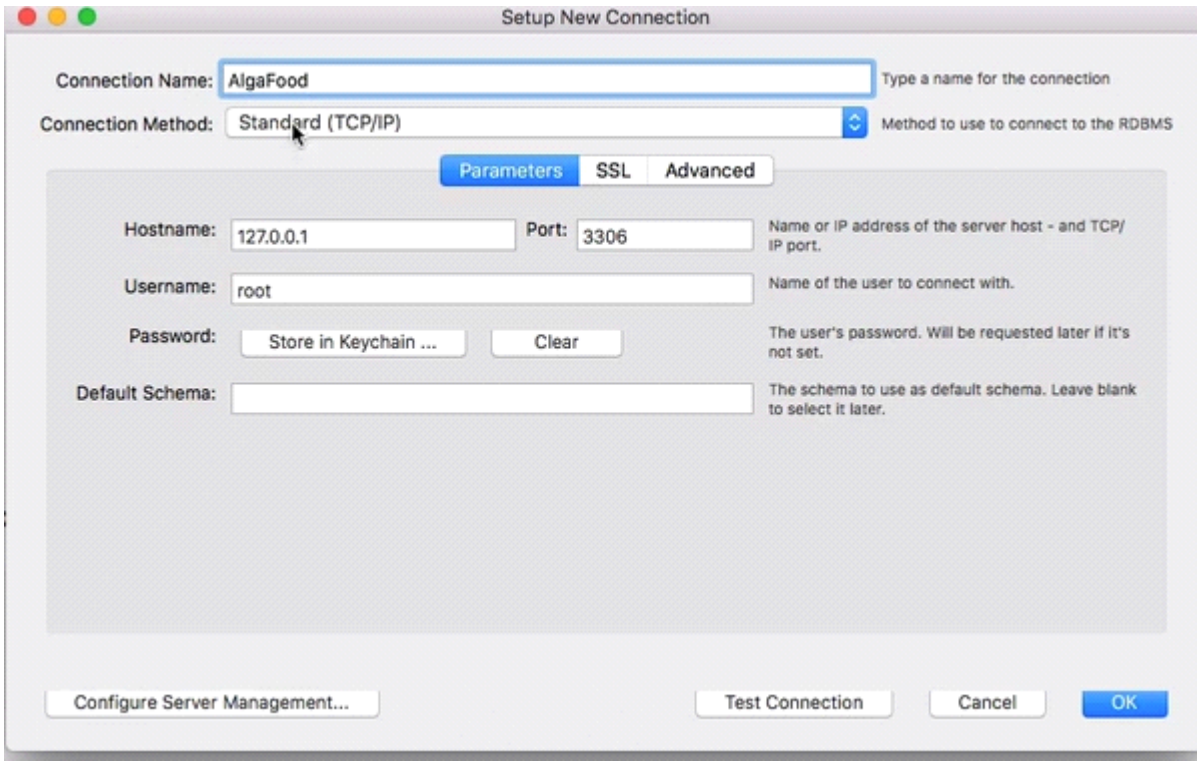
3.1. Instalando o MySQL Server e MySQL Workbench

sexta-feira, 3 de fevereiro de 2023 17:33



3.2. O que é JPA e Hibernate

sexta-feira, 3 de fevereiro de 2023 17:34

O que é persistência de dados?

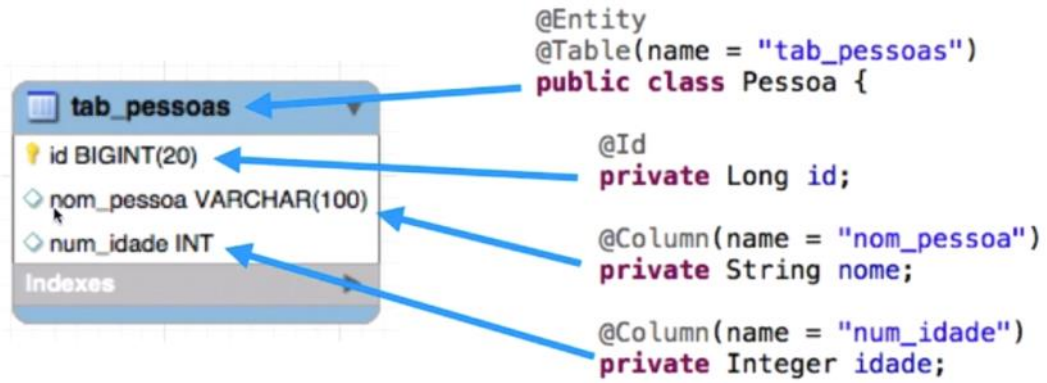
Persistência com banco de dados usando Java

O que é ORM?

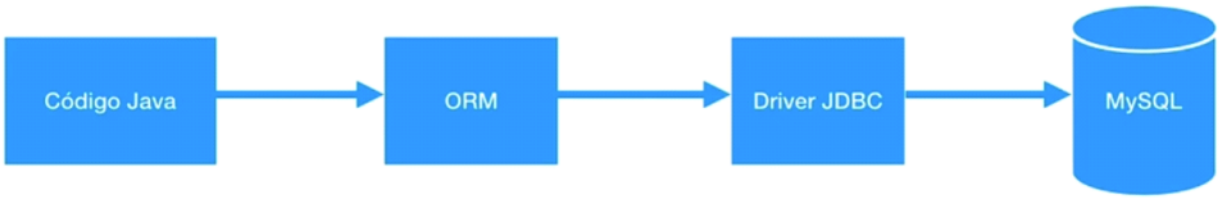
Mapeamento Objeto-Relacional

Modelo Relacional	Modelo OO
Tabela	Classe
Linha	Objeto
Coluna	Atributo
	Método
Chave estrangeira	Associação

Mapeamento Objeto-Relacional



Persistência com ORM



Persistência com ORM

```
public void adicionar(Pessoa pessoa) {
    manager.persist(pessoa);
}
```

Persistência com ORM

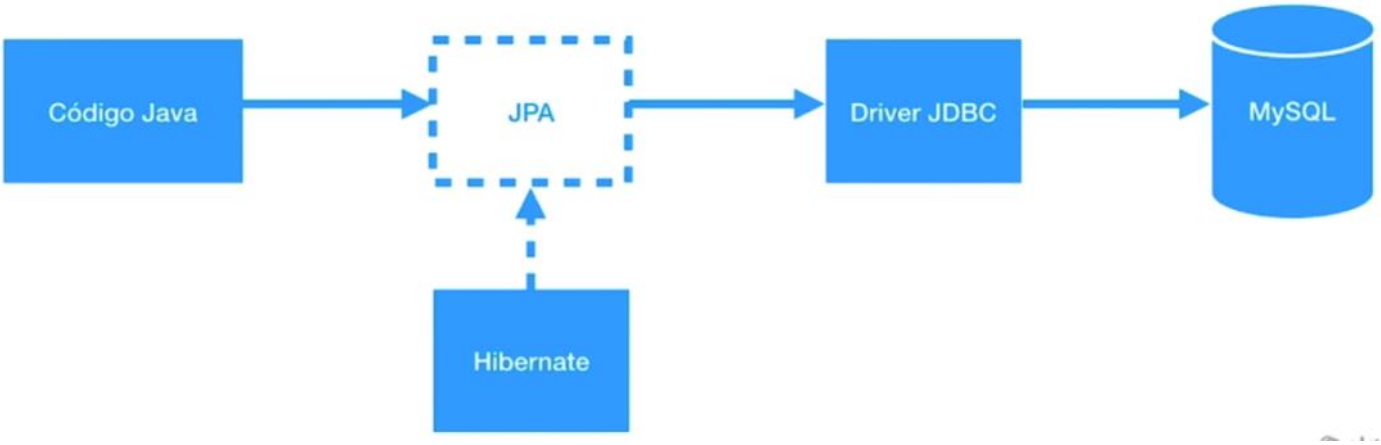
```
public List<Pessoa> consultar() {
    return manager.createQuery("from Pessoa", Pessoa.class).getResultList();
}
```

O que é Java Persistence API (JPA)?

É uma especificação JEE, uma solução ORM para persistência de dados que é padronizada. Define a forma de mapeamento objeto-relacional, possui api de consulta e modificação de dados e a linguagem JPQL.

JPA é uma especificação e descreve como algo deve funcionar
Hibernate é uma implementação do JPA

Persistência com banco de dados com JPA.



3.3. Adicionando JPA e configurando o Data Source

sábado, 4 de fevereiro de 2023 23:06

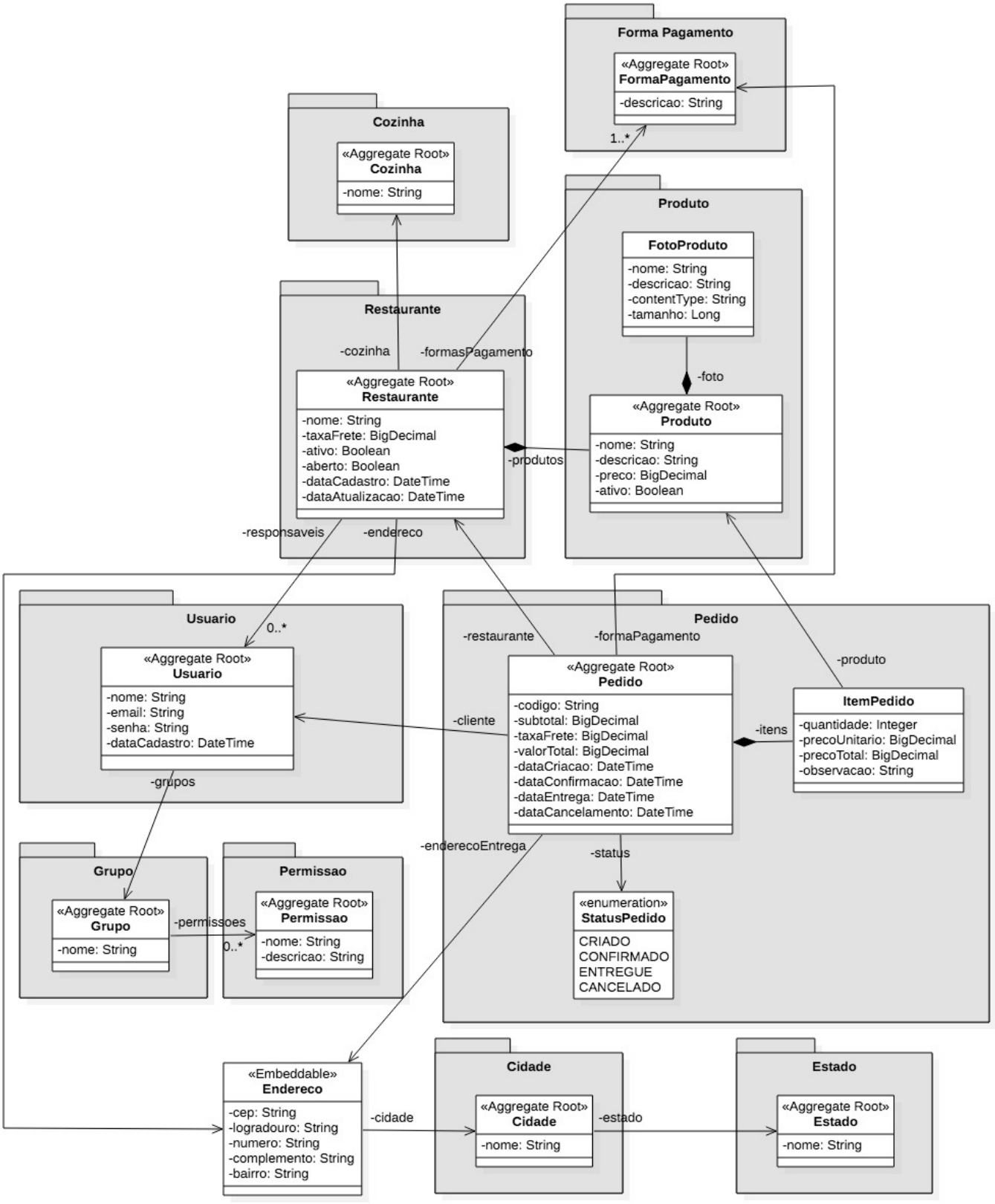
Dependência adicionada

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
		<groupId>mysql</groupId>
		<artifactId>mysql-connector-java</artifactId>
		<version>8.0.32</version>
	</dependency>
```

para iniciar um projeto Spring Data JPA, é necessário criar a configuração no datasource (application.properties)

3.4. Mapeando entidades com JPA

segunda-feira, 6 de fevereiro de 2023 15:32



Classe Restaurante e Cozinha adicionada representando as tabelas do diagrama

3.5. Criando as tabelas do banco a partir das entidades

segunda-feira, 6 de fevereiro de 2023 15:35

Gerando tabelas a partir das classes mapeadas (não é uma boa prática fazer isso em produção)

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/algafood?createDatabaseIfNotExist=true&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=1042

#gerando as tabelas automaticamente script de criacao
spring.jpa.generate-ddl=true

#configuracao especifica do hibernate p/ qual forma o ddl vai ser executado
spring.jpa.hibernate.ddl-auto=create
```

Modo DDL. Este é realmente um atalho para a propriedade "hibernate.hbm2ddl.auto". O padrão é "create-drop" ao usar um banco de dados incorporado e nenhum gerenciador de esquema foi detectado. Caso contrário, o padrão é "nenhum".

Valores:

nenhum Desativa a manipulação de DDL.

validar Valida o esquema, não faz alterações no banco de dados.

update Atualize o esquema, se necessário.

create Cria o esquema e destrói os dados anteriores.

create-drop Crie e destrua o esquema no final da sessão.

```
#mostrando o sql no console da aplicacao
spring.jpa.show-sql=true
```

3.6. Mapeando o id da entidade para autoincremento

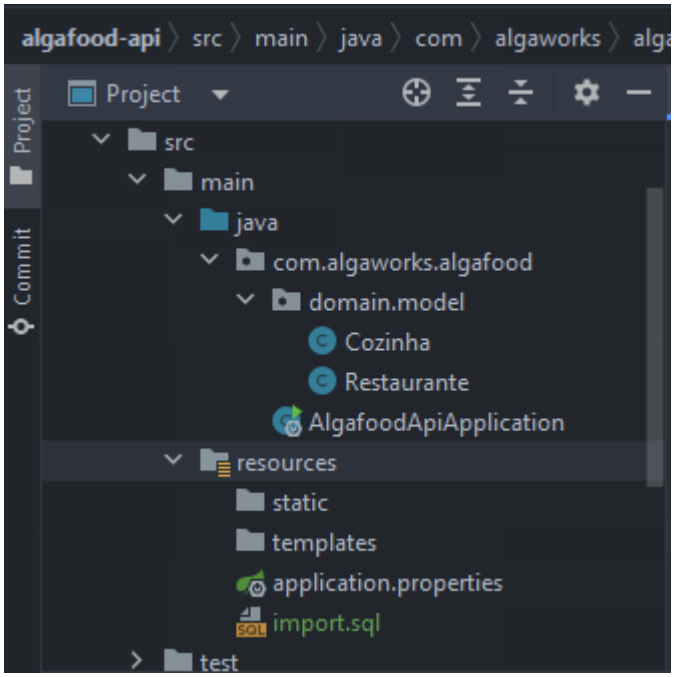
segunda-feira, 6 de fevereiro de 2023 16:06

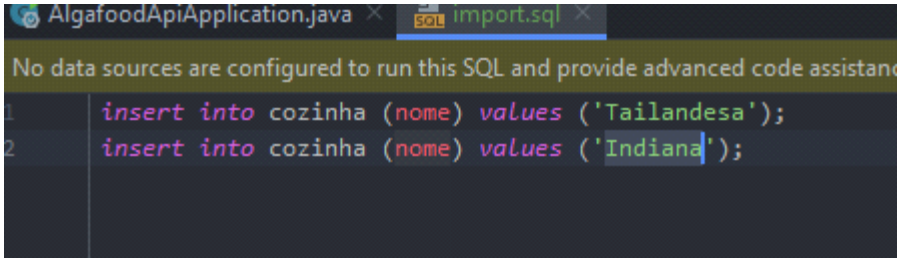
Foi adicionado a estratégia de geração de identificador

3.7. Importando dados de teste com import

segunda-feira, 6 de fevereiro de 2023 16:06

Para fazer inserção de dados a partir de um arquivo é necessário criar um arquivo import.sql na pasta src/main/resources





3.8. Consultando objetos do banco de dados

segunda-feira, 6 de fevereiro de 2023 16:28

Fazendo uma consulta através da interface EntityManager do JPA pelo Spring.

É necessário ter uma instância da interface, e podemos injetar em um ponto de injeção com @persistenceContext (diferente de @Autowried) e usar JPQL para fazer uma busca avançada com objetos.

```
@Component
public class CadastroCozinha {

    1 usage
    @PersistenceContext
    private EntityManager entityManager;

    List<Cozinha> listar(){
        final TypedQuery<Cozinha> cozinhas = entityManager.createQuery( qlString: "from Cozinha", Cozinha.class);

        return cozinhas.getResultList();
    }
}
```

Para utilizar a classe, vamos iniciar uma classe com a ideia de não ser web (aplicação não web) pois na classe de inicialização do Spring, ele inicia uma app web com @SpringBootApplication e fica esperando requisições

```
1 usage
@SpringBootApplication
public class AlgafoodApiApplication {

    public static void main(String[] args) { SpringApplication.run(AlgafoodApiApplication.class, args); }

}
```

Vamos utilizar uma não web (Inicia, roda e finaliza) é necessário ter um contexto do Spring

```
AlgafoodApiApplication.java  CadastroCozinha.java  ConsultaCozinhaMain.java
1   package com.algaworks.algafood.jpa;
2
3   import com.algaworks.algafood.AlgafoodApiApplication;
4   import org.springframework.boot.WebApplicationType;
5   import org.springframework.boot.builder.SpringApplicationBuilder;
6   import org.springframework.context.ApplicationContext;
7
8   public class ConsultaCozinhaMain {
9       public static void main(String[] args) {
10
11          ApplicationContext applicationContext = new SpringApplicationBuilder(AlgafoodApiApplication.class)
12                  .web(WebApplicationType.NONE)
13                  .run(args);
14      }
15  }
16
```

ApplicationContext = new SpringApplicationBuilder(AlgafoodApiApplication.class)
.web(WebApplicationType.NONE)
.run(args);

O construtor de SpringApplicationBuilder espera como argumento uma classe com configurações básicas iniciais do contexto Spring, usamos aquela classe pois está anotada com @SpringBootApplication mas podemos usar qualquer classe usando essa anotação inclusive a própria classe e o método web dizendo que esta não é uma aplicação web.

```
final CadastroCozinha bean = applicationContext.getBean(CadastroCozinha.class);

for (Cozinha cozinha : bean.listar()) {
    System.out.println(cozinha.getNome());
}
```

Pegando um Bean do contexto da aplicação

3.9. Adicionando um objeto no banco de dados

segunda-feira, 6 de fevereiro de 2023 16:57

Persistindo uma instância pelo JPA

Podemos utilizar o método merge do EntityManager

```
2 usages
public Cozinha adicionar(Cozinha cozinha){
    return entityManager.merge(cozinha);
}
```

para persistir

```
public class InclusaoCozinhaMain {
    public static void main(String[] args) {

        ApplicationContext applicationContext = new SpringApplicationBuilder(AlgafoodApiApplication.class)
                .web(WebApplicationType.NONE)
                .run(args);

        final CadastroCozinha bean = applicationContext.getBean(CadastroCozinha.class);

        Cozinha cozinha1 = new Cozinha();
        Cozinha cozinha2 = new Cozinha();

        cozinha1.setNome("Brasileira");
        cozinha2.setNome("Japonesa");

        bean.adicionar(cozinha1);
        bean.adicionar(cozinha2);
```

```
ion in thread "restartedMain" java.lang.reflect.InvocationTargetException <4 internal lines>
 org.springframework.boot.devtools.restart.RestartLauncher.run(RestartLauncher.java:49)
 by: javax.persistence.TransactionRequiredException Create breakpoint : No EntityManager with actual transaction available for current thread - cannot reliably process 'merge' call
 org.springframework.orm.jpa.SharedEntityManagerCreator$SharedEntityManagerInvocationHandler.invoke(SharedEntityManagerCreator.java:295)
 jdk.proxy4/jdk.proxy4.$Proxy81.merge(Unknown Source)
 com.algaworks.algafood.jpa.CadastroCozinha.adicionar(CadastroCozinha.java:24)
 com.algaworks.algafood.jpa.InclusaoCozinhaMain.main(InclusaoCozinhaMain.java:24)
 . 5 more
2-06 17:02:16.138  INFO 9236 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for persistence unit 'default'
2-06 17:02:16.143  INFO 9236 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
2-06 17:02:16.160  INFO 9236 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.

s finished with exit code 0
```

O erro acontece pois o método não acontece dentro de uma transação

```
    public List<Cozinha> listar(){
        final TypedQuery<Cozinha> cozinhas = e

        return cozinhas.getResultList();
    }

    2 usages
    @Transactional
    public Cozinha adicionar(Cozinha cozinha){
        return entityManager.merge(cozinha);
    }
}
```

```
@Target({ElementType.TYPE,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Transactional
extends java.lang.annotation.Annotation

org.springframework.transaction.annotation

Describes a transaction attribute on an individual method or on a class.
When this annotation is declared at the class level, it applies as a default to all methods of the declaring class and its subclasses. Note that it does not apply to ancestor classes up the class hierarchy; inherited methods need to be locally redeclared in order to participate in a subclass-level annotation. For details on method visibility constraints, consult the Transaction Management ↗ section of the reference manual.

This annotation is generally directly comparable to Spring's org.springframework.transaction.interceptor.RuleBasedTransactionAttribute class, and in fact AnnotationTransactionAttributeSource will directly convert this annotation's attributes to properties in RuleBasedTransactionAttribute, so that Spring's transaction support code does not have to know about annotations.

Attribute Semantics

If no custom rollback rules are configured in this annotation, the transaction will roll back on RuntimeException and Error but not on checked exceptions.

Rollback rules determine if a transaction should be rolled back when a given exception is thrown, and the rules are based on patterns. A pattern can be a fully qualified class name or a substring of a fully qualified class name for an exception type (which must be a subclass of Throwable), with no wildcard support at present. For example, a value of "javax.servlet.ServletException" or "ServletException" will match javax.servlet.ServletException and its subclasses.

Rollback rules may be configured via rollbackFor/noRollbackFor and
```

```
.restart.RestartLauncher.run(RestartLauncher.java
RequiredException Create breakpoint : No EntityManage
edEntityManagerCreator$SharedEntityManagerInvocati
erge(Unknown Source)
roCozinha.adicionar(CadastroCozinha.java:24)
aoCozinhaMain.main(InclusaoCozinhaMain.java:24)

ionShutdownHook] j.LocalContainerEntityManagerFac
ionShutdownHook] com.zaxxer.hikari.HikariDataSour
ionShutdownHook] com.zaxxer.hikari.HikariDataSour
```

3.10. Buscando um objeto pelo id no banco de dados

terça-feira, 7 de fevereiro de 2023 13:51

Buscando uma instância de Cozinha do banco de dados pelo método find do EntityManager

```
public Cozinha buscar(Long id){
    return entityManager.find(Cozinha.class, id);
}
```

3.11. Atualizando um objeto no banco de dados

terça-feira, 7 de fevereiro de 2023 16:19

```
ApplicationContext applicationContext = new SpringApplicationBuilder(AlgafoodApiApplication.class)
        .web(WebApplicationType.NONE)
        .run(args);

final CadastroCozinha bean = applicationContext.getBean(CadastroCozinha.class);

Cozinha cozinha1 = new Cozinha();

cozinha1.setNome("Brasileira");
cozinha1.setId(1L);
bean.salvar(cozinha1);
```

```
Hibernate: select cozinha0_.id as id1_0_0_, cozinha0_.nome as nome2_0_0_ from cozinha cozinha0_ where cozinha0_.id=?
Hibernate: update cozinha set nome=? where id=?
2023-02-07 16:18:55.457  INFO 9948 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManag
2023-02-07 16:18:55.462  INFO 9948 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown
2023-02-07 16:18:55.481  INFO 9948 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown

Process finished with exit code 0
```

O hibernate fez um select implicitamente pois tentamos atualizar um objeto sem o mesmo ser monitorado pelo contexto de persistência

```
3 usages
@Transactional
public Cozinha salvar(Cozinha cozinha){
    return entityManager.merge(cozinha);
}
```

o método merge serve tanto para atualizar quanto para salvar um objeto na base de dados, se o objeto ter um id, o hibernate busca o id antes de salvar

3.12. Excluindo um objeto do banco de dados

terça-feira, 7 de fevereiro de 2023 16:30

```java
Cozinha cozinha1 = new Cozinha();

cozinha1.setId(1L);
bean.remover(cozinha1);
```

```java
1 usage
@Transactional
public void remover (Cozinha cozinha){
    cozinha = buscar(cozinha.getId());
    entityManager.remove(cozinha);
}
```

Estados de uma entidade

terça-feira, 7 de fevereiro de 2023 19:01

Uma entidade pode assumir alguns estados com relação ao EntityManager. Os estados podem ser:

- Novo (*new* ou *transient*)
- Gerenciado (*managed*)
- Removido (*removed*)
- Desanexado *(detached)*

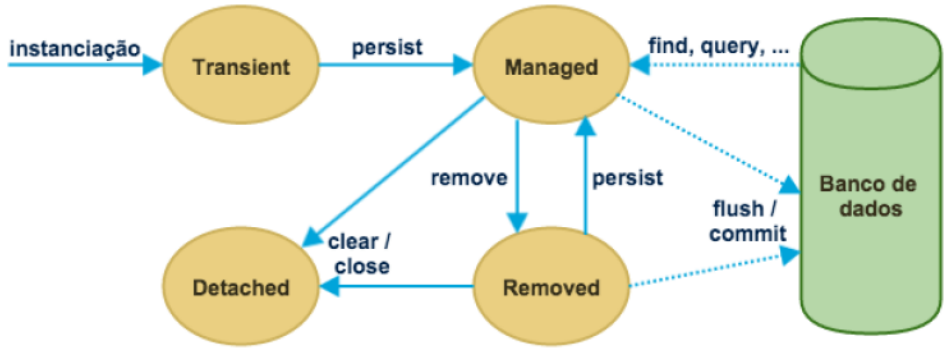
→ Novo
  - O estado "novo" é o mais natural. É simplesmente quando construímos um objeto qualquer usando o operador newj.

→ Gerenciado
  - Para um objeto estar gerenciado, podemos chamar o método persist, merge ou buscar a entidade usando o EntityManager.

→ Removido
  - O estado removido é alcançado quando chamados o método remove.

→ Desanexado
  - A entidade fica desanexada quando é passada para o método detached.

Uma entidade desanexada pode voltar ao estado gerenciado ao chamar o método merge.



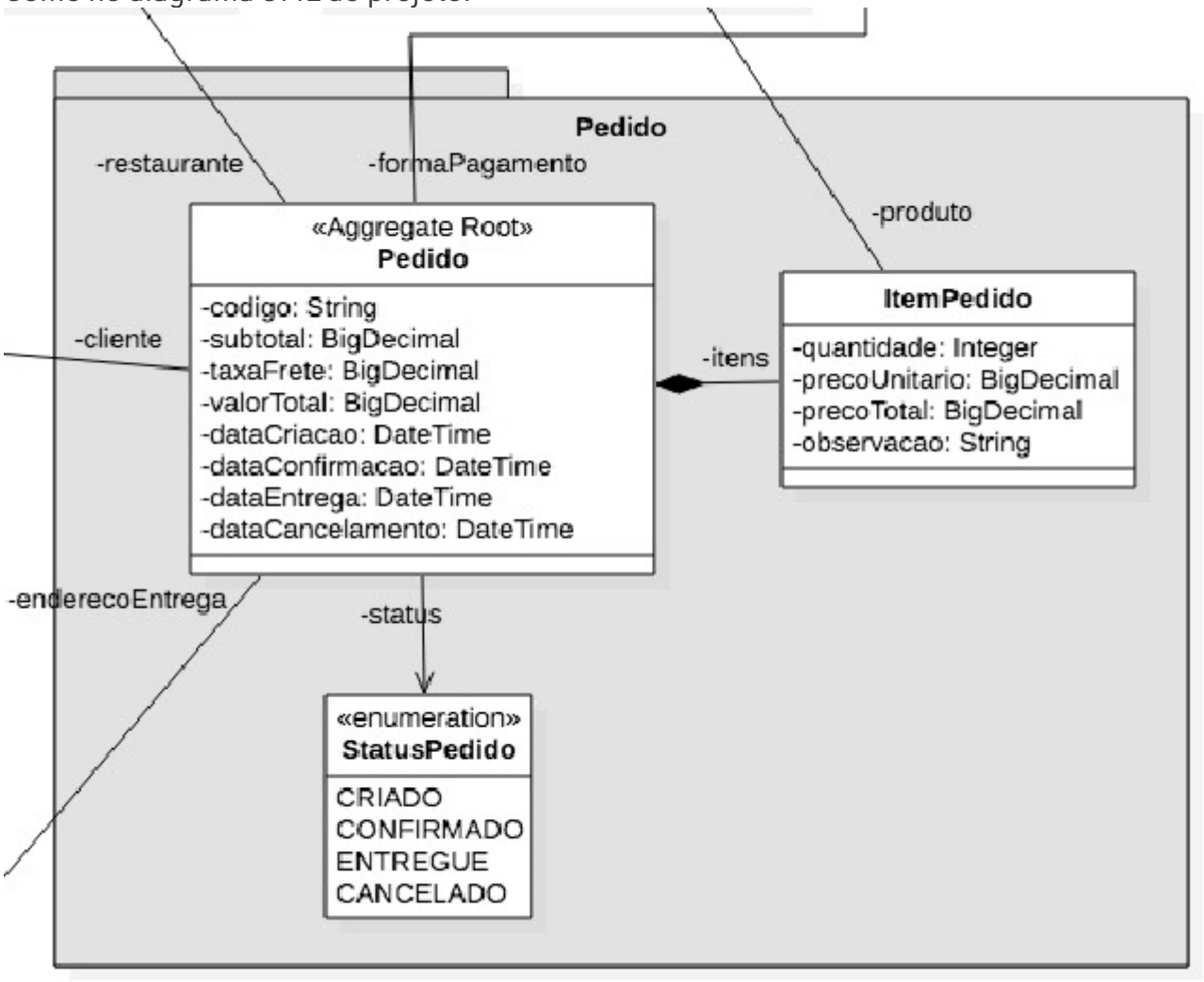
Leitura adicional:
https://blog.algaworks.com/tutorial-jpa/

3.13. Conhecendo o padrão Aggregate do DDD

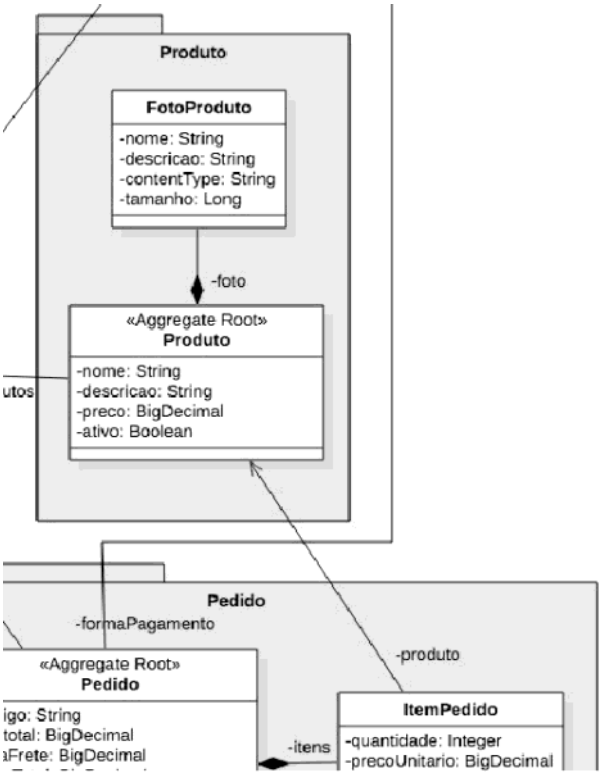
terça-feira, 7 de fevereiro de 2023 18:23

Aggregate é um padrão em Design Orientado a Domínio. Um aggregate DDD é um amontoa de objetos de domínio que podem ser tratados como uma única unidade.
Um exemplo pode ser um pedido e seus itens de linha, que serão objetos separados, mas é útil tratar o pedido (junto com seus itens de linha) como um único agregado.

Como no diagrama UML do projeto:



Um agregado terá um de seus objetos componentes como a aggregate root. Quaisquer referências de fora do agregado devem ir apenas para a raiz do agregado. A agregado raiz pode, assim, garantir a integridade do agregado como um todo. Como no aggregate de **Produto** que é referenciado por outro componente de fora do agregado, logo, agregados raiz podem ser referenciados por outros componentes de fora, o ideal dessa referência é para o aggregate root.



Agregados são o elemento básico de transferência de armazenamento de dados - você solicita carregar ou salvar agregados inteiros. As transações não devem cruzar os limites agregados.

DDD Aggregates às vezes são confundidos com classes de coleção (listas, mapas, etc). Os agregados DDD são conceitos de domínio (pedido, visita clínica, lista de reprodução), enquanto as coleções são genéricas. Uma agregação geralmente contém várias coleções, juntamente com campos simples. O termo "agregado" é comum e é usado em vários contextos diferentes (por exemplo, UML), caso em que não se refere ao mesmo conceito de um agregado DDD.

Leitura adicional:
https://martinfowler.com/bliki/DDD_Aggregate.html

3.14. Conhecendo e implementando o padrão Repository

terça-feira, 7 de fevereiro de 2023 20:14

Faz a mediação entre o domínio e as camadas de mapeamento de dados usando uma interface semelhante a uma coleção para acessar objetos de domínio.

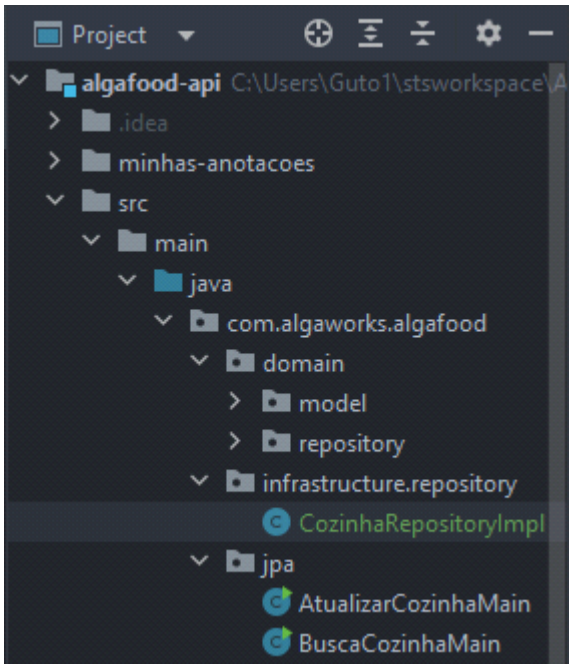
Padrão do DDD, adiciona uma camada de abstração de acesso a dados e que imita uma coleção de forma que quem usa o repositório, não precisa saber o mecanismo de persistência que está sendo usado pelo repositório.

Podemos pensar no que um repositório deve conter em relação a tarefas básicas de persistência:

```java
import com.algaworks.algafood.domain.model.Cozinha;

import java.util.List;

public interface CozinhaRepository {
    List<Cozinha> listar();
    Cozinha buscar(Long id);
    Cozinha salvar(Cozinha cozinha);
    void remover(Cozinha cozinha);
}
```

Como um repositório é algo que representa uma abstração do acesso aos dados, a interface pode ser considerada como parte da camada de negócio da aplicação.

- Com camada de domínio, queremos dizer que não tem tantos detalhes técnicos da aplicação, não há informações sobre mecanismos de persistências. O repositório pode ser chamado de repositório orientado a persistência.



Foi criado um pacote chamado infrastructure para adicionar implementações que são fora do escopo de regra de negócio

```java
package com.algaworks.algafood.domain.repository;

import com.algaworks.algafood.domain.model.Cozinha;

import java.util.List;

17 usages  1 implementation
public interface CozinhaRepository {
    1 usage  1 implementation
    List<Cozinha> todas();
    2 usages  1 implementation
    Cozinha porId(Long id);
    3 usages  1 implementation
    Cozinha adicionar(Cozinha cozinha);
    1 usage  1 implementation
    void remover(Cozinha cozinha);
}
```

Como o padrão repository imita uma coleção de acesso a dados, podemos refatorar o trecho do código para se assemelhar a uma coleção de cozinhas, para dar mais semântica na usabilidade do repositório.

```java
final CozinhaRepository cozinhas = applicationContext.getBean(CozinhaRepository.class);

for (Cozinha cozinha : cozinhas.todas()) {
    System.out.println(cozinha.getNome());
}
```

utilizando o método *todas* para se assemelhar a um método de uma coleção list.all ou list.find

3.15. Conhecendo e usando o Lombok

terça-feira, 7 de fevereiro de 2023 22:06

Instalar o Lombok no Intellij:

- Adicionar dependência

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

- adicionar o plugin pelo marketplace

3.16. Desafio Lombok e repositório de restaurantes

terça-feira, 7 de fevereiro de 2023 22:38

Entidade Restaurante adicionada com lombok e seu repositório

```java
@Data
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Restaurante {

    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) //provedor de persistencia
    private Long Id;

    private String nome;

    @Column(name = "taxa_frete")
    private BigDecimal taxaFrete;

}
```

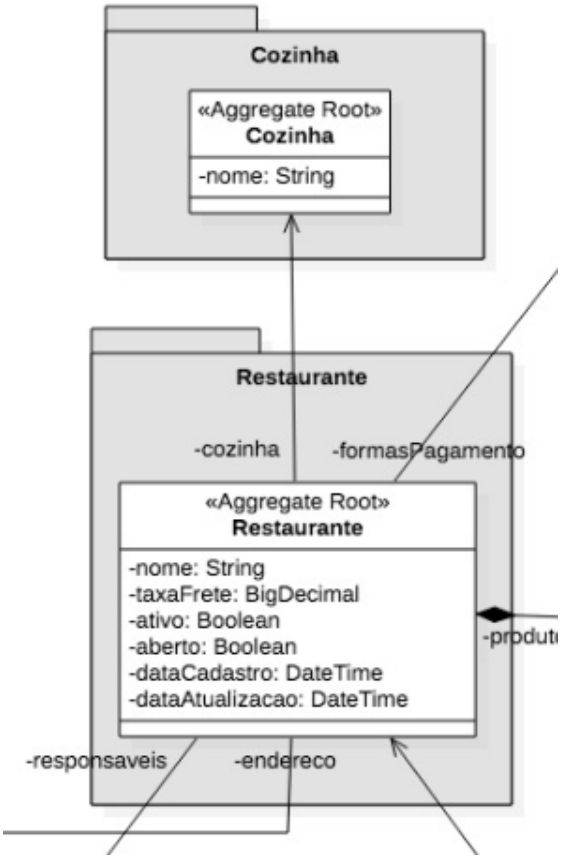
```java
5 usages  1 implementation
public interface RestauranteRepository {
    1 implementation
    List<Restaurante> todas();
    2 usages  1 implementation
    Restaurante porId(Long id);
    1 implementation
    Restaurante adicionar(Restaurante restaurante);
    1 implementation
    void remover(Restaurante restaurante);
}
```

```java
@Component
public class RestauranteRepositoryImpl implements RestauranteRepository {

    4 usages
    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public List<Restaurante> todas(){
        final TypedQuery<Restaurante> restaurantes = entityManager.createQuery( qlString: "from Restau

        return restaurantes.getResultList();
    }

    @Transactional
    @Override
    public Restaurante adicionar(Restaurante restaurante){
        return entityManager.merge(restaurante);
    }

    2 usages
    @Override
    public Restaurante porId(Long id){
        return entityManager.find(Restaurante.class, id);
    }

    @Transactional
    @Override
    public void remover (Restaurante restaurante){
        restaurante = porId(restaurante.getId());
        entityManager.remove(restaurante);
    }
```

```java
public class ConsultaRestauranteMain {
    public static void main(String[] args) {

        ApplicationContext applicationContext = new SpringApplicationBuilder(AlgafoodApiApplication.class)
                .web(WebApplicationType.NONE)
                .run(args);

        final RestauranteRepository restaurantes = applicationContext.getBean(RestauranteRepository.class);

        final Restaurante restaurante = restaurantes.porId(1L);

        System.out.println(restaurante.getNome());
        System.out.println(restaurante.getTaxaFrete());

    }
}
```

3.17. Mapeando relacionamento com @ManyToOne

terça-feira, 7 de fevereiro de 2023 22:42

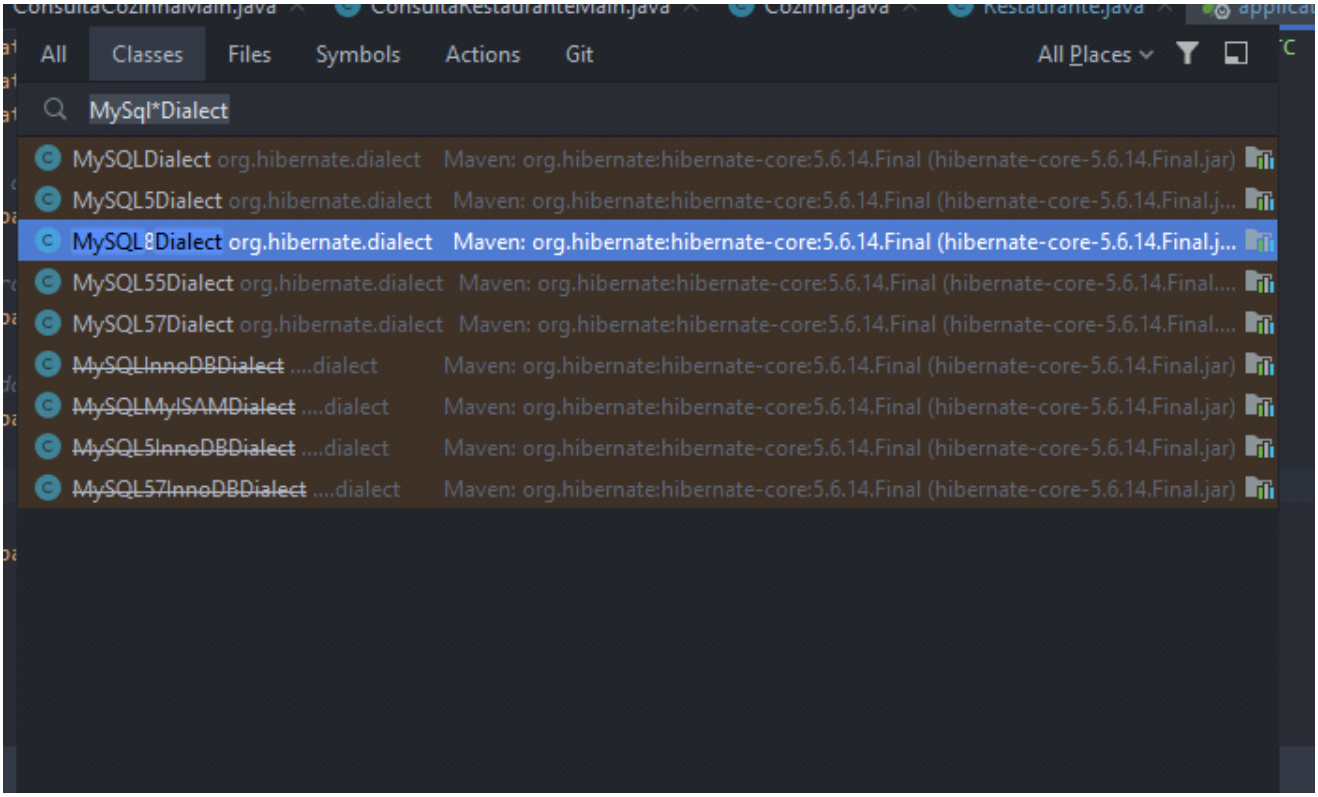


Vamos fazer o relacionamento entre Restaurante e Cozinha.

```java
@Data
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Restaurante {

    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) //provedor de persistencia
    private Long Id;

    private String nome;

    @Column(name = "taxa_frete")
    private BigDecimal taxaFrete;

    @ManyToOne
    private Cozinha cozinha;

}
```

Anotando a propriedade Cozinha com @ManyToOne

- Dialeto do Mysql no Hibernate




```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

import.sql

No data sources are configured to run this SQL and provide advanced code assistance.

```
insert into cozinha (id, nome) values (1, 'Tailandesa');
insert into cozinha (id, nome) values (2, 'Indiana');

insert into restaurante (nome, taxa_frete, cozinha_id) values ('Thai Gourmet', 10.00, 1);
insert into restaurante (nome, taxa_frete, cozinha_id) values ('Thai Delivery', 9.50, 1);
insert into restaurante (nome, taxa_frete, cozinha_id) values ('Tuk Tuk Comida Indiana', 15.00, 2);
```

```
public class ConsultaRestauranteMain {
    public static void main(String[] args) {

        ApplicationContext applicationContext = new SpringApplicationBuilder(AlgafoodApiApplication.class)
                .web(WebApplicationType.NONE)
                .run(args);

        final RestauranteRepository restaurantes = applicationContext.getBean(RestauranteRepository.class);

        final List<Restaurante> todos = restaurantes.todos();

        for (Restaurante r : todos) {
            System.out.printf("Nome: %s Cozinha: %s Taxa Frete: %.2f\n", r.getNome(), r.getCozinha().getNome(), r.getTaxaFrete());
        }


    }
}
```

3.18. A anotação @JoinColumn

quarta-feira, 8 de fevereiro de 2023 08:40

Essa notação permite a mudança no nome da tabela estrangeira quando o atributo é anotado com alguma notação de relacionamento (ManyToOne, ManyToMany...)

```
    @ManyToOne
    @JoinColumn(name = "cozinha_codigo")
    private Cozinha cozinha;

}
```

```
+----------------+---------------+------+-----+---------+----------------+
| Field          | Type          | Null | Key | Default | Extra          |
+----------------+---------------+------+-----+---------+----------------+
| id             | bigint        | NO   | PRI | NULL    | auto_increment |
| nome           | varchar(255)  | YES  |     | NULL    |                |
| taxa_frete     | decimal(19,2) | YES  |     | NULL    |                |
| cozinha_codigo | bigint        | YES  | MUL | NULL    |                |
+----------------+---------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

3.19. Propriedade nullable de @Column e @JoinColumn

quarta-feira, 8 de fevereiro de 2023 08:48

Criar colunas not null funcionam tanto na anotação Column quanto JoinColumn, mas se o script sql for feito de outra forma o JPA pode sobrescrever de acordo com a forma de execução do hibernate ddl.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY) //provedor de persistencia
private Long Id;

@Column(nullable = false)
private String nome;

@Column(name = "taxa_frete", nullable = false)
private BigDecimal taxaFrete;

@ManyToOne
@JoinColumn(name = "cozinha_id", nullable = false)
private Cozinha cozinha;
```

essas propriedades podem influenciar a geração de JSON.

3.20. Desafio mapeando entidades

quarta-feira, 8 de fevereiro de 2023 09:15

Criar as entidades junto com seus repositórios e classes de teste: Cidade, Estado, FormaPagamento e Permissao a partir do diagrama

Resolução do desafio

1. Adicionando restrições em Restaurante.

Primeiro, vamos adicionar restrições para não permitir valores nulos nos atributos de restaurante

```
@Column(nullable = false)
private String nome;

@Column(name = "taxa_frete", nullable = false)
private BigDecimal taxaFrete;

@ManyToOne
@JoinColumn(name = "cozinha_id", nullable = false)
private Cozinha cozinha;
```

2. Criando as entidades

Nessa etapa, iremos seguir o diagrama e criar nossas entidades baseadas nele.

Assim como fizemos anteriormente, adicionaremos as anotações do JPA e do Lombok

```
Estado
  @Data
  @EqualsAndHashCode(onlyExplicitlyIncluded = true)
  @Entity
  public class Estado {

    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String nome;

  }
```

Cidade

```
@Data
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Cidade {

    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String nome;

    @ManyToOne
    @JoinColumn(nullable = false)
    private Estado estado;
}
```

FormaPagamento

```
@Data
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class FormaPagamento {

    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String descricao;
}
```

Permissao

```
@Data
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Permissao {

    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String nome;

    @Column(nullable = false)
```

```
    private String descricao;

}
```

3. Criando as interfaces dos repositórios

Agora, vamos criar as interfaces dos repositórios

Utilizaremos os mesmos métodos, como feito anteriormente

EstadoRepository

```
public interface EstadoRepository {

    List<Estado> listar();
    Estado buscar(Long id);
    Estado salvar(Estado estado);
    void remover(Estado estado);

}
```

CidadeRepository

```
public interface CidadeRepository {

    List<Cidade> listar();
    Cidade buscar(Long id);
    Cidade salvar(Cidade cidade);
    void remover(Cidade cidade);

}
```

FormaPagamentoRepository

```
public interface FormaPagamentoRepository {

    List<FormaPagamento> listar();
    FormaPagamento buscar(Long id);
    FormaPagamento salvar(FormaPagamento formaPagamento);
    void remover(FormaPagamento formaPagamento);

}
```

PermissaoRepository

```
public interface PermissaoRepository {

    List<Permissao> listar();
    Permissao buscar(Long id);
    Permissao salvar(Permissao permissao);
    void remover(Permissao permissao);

}
```

4. Implementando os repositórios

Chegou a hora de implementarmos as interfaces que acabamos de criar

```
EstadoRepositoryImpl
  @Component
  public class EstadoRepositoryImpl implements EstadoRepository {

    @PersistenceContext
    private EntityManager manager;

    @Override
    public List<Estado> listar() {
      return manager.createQuery("from Estado", Estado.class)
          .getResultList();
    }

    @Override
    public Estado buscar(Long id) {
      return manager.find(Estado.class, id);
    }

    @Transactional
    @Override
    public Estado salvar(Estado estado) {
      return manager.merge(estado);
    }

    @Transactional
    @Override
    public void remover(Estado estado) {
      estado = buscar(estado.getId());
      manager.remove(estado);
    }

  }

CidadeRepositoryImpl
  @Component
  public class CidadeRepositoryImpl implements CidadeRepository {

    @PersistenceContext
    private EntityManager manager;

    @Override
    public List<Cidade> listar() {
      return manager.createQuery("from Cidade", Cidade.class)
          .getResultList();
```

```
    }

    @Override
    public Cidade buscar(Long id) {
      return manager.find(Cidade.class, id);
    }

    @Transactional
    @Override
    public Cidade salvar(Cidade cidade) {
      return manager.merge(cidade);
    }

    @Transactional
    @Override
    public void remover(Cidade cidade) {
      cidade = buscar(cidade.getId());
      manager.remove(cidade);
    }
  }
```

FormaPagamentoRepositoryImpl

```
  @Component
  public class FormaPagamentoRepositoryImpl implements
FormaPagamentoRepository {

    @PersistenceContext
    private EntityManager manager;

    @Override
    public List<FormaPagamento> listar() {
      return manager.createQuery("from FormaPagamento",
FormaPagamento.class)
          .getResultList();
    }

    @Override
    public FormaPagamento buscar(Long id) {
      return manager.find(FormaPagamento.class, id);
    }

    @Transactional
    @Override
    public FormaPagamento salvar(FormaPagamento formaPagamento) {
      return manager.merge(formaPagamento);
    }

    @Transactional
    @Override
```

```
    public void remover(FormaPagamento formaPagamento) {
      formaPagamento = buscar(formaPagamento.getId());
      manager.remove(formaPagamento);
    }

  }
```

PermissaoRepositoryImpl

```
  @Component
  public class PermissaoRepositoryImpl implements PermissaoRepository {

    @PersistenceContext
    private EntityManager manager;

    @Override
    public List<Permissao> listar() {
      return manager.createQuery("from Permissao", Permissao.class)
          .getResultList();
    }

    @Override
    public Permissao buscar(Long id) {
      return manager.find(Permissao.class, id);
    }

    @Transactional
    @Override
    public Permissao salvar(Permissao permissao) {
      return manager.merge(permissao);
    }

    @Transactional
    @Override
    public void remover(Permissao permissao) {
      permissao = buscar(permissao.getId());
      manager.remove(permissao);
    }
  }
```

5. Criando as classes para testarmos

Chegou a hora de validarmos se tudo correu bem, pra isso criaremos as seguintes classes

ConsultaCidadeMain

```
  public class ConsultaCidadeMain {

    public static void main(String[] args) {
      ApplicationContext applicationContext = new
```

```
SpringApplicationBuilder(AlgafoodApiApplication.class)
            .web(WebApplicationType.NONE)
            .run(args);

        CidadeRepository cidadeRepository =
applicationContext.getBean(CidadeRepository.class);

        List<Cidade> todasCidades = cidadeRepository.listar();

        for (Cidade cidade : todasCidades) {
          System.out.printf("%s - %s\n", cidade.getNome(),
cidade.getEstado().getNome());
        }
    }

  }
```

ConsultaFormaPagamentoMain

```
  public class ConsultaFormaPagamentoMain {

    public static void main(String[] args) {
        ApplicationContext applicationContext = new
SpringApplicationBuilder(AlgafoodApiApplication.class)
            .web(WebApplicationType.NONE)
            .run(args);

        FormaPagamentoRepository formaPagamentoRepository =
applicationContext.getBean(FormaPagamentoRepository.class);

        List<FormaPagamento> todasFormasPagamentos =
formaPagamentoRepository.listar();

        for (FormaPagamento formaPagamento : todasFormasPagamentos) {
          System.out.println(formaPagamento.getDescricao());
        }
    }

  }
```

ConsultaPermissaoMain

```
  public class ConsultaPermissaoMain {

    public static void main(String[] args) {
        ApplicationContext applicationContext = new
SpringApplicationBuilder(AlgafoodApiApplication.class)
            .web(WebApplicationType.NONE)
            .run(args);

        PermissaoRepository permissaoRepository =
```

```
applicationContext.getBean(PermissaoRepository.class);

        List<Permissao> todasPermissoes = permissaoRepository.listar();

        for (Permissao permissao : todasPermissoes) {
          System.out.printf("%s - %s\n", permissao.getNome(),
permissao.getDescricao());
        }
    }

  }
```

6. Adicionando dados de restaurantes

Aqui, iremos alterar o arquivo import.sql e adicionaremos comandos insert para popularmos as tabelas referentes as nossas novas entidades.

Mas primeiro, iremos forçar os IDs dos restaurantes para usarmos nos relacionamentos

```
  insert into restaurante (id, nome, taxa_frete, cozinha_id) values (1, 'Thai Gourmet',
10, 1);
  insert into restaurante (id, nome, taxa_frete, cozinha_id) values (2, 'Thai Delivery',
9.50, 1);
  insert into restaurante (id, nome, taxa_frete, cozinha_id) values (3, 'Tuk Tuk Comida
Indiana', 15, 2);
```

Com esses IDs podemos, continuar a popular nossas tabelas

```
  insert into estado (id, nome) values (1, 'Minas Gerais');
  insert into estado (id, nome) values (2, 'São Paulo');
  insert into estado (id, nome) values (3, 'Ceará');

  insert into cidade (id, nome, estado_id) values (1, 'Uberlândia', 1);
  insert into cidade (id, nome, estado_id) values (2, 'Belo Horizonte', 1);
  insert into cidade (id, nome, estado_id) values (3, 'São Paulo', 2);
  insert into cidade (id, nome, estado_id) values (4, 'Campinas', 2);
  insert into cidade (id, nome, estado_id) values (5, 'Fortaleza', 3);

  insert into forma_pagamento (id, descricao) values (1, 'Cartão de crédito');
  insert into forma_pagamento (id, descricao) values (2, 'Cartão de débito');
  insert into forma_pagamento (id, descricao) values (3, 'Dinheiro');

  insert into permissao (id, nome, descricao) values (1, 'CONSULTAR_COZINHAS',
'Permite consultar cozinhas');
  insert into permissao (id, nome, descricao) values (2, 'EDITAR_COZINHAS',
'Permite editar cozinhas');
```