificação"; isto é, a entidade pode permitir que o seu comportamento seja estendido sem modificar seu código-fonte. (Wikipedia).

No lugar de polimorfismo simples, podemos adotar o padrão Specification. Nesse caso, a lógica que "muda" com o tempo fica isolada e adequadamente identificada.

5.17. Implementando Specifications com SDJ

quinta-feira, 16 de fevereiro de 2023 20:05

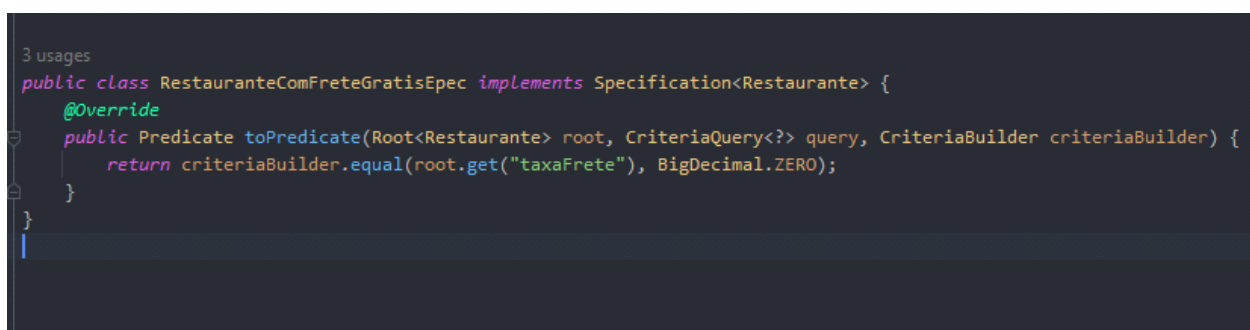
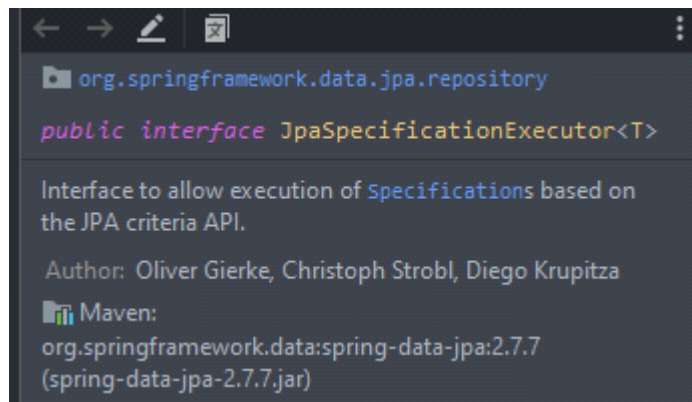
O próprio Spring Data Jpa provê uma interface para trabalhar com o padrão Specification



Como a documentação explicita, é um modelo orientado a domínio, no sentido do DDD.

A ideia é fornecer predicados para cada situação, a ideia é utilizar de forma encadeada para cada predicate, que foi utilizado em [5.15. Tornando a consulta com Criteria API com filtros dinâmicos](#)

Basta implementar a interface Specification<T> e sobrescrever o método toPredicate que retorna um predicate para os métodos de JpaSpecificationExecutor<T> baseados na CRITERIA API.



Ao sobrescrever o método, ele espera um root, uma query e um criteriaBuilder para formar um predicate. No predicate acima a ideia é listar todos os restaurantes com a taxaFrete igual a zero, ou seja, frete grátis. Logo podemos criar um predicate chamando a instância de CriteriaBuilder com o método equal(igual) passando a propriedade do root, e o 2º parâmetro da comparação, BigDecimal.ZERO retorna 0.0 do tipo BigDecimal.

```
@AllArgsConstructor
public class RestauranteComNomeSemelhanteEspec implements Specification<Restaurante> {

    1 usage
    private String nome;
    @Override
    public Predicate toPredicate(Root<Restaurante> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {

        return criteriaBuilder.like(root.get("nome"), pattern: "%" + nome + "%");
    }
}
```

Outra classe implementada por Specification para a criação de um predicate. A lógica é trazer todos os Restaurantes com nome parecido com o valor da variável nome (passando pelo construtor feito com Lombok)

Para utilizar os predicados criados é necessário implementar a interface SpecificationExecutor no repositório da entidade

```
6 usages
@Repository
public interface RestauranteRepository extends JpaRepository<Restaurante, Long>,
    RestauranteRepositoryQueries, JpaSpecificationExecutor<Restaurante> {

    List<Restaurante> findByTaxaFreteBetween(BigDecimal taxaFreteMinima, BigDecimal taxaFreteMaxima);
    List<Restaurante> findByNomeAndCozinhaId(String nome, Long cozinhaId);

    // @Query("from Restaurante where nome like %:nome%")
    List<Restaurante> consultarPorNome(String nome, @Param("nome") String nomeParam);

    Optional<Restaurante> findFirstRestauranteByNomeContaining(String nome);

    List<Restaurante> findTop2ByNomeContaining(String nome);

    int countByCozinhaId(Long cozinha);

    1 usage
```

```
public interface JpaSpecificationExecutor<T>
```

org.springframework.data.jpa.repository

Interface to allow execution of specifications based on the JPA criteria API.

Author: Oliver Gierke, Christoph Strobl, Diego Krupitza

Maven:

org.springframework.data:spring-data-jpa:2.7.7 (spring-data-jpa-2.7.7.jar)

Com os métodos disponíveis

```
JpaSpecificationExecutor.java
```

☒ Inherited members (Ctrl+F12) ☐ Anonymous Classes (Ctrl+I) ☐ Lambdas (Ctrl+L) ⚙

▼ **JpaSpecificationExecutor**

- count(Specification<T>): long
- exists(Specification<T>): boolean
- findAll(Specification<T>): List<T>
- findAll(Specification<T>, Pageable): Page<T>
- findAll(Specification<T>, Sort): List<T>
- findOne(Specification<T>): Optional<T>

Vamos chamar o método `findAll` que recebe um predicate

```
@GetMapping("/restaurantes/com-frete-gratis")
public List<Restaurante> restaurantesComFreteGratis(String nome) {
    final RestauranteComFreteGratisEpec comFreteGratis = new RestauranteComFreteGratisEpec();
    final RestauranteComNomeSemelhanteEspec comNomeSemelhante = new RestauranteComNomeSemelhanteEspec(nome);

    return restauranteRepository.findAll(comFreteGratis.and(comNomeSemelhante));
}
```

e podemos concatenar os predicados

5.18. Criando uma fábrica de Specifications

sexta-feira, 17 de fevereiro de 2023 08:35

Na aula anterior foi implementado algumas classes que implementavam a interface funcional Specification<T>

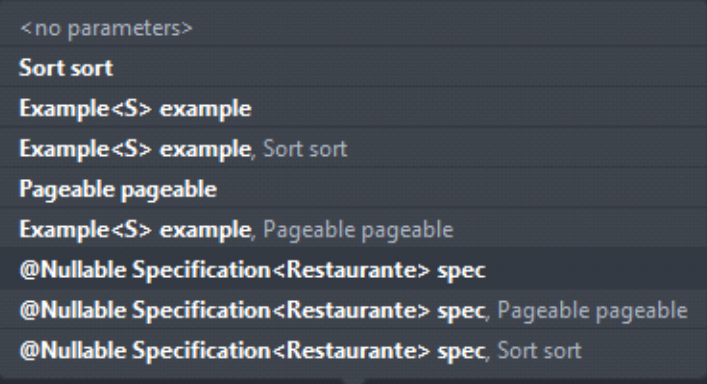
```
public class RestauranteComFreteGratisEspec implements Specification<Restaurante> {  
    @Override  
    public Predicate toPredicate(Root<Restaurante> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {  
        return criteriaBuilder.equal(root.get("taxaFrete"), BigDecimal.ZERO);  
    }  
}
```

```
@AllArgsConstructor  
public class RestauranteComNomeSemelhanteEspec implements Specification<Restaurante> {  
    1 usage  
    private String nome;  
    @Override  
    public Predicate toPredicate(Root<Restaurante> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {  
        return criteriaBuilder.like(root.get("nome"), pattern: "%" + nome + "%");  
    }  
}
```

Para serem usadas com o repositório de restaurante, através do método findAll provido pela interface JpaSpecificationExecutor<T> que recebe uma instância de Specification

```
@Repository  
public interface RestauranteRepository extends JpaRepository<Restaurante, Long>,  
    RestauranteRepositoryQueries, JpaSpecificationExecutor<Restaurante> {
```

```
@Autowired  
private RestauranteRepository restauranteRepository;  
  
@GetMapping("restaurantes")  
public List<Restaurante> getAll() {  
    return restauranteRepository.findAll();  
}  
  
@GetMapping("restaurantes/{nome}")  
public List<Restaurante> getByName(@PathVariable String nome) {  
    return restauranteRepository.findAll(RestauranteSpecs.restaurantesComNomeSemelhante(nome));  
}
```



mas podemos utilizar expressões lambdas nesse caso, pois Specification é uma interface funcional

```

public static Specification<Restaurante> restaurantesComFreteGratis(){
    return (root, query, criteriaBuilder) -> criteriaBuilder.equal(root.get("taxaFrete"), BigDecimal.ZERO);
}

1 usage
public static Specification<Restaurante> restaurantesComNomeSemelhante(String nome){
    return new RestauranteComNomeSemelhanteEspec(nome);
}

```

métodos com expressão lambda no retorno

```

public class RestauranteSpecs {

    public static Specification<Restaurante> restaurantesComFreteGratis(){
        return (root, query, criteriaBuilder) -> criteriaBuilder.equal(root.get("taxaFrete"), BigDecimal.ZERO);
    }

    1 usage
    public static Specification<Restaurante> restaurantesComNomeSemelhante(String nome){
        return (root, query, criteriaBuilder) -> criteriaBuilder.like(root.get("nome"), pattern: "%" + nome + "%");
    }
}

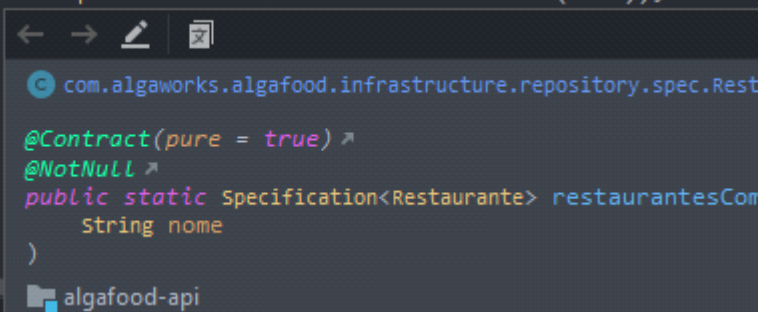
```

Agora o método retorna uma instância feita com expressão lambda

```

.findAll(RestauranteSpecs.restaurantesComNomeSemelhante(nome));

```



```

@GetMapping("/restaurantes/com-frete-gratis")
public List<Restaurante> restaurantesComFreteGratis(String nome) {

    return restauranteRepository.findAll(
        RestauranteSpecs.restaurantesComNomeSemelhante(nome)
        .and(RestauranteSpecs.restaurantesComNomeSemelhante(nome)));
}

```

5.19. Injetando o próprio repositório na implementação customizada e a anotação @Lazy

sexta-feira, 17 de fevereiro de 2023 11:15

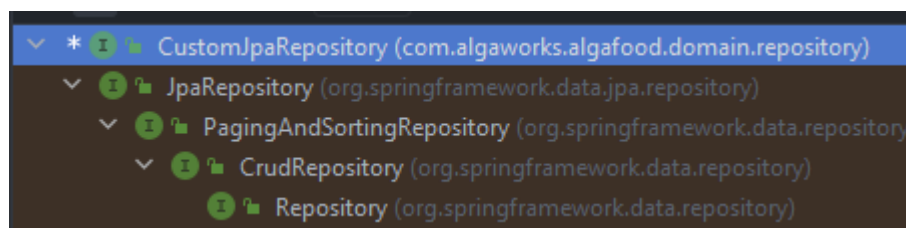
Para dependências circulares adicionar @Lazy na injeção

5.20. Estendendo o JpaRepository para customizar o repositório base

sexta-feira, 17 de fevereiro de 2023 11:31

Podemos adicionar métodos no repositório base de qualquer entidade e sempre usarmos a partir da necessidade.

Primeiro criamos uma interface que implementa JpaRepository<T, ID> uma interface customizada do JpaRepository (CustomJpaRepository) e adicionamos a assinatura do nosso próprio método, agora, em vez da entidade herdar JpaRepository, ela vai implementar CustomJpaRepository, nosso repositório customizado (usando Generics para tipar). Fazendo isso, estamos criando apenas mais uma camada de abstração na hierarquia.



```
3 usages 2 implementations
@NoRepositoryBean // ignorando a instanciação de um bean pelo Spring
public interface CustomJpaRepository<T, ID> extends JpaRepository<T, ID> {

    1 usage 1 implementation
    Optional<T> buscarPrimeiro ();
}
```

@NoRepositoryBean: a documentação define:

Anotação para excluir as interfaces do repositório de serem selecionadas e, portanto, obter uma instância sendo criada. Isso normalmente será usado ao fornecer uma interface base estendida para todos os repositórios em combinação com uma classe base de repositório personalizada para implementar métodos declarados nessa interface intermediária. Nesse caso, você normalmente deriva suas interfaces de repositório concretas da interface intermediária, mas não deseja criar um bean Spring para a interface intermediária.

Basicamente, definimos para que o Spring não crie um bean para nossa interface intermediária, pois nós iremos criar "na mão" a nossa própria implementação.

Agora precisamos implementar a interface para que ela sirva para qualquer repositório, criamos uma classe que estende a classe (SimpleJpaRepository) que por padrão

implementa JpaRepository e implementaremos a nossa interface customizada do JpaRepository. Após isso implementar o método herdado da interface customizada. Fazendo isso, a classe criada terá todos os métodos (SimpleJpaRepository) implementados ainda disponíveis e a nossa própria implementação do método da interface customizada.

```
2 usages
public class CustomJpaRepositoryImpl<T, ID>
    extends SimpleJpaRepository<T, ID>
    implements CustomJpaRepository<T, ID> {

    2 usages
    private EntityManager entityManager;

    public CustomJpaRepositoryImpl(JpaEntityInformation<T, ?> entityInformation,
        EntityManager entityManager) {
        super(entityInformation, entityManager);
        this.entityManager = entityManager;
    }
}
```

Agora precisamos criar nossa consulta baseada na assinatura do método através de um EntityManager, não precisamos injetar e nem criar instâncias, pois a classe que estendemos possui um construtor que recebe uma instância de entityManager.

```
@Override
public Optional<T> buscarPrimeiro() {
    //getDomainClass retorna a classe que representa a entidade, getName retorna o nome em String
    String jpql = "from " + getDomainClass().getName();

    final TypedQuery<T> query = entityManager.createQuery(jpql, getDomainClass());
    final T singleResult = query
        .setMaxResults(1)//somente 1 entidade (sql Limit)
        .getSingleResult();
    return Optional.ofNullable(singleResult);
}
```

A nossa consulta tem que ser dinâmica, pois a classe implementadora é genérica (Generics)

Agora temos que sinalizar ao Spring que temos outro repositório base para

```
@SpringBootApplication
@EnableJpaRepositories(repositoryBaseClass = CustomJpaRepositoryImpl.class)
public class AlgafoodApiApplication {
```

Dessa forma, substituímos a implementação do repositório padrão (SimpleJpaRepository), que de certa forma é um SimpleJpaRepository, porém, uma especialização.

Agora temos que utilizar nossa implementação em uma entidade, como de praxe, estendemos o repositório e anotamos com @Repository

```

@Repository
public interface RestauranteRepository extends CustomJpaRepository<Restaurante, Long>,
    RestauranteRepositoryQueries, JpaSpecificationExecutor<Restaurante> {

```

Agora podemos fazer a consulta pelo nosso repositório da entidade e de várias outras entidades

```

@GetMapping("/restaurantes/primeiro")
public Optional<Restaurante> restaurantePrimeiro() {
    return restauranteRepository.buscarPrimeiro();
}

```

← → ↗ 🗑
⋮

com.algaworks.algafood.domain.repository.CustomJpaRepository<T, ID>

Optional<T> buscarPrimeiro()

algafood-api