CADERNO CURSO ESPECIALISTA SPRING REST - ALGAWORKS

**SUMÁRIO**

[1 SPRING E INJEÇÃO DE DEPENDÊNCIAS 6](#_Toc102742361)

[1.1 CRIANDO UM PROJETO SPRING BOOT COM SPRING INITIALIZR 6](#_Toc102742362)

[1.2 Arquivo mvnw.cmd 8](#_Toc102742363)

[1.3 Configurando o boot automático do servidor Spring Boot 8](#_Toc102742364)

[1.4 Spring IoC Container 10](#_Toc102742365)

[1.5 Usando @Configuration e @Bean para definir beans 11](#_Toc102742366)

[1.6 Pontos de injeção 13](#_Toc102742367)

[1.7 Dependência opcional 13](#_Toc102742368)

[1.8 Problema de ambiguidade nos Beans 13](#_Toc102742369)

[1.8.1 Desambiguação de beans com @Primary 13](#_Toc102742370)

[1.8.2 Desambiguação de beans com @Qualifier 14](#_Toc102742371)

[1.9 Desambiguação de beans com anotação customizada 14](#_Toc102742372)

[1.10 Mudando comportamento da aplicação com Spring Profiles 15](#_Toc102742373)

[1.11 Criando métodos de callback do ciclo de vida dos beans 16](#_Toc102742374)

[1.12 Publicando e consumindo eventos customizados 16](#_Toc102742375)

[1.13 Configurando projetos com application.properties 17](#_Toc102742376)

[1.14 Criando e acessando propriedades customizadas com @Value 17](#_Toc102742377)

[2 INTRODUÇÃO AO JPA E HIBERNATE 18](#_Toc102742378)

[2.1 O que é JPA e Hibernate 18](#_Toc102742379)

[2.2 Adicionando Spring Data JPA no pom.xml 18](#_Toc102742380)

[2.3 Importando dados de teste com import.sql 18](#_Toc102742381)

[2.4 Padrão DDD Aggregate 19](#_Toc102742382)

[2.5 Projeto Lombok 20](#_Toc102742383)

[2.6 Escondendo código boilerplate com Lombok 20](#_Toc102742384)

[2.7 A annotation @PersistenceContext 21](#_Toc102742385)

[2.8 Annotations de relacionamentos de bancos de dados 21](#_Toc102742386)

[2.8.1 @ManyToOne 21](#_Toc102742387)

[2.9 Suporte ao Foreign Key do motor SQL 22](#_Toc102742388)

[2.10 Utilizando o @JoinColumn 23](#_Toc102742389)

[2.11 Propriedade nullable de @Column e @JoinColumn 23](#_Toc102742390)

[3 ATALHOS DO INTELLIJ IDEA 24](#_Toc102742391)

[3.1 Geral 24](#_Toc102742392)

[3.2 Na aba de projetos 24](#_Toc102742393)

[3.3 Templates de código 24](#_Toc102742394)

[3.4 Na aba de código 24](#_Toc102742395)

[4 REST COM SPRING 26](#_Toc102742396)

[4.1 O que é REST? 26](#_Toc102742397)

[4.2 Constraints do REST 27](#_Toc102742398)

[4.3 Diferença entre REST e RESTful 27](#_Toc102742399)

[4.4 Conhecendo o protocolo HTTP 27](#_Toc102742400)

[4.5 Fazendo testes de requisição HTTP com GnuTLS 28](#_Toc102742401)

[4.6 Resources REST 29](#_Toc102742402)

[4.6.1 Identificando Recursos REST 29](#_Toc102742403)

[4.7 Adicionando suporte a respostas XML 29](#_Toc102742404)

[4.8 Especificando o tipo de retorno nos métodos de requisição 29](#_Toc102742405)

[4.9 Consultando Singleton Resource com GET e @PathVariable 29](#_Toc102742406)

[4.10 Mudando a representação dos atributos com @JsonProperty 30](#_Toc102742407)

[4.11 Ignorando valores no retorno Json com @JsonIgnore 30](#_Toc102742408)

[4.12 Modificando o nome do recurso com @JsonRootName 30](#_Toc102742409)

[4.13 A annotation @NonNull 31](#_Toc102742410)

[4.14 Criando Wrapper de lista de objetos para modificar nomes de atributos XML 31](#_Toc102742411)

[4.15 Métodos HTTP 32](#_Toc102742412)

[4.15.1 Idempotência 32](#_Toc102742413)

[4.16 Códigos de status HTTP 35](#_Toc102742414)

[4.17 Definindo o status da resposta HTTP com @ResponseStatus 37](#_Toc102742415)

[4.18 Manipulando a resposta com @ResponseEntity 37](#_Toc102742416)

[4.19 Adicionando recursos com @PostMapping 38](#_Toc102742417)

[4.20 Atualização de um recurso com @PutMapping 38](#_Toc102742418)

[4.21 Exclusão de recurso com @DeleteMapping 38](#_Toc102742419)

[4.22 Atualização de recursos com PATCH utilizando Reflections 38](#_Toc102742420)

[4.23 Introdução ao Modelo de Maturidade de Richardson (RMM) 38](#_Toc102742421)

[5 SUPER PODERES DO SPRING DATA JPA 45](#_Toc102742422)

[5.1 Annotation @Repository 45](#_Toc102742423)

[5.2 Spring Data JPA 45](#_Toc102742424)

[5.2.1 Criando um repositório com Spring Data JPA (SDJ) 45](#_Toc102742425)

[5.2.2 Criando consultas com query methods 45](#_Toc102742426)

[5.2.3 Consultas utilizando JPQL 46](#_Toc102742427)

[5.2.4 Externalizando consultas JPQL em arquivo .xml 46](#_Toc102742428)

[5.2.5 Criando Interface avançada de repository 47](#_Toc102742429)

[5.2.6 Implementando consulta dinâmica com JPQL 49](#_Toc102742430)

[5.3 Implementando consulta com Criteria API 50](#_Toc102742431)

[5.3.1 Implementando Specifications 51](#_Toc102742432)

[5.3.1.1 Criando uma fábrica de specifications 52](#_Toc102742433)

[5.4 Utilizando specifications com repository interface 53](#_Toc102742434)

[5.4.1 Arrumando erro de Referência Circular com @Lazy 54](#_Toc102742435)

[5.5 Annotation @NoRepositoryBean 54](#_Toc102742436)

[5.6 Habilitando JPA Repositories 54](#_Toc102742437)

[6 EXPLORANDO MAIS DO JPA E HIBERNATE 55](#_Toc102742438)

[6.1 Mapeando relacionamento bidirecional com @OneToMany 55](#_Toc102742439)

[6.2 Mapeando relacionamento @ManyToMany 55](#_Toc102742440)

[6.3 Mapeando classes incorporáveis com @Embedded e @Embeddable 56](#_Toc102742441)

[6.4 Annotation @CreationTimestamp e @UpdateTimestamp 57](#_Toc102742442)

[6.5 Entendendo o Eager Loading 57](#_Toc102742443)

[6.5.1 Alterando a estratégia de fetching Eager para Lazy Loading 57](#_Toc102742444)

[6.6 Entendendo o Lazy Loading 58](#_Toc102742445)

[6.6.1 Alterando a estratégia de fetching Lazy para Eager Loading 58](#_Toc102742446)

[6.7 Resolvendo problema N+1 com fetch join na JPQL 59](#_Toc102742447)

[7 POOL DE CONEXÕES E FLYWAY 60](#_Toc102742448)

[7.1 Pool de conexão 60](#_Toc102742449)

[7.2 Conhecendo o Hikari solução Pool de conexões Springboot 62](#_Toc102742450)

[7.3 Flyway ferramenta de versionamento de schemas de banco de dados 62](#_Toc102742451)

[7.4 CRIANDO ARQUIVO DE POPULAÇÃO DE DADOS NO FLYWAY 63](#_Toc102742452)

[7.4.1 Nota - Adicionando arquivos untracked no git hub 64](#_Toc102742453)

[8 TRATAMENTO E MODELAGEM DE ERROS DA API 65](#_Toc102742454)

[8.1 Exceções customizadas com a annotation @ResponseStatus 65](#_Toc102742455)

[8.2 Lançando exceções do tipo ResponseStatusException 66](#_Toc102742456)

[8.2.1 Nota - Services com conhecimento de http status: 66](#_Toc102742457)

[8.3 Estendendo ResponseStatusException 67](#_Toc102742458)

[8.4 Tratando exceções dentro do controlador com @ExceptionHandler 67](#_Toc102742459)

[8.5 Tratando exceções globais com @ExceptionHandler e @ControllerAdvice 69](#_Toc102742460)

[8.6 Estendendo com a classe ResponseEntityExceptionHandler 70](#_Toc102742461)

[8.7 Customizando a resposta de ResponseEntityExceptionHandler 70](#_Toc102742462)

[8.8 Conhecendo a RFC 7807 (Problem Details for HTTP APIs) 72](#_Toc102742463)

[8.9 Padronizando o formato no corpo com a RFC 7807 73](#_Toc102742464)

[8.10 Ocultando valores nulos no corpo da resposta HTTP 74](#_Toc102742465)

[8.11 Tratando a causa da exception na desserialização 74](#_Toc102742466)

[8.12 Habilitar erros na desserialização de propriedades null, ignored 76](#_Toc102742467)

[8.13 Habilitando o stacktrace da exception NoHandlerFoundException 77](#_Toc102742468)

[9 VALIDAÇÕES COM BEAN VALIDATION 78](#_Toc102742469)

[9.1 Adicionando constraint e validando com @Valid 78](#_Toc102742470)

[9.2 Estendendo o Problem Details para adicionar as propriedades com constraints violadas 79](#_Toc102742471)

[9.3 Conhecendo mais constraints de validação 80](#_Toc102742472)

[9.4 Validando em Cascata 80](#_Toc102742473)

[9.5 Criando grupos de validação 80](#_Toc102742474)

[9.6 Convertendo grupos de constraints para @ConvertGroup 81](#_Toc102742475)

[9.7 Formas de customizar mensagens de constraints 82](#_Toc102742476)

[9.8 Criando constraints de validação customizadas usando composição 84](#_Toc102742477)

[9.8.1 Conflito entre Bean Validation do Javax entre o Spring 85](#_Toc102742478)

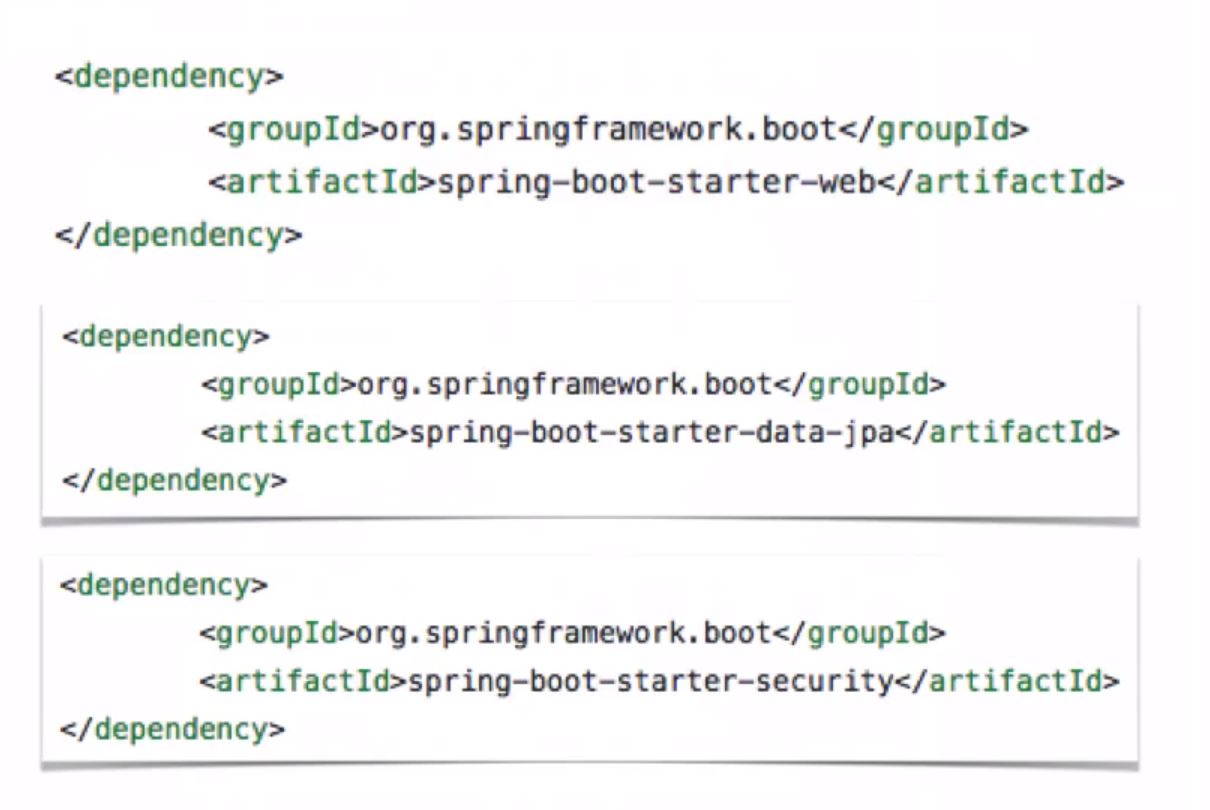
[9.9 Constraints de validação customizadas com implementação de ConstraintValidator 85](#_Toc102742479)

[9.10 Constraints de validação customizadas em nível de classe 87](#_Toc102742480)

[10 TESTES DE INTEGRAÇÃO 90](#_Toc102742481)

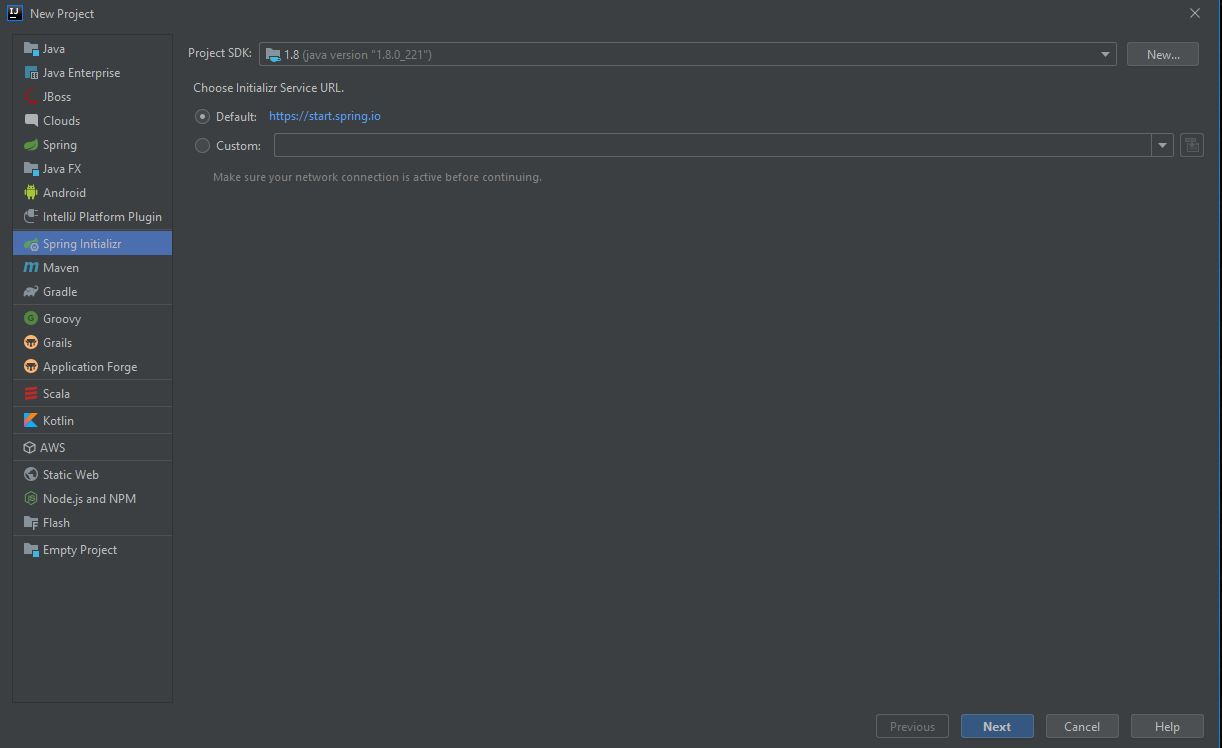
# SPRING E INJEÇÃO DE DEPENDÊNCIAS

O Spring Boot oferece os starters, que são dependências que agrupam outras dependências, utilizando no arquivo pom.xml do Maven.

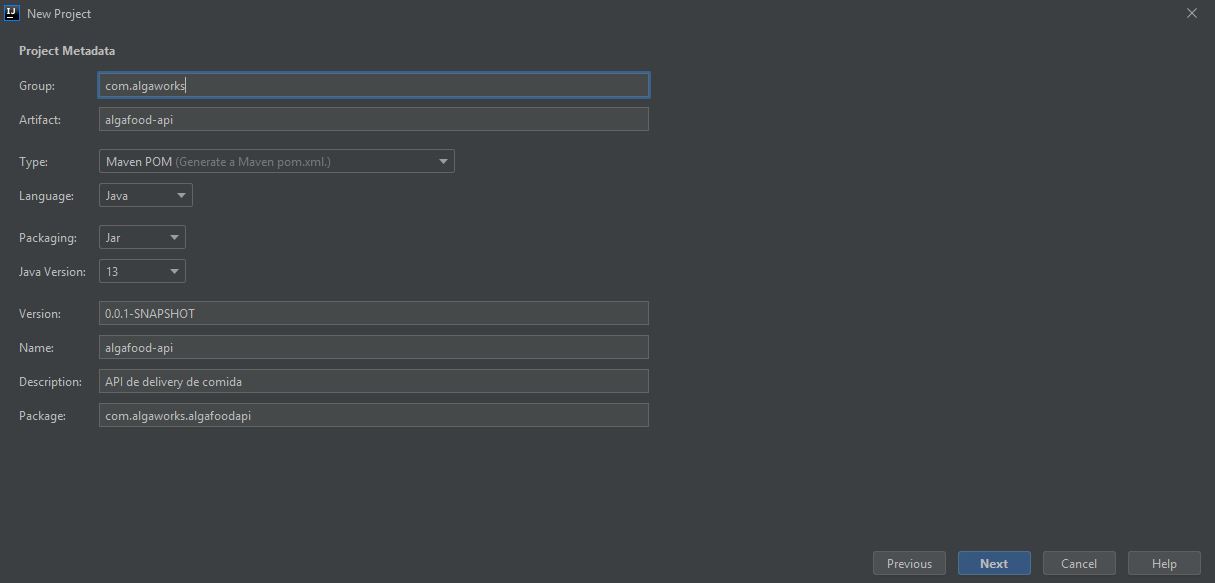


## CRIANDO UM PROJETO SPRING BOOT COM SPRING INITIALIZR

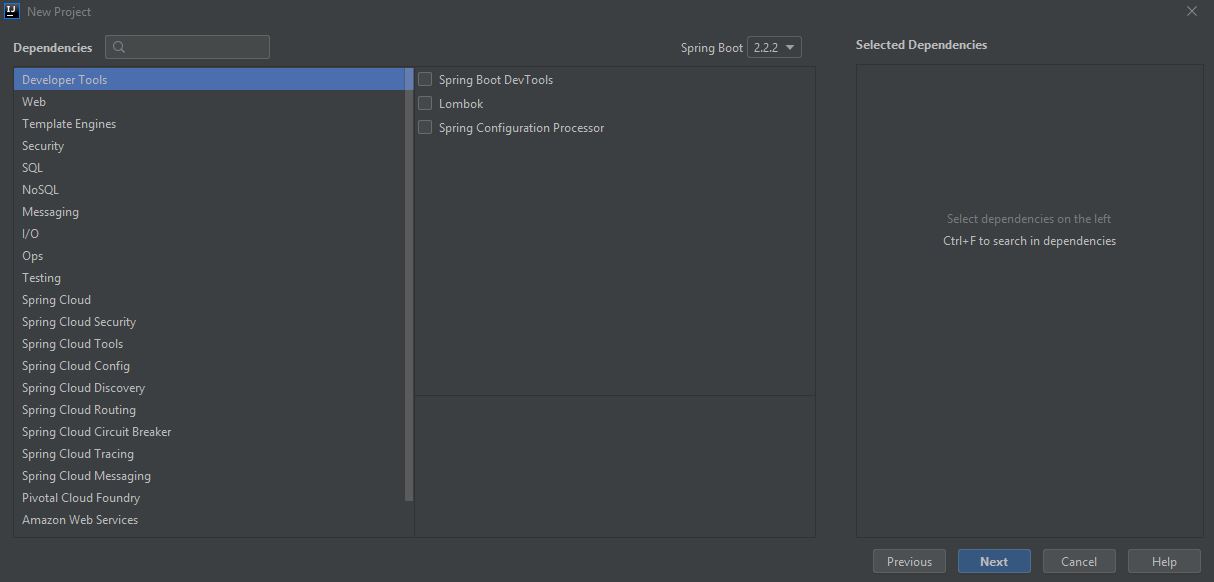
Neste curso o Thiago utiliza o STS (Spring Tool Suite) mas eu vou utilizar o IntellijIDEA, para iniciarmos um projeto Spring no Intellij basta criarmos um novo projeto:



Iremos utilizar o Spring boot, por isso, utilizamos o Spring Initializr para configurarmos o projeto. Na tela da imagem acima você escolhe o SDK que vai trabalhar e a url de serviço que irá conectar para baixar as dependências do Spring, por padrão deixe como está, pois é a url padrão para download das dependências do Spring.

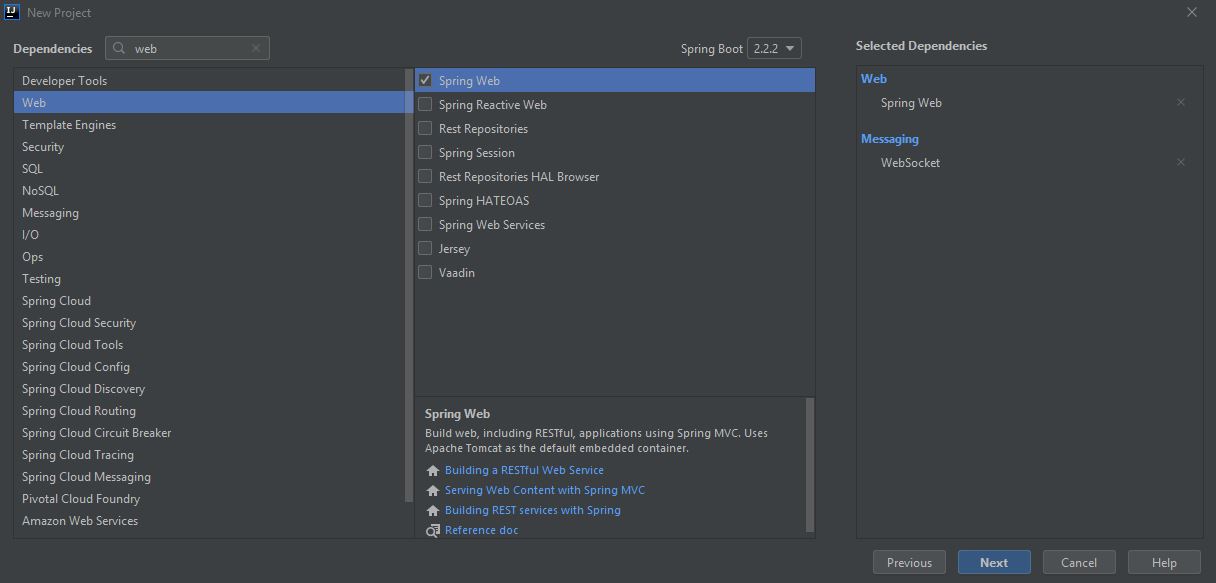


Na tela acima temos a configuração do projeto, nome, group id, linguagem, versão do Java, nome do pacote, etc. Ajuste como necessitar e siga em frente.



Nesta tela acima você vai selecionar a versão do Spring Boot, não se preocupe se a versão utilizada no treinamento é mais antiga e não consta no select box, depois da para alterar a versão utilizada pelo pom.xml e obter a versão necessária para o seu projeto.

É nesta tela que selecionamos os projetos do Spring que queremos trabalhar, podemos selecionar o Web e o Spring Web, se quiser pode adicionar outras dependências mais tarde pelo pom.xml, não é necessário adicionar tudo de uma vez nesta tela.



Após clicar em next o projeto vai começara a ser configurado, e vai baixar as dependências automaticamente.

## Arquivo mvnw.cmd

O arquivo mvnw.cmd é utilizado para rodar o Maven no terminal do Windows e o outro mvnw é utilizado para rodar o Maven no terminal do Mac e do Linux, com isso você pode gerar o build do projeto, o arquivo .jar sem precisar de uma IDE para isso, pode fazer configurações pelo terminal.

Como exemplo, podemos olhar a arvore de dependências do projeto pelo prompt de comando, com o comando:

mvnw dependency:tree

E para visualizar a arvore de dependências resolvidas, podemos utilizar o comando:

mvnw dependency:resolve

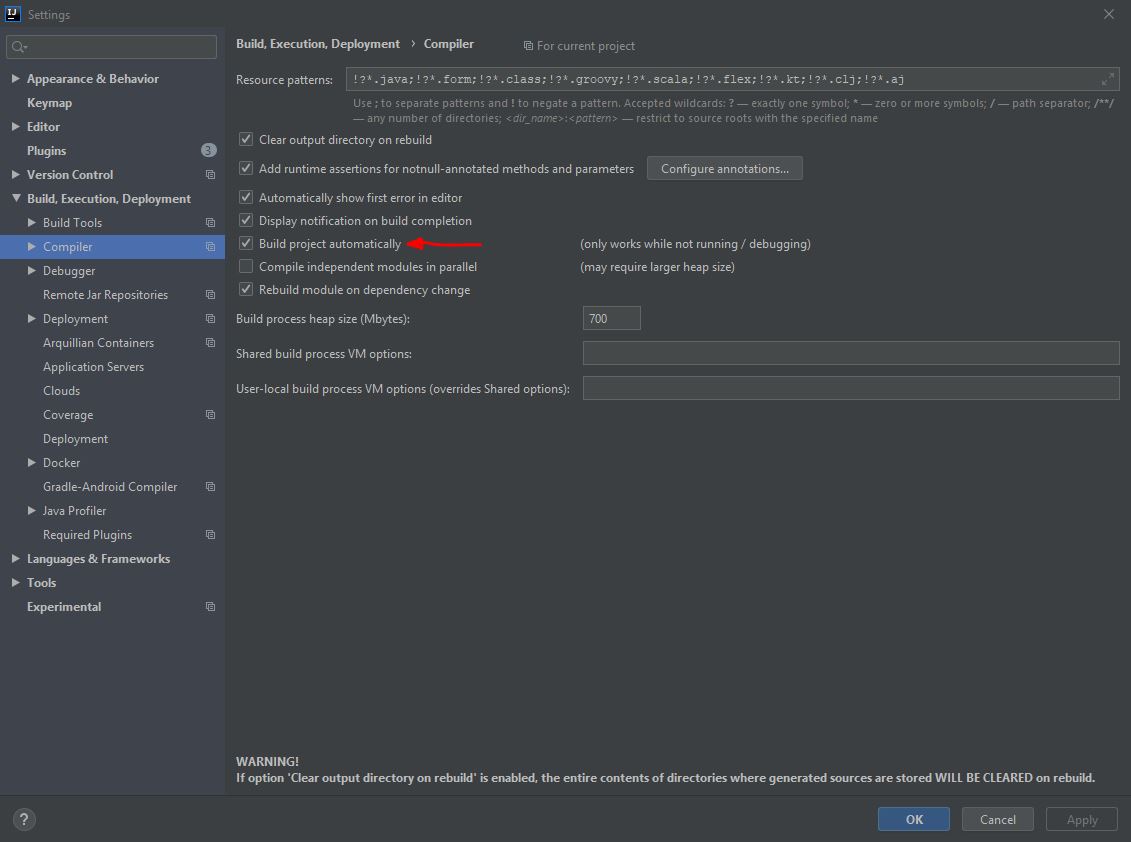
## Configurando o boot automático do servidor Spring Boot

Para que o processo de desenvolvimento de uma aplicação seja mais rápido na hora de executar o servidor depois de uma modificação nos arquivos, para que não precisemos reiniciar o servidor toda vez após uma alteração, podemos utilizar o Spring DevTools, ele permite que o servidor dê o boot automaticamente após uma alteração. Na IDE Intellij é necessário fazer algumas modificações para permitir o uso desta ferramenta, veja como:

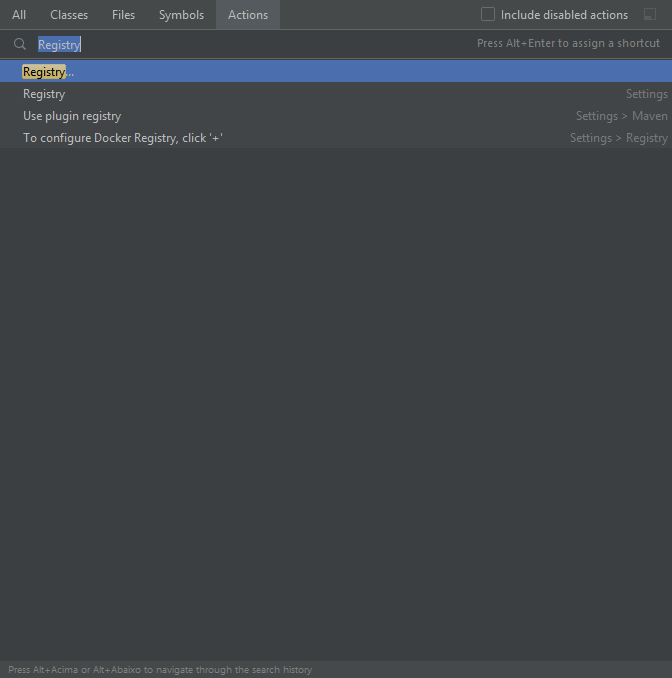
Primeiro adicione esta dependência no pom.xml:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>  
</dependency>

Após abra as configurações da IDE Intellij no caminho: Settings / Build, Execution, Deployment / Compiler. E ative o Build project automatically.



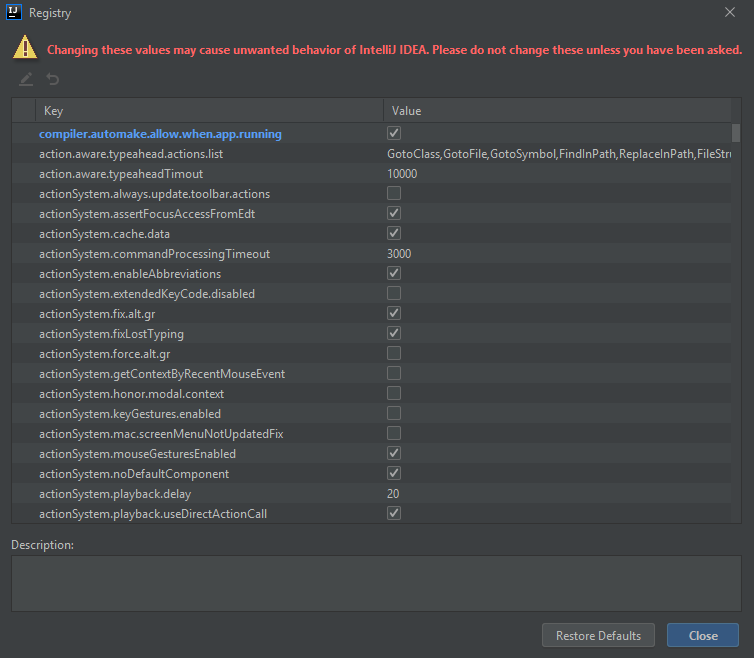
E após isso, é necessário alterar o registro da IDE, para entrar no registro aperte (Ctrl + Shift + A). E pesquise por Registry.



Após você vai entrar em uma janela de registros, procure por:

compiler.automake.allow.when.app.running

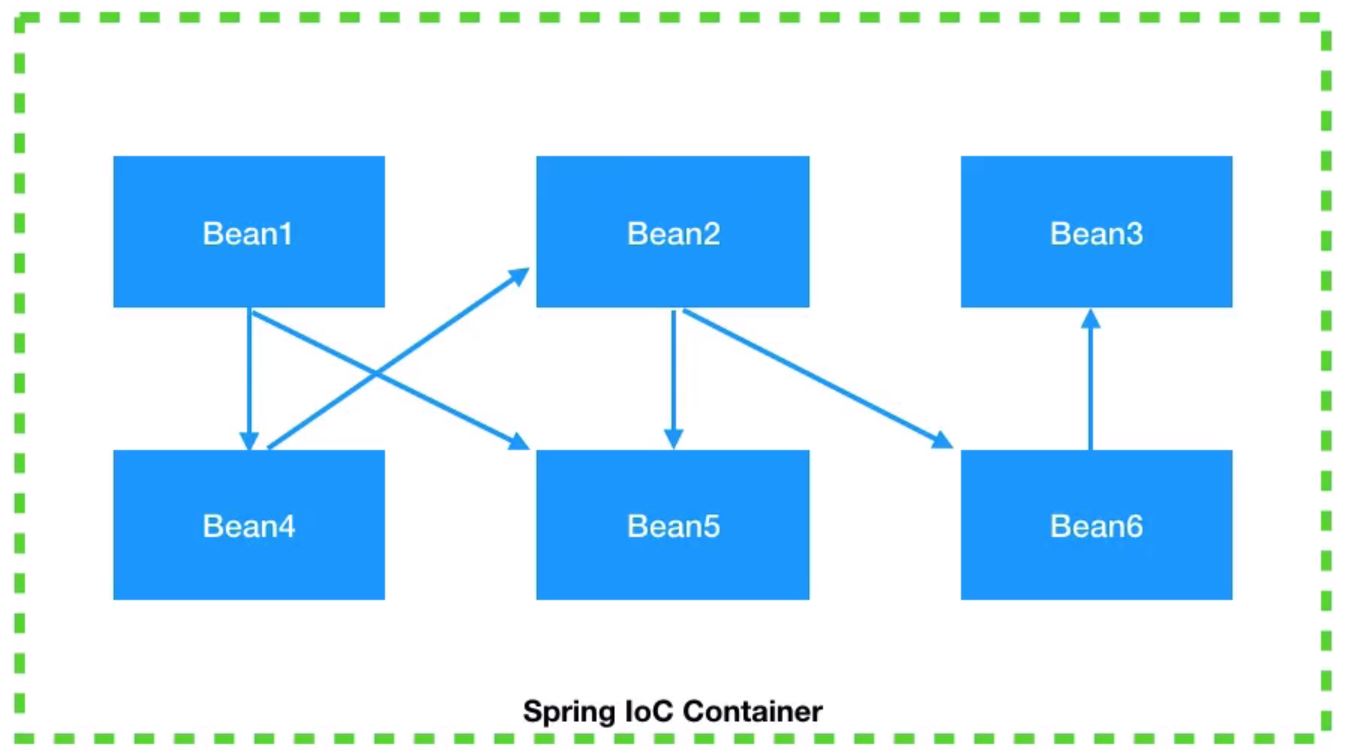
Selecione esse registro marcando o checkbox, feche no close e pronto. Feito isso agora será possível desenvolver sem precisar de reiniciar o servidor manualmente.



O DevTools é usado somente durante o desenvolvimento do projeto.

## Spring IoC Container

O Spring IoC Container é o container de controle de injeção de dependências do Spring.



Beans são objetos gerenciados pelo container do Spring.

Para configurar uma classe como um objeto bean do Spring utilizamos a annotation @Component, com essa annotation a classe se transforma em um bean Spring.

import org.springframework.stereotype.Component;  
  
@Component  
public class NotificadorEmail {  
  
 public void notificar(Cliente cliente, String mensagem) {  
 System.*out*.printf("Notificando %s através do e-mail %s: %s\n",  
 cliente.getNome(), cliente.getEmail(), mensagem);  
 }  
}

## Usando @Configuration e @Bean para definir beans

As vezes é necessário definir um bean com configurações especiais, para isso utilizamos a annotation @Configuration, com isso podemos criar uma classe com a annotation @Configuration para que sirva para criar beans, dentro da classe criamos métodos que recebem a annotation @Bean, que seta que o método é um bean, veja o código de exemplo:

import com.algaworks.algafoodapi.notificacao.NotificadorEmail;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class AlgaConfig {

*/\*\*  
 \* Dentro do IoC Container, por padrão o bean será nomeado como "notificadorEmail",   
 \* que é exatamente o nome do método de definição do bean.  
 \** ***@return*** *\*/* @Bean  
 public NotificadorEmail notificadorEmail(){  
 NotificadorEmail notificador = new NotificadorEmail("smtp.algamail.com.br");  
 notificador.setCaixaAlta(true);  
 return notificador;  
 }  
}

Podemos utilizar um bean na própria classe configuration para inicializar uma dependência de bean, veja:

@Configuration  
public class AlgaConfig {  
  
 */\*\*  
 \* Dentro do IoC Container, por padrão o bean será nomeado como "notificadorEmail",   
 \* que é exatamente o nome do método de definição do bean.  
 \** ***@return*** *\*/* @Bean  
 public NotificadorEmail notificadorEmail(){  
 NotificadorEmail notificador = new NotificadorEmail("smtp.algamail.com.br");  
 notificador.setCaixaAlta(true);  
 return notificador;  
 }  
   
 @Bean  
 public AtivacaoClienteService ativacaoClienteService(){  
 return new AtivacaoClienteService(notificadorEmail());  
 }  
}

Para utilizar uma dependência de bean que não está na mesma classe de configuration ou está em uma classe componente, basta passar como parâmetro o objeto gerenciado pelo Spring, que irá entender que o objeto pode ser injetado.

Classe de configuração 1:

@Configuration  
public class NotificacaoConfig {  
  
 @Bean  
 public NotificadorEmail notificadorEmail(){  
 NotificadorEmail notificador = new NotificadorEmail("smtp.algamail.com.br");  
 notificador.setCaixaAlta(true);  
 return notificador;  
 }  
}

Classe de configuração 2:

@Configuration  
public class ServiceConfig {  
  
 @Bean  
 public AtivacaoClienteService ativacaoClienteService(Notificador notificador){  
 return new AtivacaoClienteService(notificador);  
 }  
}

## Pontos de injeção

O construtor da classe é um ponto de injeção.

Os métodos seters são um ponto de injeção

O atributo é um ponto de injeção.

Utilizando a annotation @Autowired é possível setar para o Spring que ele injete no método, construtor ou atributo, ele vai dar preferência para quem estiver com a annotation @Autowired, um caso de exemplo seria uma classe com dois construtores se sobrescrevendo, neste caso é necessário anotar o método que deseja que seja inicializado pelo Spring, utilizando o @Autowired.

## Dependência opcional

Podemos setar que uma dependência seja opcional adicionando um atributo para a annotation @Autowired:

@Autowired(required = false)

Private Notificador notificado;

Com o atributo required = false definimos que caso não tenha o objeto para ser injetado a classe vai ser compilada da mesma forma, mas se tiver ela vai ser utilizada.

## Problema de ambiguidade nos Beans

Caso a aplicação possua mais de um bean e você tenha injetado a interface que vai receber os beans, o Spring não vai conseguir lidar com isso e vai acontecer um erro, porque existe mais de um bean e o Spring suporta apenas um Single bean, podemos resolver este problema de ambiguidade de diferentes formas, uma delas é criando uma lista de beans, com o uso de uma lista o Spring ira utilizar todas as implementações da interface que você deseja injetar.

### Desambiguação de beans com @Primary

Caso você não vá utilizar uma lista você pode utilizar a annotation @Primary para configurar que a implementação seja injetada por preferência, o Spring irá utilizar esta implementação por prioridade.

### Desambiguação de beans com @Qualifier

E podemos utilizar a annotation @Qualifier para qualificar os beans, damos um nome, veja no exemplo:

@Qualifier("urgente")  
@Component  
public class NotificadorEmail implements Notificador {  
   
 @Override  
 public void notificar(Cliente cliente, String mensagem) {  
 System.*out*.printf("Notificando %s através do e-mail %s usando SMTP %s: %s\n",  
 cliente.getNome(), cliente.getEmail(), mensagem);  
 }  
}

Com o bean anotado acima chamamos a annotation na classe que está injetando o bean.

@Component  
public class AtivacaoClienteService {  
   
 @Qualifier("urgente")  
 @Autowired  
 private Notificador notificador;  
  
 @Autowired  
 public AtivacaoClienteService(Notificador notificador) {  
 this.notificador = notificador;  
 }  
  
 public void ativar(Cliente cliente){  
 cliente.ativar();  
 notificador.notificar(cliente, "Seu cadastro no sistema está ativo!");  
 }  
}

## Desambiguação de beans com anotação customizada

Podemos criar uma annotation customizada para podermos remover a ambiguidade dos múltiplos beans, a nova anotação vai funcionar como o qualifier mas com uma característica melhor, que é poder fazer refatoração, a annotation @Qualifier possui o parâmetro em String, portanto não é possível realizar refatoração caso precisemos refatorar e mudar o nome em todos os lugares que a propriedade estava, para resolver isto podemos criar uma annotation e utilizar enums como forma de parâmetro, veja o exemplo:

Aqui criamos uma annotation customizada que recebe um Enum como valor padrão, anotamos a nossa annotation com o @Qualifier transformando nossa anotação em um qualificador, Temos que utilizar a annotation @Retention para dizer quanto tempo essa anotação deve ficar sendo utilizada, incluímos como parâmetro RUNTIME que significa que esta anotação pode ser lida em tempo de execução.

@Retention(RetentionPolicy.*RUNTIME*)  
@Qualifier  
public @interface TipoDoNotificador {  
 NivelUrgencia value();  
}

E utilizamos a nossa anotação na classe que recebe a injeção de dependência:

@TipoDoNotificador(NivelUrgencia.*NORMAL*)  
@Component  
public class NotificadorEmail implements Notificador {  
  
 @Override  
 public void notificar(Cliente cliente, String mensagem) {  
 System.*out*.printf("Notificando %s através do e-mail %s usando SMTP %s: %s\n",  
 cliente.getNome(), cliente.getEmail(), mensagem);  
 }  
}

## Mudando comportamento da aplicação com Spring Profiles

A anotação @Profile é utilizada para definir o perfil de uma classe, bean do Spring, com ela nomeamos o perfil da classe que queremos. Isso é útil para conseguirmos modificar o uso das classes de nossa aplicação, que dependendo do contexto de utilização, por exemplo em desenvolvimento ou em produção podemos ter duas classes iguais mas com comportamentos diferentes, por exemplo, uma classe MOCK que possui métodos de teste que retornam objetos simulados diferentes da classe que está no ambiente de desenvolvimento, a sacada é separar os componentes da aplicação para determinado uso.

@Profile("dev")  
@TipoDoNotificador(NivelUrgencia.*NORMAL*)  
@Component  
public class NotificadorEmailMock implements Notificador{  
   
 @Override  
 public void notificar(Cliente cliente, String mensagem) {  
 System.*out*.printf("Notificando %s através do e-mail %s usando SMTP %s: %s\n",  
 cliente.getNome(), cliente.getEmail(), mensagem);  
 }  
}

Veja o código acima, ele está anotado com a annotation @Profile(“dev”) o nome dev é inventado pelo programador, quer dizer que esta classe é utilizada no ambiente de desenvolvimento. E para a gente inicializar, dizer para o Spring que ele deve reconhecer esta classe temos que editar o arquivo application.properties, veja:

spring.profiles.active=dev

Com isso o Spring vai iniciar as classes anotadas com o perfil “dev”, podemos adicionar diversos profiles na mesma linha apenas separando com virgula, veja:

spring.profiles.active=dev,mysql,s3

Veja que temos 3 possíveis perfis de comportamento, um de uso de cliente de banco de dados, e outro de armazenamento na Amazon S3.

## Criando métodos de callback do ciclo de vida dos beans

Um bean possui três fases de vida:

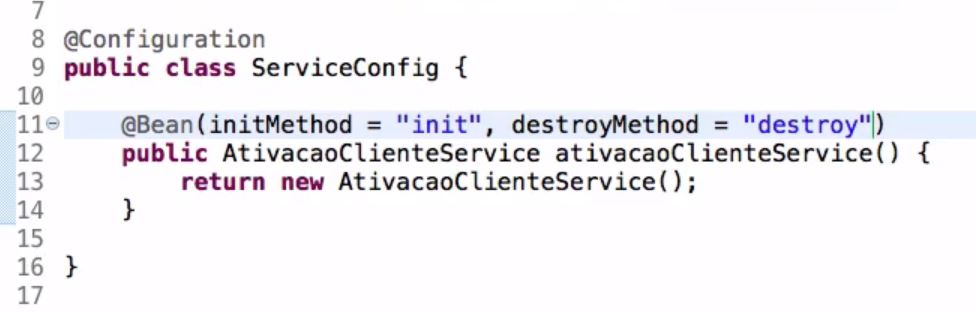
1. Inicialização
2. Uso
3. Destruição

Podemos capturar os momentos das fases do ciclo de vida de um bean do Spring, uma das formas de fazer isso é utilizando métodos com anotações @PostConstruct ou @PreDestroy, a criação da nomenclatura dos métodos deve ser feita com um padrão, não nomeie de qualquer jeito, veja um exemplo:

@PostConstruct  
public void init(){  
 System.*out*.println("INIT");  
}  
  
@PreDestroy  
public void destroy(){  
 System.*out*.println("DESTROY");  
}

Com a utilização destas anotações, você pode trabalhar em cima de cada momento e definir sua lógica de negócio para tratar o que deseja que aconteça em determinado momento.

Existe outra forma de fazer isso utilizando parâmetros na annotation @Bean



## Publicando e consumindo eventos customizados

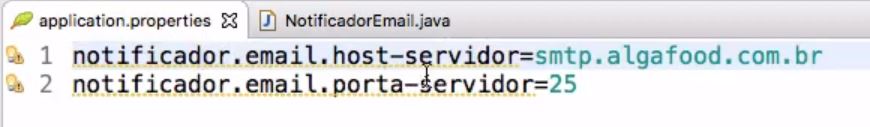
Voltar na aula 2.22 e escrever as anotações

## Configurando projetos com application.properties

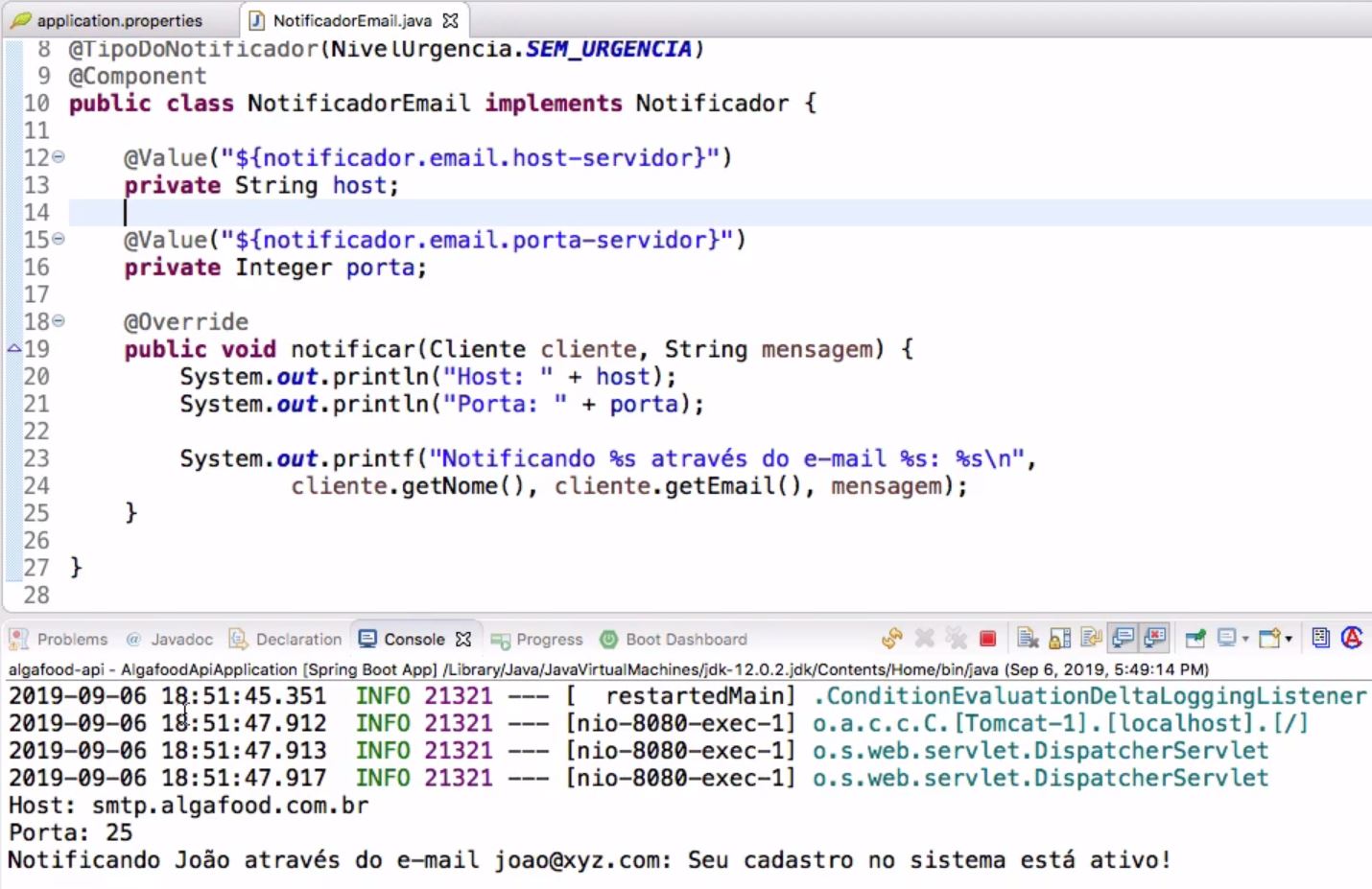
Neste link: <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html> você encontra a documentação com todos os tipos de parâmetros de properties que podem ser utilizadas no arquivo application.properties.

## Criando e acessando propriedades customizadas com @Value

Podemos criar propriedades customizadas no arquivo application.properties, veja:



Para acessarmos estes atributos, utilizamos a anotação @Value e passamos como parâmetro a chave da propriedade customizada do properties, em uma expression do Spring, veja:



# INTRODUÇÃO AO JPA E HIBERNATE

## O que é JPA e Hibernate

## Adicionando Spring Data JPA no pom.xml

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

Após isso é necessário adicionar um driver de banco de dados no pom.xml, basta escolher o banco que deseja utilizar no seu projeto, aqui vou incluir o mySQL, veja o código:

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.18</version>  
</dependency>

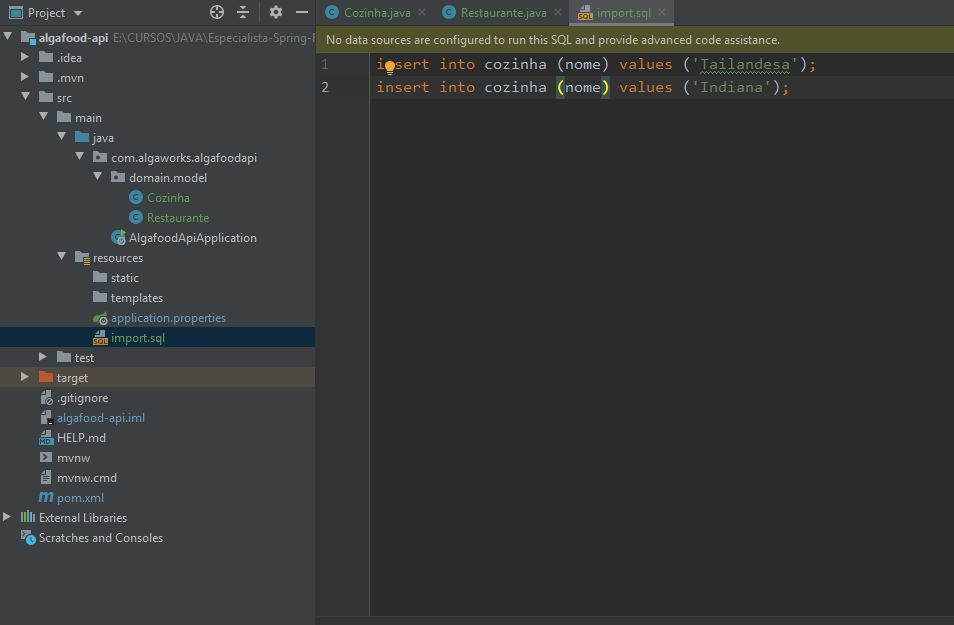
Para outros bancos de dados ou outras versões, basta pesquisar no site do Maven pela dependência do connector/J.

E após isso é necessário configurar o arquivo application.properties, inserindo estas configurações. O comando createDatabaseIfNotExist=true vai criar uma base de dados se ela não existir no mysql, e o serverTimezone=UTC é para evitar erros de localização do banco de dados MySQL. Você deve especificar o username e o password do seu banco de dados.

spring.datasource.url=jdbc:mysql://localhost/algafood?createDatabaseIfNotExist=true&serverTimezone=UTC  
spring.datasource.username=root  
spring.datasource.password=7515

## Importando dados de teste com import.sql

Em ambientes de teste podemos popular um banco de dados com um arquivo .sql com os ddl sql, inserts para inserir dados na tabela, esse arquivo tem que ser criado com o nome import.sql



Para evitar erros de transação ao fazer uma consulta com um método utilizando o JPQL, podemos anotar o método com a annotation @Transactional

@Transactional  
public Cozinha adicionar(Cozinha cozinha){  
 return manager.merge(cozinha);  
}

## Padrão DDD Aggregate

O padrão Aggregate do DDD é utilizado para tratar objetos de um domínio como uma única unidade, um exemplo para exemplificar é um pedido e seus itens de linha, esses são objetos separados, mas é útil tratar o pedido junto com os itens de linha como um único aggregate.

Um aggregate tem que ter um de seus objetos componentes como a raiz aggregate, qualquer referência externa deve apenas ir para a raiz aggregate, a raiz garante a integridade do agregado como um todo.

Aggregate é o elemento básico da transferência de armazenamento de dados, você solicita carregar ou salvar aggregates inteiros. As transações não devem cruzar fronteiras aggregate.

Às vezes, os DDD Aggregates são confundidos com as classes de coleção (lists, maps, etc). DDD Aggregates são conceitos de domínio (order, clinic visit, playlist), enquanto as coleções são genéricas. Um Aggregate geralmente contém coleções múltiplas, junto com campos simples. O termo "Aggregate" é comum e é usado em vários contextos diferentes (por exemplo, UML), caso em que não se refere ao mesmo conceito que um DDD Aggregate.

## Projeto Lombok

O Project Lombok é uma biblioteca java que se conecta automaticamente ao seu editor e cria ferramentas, apimentando o seu java. Nunca escreva outro método getter ou igual novamente, com uma anotação em que sua classe possui um construtor completo, automatize suas variáveis ​​de log e muito mais.

## Escondendo código boilerplate com Lombok

Podemos utilizar uma ferramenta chamada Lombok para diminuir os códigos boilerplates do Java, como os Getters e Setters, Equals and Hashcode e etc. Para isso você precisa adicionar no pom.xml a dependência do Lombok pelo Maven.

<dependencies>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.10</version>

<scope>provided</scope>

</dependency>

</dependencies>

E depois adicionar o plugin do Lombok na IDE que você está utilizando, para isso entre no site oficial do projeto Lombok e veja as instruções para a sua IDE preferida, no Intellij basta adicionar o plugin do Lombok no repositório de plugins do Intellij. Site oficial do Lombok: <https://projectlombok.org/>

Em um código normal do Java tem muitos boilerplates, e com a utilização do Lombok podemos deletar estes códigos e utilizar annotations referentes ao que desejamos substituir para diminuir o código, veja:

package com.algaworks.algafoodapi.domain.model;  
  
import lombok.EqualsAndHashCode;  
import lombok.Getter;  
import lombok.Setter;  
  
import javax.persistence.\*;  
import java.util.Objects;  
  
@Getter  
@Setter  
@EqualsAndHashCode  
@Entity  
public class Cozinha {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 private String nome;  
}

A classe do código acima ficou bem enxuta, e repare nas annotations acima.

Podemos também configurar os atributos dos métodos se precisarmos omitir ou adicionar alguma propriedade, como exemplo o método equal and hashcode, onde na annotation @EqualsAndHashCode podemos inserir parâmetros para omitir atributos, veja o exemplo:

@Data  
@EqualsAndHashCode(onlyExplicitlyIncluded = true)  
@Entity  
public class Cozinha {  
  
 @EqualsAndHashCode.Include  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 private String nome;  
}

Repare que estamos utilizando o atributo onlyExplicitlyIncluded = true, que serve para você poder especificar os atributos que você deseja utilizar no método Equals and Hashcode, e em seguida você utiliza a annotation @EqualsAndHashCode.Include no atributo que deseja utilizar no método Equals and Hashcode.

## A annotation @PersistenceContext

A annotation @PersistenceContext (Contexto Persistente) é um local onde ficam armazenados os objetos (entidades) que estão sendo manipulados pelo EntityManager corrente.

EntityManager é o recurso responsável por realizar as operações de sincronismo com o banco de dados (inserir, remover, atualizar ou consultar - CRUD) e gerenciar o ciclo de vida das entidades.

@PersistenceContext  
private EntityManager manager;

## Annotations de relacionamentos de bancos de dados

### @ManyToOne

É responsável por representar o relacionamento de Muitos para Um do SQL, sempre o primeiro relacionamento no caso aqui o Many representa a classe que você está adicionando, e o segundo a classe de referência. Veja um exemplo:

@Data  
@EqualsAndHashCode(onlyExplicitlyIncluded = true)  
@Entity  
public class Restaurante {  
  
 @EqualsAndHashCode.Include  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 private String nome;  
  
 private BigDecimal taxaFrete;  
  
 @ManyToOne  
 private Cozinha cozinha;  
}

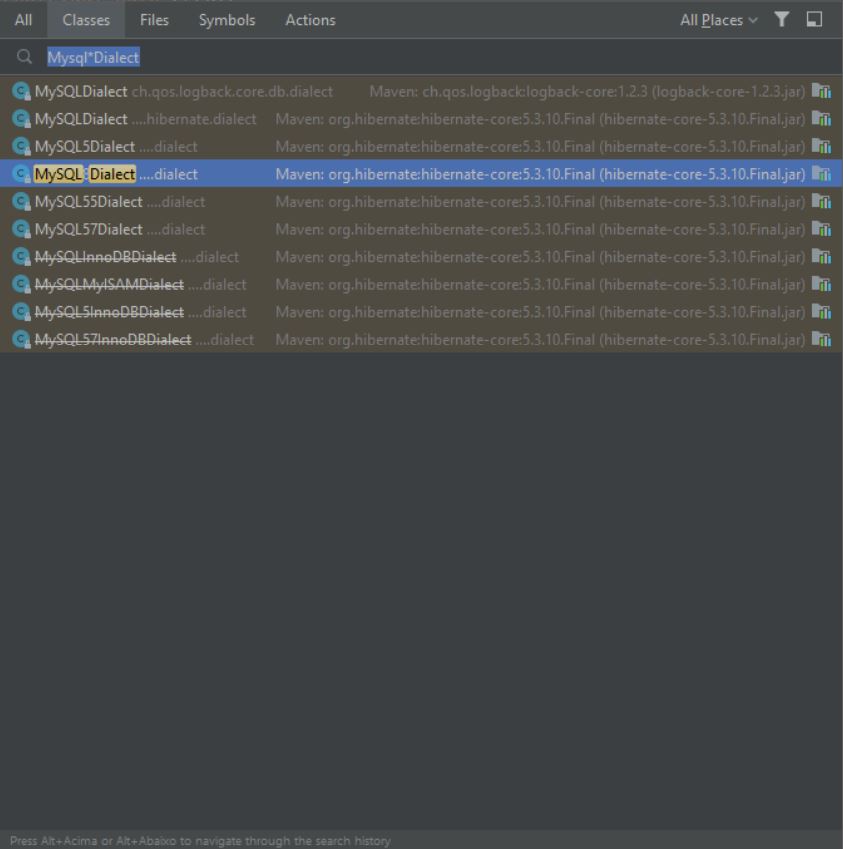
Repare que estamos referenciando a classe Cozinha como um relacionamento de muitos para um, ou seja: Muitos Restaurantes possuem uma Cozinha.

## Suporte ao Foreign Key do motor SQL

Pode acontecer por padrão do motor do MySQL estar setado com o motor MyISAM, que não suporta a chave estrangeira, e devemos utilizar a InnoDB. Para resolver isso você deve inserir no arquivo propperties uma configuração de dialeto do hibernate, veja:

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

Para você encontrar isso você deve pesquisar as classes no seu projeto como na imagem abaixo, cada IDE tem um modo de pesquisar classes, no Intellij utilizamos o atalho Ctrl + N para abrir a pesquisa de classes, pesquise por: Mysql\*Dialect e escolha qual utilizar, no caso pode escolher conforme a versão do seu Mysql que no momento da escrita é o 8.



Você só vai fazer isso para pegar o pacote e o nome da classe, para inserir a propriedade no arquivo propperties.

## Utilizando o @JoinColumn

Para nomear uma coluna foreign key precisamos utilizar a annotation @JoinColumn, veja:

@ManyToOne  
@JoinColumn(name = "cozinha\_codigo")  
private Cozinha cozinha;

## Propriedade nullable de @Column e @JoinColumn

Para gerar atributos que não aceitam null no banco de dados, devemos utilizar um parâmetro na anotação @Column e @JoinColumn que é o nullable = false.

@Column(nullable = false)  
private String nome;  
  
@Column(nullable = false)  
private BigDecimal taxaFrete;  
  
@ManyToOne  
@JoinColumn(nullable = false)  
private Cozinha cozinha;

# ATALHOS DO INTELLIJ IDEA

## Geral

* Alt + 1: Focar aba de projetos
* Alt + Shift + Insert: Alterna entre modo de seleção de linha/coluna
* Ctrl + Shift + F12: Expandir a tela

## Na aba de projetos

* Alt + Insert: Para criar um novo arquivo

## Templates de código

* psvm: Criar método main
* sout: System.out.println()
* Ctrl + Alt + C: Cria constante (precisa estar sobre código que permite atribuição)
* Ctrl + Alt + F: Cria field - propriedade da classe (precisa estar sobre código que permite atribuição)
* Ctrl + Alt + V: Cria variável (precisa estar sobre código que permite atribuição)
* Ctrl + Alt + T: Templates de IF, for, while, etc (precisa selecionar o código antes)
* Ctrl + J: Exibe todos os templates disponíveis

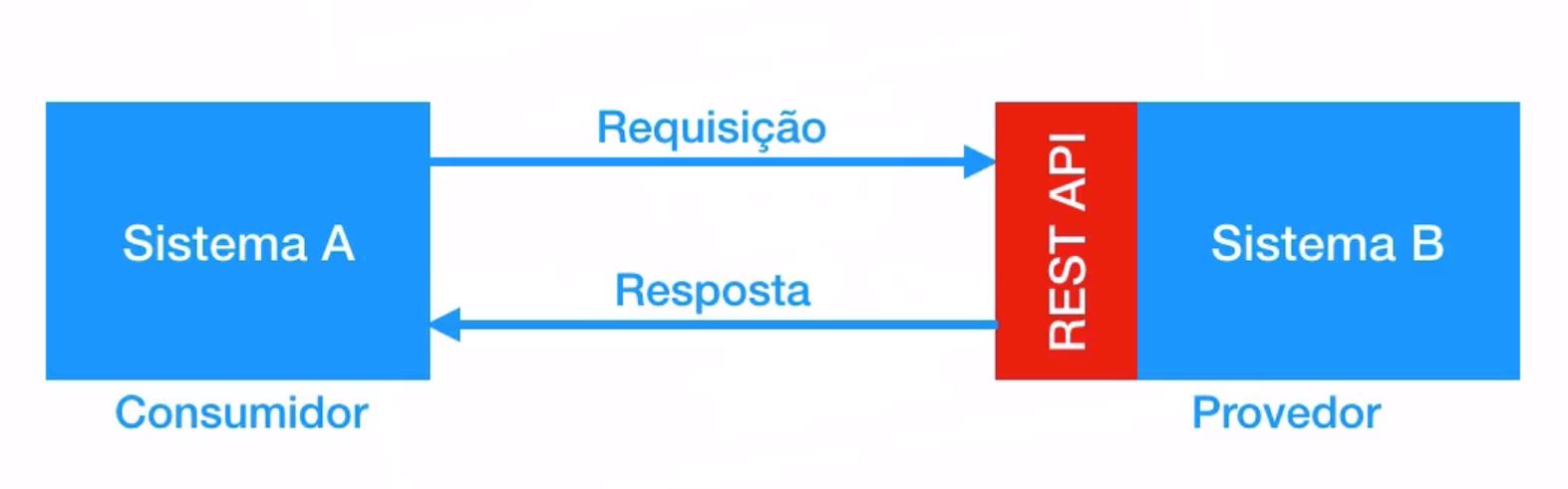
## Na aba de código

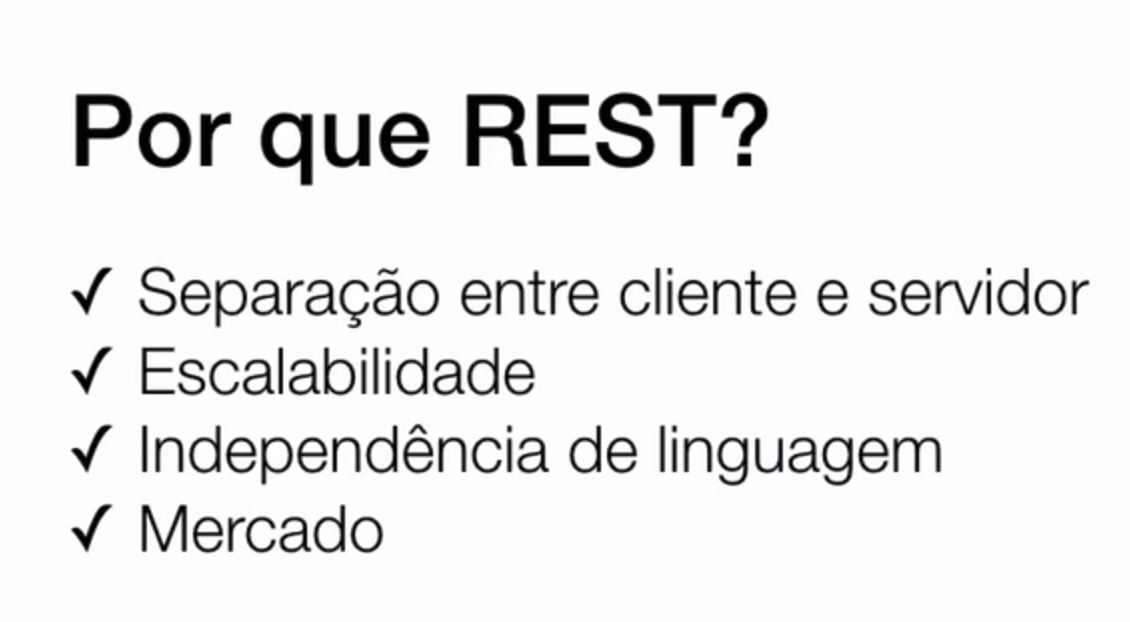
* Ctrl + Y: Apaga a linha inteira
* Ctrl + D: Duplicar linha atual
* Alt + Shift + Setas: Mover linha atual
* Ctrl + Shift + Setas: Mover linha atual mantendo contexto
* Ctrl + Shift + A: Find Action - mostra todas as ações da IDE em uma lista
* Ctrl + N: Busca rápida de classes
* Ctrl + Shift + N: Busca rápida de arquivos
* Ctrl + B: Inspecionar elemento - go to class, method, declaration, etc
* Ctrl + Alt + L: Reformatar código
* Ctrl + Alt + M: Extrair método - transforma código selecionado em método
* Shift + F6: Renomeia classe, variável, método, etc
* Ctrl + Shift + F6: Usar referência de outra classe
* Shift & Shift: Search everywhere
* Ctrl + Shift + F: Busca por conteúdo de arquivo
* Alt + Enter: Intention actions
* Alt + Insert: Generate shortcut
* Ctrl + F12: Mostra a estrutura do arquivo - por exemplo, exibe métodos de uma classe
* Alt + F7: Find usages - mostra a lista de quem está chamando o método, usando a variável, etc
* Ctrl + G: Ir para linha
* Ctrl + H: Mostrar hierarquia da classe (classes pais)
* Alt + Home: Ir para a barra de navegação
* Ctrl + P: Mostrar parâmetros do método
* Ctrl + Q: Documentação rápida
* Ctrl + / ou Ctrl + Shift + /: Comentar código
* Ctrl + Shift + Alt + T: Métodos de refatoração
* Ctrl + Alt + O: Reajustar imports da classe (remove imports não utilizados)

# REST COM SPRING

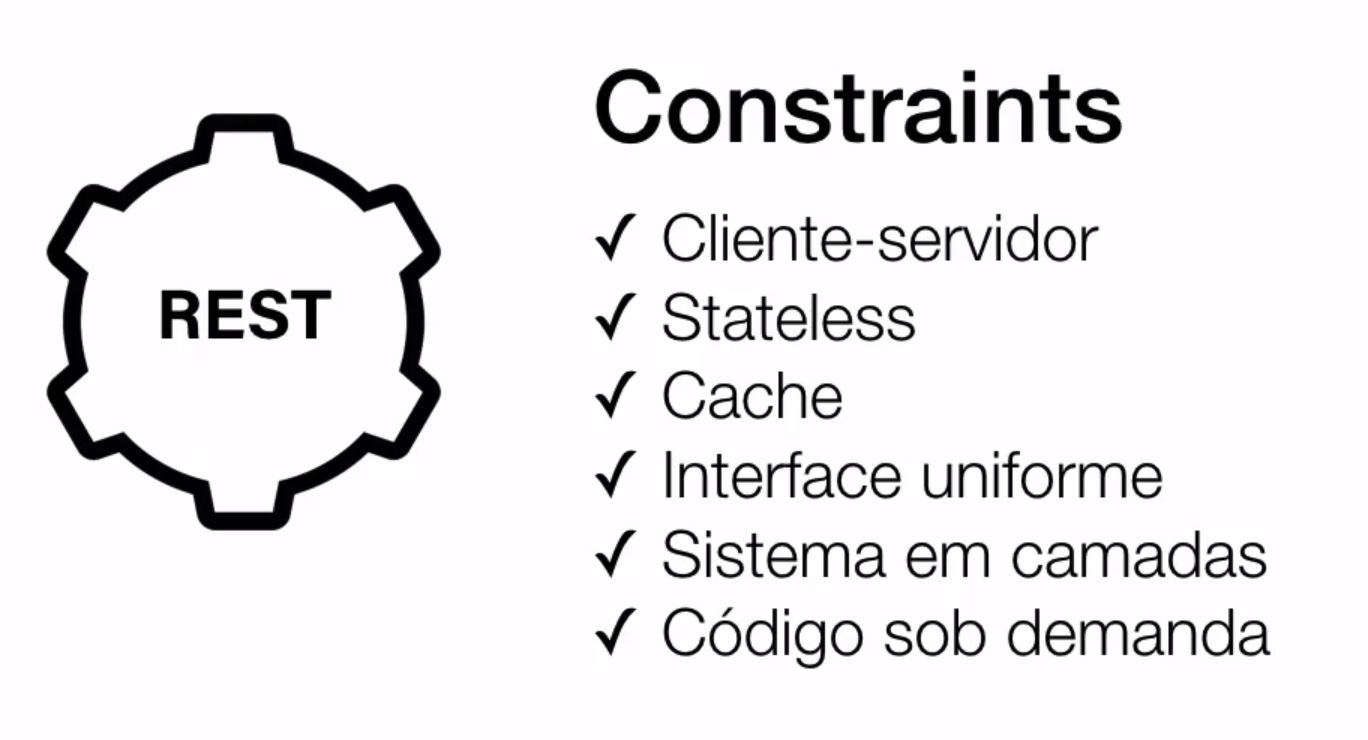
## O que é REST?

REST vem de REpresentational State Transfer, é um modelo arquitetural para desenvolvimento de web services, surgiu em meados do ano 2000 criado por Roy Fielding.





## Constraints do REST

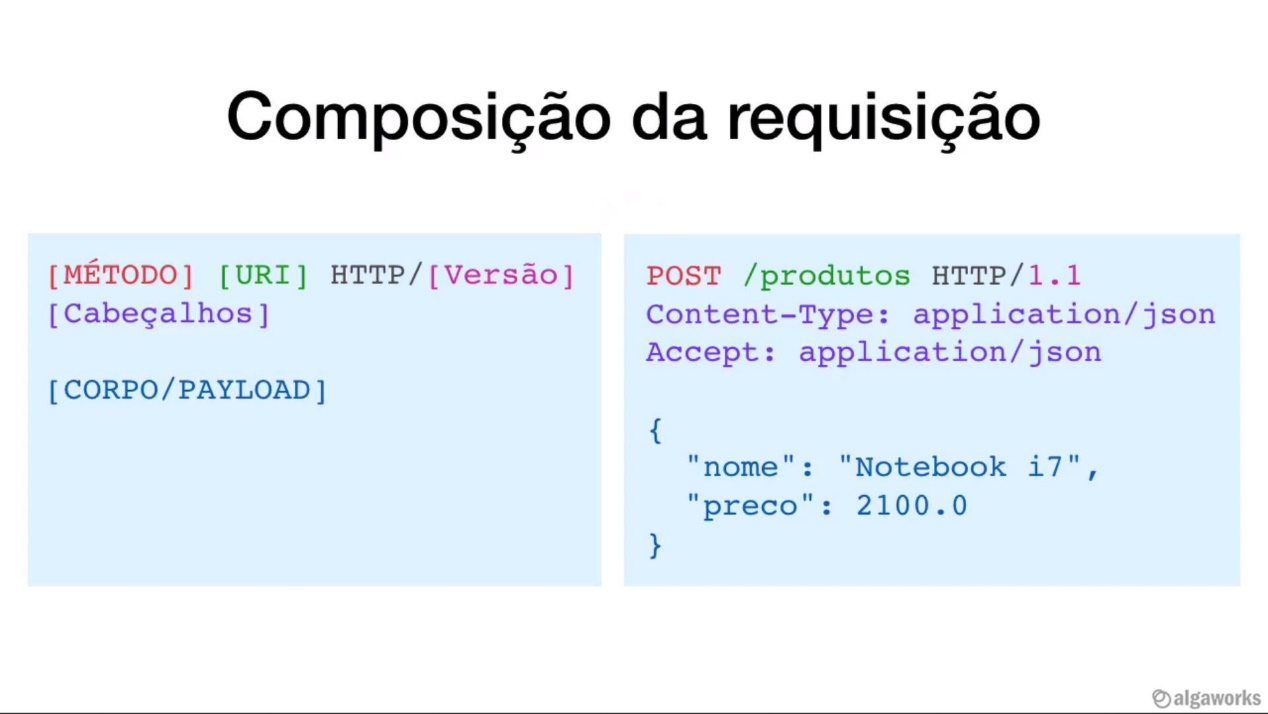


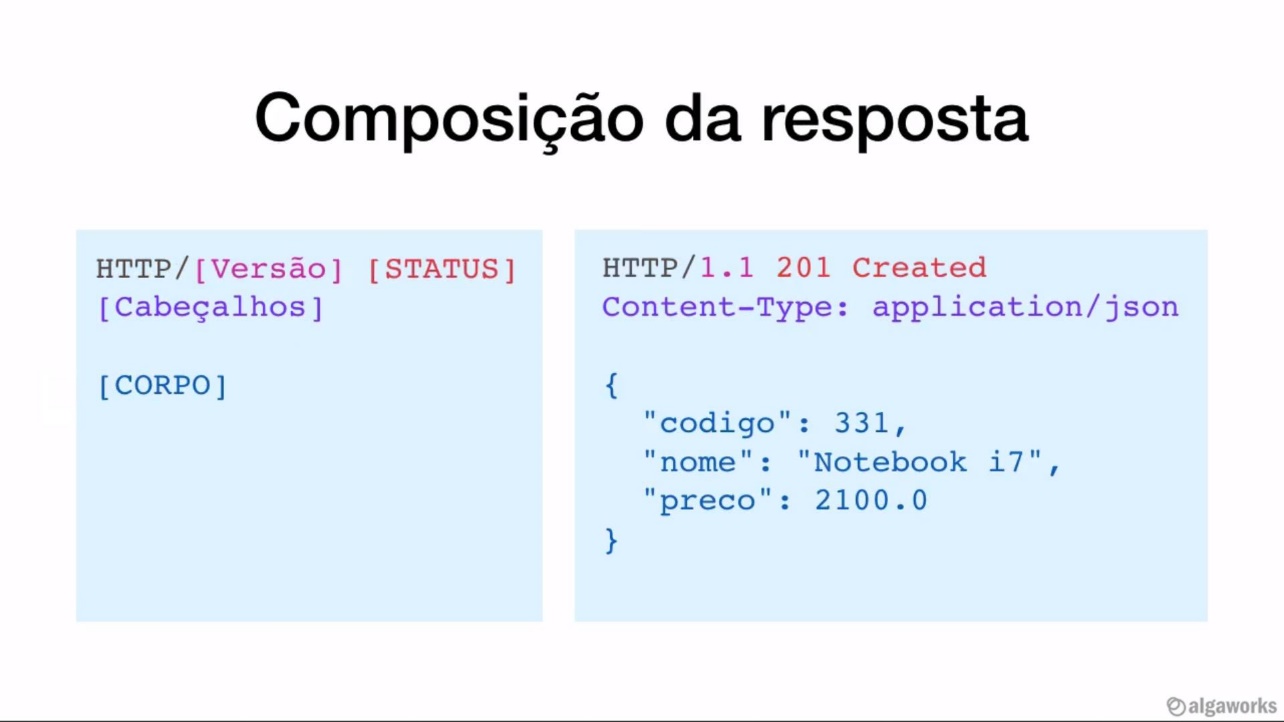
## Diferença entre REST e RESTful

A diferença entre os dois termos é conceitual, mas muita gente confunde, REST é o estilo arquitetural que possui as constraint vistas no tópico acima, é a especificação.

RESTful é uma API desenvolvida em conformidade com as constrains, ela segue todas as constraints.

## Conhecendo o protocolo HTTP





## Fazendo testes de requisição HTTP com GnuTLS

Podemos utilizar uma ferramenta para realizar testes de requisição HTTP via linha de comando, utilizando o GnuTLS, para instalar o GnuTLS no Windows é interessante você ter instalado o Chocolatey para Windows, com ele você pode instalar muitos pacotes via linha de comando, por exemplo você pode instalar o Node.JS, o Yarn via linha de comando pelo Chocolatey, uma vez instalado você vai utilizar a linha de comando para fazer instalações no Windows.

O site do GnuTLS é: <https://www.gnutls.org/> neste site você pode fazer o download do arquivo .zip para o Windows, tem para as duas versões x86 e x64. Para facilitar a instalação já que você possui o Chocolatey, basta rodar o comando abaixo:

choco install emacs

O Chocolatey irá instalar os emacs que contam com várias ferramentas, incluindo o gnutls-cli, após a instalação você deve configurar as variáveis de ambiente apontando para a pasta onde foi instalado os emacs, mais precisamente a pasta emacs\bin, veja o caminho padrão para adicionar nas variáveis de ambiente, (o caminho pode mudar de sistema para sistema, basta procurar no local onde foi instalado as dependências no chocolatey, e apontar). Edite a variável Path e adicione o caminho:

C:\ProgramData\chocolatey\lib\Emacs\tools\emacs\bin

## Resources REST

É qualquer coisa exposta na web, como um documento, página, vídeo ou até mesmo um processo de negócio, a forma de um cliente interagir com uma REST API é através de seus resources.

### Identificando Recursos REST

REST utiliza URI para identificar recursos, URI (Uniform Resource Identifier)

## Adicionando suporte a respostas XML

As vezes podemos precisar que as respostas as requisições venham em formato .xml também, além do formato .json, para habilitarmos o formato xml basta adicionar a dependência no pom.xml:

<dependency>  
 <groupId>com.fasterxml.jackson.dataformat</groupId>  
 <artifactId>jackson-dataformat-xml</artifactId>  
</dependency>

## Especificando o tipo de retorno nos métodos de requisição

Para poder designar um retorno especifico em um método, ou seja quando você quiser que um método de requisição retorne seus valores apenas em determinado formato, você pode especificar isso como um parâmetro na annotation @GetMapping:

@GetMapping(produces = MediaType.*APPLICATION\_JSON\_VALUE*)  
public List<Estado> listar(){  
 return estadoRepository.todas();  
}

Com o atributo produces você especifica um tipo de retorno, você pode especificar vários tipos de retorno em um array, entre chaves:

@GetMapping(produces = { MediaType.*APPLICATION\_JSON\_VALUE*, MediaType.*APPLICATION\_XML\_VALUE*})  
public List<Estado> listar(){  
 return estadoRepository.todas();  
}

## Consultando Singleton Resource com GET e @PathVariable

Para fazer o get de apenas um recurso podemos criar um método e capturar o valor da chave única do recurso, que seria o id ou qualquer outra chave única, e chamar nos parâmetros do path variable, especificando a chave.

@GetMapping("/{id}")  
public Cozinha buscar(@PathVariable("id") Long id){  
 return cozinhaRepository.porId(id);  
}

A annotation @PathVariable é uma forma de mapear o path que está nos parâmetros da annotation @GetMapping, o “/{id}”, para que o Spring entenda que o parâmetro path é o que você está especificando no método. Podemos encurtar o parâmetro assim:

@GetMapping("/{id}")  
public Cozinha buscar(@PathVariable Long id){  
 return cozinhaRepository.porId(id);  
}

## Mudando a representação dos atributos com @JsonProperty

As vezes podemos precisar de mudar o nome dos atributos de um recurso da nossa API, mas somente na representação, no retorno JSON ou XML, e os nomes dos atributos no domínio da aplicação continuam os mesmos, mudamos somente no retorno, para isso usamos a annotation @JsonProperty, indicando como parâmetro o nome que queremos dar ao atributo do recurso.

@JsonProperty("titulo")  
@Column(nullable = false)  
private String nome;

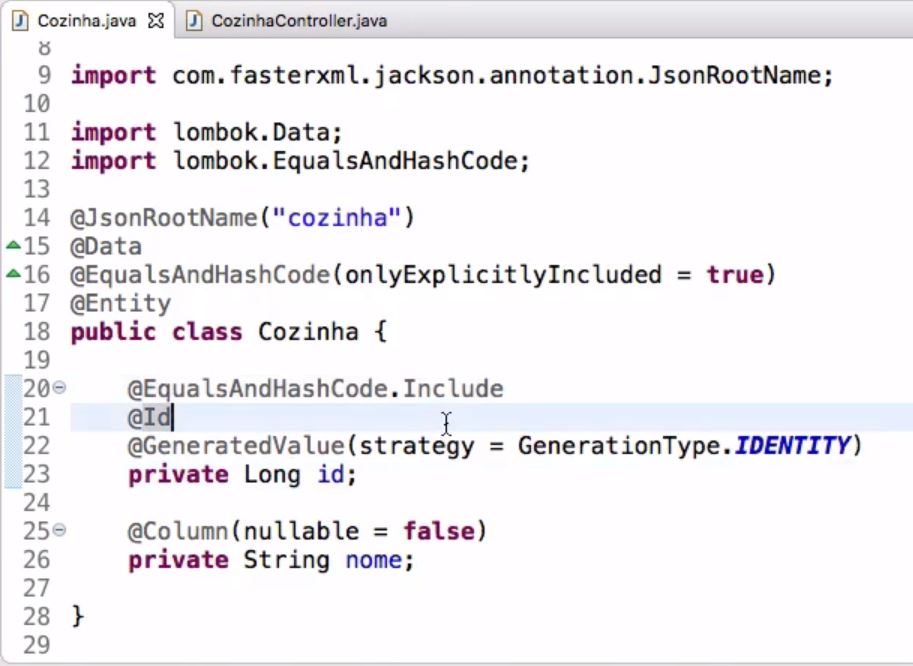
Isso foi feito na classe de modelo do objeto, a @Entity, com isso o retorno em Json ou xml vai ser titulo: valor e não nome: valor.

## Ignorando valores no retorno Json com @JsonIgnore

Podemos ignorar valores nos retornos, especificando uma annotation @JsonIgnore com esta annotation o atributo não vai mais aparecer nos resultados de retorno da chamada do recurso.

## Modificando o nome do recurso com @JsonRootName

Podemos modificar o nome por referência do recurso, com a annotation @JsonRootName especificando como parâmetro o nome que desejamos.



## A annotation @NonNull

A annotation @NonNull do Lombok serve para gerar um construtor para a classe com referência no atributo que você deseja adicionar como parâmetro, veja:

@Data  
public class CozinhasXmlWrapper {  
  
 @NonNull  
 private List<Cozinha> cozinhas;  
   
}

A annotation acima da lista de cozinhas vai ser utilizada como parâmetro em um construtor.

## Criando Wrapper de lista de objetos para modificar nomes de atributos XML

Para modificarmos os nomes de atributos nas tags xml que são geradas de respostas de requisição de recursos, podemos utilizar a abordagem de criar uma classe de embrulho, que vai servir para empacotar a lista de objetos original, um wrapper.

@JacksonXmlRootElement(localName = "cozinhas")  
@Data  
public class CozinhasXmlWrapper {  
  
 @JacksonXmlProperty(localName = "cozinha")  
 @JacksonXmlElementWrapper(useWrapping = false)  
 @NonNull  
 private List<Cozinha> cozinhas;  
}

Após a criação do wrapper, criamos um método na classe de controller do recurso que deseja.

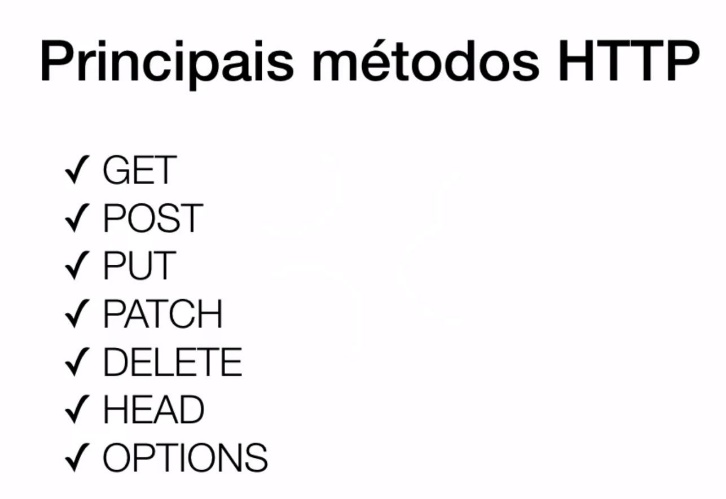
@RestController  
@RequestMapping("/cozinhas")  
public class CozinhaController {  
  
 @Autowired  
 private CozinhaRepository cozinhaRepository;  
  
 @GetMapping  
 public List<Cozinha> listar(){  
 return cozinhaRepository.listar();  
 }  
  
 /\* Responde apenas quando o consumidor pede xml \*/  
 @GetMapping(produces = MediaType.*APPLICATION\_XML\_VALUE*)  
 public CozinhasXmlWrapper listarXml(){  
 return new CozinhasXmlWrapper(cozinhaRepository.listar());  
 }  
  
 @GetMapping("/{id}")  
 public Cozinha buscar(@PathVariable Long id){  
 return cozinhaRepository.porId(id);  
 }  
}

A annotation @JacksonXmlElementWrapper(useWrapping = false) desabilita o embrulho e remove uma tag repetida. E a annotation @JacksonXmlProperty(localName = "cozinha") seta o nome da tag que você deseja.

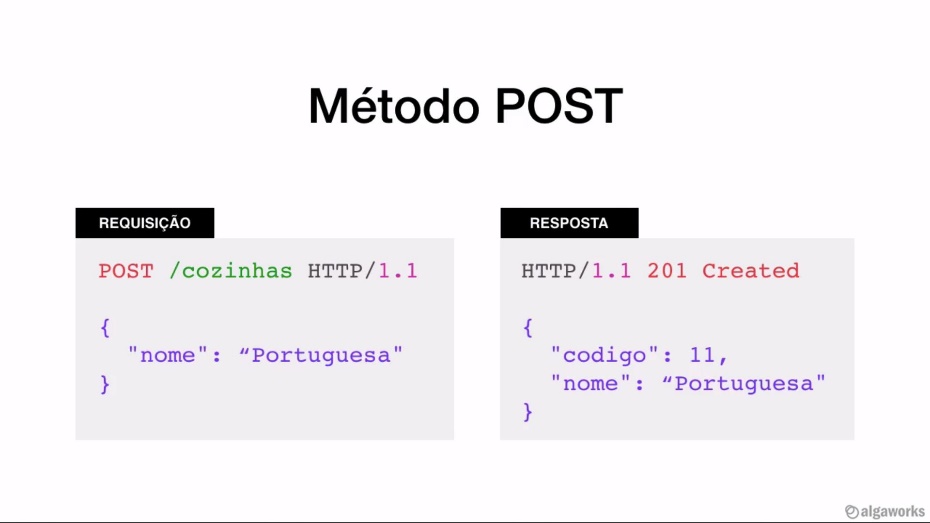
## Métodos HTTP

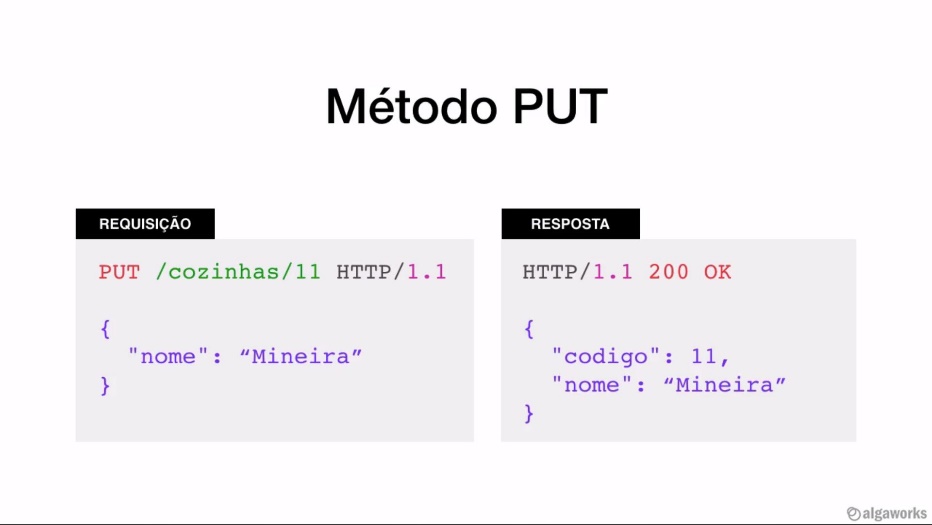
### Idempotência

Significa que pode ser aplicado mais de uma vez, sem que o resultado da primeira aplicação se altere.











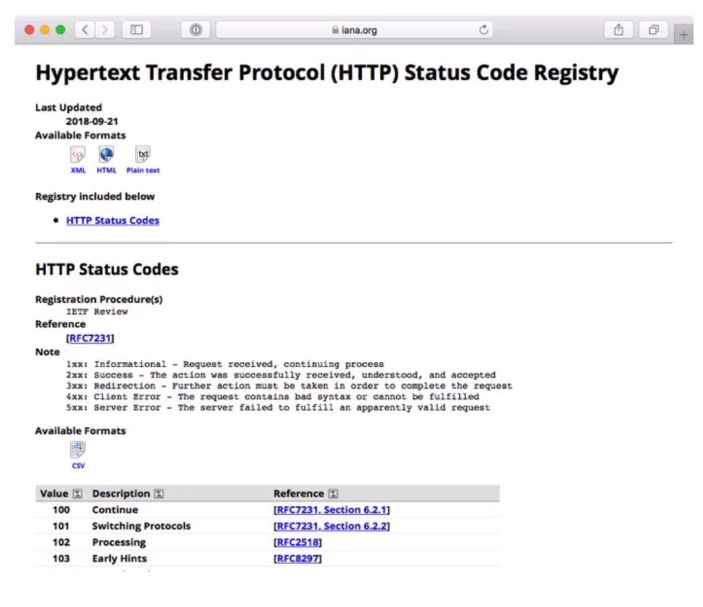


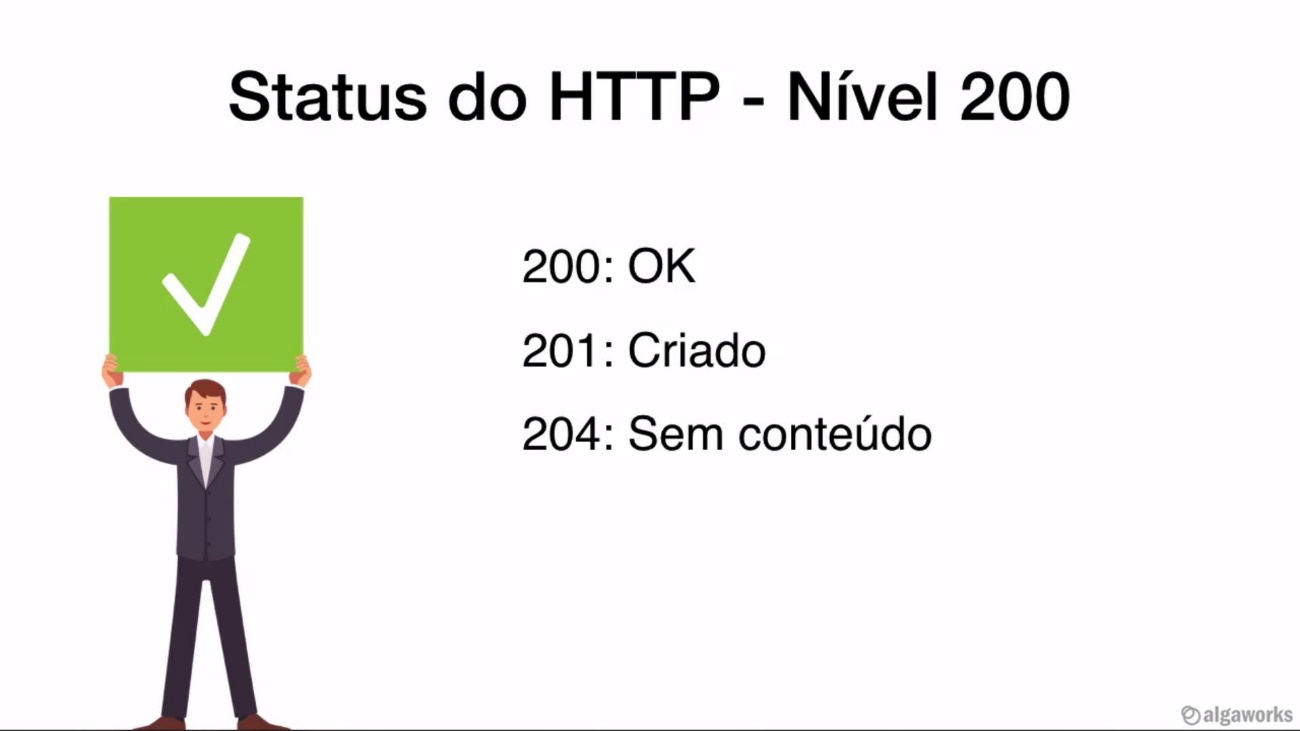


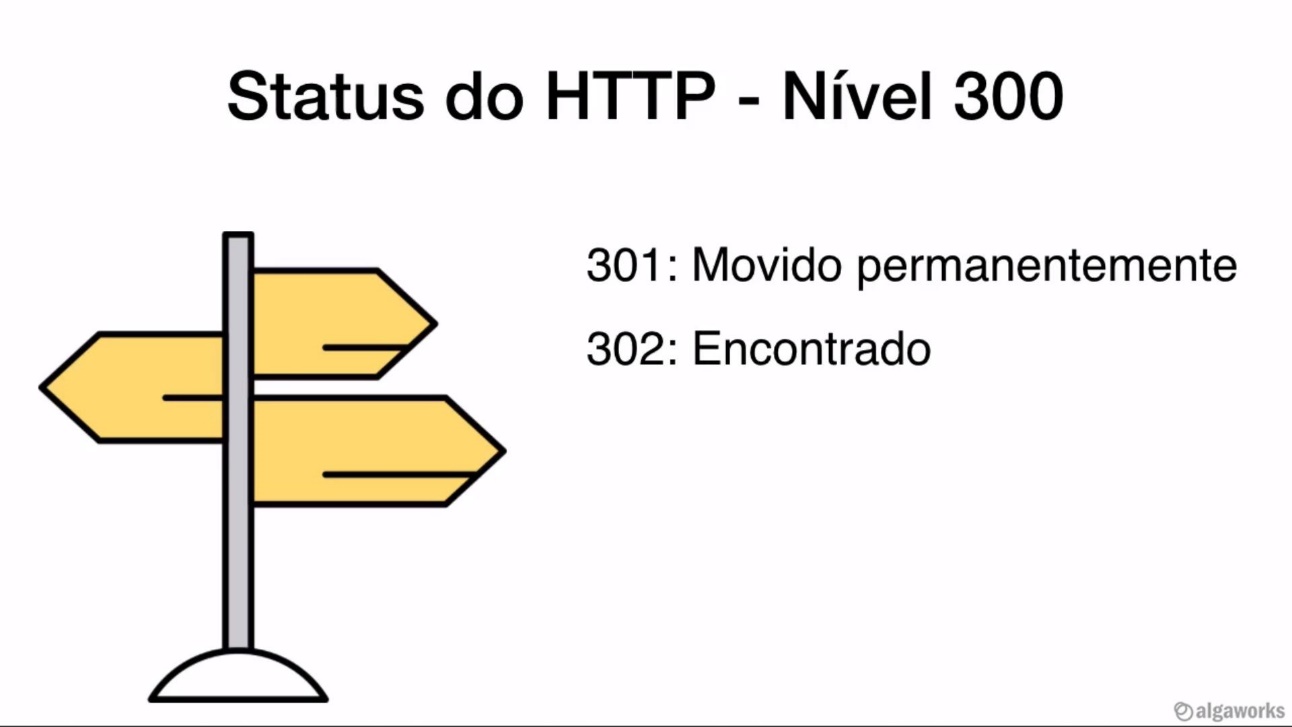


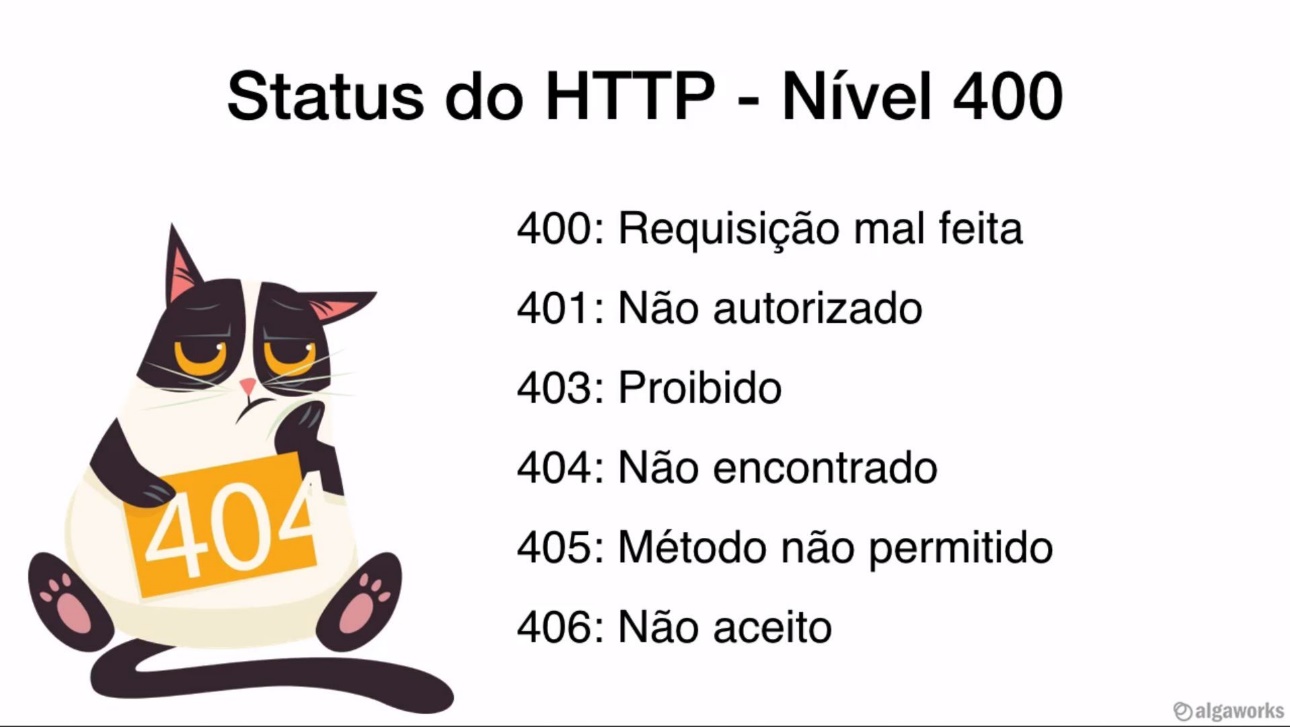
## Códigos de status HTTP











## Definindo o status da resposta HTTP com @ResponseStatus

Para alterarmos o código de status da resposta da requisição utilizamos a annotation @ResponseStatus, veja:

@ResponseStatus(HttpStatus.*ACCEPTED*) // Ira retornar 200 ok.  
@GetMapping("/{id}")  
public Cozinha buscar(@PathVariable Long id){  
 return cozinhaRepository.porId(id);  
}

## Manipulando a resposta com @ResponseEntity

Podemos utilizar o ResponseEntity para manipular as respostas HTTP, utilizando em conjunto com o HttpHeaders que serve para manipular o header, cabeçalho da resposta.

@GetMapping("/{id}")  
public ResponseEntity<Cozinha> buscar(@PathVariable Long id){  
 Cozinha cozinha = cozinhaRepository.porId(id);  
 //return ResponseEntity.status(HttpStatus.OK).body(cozinha); // Muito utilizado para lógicas condicionais  
 //return ResponseEntity.ok(cozinha); // Atalho para o mesmo método acima.  
  
 HttpHeaders headers = new HttpHeaders();  
 headers.add(HttpHeaders.*LOCATION*, "http://api.algafood.local:8080/cozinhas");  
  
 return ResponseEntity.*status*(HttpStatus.*FOUND*).headers(headers).build();  
  
}

## Adicionando recursos com @PostMapping

Para adicionar recursos no banco de dados através do corpo da requisição, utilizamos o @PostMapping, veja o método de adicionar na classe de controller:

@PostMapping  
@ResponseStatus(HttpStatus.*CREATED*)  
public Cozinha adicionar(@RequestBody Cozinha cozinha) {  
 return cozinhaRepository.adicionar(cozinha);  
}

Anotamos com @PostMapping e setamos a resposta como 201 created, precisamos também adicionar a anotação @RequestBody que tem como finalidade esperar, requer o corpo do recurso passado no corpo da requisição, o corpo definido lá vai ser passado para o parâmetro do método.

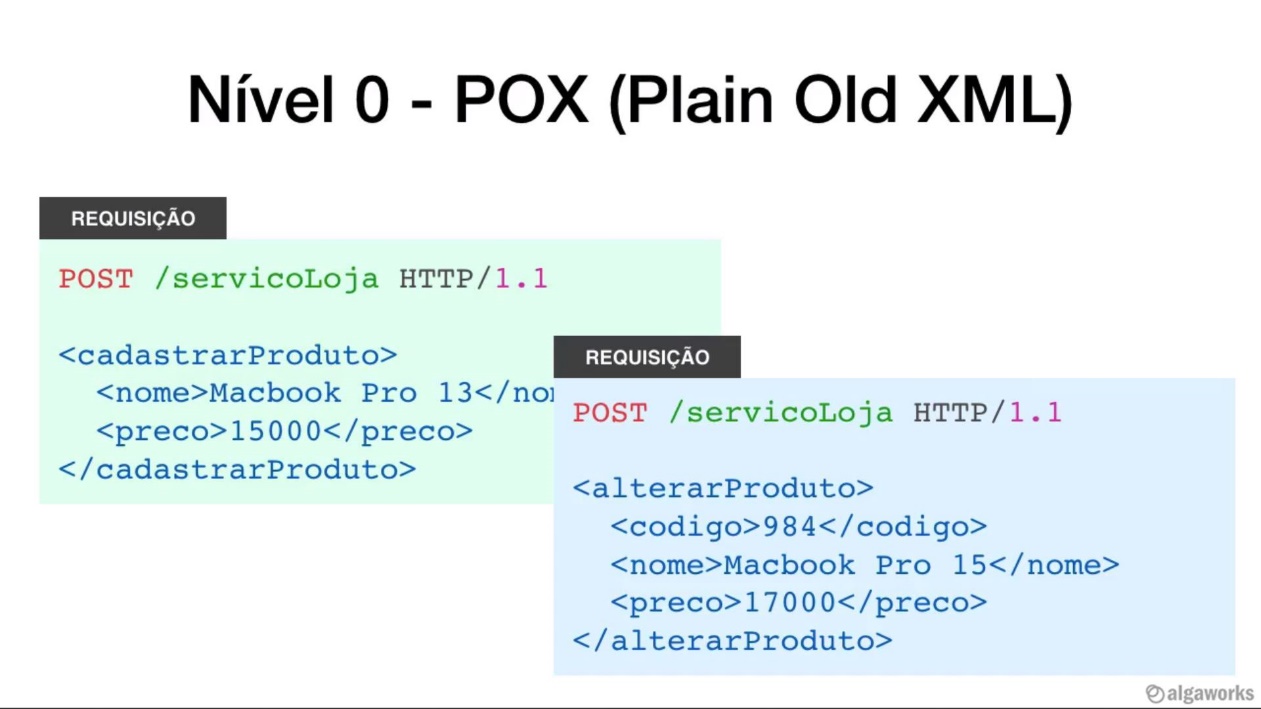
## Atualização de um recurso com @PutMapping

## Exclusão de recurso com @DeleteMapping

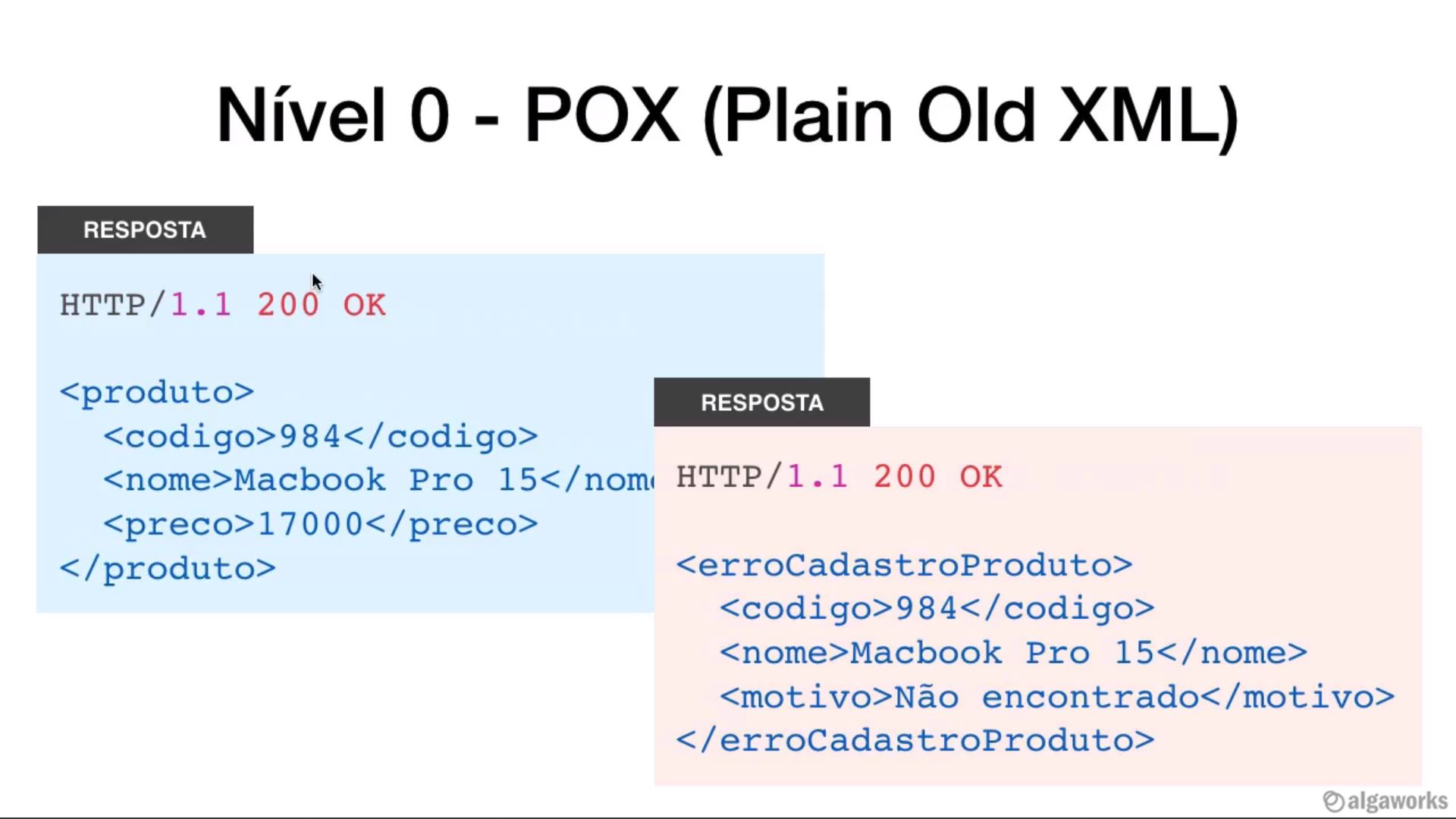
## Atualização de recursos com PATCH utilizando Reflections

## Introdução ao Modelo de Maturidade de Richardson (RMM)







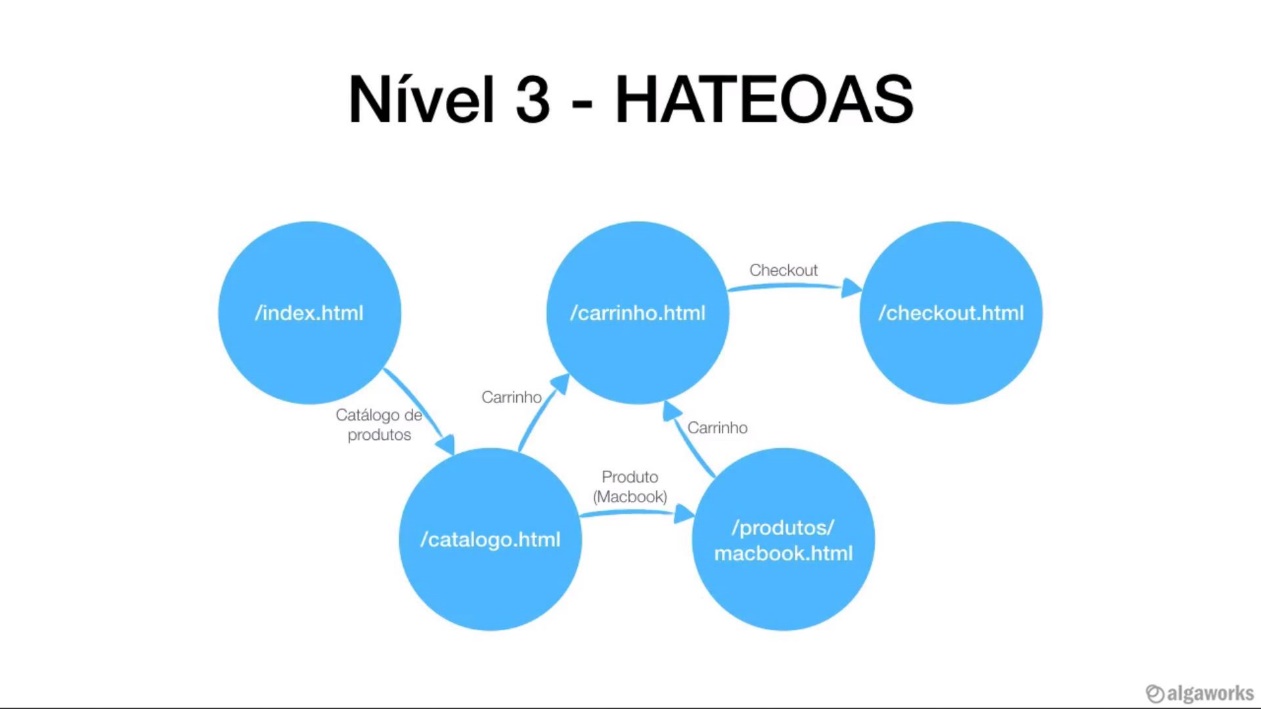




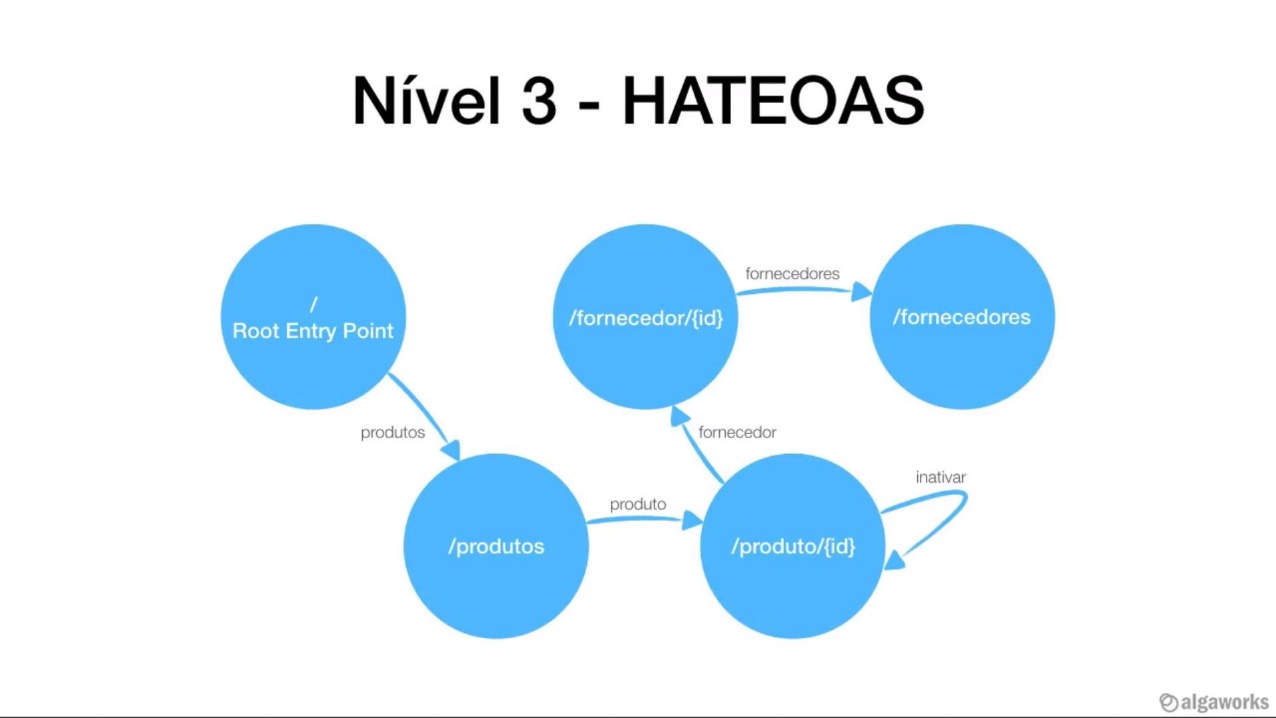














# SUPER PODERES DO SPRING DATA JPA

## Annotation @Repository

A annotation @Repository é utilizada para definir semanticamente classes que são repositórios ou implementações de repositórios, além de ser semântico esta annotation trás benefícios de tradução de exceptions do Spring.

## Spring Data JPA

### Criando um repositório com Spring Data JPA (SDJ)

O Spring Data JPA fornece uma implementação de repositórios JPA em tempo de execução, ele é uma interface e para criarmos um repositório do SDJPA temos que extender a interface JpaRepository:

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

Ela recebe com parâmetro <T, ID> a entidade, classe que você pretende utilizar e o tipo do ID que a entidade utiliza, no caso acima utilizamos a classe, entidade Cozinha, que possui um ID de tipo Long.

### Criando consultas com query methods

Podemos criar métodos nas interfaces que estendem JpaRepository, para criar consultas personalizadas além das que a interface já possui, um dos meios é utilizar o nome do método como padrão de nomenclatura de consulta, o JPA entende por meio do nome da função o tipo de consulta de acordo com a classe que você deseja pesquisar.

Funciona com o findBy, você pode encadear consultas entre o find e o By, por exemplo: findNomeById e o JPA vai entender de acordo com a classe que você está consultando.

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

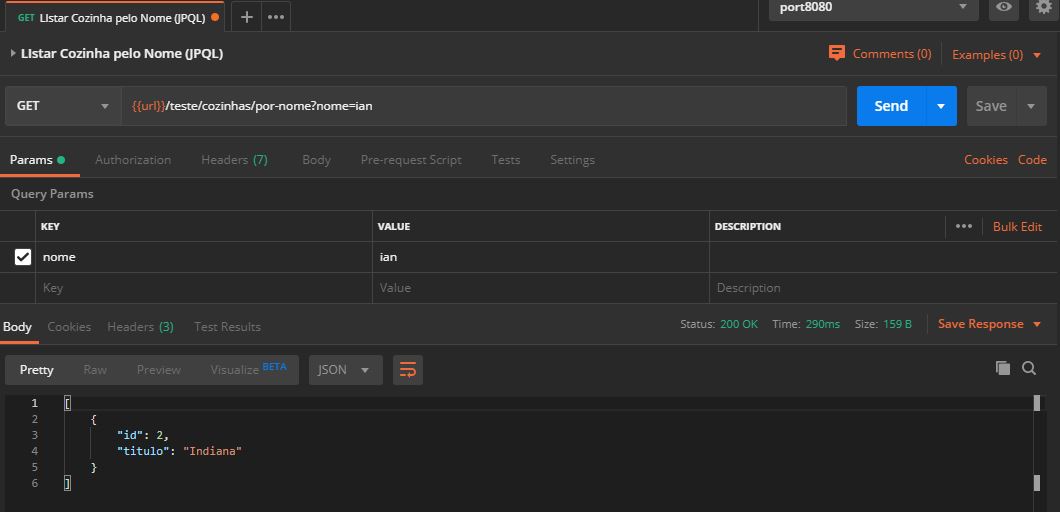
O JPA interpreta o nome do método como uma expressão, ele interpreta isso em tempo de execução e cria uma implementação da consulta.

Podemos utilizar também Keywords (palavras chave) que são utilizadas para montar querys de forma mais especifica.

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

No código acima a palavra chave containing possui a mesma função do Like do sql, a consulta acima significa, busque qualquer coisa pelo Nome que contenha(parametro) veja o controller:

@Autowired  
private CozinhaRepository cozinhaRepository;  
  
@GetMapping("/cozinhas/por-nome")  
public List<Cozinha> cozinhasPorNome(@RequestParam("nome") String nome){  
 return cozinhaRepository.findTodasByNomeContaining(nome);  
}



Veja que funciona como o like, passamos como parâmetro =ian e a consulta buscou o dado correspondente no banco.

Veja o link para a documentação dos parâmetros de nomenclatura de métodos do JPA:

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

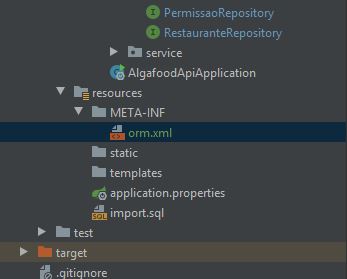
### Consultas utilizando JPQL

Podemos realizar consultas utilizando JPQL, criando métodos na interface com a annotation @Query, isso abre a possibilidade de realizar querys mais elaboradas o que traz uma maior flexibilidade na criação de consultas avançadas, veja um exemplo:

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

### Externalizando consultas JPQL em arquivo .xml

Podemos criar um arquivo .xml para separar as querys JPQL dos métodos, serve caso você queira organizar suas consultas e separar em outro arquivo, pois algumas consultas podem ficar muito grandes e ocupar muitas linhas de código acima do método e conforme haja muitas consultas fica feio e confuso. Para fazer isso você tem que criar uma pasta META-INF ou se ela já existir também você irá criar um arquivo .xml chamado orm.xml, veja a imagem:



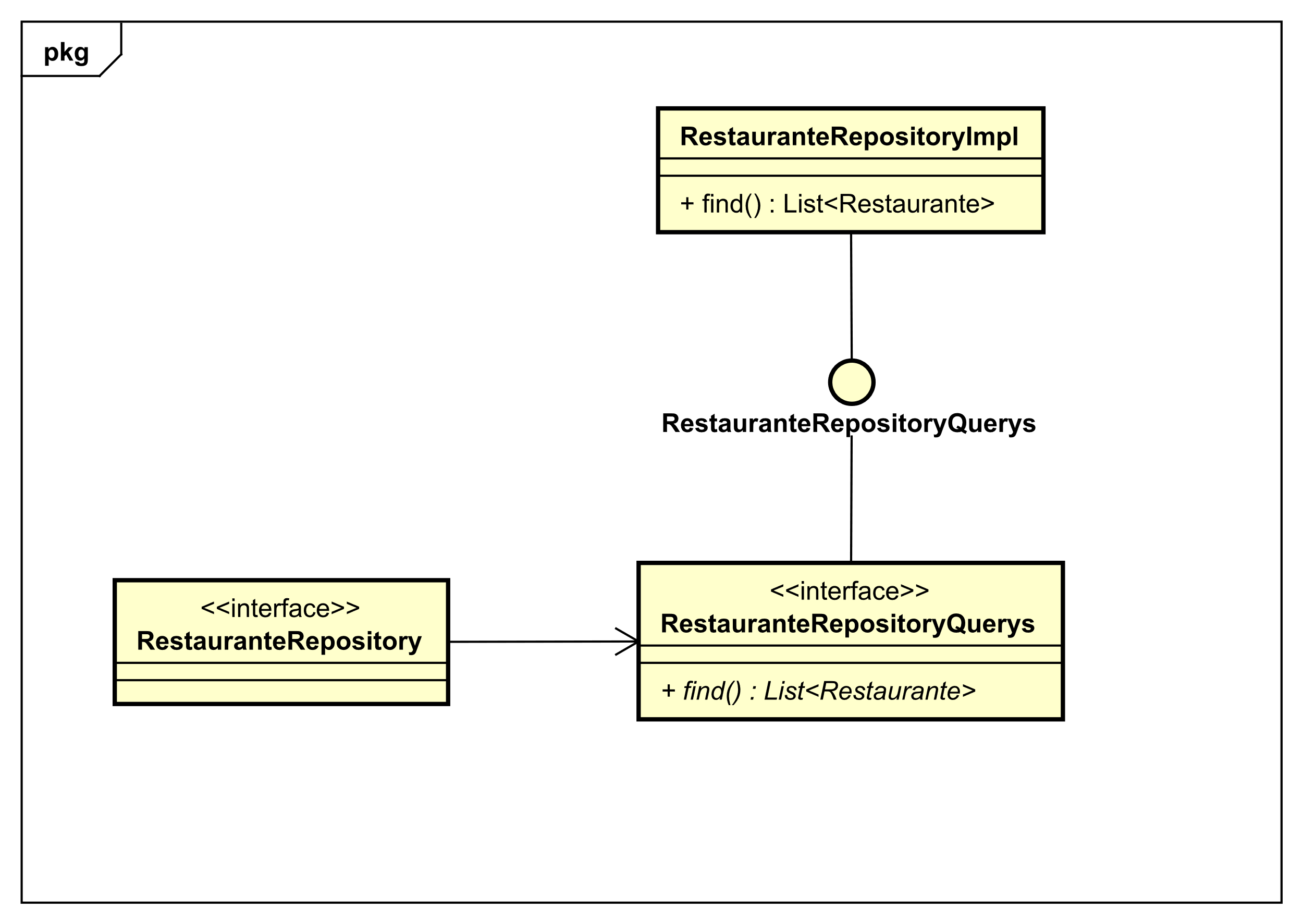
Feito isso você deve colocar o código de cabeçalho que configura o documento para receber querys do JPA, é um .xml padrão do JPA para isso.

<?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>

Veja que no exemplo temos as tags que criam a consulta, a tag <named-query> é responsável por mapear a classe e o nome do método que vai receber a query, e na tag <query> temos a query JPQL, o concat é utilizado para evitar erro de compilação, pois o formato xml da erro com o % ai temos que mascarar para evitar o erro, com isso podemos remover a annotation @Query acima do método que estava utilizando para realizar a consulta JPQL.

### Criando Interface avançada de repository

Podemos criar uma interface que vai servir como uma implementação da interface Repository, mas que não é necessário estender com o extends, basta simplesmente criar uma classe que vai implementar uma interface com os métodos de consulta avançado e implementar ela na interface repositor já que uma interface em Java pode estender várias interfaces, o diagrama abaixo explica o que foi dito:



Criamos uma classe que ira ter métodos avançados de consulta, e extraímos uma interface desta classe, que agora vai passar a implementar a interface extraída, onde ira conter as assinaturas de métodos desta classe, que por sua vez vai ser estendida na interface repositor do JPA, fazemos isto para não termos problemas caso haja mudança de nomes dos métodos futuramente.

*/\*\*  
 \* Classe de implementação do repositório, que não necessita de implements para funcionar, porque se fossemos  
 \* implementar a interface de repository teriamos que sobrescrever todos os métodos da interface, e não queremos isto  
 \* vamos utilizar esta classe para criar querys mais avançadas e dinamicas.  
 \* O Spring Data JPA é inteligente para que a gente apenas precise copiar a assinatura dos métodos abaixo para a  
 \* interface do repository, que ele vai fazer a ligaçào automaticamente.  
 \*/*@Repository  
public class RestauranteRepositoryImpl implements RestauranteRepositoryQuerys {  
  
 @PersistenceContext  
 private EntityManager manager;  
  
 @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("nome", "%" + nome + "%")  
 .setParameter("taxaInicial", taxaFreteInicial)  
 .setParameter("taxaFinal", taxaFreteFinal)  
 .getResultList();  
 }  
}

Temos a interface extraída:

public interface RestauranteRepositoryQuerys {  
 List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal);  
}

E o extends na interface JPA

@Repository  
public interface RestauranteRepository extends JpaRepository<Restaurante, Long>, RestauranteRepositoryQuerys {

### Implementando consulta dinâmica com JPQL

Podemos criar consultas dinâmicas que podem deixar de receber parâmetros ou receber qualquer objeto com parâmetro, para isso podemos utilizar lógica de código Java no método de consulta JPQL, veja o código de exemplo abaixo:

@Repository  
public class RestauranteRepositoryImpl implements RestauranteRepositoryQuerys {  
  
 @PersistenceContext  
 private EntityManager manager;  
  
 @Override  
 public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {  
 var jpql = new StringBuilder();  
 jpql.append("from Restaurante where 0 = 0 ");  
  
 var parametros = new HashMap<String, Object>();  
  
 if (StringUtils.*hasLength*(nome)) {  
 jpql.append("and nome like :nome ");  
 parametros.put("nome", "%" + nome + "%");  
 }  
  
 if (taxaFreteInicial != null){  
 jpql.append("and taxaFrete >= :taxaInicial ");  
 parametros.put("taxaInicial", taxaFreteInicial);  
 }  
  
 if (taxaFreteFinal != null){  
 jpql.append("and taxaFrete <= :taxaFinal ");  
 parametros.put("taxaFinal", taxaFreteFinal);  
 }  
  
 TypedQuery<Restaurante> query = manager.createQuery(jpql.toString(), Restaurante.class);  
  
 parametros.forEach((chave, valor) -> query.setParameter(chave, valor));  
  
 return query.getResultList();  
 }  
}

Repare que utilizamos um HashMap para guardar os parâmetros para passar para o setParemeter de forma dinâmica em um laço de repetição, a cada condição dos if, guardamos um valor no HashMap, e no final abaixo utilizamos um laço for com lambda para percorrer o HashMap e setar a cada iteração o valor de (chave e valor) com isso caso algum campo esteja nulo não ira quebrar a query, ou seja a query é totalmente dinâmica.

## Implementando consulta com Criteria API

Podemos criar consultas utilizando a API Criteria do JPA, ela é muito útil para criar consultas complexas, mas ela é bastante burocrática em sua implementação, veja um código de exemplo:

@Repository  
public class RestauranteRepositoryImpl implements RestauranteRepositoryQuerys {  
  
 @PersistenceContext  
 private EntityManager manager;  
  
 @Override  
 public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {  
  
 CriteriaBuilder builder = manager.getCriteriaBuilder();  
  
 CriteriaQuery<Restaurante> criteria = builder.createQuery(Restaurante.class);  
 Root<Restaurante> root = criteria.from(Restaurante.class);  
  
 var predicates = new ArrayList<Predicate>();  
  
 if (StringUtils.*hasText*(nome)) {  
 predicates.add(builder.like(root.get("nome"), "%" + nome + "%"));  
 }  
  
 if (taxaFreteInicial != null) {  
 predicates.add(builder.greaterThanOrEqualTo(root.get("taxaFrete"), taxaFreteInicial));  
 }  
  
 if (taxaFreteFinal != null) {  
 predicates.add(builder.lessThanOrEqualTo(root.get("taxaFrete"), taxaFreteFinal));  
 }  
  
 criteria.where(predicates.toArray(new Predicate[0]));  
  
 TypedQuery<Restaurante> query = manager.createQuery(criteria);  
 return query.getResultList();  
 }  
}

A linha:

CriteriaBuilder builder = manager.getCriteriaBuilder();

É utilizada para criar um builder do Criteria, é como se fosse uma fabrica de criterias, temos que instanciar para poder utilizar objetos Criterias e obter as funcionalidades dos métodos, utilizamos o objeto do EntityManager para pegar a instancia do CriteriaBuilder.

A linha:

CriteriaQuery<Restaurante> criteria = builder.createQuery(Restaurante.class);  
Root<Restaurante> root = criteria.from(Restaurante.class);

Serve para criar uma CriteriaQuery, do tipo do Objeto a ser utilizado, ela utiliza os métodos do criteria builder para definir a criação das querys para determinada classe. O objeto Root é como se fosse o objeto raiz, que vai ser utilizado, é necessário setar o objeto raiz, que no caso acima é a classe Restaurante.

A linha:

var predicates = new ArrayList<Predicate>();  
  
if (StringUtils.*hasText*(nome)) {  
 predicates.add(builder.like(root.get("nome"), "%" + nome + "%"));  
}

Serve para definir os requests param, o criteria recebe um array de predicates, por isso criamos um ArrayList para poder gravar os requestParam de forma dinâmica, e a cada condição adicionamos o predicate no arrayList, pegando pelo objeto raiz (root), e utilizando os métodos do builder para montar as querys.

A linha:

criteria.where(predicates.toArray(new Predicate[0]));

É necessário converter o arrayList em Array para que o criteria possa receber, nesta linha convertemos a lista em array o new Predicate[0] é só para iniciar o array, poderia ser qualquer valor que daria certo.

A linha:

TypedQuery<Restaurante> query = manager.createQuery(criteria);  
return query.getResultList();

Precisamos de um objeto TypedQuery para poder criar a query com o EntityManager, e retornamos o getResultList() para obter o resultado da query.

### Implementando Specifications

Os Specifications é mais uma forma de criar consultas ao banco de dados, uma das vantagens de utilizar specifications é que podemos combinar consultas, reaproveitar consultas para utilizar novamente e realizar combinações entre elas, para criar specifications é necessário criar uma classe que implementa a interface Specification<Classe>, e dentro desta classe utilizar a sobrecarga dos métodos da interface Specification, no caso o mais utilizado é o toPredicate, que cria um predicate, que é uma query para implementar no requestParam, veja o código:

public class RestauranteComFreteGratisSpec implements Specification<Restaurante> {  
  
 @Override  
 public Predicate toPredicate(Root<Restaurante> root, CriteriaQuery<?> criteriaQuery, CriteriaBuilder builder) {  
 return builder.equal(root.get("taxaFrete"), BigDecimal.*ZERO*);  
 }  
}

Perceba que ele possui os mesmos parâmetros que já estudamos, ele necessita de um Root, de uma criteriaQuery e de um builder, com isso montamos uma query que é um predicate. A partir desta classe instanciamos manualmente na classe controller dentro dos métodos que desejarmos, veja a utilização de dois predicates:

@AllArgsConstructor  
public class RestauranteComNomeSemelhanteSpec implements Specification<Restaurante> {  
  
 private String nome;  
  
 @Override  
 public Predicate toPredicate(Root<Restaurante> root, CriteriaQuery<?> criteriaQuery, CriteriaBuilder builder) {  
 return builder.like(root.get("nome"), "%" + nome + "%");  
 }  
}

//Método utilizando specifications  
@GetMapping("/restaurantes/com-frete-gratis")  
public List<Restaurante> restauranteComFreteGratis(String nome) {  
 var comFreteGratis = new RestauranteComFreteGratisSpec();  
 var comNomeSemelhante = new RestauranteComNomeSemelhanteSpec(nome);  
  
 return restauranteRepository.findAll(comFreteGratis.and(comNomeSemelhante));  
}

Veja que apenas necessitamos de instanciar em variável a classe do specification, repare que estamos utilizando 2 classes de specification, e abaixo no retorno estamos utilizando o findAll do JPA repository e passando como paramêtro os dois specifications, unindo-os com o .and, para utilizar os dois ao mesmo tempo, tornando os dois uma única query, esse é o poder do specification, podemos utilizar combinações de classes com predicates para formar querys e até reaproveitar. Para poder utilizar ainda precisamos fazer o extends da interface Specifications na interface repositor do JPA, extends na interface JpaSpecificationExecutor<Classe>, veja:

public interface RestauranteRepository extends JpaRepository<Restaurante, Long>, RestauranteRepositoryQuerys, JpaSpecificationExecutor<Restaurante> {

Com isso está tudo pronto para utilizar o specification.

#### Criando uma fábrica de specifications

Tudo que foi dito acima pode ser melhorado com a criação de uma fábrica de spcifications, ai não vamos precisar de criar várias classes para poder criar um predicate, agora podemos criar apenas uma classe e inserir métodos que seram utilizados como os predicates, para isso criamos uma classe para poder criar métodos estáticos que serão as querys personalizadas. Veja a classe:

public class RestauranteSpecs {  
  
 public static Specification<Restaurante> comFreteGratis() {  
 return (root, criteriaQuery, criteriaBuilder) -> criteriaBuilder.equal(root.get("taxaFrete"), BigDecimal.*ZERO*);  
 }  
  
 public static Specification<Restaurante> comNomeSemelhante(String nome) {  
 return (root, criteriaQuery, criteriaBuilder) -> criteriaBuilder.like(root.get("nome"), "%" + nome + "%");  
 }  
}

Repare que temos dois métodos que fazem exatamente a mesma coisa que as duas classes anteriores faziam, agora no mesmo método da classe de controller precisamos apenas retornar os métodos veja:

//Método utilizando specifications  
@GetMapping("/restaurantes/com-frete-gratis")  
public List<Restaurante> restauranteComFreteGratis(String nome) {  
 return restauranteRepository.findAll(*comFreteGratis*().and(*comNomeSemelhante*(nome)));  
}

Repare que removemos a instanciação das classes e removemos elas do projeto, agora estamos utilizando somente essa única classe que criamos, e para utilizar a chamada do método de forma direta, precisamos fazer o import static da classe:

import static com.algaworks.algafoodapi.infrastructure.repository.spec.RestauranteSpecs.\*;

Dessa forma podemos chamar o método diretamente sem precisar especificar o caminho da classe. Agora podemos criar diferentes querys dentro desta única classe e utilizar da melhor forma que desejarmos.

## Utilizando specifications com repository interface

Utilizar a abordagem de criar uma classe para gerar os predicates, pode ocasionar um problema de que a responsabilidade dos métodos não está atrelada a uma interface, e não é controlado pelo repositor, ou seja se futuramente precisarmos de modificar um método temos que alterar todos os que utilizam a mesma combinação, e isso é inadequado para projetos grandes, por isso podemos continar a utilizar os benefícios do specification de uma forma que possamos delegar responsabilidade para a interface repository.

Só precisamos fazer uma pequena mudança, apenas criamos uma assinatura na interface que criamos para a classe de implementação de repository, veja:

public interface RestauranteRepositoryQuerys {  
 List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal);  
  
 List<Restaurante> findComFreteGratis(String nome);  
}

Esta interface é implementada na classe:

@Repository  
public class RestauranteRepositoryImpl implements RestauranteRepositoryQuerys {  
  
 @PersistenceContext  
 private EntityManager manager;  
  
 @Autowired @Lazy  
 private RestauranteRepository restauranteRepository;  
  
 @Override  
 public List<Restaurante> find(String nome, BigDecimal taxaFreteInicial, BigDecimal taxaFreteFinal) {  
 }  
  
 @Override  
 public List<Restaurante> findComFreteGratis(String nome) {  
 return restauranteRepository

.findAll(*comFreteGratis*().and(*comNomeSemelhante*(nome)));  
 }  
}

Com a adição de uma nova assinatura, foi necessário sobrescrever o novo método, e para utilizar o repository precisamos injetar ele na dependência, com

@Autowired @Lazy

private RestauranteRepository restauranteRepository;

### Arrumando erro de Referência Circular com @Lazy

A annotation @Lazy do Spring serve para injetar uma dependência só quando for necessário, a dependência só será instanciada na hora que precisar. No caso acima foi utilizada para corrigir erro de referência circular.

Com a dependência injetada, podemos utilizar os métodos de repositor nesta implementação, para poder delegar para a interface a responsabilidade dos métodos.

E no controller apenas chamamos o método da interface query:

//Método utilizando specifications  
@GetMapping("/restaurantes/com-frete-gratis")  
public List<Restaurante> restauranteComFreteGratis(String nome) {  
 return restauranteRepository.findComFreteGratis(nome);  
}

Repare que agora utilizamos o método da interface RestauranteRepositoryQuerys.

## Annotation @NoRepositoryBean

A annotation @NoRepositoryBean serve para setar que o Spring não leve a interface anotada em conta para fins de instanciação de um repositório Spring Data JPA, ou seja o Spring Data Jpa não deve instanciar uma implementação para esta interface, ele deve ignorar.

## Habilitando JPA Repositories

Quando criamos novos repositórios como no caso de um repositório personalizado, temos que habilitar no arquivo raiz da aplicação do Spring:

@SpringBootApplication  
@EnableJpaRepositories(repositoryBaseClass = CustomJpaRepositoryImpl.class)  
public class AlgafoodApiApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(AlgafoodApiApplication.class, args);  
 }  
  
}

A annotation @EnableJpaRepositories serve para você passar o nome da implementação da interface personalizada que você criou, deste modo o Spring vai utilizar ela.

# EXPLORANDO MAIS DO JPA E HIBERNATE

## Mapeando relacionamento bidirecional com @OneToMany

Temos o relacionamento Muitos para Um e Um para Muitos, esse relacionamento é utilizado quando duas classes precisam conversar entre si, e cada uma delas precisa acessar os dados da outra classe, veja no diagrama abaixo:

COZINHA

RESTAURANTE

Vamos supor que vários restaurantes podem ter uma cozinha então anotamos a propriedade cozinha da classe Restaurante assim:

@ManyToOne  
@JoinColumn(nullable = false)  
private Cozinha cozinha;

E uma cozinha pode ter vários restaurantes, utilizamos a annotation @OneToMany na classe de cozinha assim:

@OneToMany(mappedBy = "cozinha")  
private List<Restaurante> restaurantes = new ArrayList<>();

Utilizamos o parâmetro mappedBy para definir a qual parâmetro estamos ligando este atributo na outra tabela, no nosso caso estamos ligando com o atributo de cozinha na classe Restaurante, o JPA vai entender esta relação e vai fazer a ligação.

Repare que utilizamos uma lista de Restaurantes e isso pode trazer problemas de referência cíclica na hora da serialização do JSON, ficando em um loop infinito trazendo dados repetidos infinitamente. Para resolver isto podemos utilizar o @JsonIgnore no atributo que desejamos ocultar da serialização do JSON, mas existem outras formas de tratar este problema, iremos ver isso mais para frente.

## Mapeando relacionamento @ManyToMany

Temos o relacionamento Muitos para Muitos, onde muitas determinada entidade possuem muitas de outra entidade, um exemplo seria, muitos restaurantes possuem muitas formas de pagamento, e não necessariamente a classe forma de pagamento precisa ter acesso a classe restaurante, neste caso só o restaurante tem acesso a classe formas de pagamento e não o contrário, assim utilizamos a annotation @ManyToMany, veja:

@ManyToMany  
@JoinTable(name = "restaurante\_forma\_pagamento",  
 joinColumns = @JoinColumn(name = "restaurante\_id"),  
 inverseJoinColumns = @JoinColumn(name = "forma\_pagamento\_id"))  
private List<FormaPagamento> formaPagamentos = new ArrayList<>();

Repare que também utilizamos a annotation @JoinTable para definir o nome da tabela de ligação que vai unir o relacionamento ManyToMany, porque esse relacionamento de muitos para muitos cria uma tabela de ligação que contém as chaves estrangeiras das duas tabelas interligadas, utilizamos a anotação @JoinTable para definir o nome desta tabela de ligação, com o parâmetro name. E também podemos definir o nome das colunas de chave estrangeira do relacionamento, para isso utilizamos o parâmetro joinColumns para definir o nome da coluna da chave da tabela referenciada em primeiro no relacionamento, no caso o restaurante e o parâmetro inverseJoinColumns que nomeia a segunda coluna referente a forma de pagamento, com isso podemos definir o nome personalizado para a tabela e as colunas.

## Mapeando classes incorporáveis com @Embedded e @Embeddable

Na criação de um sistema podemos decidir criar uma classe que contenha atributos separados em outra classe, mas que esses dados sejam gravados em apenas uma tabela de banco de dados, imagine com este exemplo: Temos uma classe de Restaurante que tem Endereços, resolvemos então separar em uma classe o endereço, mas os dados iram ser gravados juntos na mesma tabela de banco de dados. Para separar a classe endereço da de restaurante utilizamos a annotation @Embeddable, veja abaixo:

@Data  
@Embeddable  
public class Endereco {  
  
 @Column(name = "endereco\_cep")  
 private String cep;  
  
 @Column(name = "endereco\_logradouro")  
 private String logradouro;  
  
 @Column(name = "endereco\_numero")  
 private String numero;  
  
 @Column(name = "endereco\_complemento")  
 private String complemento;  
  
 @Column(name = "endereco\_bairro")  
 private String bairro;  
  
 @ManyToOne  
 @JoinColumn(name = "endereco\_cidade\_id")  
 private Cidade cidade;  
}

Com isso separamos a classe, agora temos que unir ela com a classe que desejamos, com a annotation @Embedded, veja abaixo:

@Embedded  
private Endereco endereco;

Adicionamos este atributo na classe Restaurante e com a annotation @Embedded vai fazer a ligação com a classe Endereco, desse modo vai ficar tudo em um banco de dados só.

## Annotation @CreationTimestamp e @UpdateTimestamp

A annotation @CreationTimestamp é uma anotação do Hibernate, que atribui a propriedade anotada que ela deve ser atribuída com uma data hora atual no momento que uma entidade for salva pela primeira vez.

E a annotation @UpdateTimestamp que também é uma anotação do Hibernate, ela atribui que a propriedade anotada deve ser atribuída com uma data hora atual no momento que uma entidade for atualizada.

@CreationTimestamp  
@Column(nullable = false)  
private LocalDateTime dataCadastro;  
  
@UpdateTimestamp  
@Column(nullable = false)  
private LocalDateTime dataAtualizacao;

## Entendendo o Eager Loading

Todas as associações que terminam com ToOne, ou seja por exemplo, @ManyToOne, usam por estratégia a Eager Loading, que por tradução significa (Um carregamento antecipado), por isso toda vez que uma instancia da classe for carregada a partir do banco de dados, ele vai carregar também as associações que usam Eager Loading.

Com isso ele irá fazer várias consultas no banco de dados, é o que chamamos de problema N+1, onde você faz uma consulta de uma entidade, e o JPA retorna várias consultas, é necessário prestar atenção e analisar como isso afeta a performance do seu sistema, isso não chega a ser um bug ou problema, tudo depende de analise para saber se isto irá afetar a performance do seu projeto.

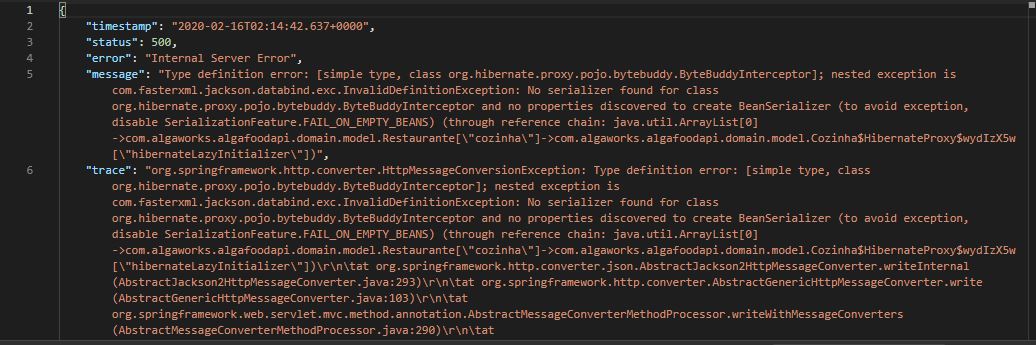
### Alterando a estratégia de fetching Eager para Lazy Loading

Para converter uma estratégia de Eager para Lazy basta adicionar o parâmetro a annotation que termina com ToOne, veja:

@JsonIgnore  
@ManyToOne(fetch = FetchType.*LAZY*)  
@JoinColumn(nullable = false)  
private Cozinha cozinha;

Agora se ela estiver anotada com @JsonIgnore o JPA não vai mais realizar a consulta no banco desta entidade anotada com o parâmetro fetch.

Mas caso a instancia acima não estiver anotada com @JsonIgnore ocorreria um erro, veja:



Em uma consulta Lazy o JPA cria uma classe proxy em tempo de execução que é como um container que aguarda para capturar as informações da instancia, e para que este erro não ocorra temos que ignorar o seguinte atributo que está sendo iniciado pela classe Proxy dinâmica do JPA, [\”hibernateLazyInitializer\”], veja:

@JsonIgnoreProperties("hibernateLazyInitializer")  
@ManyToOne(fetch = FetchType.*LAZY*)  
@JoinColumn(nullable = false)  
private Cozinha cozinha;

Com isso estamos ignorando o atributo da classe Proxy para que tudo funcione corretamente. Caso tivéssemos o @JsonIgnore não seria necessário esta annotation @JsonIgnoreProperties.

## Entendendo o Lazy Loading

Todas as associações que terminam com ToMany, ou seja por exemplo, @ManyToMany, usam por estratégia a Lazy Loading, que por tradução significa (Um carregamento preguiçoso), com isso o JPA só irá fazer a query de consulta se a gente estiver utilizando o atributo, ou seja ele faz o carregamento por demanda.

### Alterando a estratégia de fetching Lazy para Eager Loading

Podemos alterar a estratégia fetching de Lazy para Eager, apenas utilizando o parâmetro fetch nas anotações que terminam com ToMany, veja:

@JsonIgnore  
@ManyToMany(fetch = FetchType.*EAGER*)  
@JoinTable(name = "restaurante\_forma\_pagamento",  
 joinColumns = @JoinColumn(name = "restaurante\_id"),  
 inverseJoinColumns = @JoinColumn(name = "forma\_pagamento\_id"))  
private List<FormaPagamento> formaPagamentos = new ArrayList<>();

E mesmo com a annotation @JsonIgnore serão realizadas muitas consultas do JPA no banco de dados, essa abordagem de converter de Lazy para Eager quase nunca é utilizada quando possuímos associações ToMany.

## Resolvendo problema N+1 com fetch join na JPQL

Para resolver o problema de N+1, onde o JPA realiza várias consultas extras após a consulta que foi especificada devido as associações Eager e Lazy, e isso ocorre principalmente em consultas findAll que retornam todos os dados de uma entidade, e com isso pode afetar a performance, para contornar isto podemos substituir o método de consulta findAll padrão gerado pela interface do Spring Data JPA, por uma consulta JPQL personalizada, veja:

@Repository  
public interface RestauranteRepository extends CustomJpaRepository<Restaurante, Long>, RestauranteRepositoryQuerys, JpaSpecificationExecutor<Restaurante> {  
@Query("from Restaurante r join r.cozinha left join fetch r.formaPagamentos")  
 List<Restaurante> findAll();

Na interface Repository da entidade você cria uma query JPQL personalizada, e esta query deve ser feita utilizando join.

Existe uma peculiaridade, repare na associação de cozinha,

@ManyToOne  
@JoinColumn(nullable = false)  
private Cozinha cozinha;

Temos um ToOne, e com isso na query JPQL é feito o fetch automaticamente, mas na associação ToMany de formaPagamentos:

@ManyToMany  
@JoinTable(name = "restaurante\_forma\_pagamento",  
 joinColumns = @JoinColumn(name = "restaurante\_id"),  
 inverseJoinColumns = @JoinColumn(name = "forma\_pagamento\_id"))  
private List<FormaPagamento> formaPagamentos = new ArrayList<>();

É necessário forçar o fetch na query, porque se o fetch não for especificado as consultas ocorrerão da mesma forma não adiantando utilizar JPQL,

from Restaurante r join r.cozinha left join fetch r.formaPagamentos

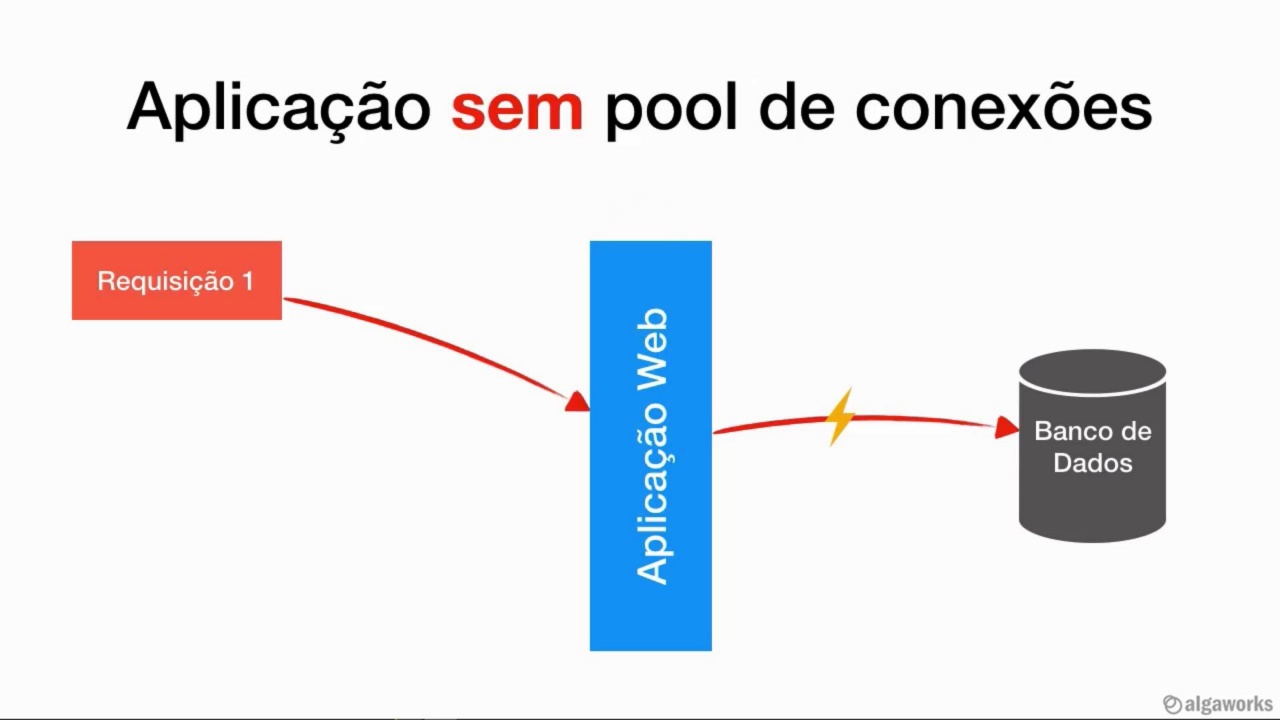
O left join serve para retornar formaPagamentos mesmo ela sendo nula, garante essa segurança, e o fetch é para forçar esse join a retornar apenas a consulta correspondente. Podemos utilizar este fetch forçado em todos os tipos sendo Eager ou Lazy, para ficar legível e garantir que funcione de forma eficaz, veja:

@Query("from Restaurante r join fetch r.cozinha left join fetch r.formaPagamentos")  
List<Restaurante> findAll();

# POOL DE CONEXÕES E FLYWAY

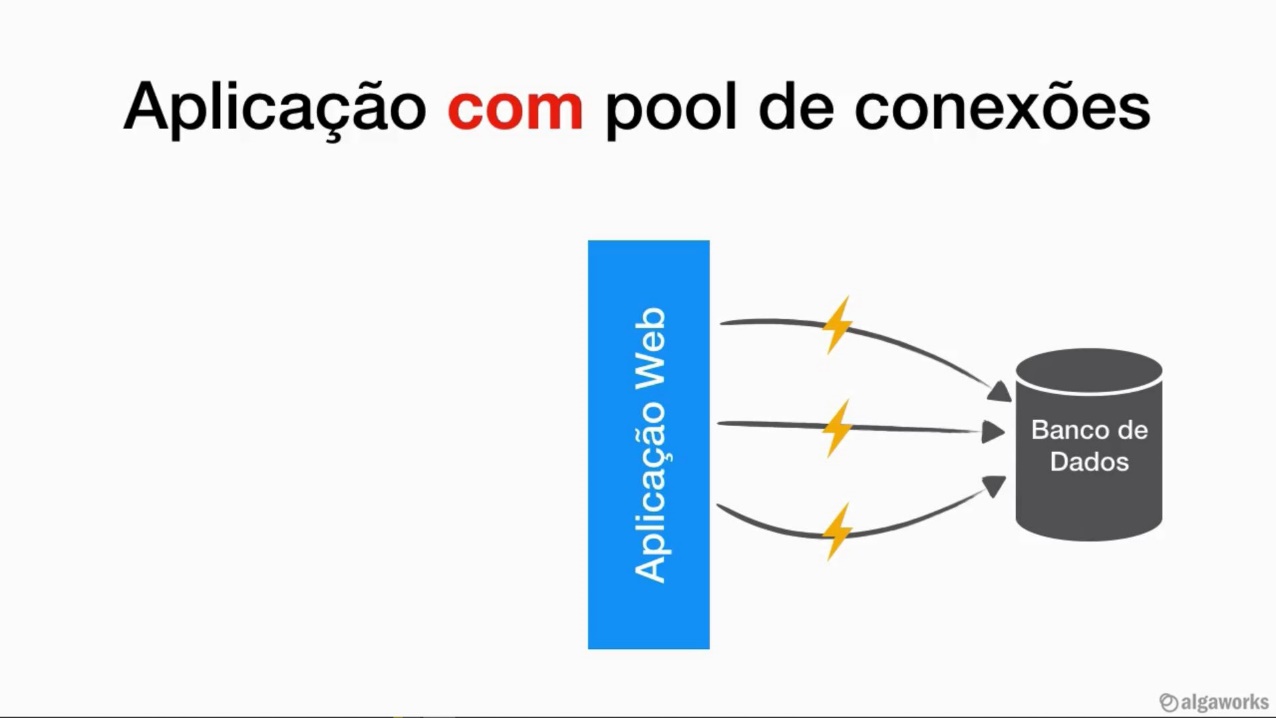
## Pool de conexão

É muito importante conhecer sobre pool de conexão quando desenvolvemos aplicações web, primeiro vamos entender o que é uma aplicação sem pool de conexões.

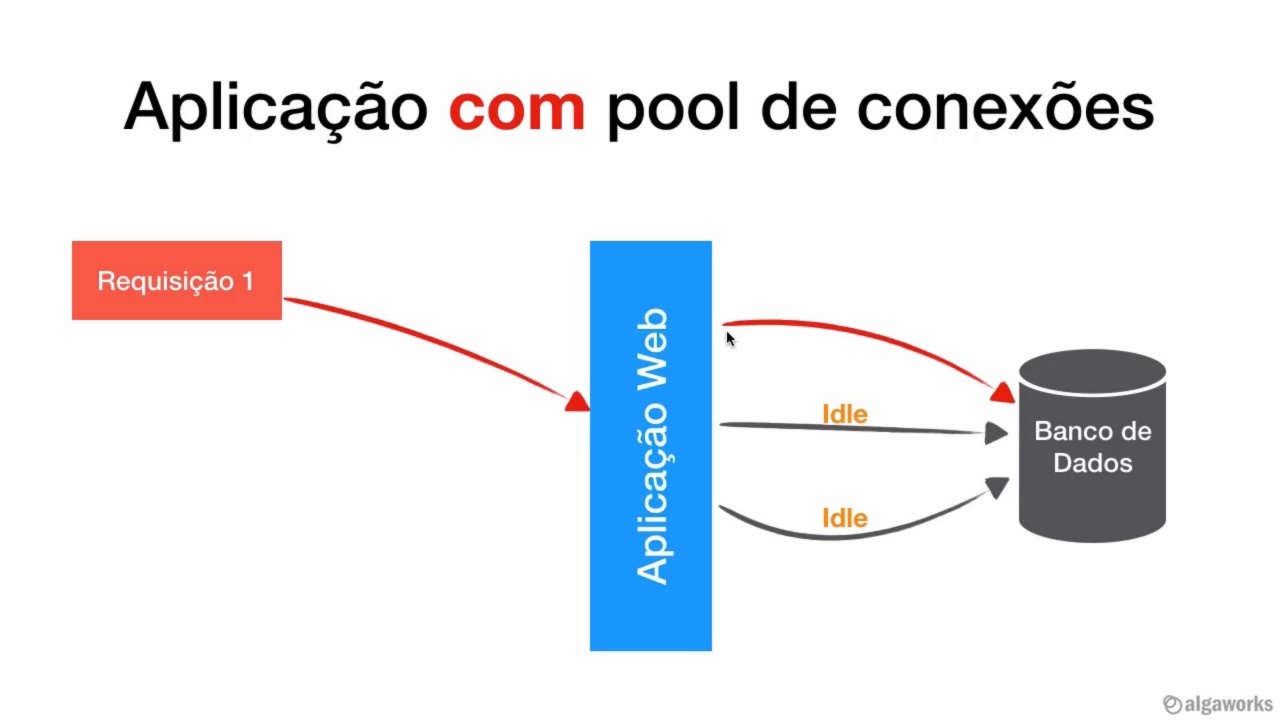


Neste modelo uma aplicação faz uma requisição a aplicação web, a aplicação se conecta com o banco de dados, e faz o que a requisição pediu, e encerra a conexão, a aplicação web continua funcionando, o banco de dados continua funcionando mas sem conexão com a aplicação, e a cada requisição o ciclo se repete.

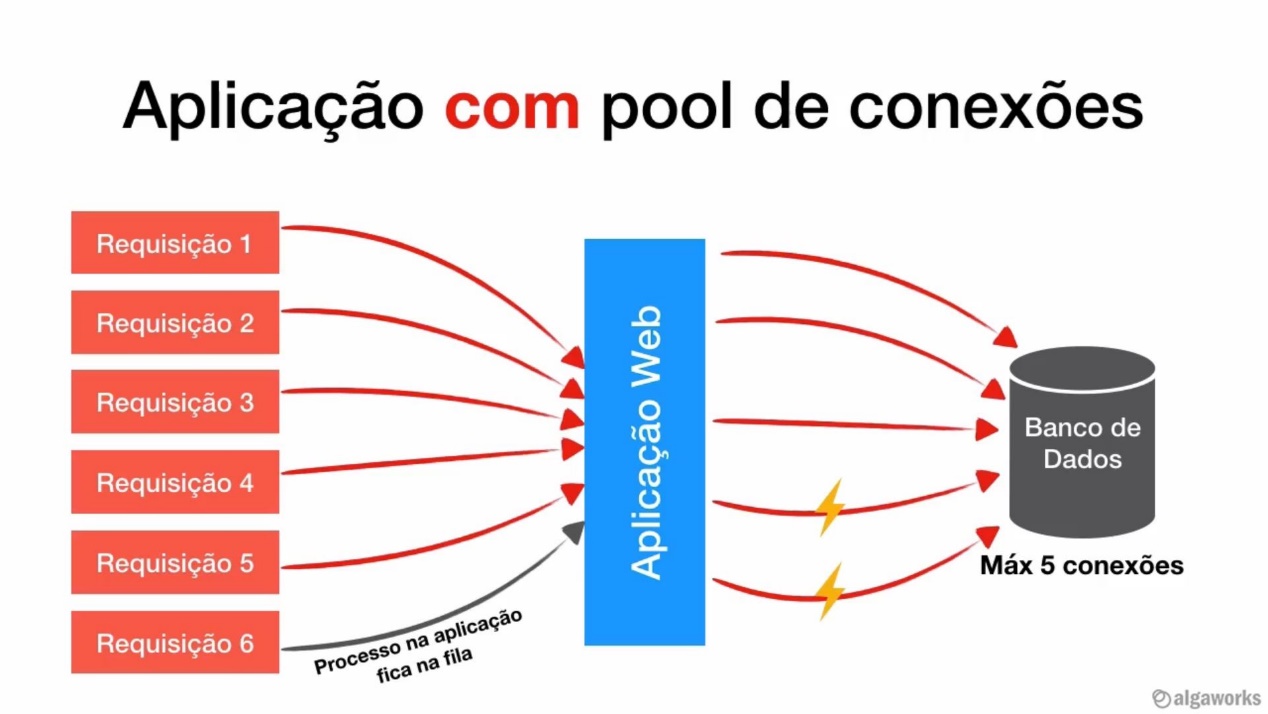
E temos também aplicações com pool de conexões.



É um componente de software que mantém um conjunto de conexões com o banco de dados para reutilização em uma aplicação, um conjunto, grupo de conexões. Essa abordagem funciona assim, uma aplicação web mantém um mínimo de conexões especificado abertas prontas para receber requisições, essas conexões se mantêm ociosas aguardando uma requisição.



E quando o banco resolve a requisição ele não fecha a conexão, ele volta para o estado de ociosidade, as conexões se mantêm abertas.



E quando há um conjunto grande de conexões que ultrapassam o limite mínimo estipulado de conexões abertas o pool abre novas conexões no valor máximo estipulado, isso tudo é configurável, e como no exemplo da imagem acima, temos 5 conexões e 6 requisições, a 6 fica na fila até que uma conexão esteja livre e ela possa ser atendida.

Quando todas as requisições são atendidas as 5 conexões ficam ociosas novamente e não fecham, o mais comum é pela configuração que as conexões extras abertas para suprir a demanda sejam fechadas depois de um certo tempo estipulado, ou seja o pool não as fecha imediatamente depois de utiliza-las.

## Conhecendo o Hikari solução Pool de conexões Springboot

O Springboot já configura automaticamente um pool de conexões, ele utiliza o HikariCP, por padrão o Hikari configura 10 conexões para a aplicação, mas podemos configurar isso.

No arquivo de application.properties do spring, você adiciona propriedades de configuração do Hikari, veja algumas:

Esta seta o máximo de conexões que podem ser abertas no pool.

spring.datasource.hikari.maximum-pool-size=5

Esta seta o mínimo de conexões iniciais abertas.

spring.datasource.hikari.minimum-idle=3

Esta configura o tempo que as conexões extras que foram abertas, fiquem ligadas.

spring.datasource.hikari.idle-timeout=10000

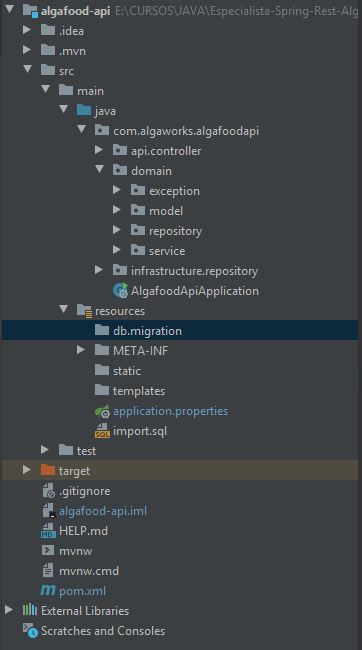
## Flyway ferramenta de versionamento de schemas de banco de dados

O Flyway serve para automatizar o versionamento de schemas de banco de dados, ele se integra muito bem com o Spring, para adicionar o Flyway no projeto Spring basta adicionar a dependência dele no arquivo pom.xml, pelo Maven ou Graddle depende do gerenciador de dependências que você estiver utilizando, neste exemplo eu uso Maven, adicione as seguintes linhas:

<dependency>  
 <groupId>org.flywaydb</groupId>  
 <artifactId>flyway-core</artifactId>  
</dependency>

A versão depende em qual época você acessar, para garantir pesquise no site do Maven e pegue o script atual para importar a ferramenta.

Após isso crie um diretório na pasta de resources do projeto, (db/migration), veja na imagem abaixo:



E dentro da pasta migration você ira inserir os arquivos de migração. Os arquivos de migração devem seguir uma nomenclatura padrão, para que o Flyway entenda e consiga ler os arquivos. Essa nomenclatura dos arquivos deve ser assim:

V001\_\_nome-arquivo.sql

As partes obrigatórias são o V maiúsculo, seguido de numeração de arquivo, depois os 2 \_ underline e por fim o .sql.

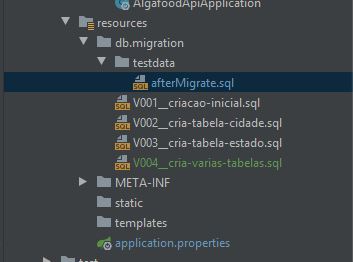
Depois do V maiúsculo pode ter o tipo de numeração que você desejar, pode ser, 1.1, 1.2... ou 1, 2, 3... ou 00001, 00002... vai depender do seu gosto, e o nome do arquivo também é a seu critério. Dentro deste arquivo você irá inserir os scripts SQL, de create, update, e automaticamente assim que a aplicação for iniciada o flyway irá ler os arquivos e fazer o trabalho no banco de dados. É importante frisar que após criar uma migration não é recomendado alterar a mesma versão, caso você precise fazer uma alteração é recomendado que você crie outra migration V02, etc, porque o flyway faz uma checagem de checksun e compara com as modificações do arquivo, caso o arquivo mude uma virgula, espaço, o checksun muda e o flyway da um erro na hora da aplicação subir. Por isso tome cuidado e sempre procure criar versões para cada mudança que for fazer no banco de dados.

## CRIANDO ARQUIVO DE POPULAÇÃO DE DADOS NO FLYWAY

Para podermos ter dados populados no banco utilizando o flyway podemos criar um arquivo de população de dados, esse arquivo precisa ser chamado de: afterMigrate.sql

Este arquivo é um arquivo de call-back que irá ser executado após o flyway finalizar todas as migrações.

Para estruturar esse arquivo no projeto de acordo com o ambiente de desenvolvimento, temos que criar uma pasta dentro da pasta de db/migrations para guardar esse arquivo afterMigrate.sql, e no arquivo de properties especifico do desenvolvimento, seja ele dev, prod, stage, ou o que for, você pode especificar a leitura deste arquivo ou omitir, facilitando assim o desenvolvimento sem interferir na produção. Veja um exemplo:



E dentro do application.properties inserir esta linha:

spring.flyway.locations=classpath:db/migration,classpath:db/testdata

E nos outros arquivos properties, como o de produção podemos omitir esta linha e ele irá ler somente os arquivos de migrations, ignorando o arquivo afterMigrate.sql que está na pasta testdata. A proposito a pasta testdata pode ter qualquer nome.

### Nota - Adicionando arquivos untracked no git hub

Pode acontecer de quando formos adicionar arquivos ao git alguns arquivos podem ficar untracked files impossibilitando adicionar tal arquivo, para esses casos basta usar este comando abaixo:

Página do stack overflow com a solução:

<https://stackoverflow.com/questions/8470547/git-commit-a-untracked-files>

git add --all

git commit -am “Sua mensagem de commit”

# TRATAMENTO E MODELAGEM DE ERROS DA API

## Exceções customizadas com a annotation @ResponseStatus

Podemos lançar um código de resposta HTTP diretamente na exception com a annotation @ResponseStatus, veja:

@ResponseStatus(value = HttpStatus.*NOT\_FOUND*, reason = "Entidade não encontrada")  
public class EntidadeNaoEncontradaException extends RuntimeException{  
 private static final long *serialVersionUID* = 1L;  
  
 public EntidadeNaoEncontradaException(String message) {  
 super(message);  
 }  
}

A linha:

@ResponseStatus(value = HttpStatus.*NOT\_FOUND*, reason = "Entidade não encontrada")

Tem a função de lançar o código de status 404 not found quando esta exception for lançada, e o reason é o texto do erro que pode ser editado para ser exibido na exception.

Inserindo o ResponseStatus nas exceptions podemos deixar o código mais elegante no controller, veja:

As exceptions:

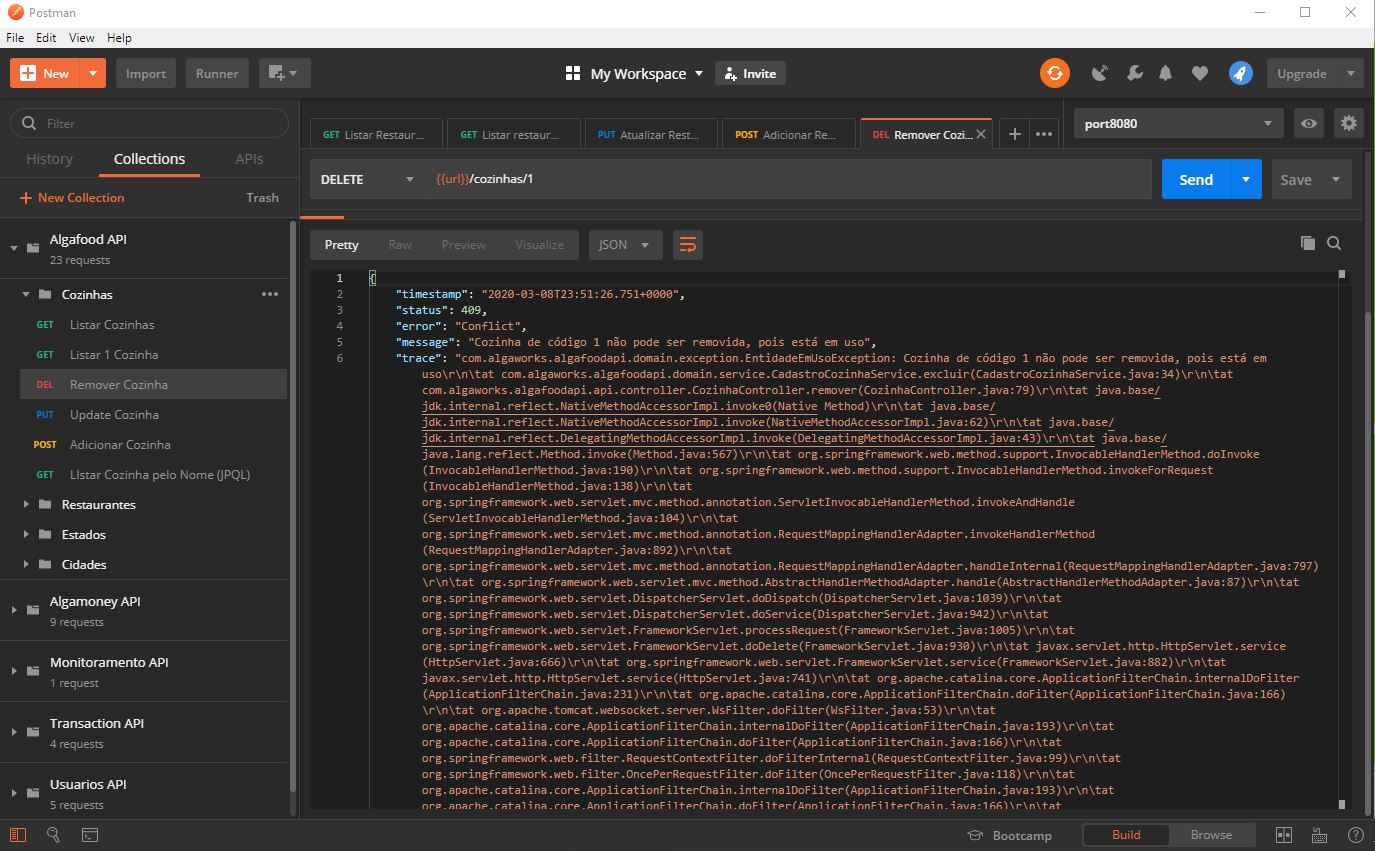
@ResponseStatus(HttpStatus.*CONFLICT*)  
public class EntidadeEmUsoException extends RuntimeException{  
 private static final long *serialVersionUID* = 1L;  
  
 public EntidadeEmUsoException(String message) {  
 super(message);  
 }  
}

@ResponseStatus(value = HttpStatus.*NOT\_FOUND*)//, reason = "Entidade não encontrada")  
public class EntidadeNaoEncontradaException extends RuntimeException{  
 private static final long *serialVersionUID* = 1L;  
  
 public EntidadeNaoEncontradaException(String message) {  
 super(message);  
 }  
}

O controller:

/\*@DeleteMapping("/{id}")  
public ResponseEntity<?> remover(@PathVariable Long id) {  
 try {  
 cadastroCozinha.excluir(id);  
 return ResponseEntity.noContent().build();  
  
 } catch (EntidadeNaoEncontradaException e) {  
 return ResponseEntity.notFound().build();  
  
 } catch (EntidadeEmUsoException e) {  
 return ResponseEntity.status(HttpStatus.CONFLICT).build();  
 }  
}\*/  
  
@DeleteMapping("/{id}")  
@ResponseStatus(HttpStatus.*NO\_CONTENT*)  
public void remover(@PathVariable Long id) {  
 cadastroCozinha.excluir(id);  
}

Removemos os try catch, mas esse método tem um porém que não conseguimos personalizar o corpo do erro, no máximo conseguimos modificar somente a descrição.



## Lançando exceções do tipo ResponseStatusException

A partir do Spring 5 foi introduzido uma nova classe de exception que seve como uma exception base sem precisar criar uma classe de exception. Essa classe é útil somente para projetos simples, ou protótipos, ou se você não quer perder tempo criando exceptions, e ela também não permite a customização do corpo da mensagem da exception.

@DeleteMapping("/{id}")  
@ResponseStatus(HttpStatus.*NO\_CONTENT*)  
public void remover(@PathVariable Long id) {  
 try {  
 cadastroCozinha.excluir(id);  
 } catch (EntidadeNaoEncontradaException e) {  
 throw new ResponseStatusException(HttpStatus.*NOT\_FOUND*, e.getMessage());  
 }  
}

### Nota - Services com conhecimento de http status:

Não é recomendável as classes do pacote de domínio, como as classes de service ter conhecimento dos códigos de HTTP, é uma má pratica, as classes de serviço não podem ter contato com http status ou qualquer forma de manipulação de status http.

## Estendendo ResponseStatusException

Podemos dar um extends na classe de exception com a classe ResponseStatusException, deste modo vamos poder modificar além da mensagem também o status http, basta fazer um extends:

public class EntidadeNaoEncontradaException extends ResponseStatusException {  
 private static final long *serialVersionUID* = 1L;  
  
 public EntidadeNaoEncontradaException(HttpStatus status, String message) {  
 super(status, message);  
 }  
  
 public EntidadeNaoEncontradaException(String message) {  
 this(HttpStatus.*NOT\_FOUND*, message);  
 }  
}

Criamos um construtor para a chamada da propriedade da classe ResponseStatusException, e modificamos o segundo construtor para jogar um código http padrão, tendo assim uma sobrecarga de métodos.

A vantagem de estender a classe ResponseStatusException é que podemos modificar o status http como quisermos, diferente de quando utilizamos a annotation @ResponseStatus na classe de exception, porque assim ela ficava estática sem poder modificar e se precisássemos de jogar outro http status teríamos de criar outra classe de exception para isso, mas estendendo a classe ResponseStatusException não precisamos fazer isto, porque podemos lançar diferentes status http no construtor.

## Tratando exceções dentro do controlador com @ExceptionHandler

Para podermos fazer a edição do corpo da mensagem de erro, podemos utilizar métodos dentro do controller que servem para tratar respostas de exceções, para isso basta criar métodos com a annotation @ExceptionHandler, veja:

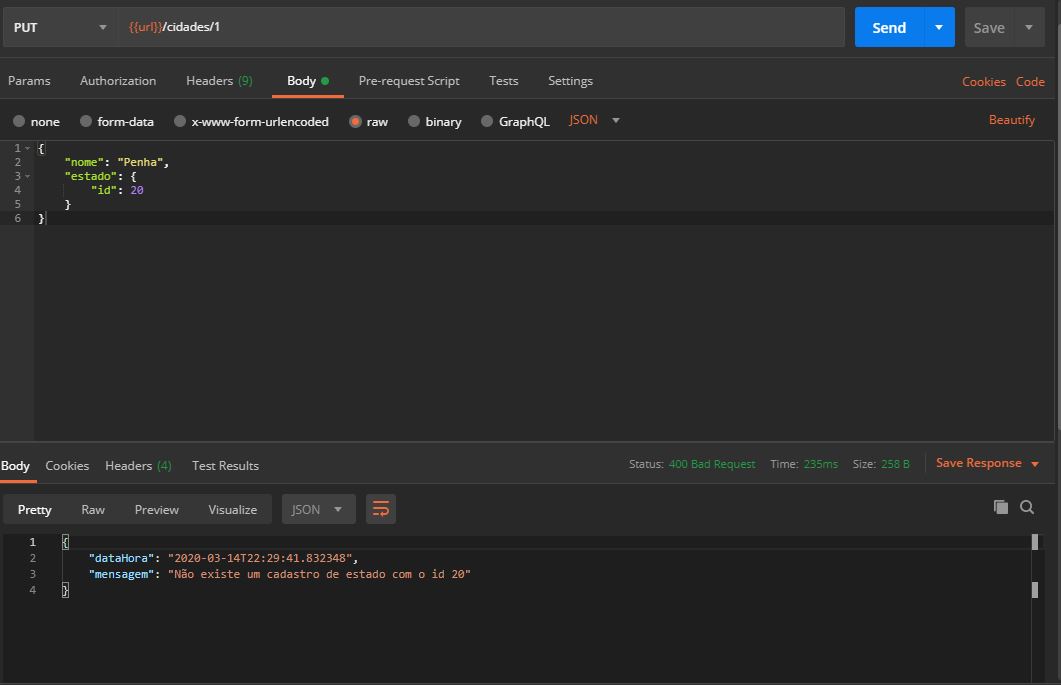
@ExceptionHandler(EntidadeNaoEncontradaException.class)  
public ResponseEntity<?> tratarEntidadeNaoEncontradaException(EntidadeNaoEncontradaException e){  
  
 Problema problema = Problema.*builder*()  
 .dataHora(LocalDateTime.*now*())  
 .mensagem(e.getMessage()).build();  
  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(problema);  
}

A annotation recebe como argumento, a classe de exception que você deseja trabalhar, e setamos como retorno a classe ResponseEntity<?>, a annotation pode receber várias classes de exceptions como argumento, não apenas uma.

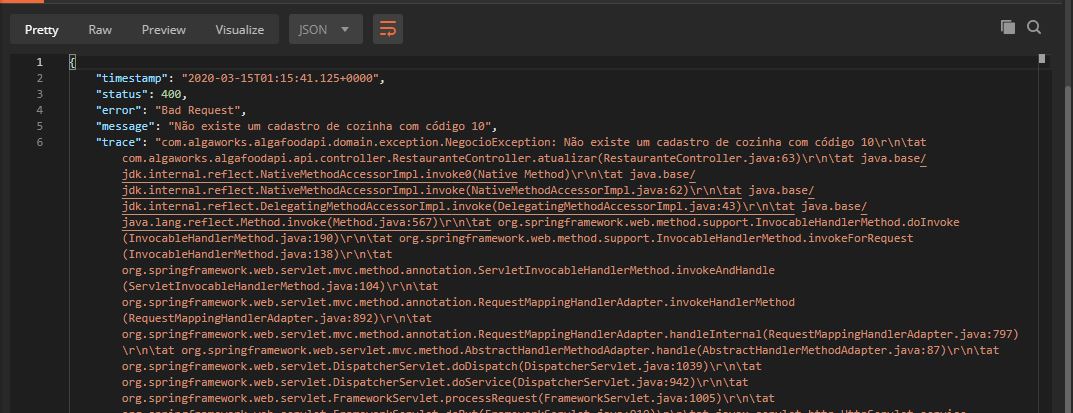
O método pode receber como parâmetro a exception que será trabalhada, e com isso podemos retornar um ResponseEntity com o body personalizado. Podemos também criar uma classe responsável por setar atributos de personalização de mensagem, como visto neste método acima, temos a classe Problema que está servindo para criar um corpo personalizado, veja a classe:

@Getter  
@Builder  
public class Problema {  
  
 private LocalDateTime dataHora;  
 private String mensagem;  
}

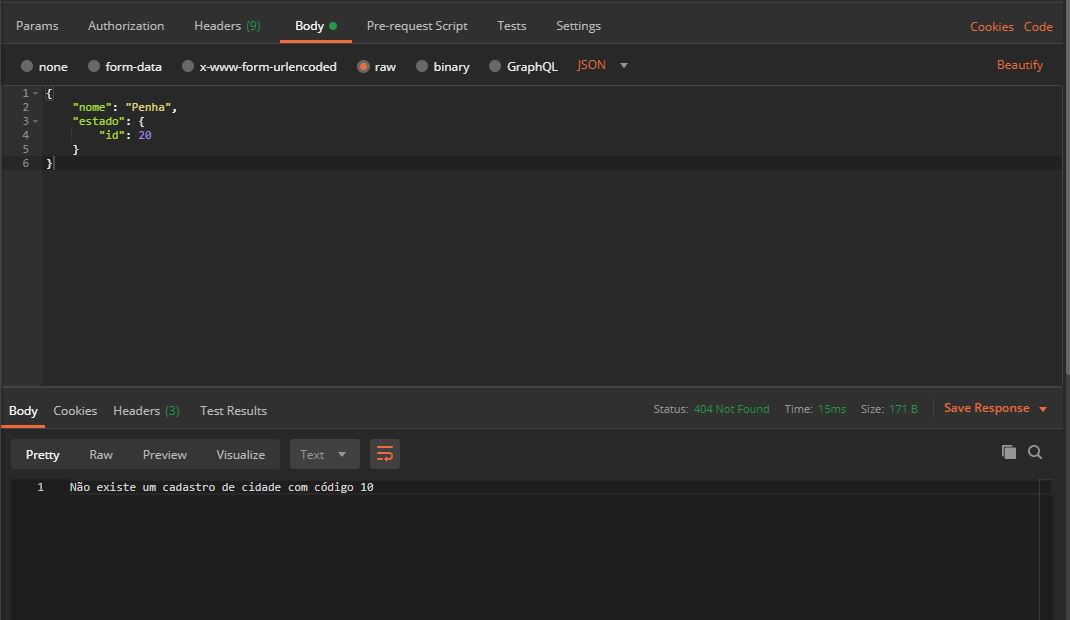
Utilizamos o Lombok para reduzir o código boilerplate, e a annotation @Builder tem como função permitir a instanciação da classe como um builder, uma forma diferente do new() comum do Java, a utilização desse método irá criar uma mensagem customizada assim, veja:



Repare que a mensagem é bem diferente desta, sem o tratamento:



A cada endpoint que é executado e dispara uma exception, se esta exception está sendo tratada em uma função de ExceptionHandler ela será exibida de forma personalizada de acordo com o tratamento no método, por padrão se não houver tratamento com objetos a mensagem será exibida assim:



## Tratando exceções globais com @ExceptionHandler e @ControllerAdvice

Para podermos utilizar os métodos de ExceptionHandler de forma global em todos os endpoints da aplicação, criamos uma classe que é anotada com a annotation @ControllerAdvice e colocamos os métodos @ExceptionHandler dentro desta classe, veja:

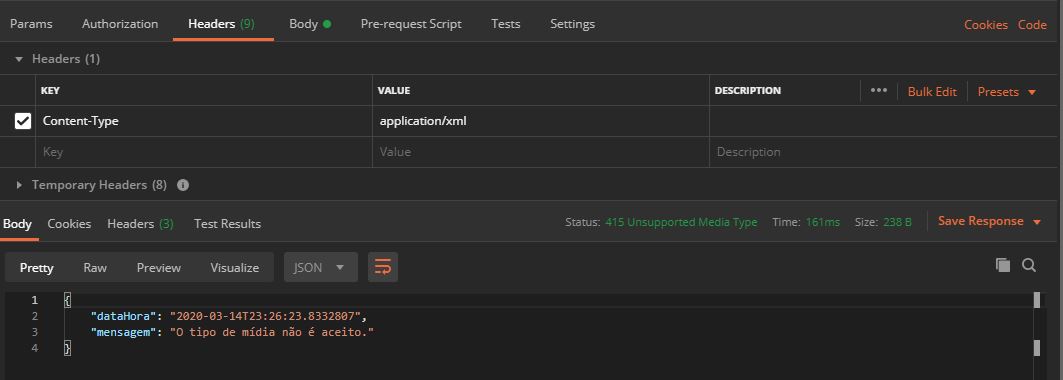
@ControllerAdvice  
public class ApiExceptionHandler {  
  
 @ExceptionHandler(EntidadeNaoEncontradaException.class)  
 public ResponseEntity<?> tratarEntidadeNaoEncontradaException(EntidadeNaoEncontradaException e){  
  
 Problema problema = Problema.*builder*()  
 .dataHora(LocalDateTime.*now*())  
 .mensagem(e.getMessage()).build();  
  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(problema);  
 }  
  
 @ExceptionHandler(NegocioException.class)  
 public ResponseEntity<?> tratarNegocioException(NegocioException e){  
  
 Problema problema = Problema.*builder*()  
 .dataHora(LocalDateTime.*now*())  
 .mensagem(e.getMessage()).build();  
  
 return ResponseEntity.*status*(HttpStatus.*BAD\_REQUEST*).body(problema);  
 }  
}

Feito isso podemos acionar qualquer endpoint na aplicação que esta classe de exception handler irá ser chamada, não apenas na classe de controller como estava antes, se esses métodos estiverem apenas em uma classe de controller especifica só ira funcionar nas chamadas de controle daquela determinada classe, mas se removermos os métodos desta determinada classe e colocarmos em uma classe global de @ControllerAdvice os métodos iram funcionar de forma global em todos os endpoints da aplicação independente da classe de controller.

Podemos customizar exceptions externas que não foram criadas por nós, exceptions geradas pelo Spring, veja:

@ExceptionHandler(HttpMediaTypeNotSupportedException.class)  
public ResponseEntity<?> tratarHttpMediaTypeNotSupportedException(){  
  
 Problema problema = Problema.*builder*()  
 .dataHora(LocalDateTime.*now*())  
 .mensagem("O tipo de mídia não é aceito.").build();  
  
 return ResponseEntity.*status*(HttpStatus.*UNSUPPORTED\_MEDIA\_TYPE*).body(problema);  
}

Veja o retorno:



## Estendendo com a classe ResponseEntityExceptionHandler

Para podermos utilizar os benefícios de tratar exceptions internas do Spring de uma forma global, podemos estender a classe ResponseEntityExceptionHandler, é uma classe de conveniência para classe de exceptions globais, é uma implementação padrão que já trata exceptions internas do Spring.

@ControllerAdvice  
public class ApiExceptionHandler extends ResponseEntityExceptionHandler {

## Customizando a resposta de ResponseEntityExceptionHandler

Podemos sobrescrever o método handleExceptionInternal para tratar todas as exceptions que são lançadas pela classe ResponseEntityExceptionHandler, o método handleExceptionInternal que está na classe ResponseEntityExceptionHandler é lançado por todas as exceptions desta classe, e podemos utilizar ele para tratar todas as exceptions lançadas, porque ela é lançada em todas os métodos, a maioria dos body que são o corpo da exception estão setados como null por padrão, podemos customizar.

O que precisamos fazer é sobrescrever o método handleExceptionInternal na nossa classe para poder customizar as nossas exceptions.

@Override  
protected ResponseEntity<Object> handleExceptionInternal(Exception ex, Object body, HttpHeaders headers, HttpStatus status, WebRequest request) {  
  
 body = Problema.*builder*()  
 .dataHora(LocalDateTime.*now*())  
 .mensagem(status.getReasonPhrase())  
 .build();  
  
 return super.handleExceptionInternal(ex, body, headers, status, request);  
}

Sobrescrevemos o método e passamos os parâmetros que queremos no body, utilizando ainda a nossa classe Problema.java.

Como o método handleExceptionInternal precisa de um parâmetro request, podemos passar uma instância de WebRequest que o Spring vai entender.

O parâmetro status.getReasonPhrase() serve para capturar a descrição que descreve o status que está sendo retornado na resposta.

Com este método sobrescrito podemos utilizar ele nos nossos métodos também, veja um exemplo da modificação que fizemos com o novo método.

@ExceptionHandler(EntidadeNaoEncontradaException.class)  
public ResponseEntity<?> tratarEntidadeNaoEncontradaException(EntidadeNaoEncontradaException ex, WebRequest request) {  
  
 return handleExceptionInternal(ex, ex.getMessage(), new HttpHeaders(), HttpStatus.*NOT\_FOUND*, request);  
}

No método sobrescrito podemos customizar a mensagem de erro que irá retornar, como agora temos o tratamento dos nossos erros e erros internos que não temos o controle do texto da mensagem, podemos utilizar os dois textos na mensagem. Veja como:

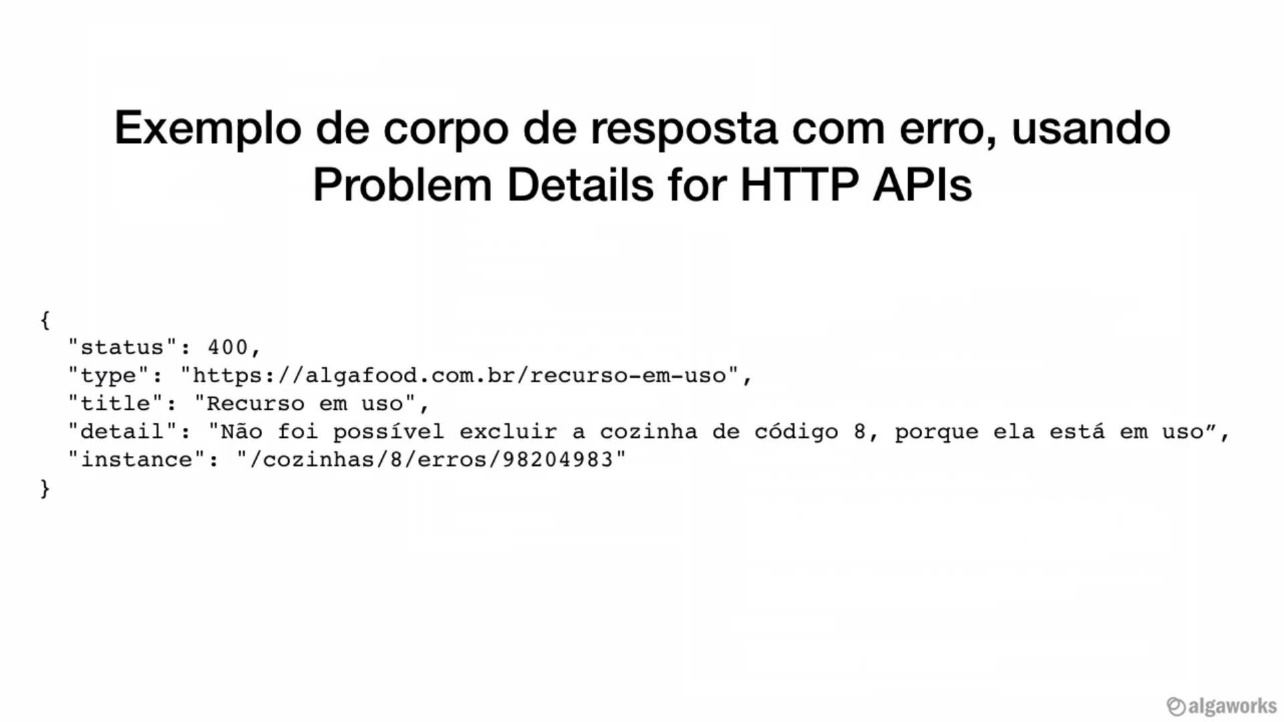
@Override  
protected ResponseEntity<Object> handleExceptionInternal(Exception ex, Object body, HttpHeaders headers, HttpStatus status, WebRequest request) {  
  
 if (body == null) {  
 body = Problema.*builder*()  
 .dataHora(LocalDateTime.*now*())  
 .mensagem(status.getReasonPhrase())  
 .build();  
 } else if (body instanceof String) {  
 body = Problema.*builder*()  
 .dataHora(LocalDateTime.*now*())  
 .mensagem((String) body)  
 .build();  
 }  
  
 return super.handleExceptionInternal(ex, body, headers, status, request);  
}

Com esta verificação, identificamos se o corpo vier null a gente seta o corpo da mensagem capturando a mensagem padrão que vai vir pelo getReasonPhrase(), ou se não vier nula e for uma instância de String a gente seta o corpo da mensagem com o body, fazendo o casting para String, com isso a nossa mensagem que foi trabalhada nas nossas exceptions será exibida quando entrar em uma exception tratada por nós.

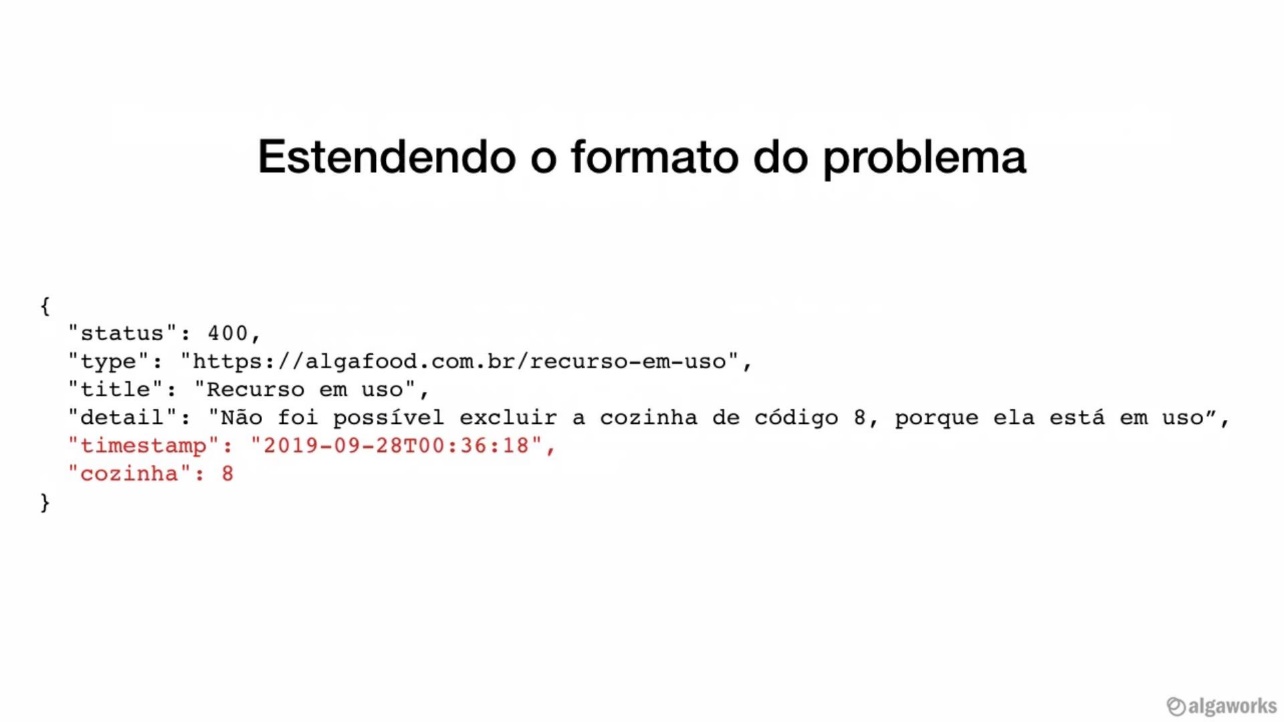
## Conhecendo a RFC 7807 (Problem Details for HTTP APIs)

A Problem Details for HTTP APIs é uma especificação que padroniza o formato de resposta de envio de erros da API, existem outras especificações como a jsonapi.org, a vnd.error, e esta que estamos falando agora, ela é especificada pela (IETF) Internet Engineering Task Force.

Um corpo de resposta padrão utilizando esta especificação é assim:



Podemos estender esse formato adicionando novos atributos, veja:



Uma pergunta recorrente é se podemos expor detalhes internos do erro, como a stacktrace? Sim pode ser incluída sem problemas, mas geralmente e especificamente em APIs públicas não é recomendável incluir detalhes técnicos internos da API, pois podemos expor dados de terceiros.

## Padronizando o formato no corpo com a RFC 7807

Podemos instanciar um ProblemBuilder nos métodos de uma forma diferente, para isso podemos criar um método para instanciar um ProblemBuilder antes de ser um builder, com isso podemos setar ele nos métodos e dar um .builder() no final para assim criarmos um builder. Veja a implementação:

Primeiro criamos um enum para representar o titulo e a uri que é o type da mensagem:

@Getter  
public enum ProblemType {  
  
 *ENTIDADE\_NAO\_ENCONTRADA*("/entidade-nao-encontrada", "Entidade não encontrada"),  
 *ERRO\_NEGOCIO*("/erro-negocio", "Violação de regra de negócio"),  
 *ENTIDADE\_EM\_USO*("/entidade-em-uso", "Entidade em Uso");  
  
 private String title;  
 private String uri;  
  
 ProblemType(String path, String title) {  
 this.uri = "https://algafood.com.br" + path;  
 this.title = title;  
 }  
}

E com isso criamos o método:

private Problem.ProblemBuilder createProblemBuilder(HttpStatus status, ProblemType problemType, String detail) {  
 return Problem.*builder*()  
 .status(status.value())  
 .type(problemType.getUri())  
 .title(problemType.getTitle())  
 .detail(detail);  
}

Esse método não instancia um builder em si, ele apenas instancia um ProblemBuilder que irá ser instanciado nos métodos acima como um builder. Veja a implementação:

@ExceptionHandler(EntidadeNaoEncontradaException.class)  
public ResponseEntity<?> handleEntidadeNaoEncontradaException(EntidadeNaoEncontradaException ex, WebRequest request) {  
  
 HttpStatus status = HttpStatus.*NOT\_FOUND*;  
  
 ProblemType problemType = ProblemType.*ENTIDADE\_NAO\_ENCONTRADA*;  
  
 String detail = ex.getMessage();  
  
 Problem problem = createProblemBuilder(status, problemType, detail).build();  
  
 return handleExceptionInternal(ex, problem, new HttpHeaders(), status, request);  
}

Com isso criamos uma instancia de Problem a partir do método e só passamos os parâmetros necessários para o método. Simplificamos o código.

## Ocultando valores nulos no corpo da resposta HTTP

Podemos ocultar atributos que estão nulos da resposta de uma entidade com a annotation @JsonInclude, e passando como parâmetro o Include.NON\_NULL que seta que propriedades com valores nulos não seram exibidas no corpo da resposta.

@JsonInclude(JsonInclude.Include.*NON\_NULL*)  
@Getter  
@Builder  
public class Problem {  
  
 private Integer status;  
 private String type;  
 private String title;  
 private String detail;  
}

## Tratando a causa da exception na desserialização

Quando lançamos uma exception ela pode conter uma descrição muito vaga do problema sem detalhes específicos que poderiam ajudar o consumidor da API, podemos customizar e trazer mais detalhes para a resposta, devemos inspecionar a exception lançada e ver qual a exception de causa que foi lançada junto com a exception seria a rootCause, essa exception contém os detalhes sobre o que ocorreu, permitindo a gente jogar mais detalhes para o consumidor da API.

Quando é lançado um erro HttpMessageNotReadable temos também o lançamento de outra exception dentro dela que é a InvalidFormatException esta contém os detalhes do ocorrido, e a partir dela podemos customizar o corpo da resposta com mais detalhes. Precisamos buscar qual é a exceção raiz e para isso podemos adicionar uma biblioteca no pom.xml do Maven que tem classes utilitárias que fornece vários métodos para trabalhar de maneira simplificada com strings, com números, com exceptions e outras coisas. A dependência é:

<dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-lang3</artifactId>  
</dependency>

Após adicionada esta biblioteca podemos utilizar seus métodos e um deles a gente utiliza para capturar a causa raiz, veja:

Throwable rootCause = ExceptionUtils.*getRootCause*(ex);

Esse método faz a busca em toda a hierarquia da exception e captura a causa raiz. É necessário prestar atenção na hora de importar o package deste método corretamente, o package é este:

import org.apache.commons.lang3.exception.ExceptionUtils;

Assim sobrescrevemos o método handleHttpMessageNotReadable para poder tratar de forma mais detalhada o erro, veja:

@Override  
protected ResponseEntity<Object> handleHttpMessageNotReadable(HttpMessageNotReadableException ex,  
 HttpHeaders headers, HttpStatus status, WebRequest request) {  
  
 Throwable rootCause = ExceptionUtils.*getRootCause*(ex);  
  
 if (rootCause instanceof InvalidFormatException) {  
 return handleInvalidFormatException((InvalidFormatException) rootCause, headers, status, request);  
 }  
  
 ProblemType problemType = ProblemType.*MENSAGEM\_INCOMPREENSIVEL*;  
 String detail = "O corpo da requisição está inválido. Verifique erro de sintaxe";  
  
 Problem problem = createProblemBuilder(status, problemType, detail).build();  
  
 return handleExceptionInternal(ex, problem, new HttpHeaders(), status, request);  
}

Se o rootCause for uma instancia de InvalidFormatException vai retornar o método que criamos especialmente para tratar com detalhes o retorno da exception, veja:

private ResponseEntity<Object> handleInvalidFormatException(InvalidFormatException ex,  
 HttpHeaders headers, HttpStatus status, WebRequest request) {  
  
 String path = ex.getPath().stream().map(ref -> ref.getFieldName()).collect(Collectors.*joining*("."));  
  
 ProblemType problemType = ProblemType.*MENSAGEM\_INCOMPREENSIVEL*;  
 String detail = String.*format*("A propriedade '%s' recebeu o valor '%s', " +  
 "que é de um tipo inválido. Corrija e informe um valor compatível com o tipo %s.",  
 path, ex.getValue(), ex.getTargetType().getSimpleName());  
  
 Problem problem = createProblemBuilder(status, problemType, detail).build();  
  
 return handleExceptionInternal(ex, problem, headers, status, request);  
}

A linha com o código:

String path = ex.getPath()  
 .stream()  
 .map(ref -> ref.getFieldName())  
 .collect(Collectors.*joining*("."));

Utiliza a API Stream do Java para manipular listas com funções lambdas utilizando técnicas de programação funcional. Está capturando o nome das propriedades para exibir concatenado no texto da descrição detail, sendo capturado os parâmetros, veja:

String detail = String.*format*("A propriedade '%s' recebeu o valor '%s', " +  
 "que é de um tipo inválido. Corrija e informe um valor compatível com o tipo %s.",  
 path, ex.getValue(), ex.getTargetType().getSimpleName());

O path captura o nome da propriedade, o ex.getValue() captura o valor que foi atribuído a propriedade e o ex.getTargetType().getSimpleName() captura o tipo especifico da propriedade que pode ser Long, String etc.

## Habilitar erros na desserialização de propriedades null, ignored

Para evitar deixar passar uma propriedade que não existe ser passada no request body ou uma propriedade que esteja anotada com @JsonIgnore, utilizamos duas propriedades no arquivo application-properties. Para entender melhor caso uma pessoa passe no json uma propriedade que não existe na entidade, exemplo:

{

"nome": "Penha",

"esteNaoExisteNaEntidade": "icsto no ecxiste",

"estado": {

"id": 10

}

}

Caso não tratarmos isto o retorno será 200 OK como se nada tivesse acontecido, isso não implica no salvamento ou qualquer outra operação porque a propriedade será ignorada, mas isso é ruim porque o consumidor da API vai pensar que está tudo certo, pensando que a propriedade existe na entidade. O mesmo acontece com propriedades que existem, mas estão anotadas com @JsonIgnore, precisamos tratar também.

Para tratar isto, utilizamos estas propriedades no arquivo application-properties.

Para tratar propriedades não existentes:

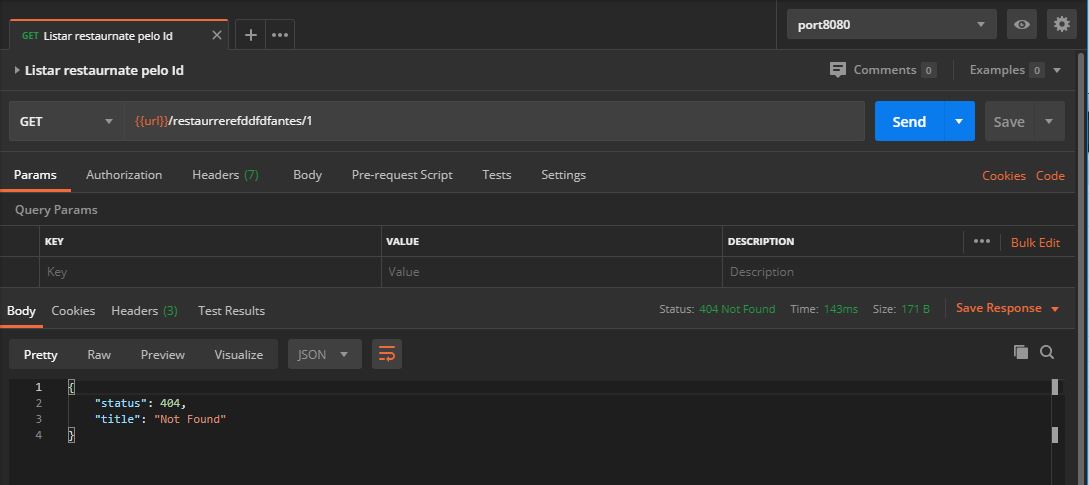
spring.jackson.deserialization.*fail-on-unknown-properties*=true

Para tratar propriedades ignoradas com @JsonIgnore

spring.jackson.deserialization.*fail-on-ignored-properties*=true

## Habilitando o stacktrace da exception NoHandlerFoundException

Para a exception NoHandlerFoundException conseguir ser lançada precisamos adicionar duas propriedades no arquivo application-properties, essa exception trata quando inserimos uma URI inválida para requisição, veja:



Repare que o recurso uri na url acima não existe na API, com isso tem que ser lançada a exception NoHandlerFoundException, mas por padrão ela não é lançada, temos que habilitar adicionando as propriedades:

spring.mvc.throw-exception-if-no-handler-found=true  
spring.resources.add-mappings=false

A linha:

spring.mvc.throw-exception-if-no-handler-found=true

Trata de habilitar o lançamento da exception NoHandlerFoundException, mas ela sozinha não é suficiente para habilitar o lançamento temos que adicionar também a linha abaixo:

A linha:

spring.resources.add-mappings=false

É responsável por desabilitar o mapeamento da pasta static, essa pasta contém arquivos de imagem e outras, que não estamos utilizando no desenvolvimento de uma API, desabilitamos e com isso a exception NoHandlerFoundException será lançada.

# VALIDAÇÕES COM BEAN VALIDATION

Quando recebemos entrada de dados é muito importante validar esses dados para garantir que estes dados estejam como desejamos, para validar os dados em uma API Java utilizamos o Bean Validation.

O Bean Validation é uma especificação que faz parte do JEE, e tem como objetivo facilitar a validação de propriedades em objetos Java, precisamos de uma implementação para usá-lo e para isso o Hibernate Validator é a implementação de referência da especificação.

**Nota importante:** Nas versões mais novas do Spring, pelo menos a que testei até agora, na versão 2.3.4 do Spring Boot é necessário adicionar a dependência spring-boot-starter-validation, para que as annotations @Valid funcionem corretamente com o pacote javax.validations.

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
</dependency>

## Adicionando constraint e validando com @Valid

Podemos validar dados, objetos antes deles serem transacionados nos services, podemos fazer isso antes no controller, para isso basta adicionar a annotation @Valid no início da propriedade que deseja validar, veja o exemplo:

@PostMapping  
@ResponseStatus(HttpStatus.*CREATED*)  
public Restaurante adicionar(@RequestBody @Valid Restaurante restaurante) {  
 try {  
 return cadastroRestaurante.salvar(restaurante);  
 } catch (CozinhaNaoEncontradaException e) {  
 throw new NegocioException(e.getMessage());  
 }  
}

Repare que adicionamos o @Valid na propriedade ResquestBody restaurante, com isso ela será validada antes de passar para a execução do código, retornando assim um erro mais adequado ao invés de retornar 500, irá retornar 400, pois se trata de um erro do usuario e não do servidor.

Podemos validar atributos em uma classe também, como por exemplo não permitir valores nulos em um atributo, para isso utilizamos a annotation @NotNull do pacote: javax.validation.constraints.NotNull

O pacote que trabalha com as classes do Bean Validation é o javax.validation.

## Estendendo o Problem Details para adicionar as propriedades com constraints violadas

Podemos estender a visualização de erros nos retornos de erros da nossa api, onde podemos visualizar com mais detalhes quais propriedades foram afetadas e jogar essa informação para o consumidor da nossa API, e não somente jogar na tela que algumas propriedades estão causando o erro, mas sim mostrar quais são essas propriedades. Para isso podemos utilizar os métodos da interface BindingResult, a classe MethodArgumentNotValidException possui como parâmetro a interface BindingResult que é responsável por possibilitar o uso de vários métodos que trazem os valores de determinadas propriedades que podemos capturar, ou seja podemos capturar e mostrar em quais propriedades estão sendo responsáveis por ocasionar as exceptions, podemos carregar em uma lista no caso de mais de uma propriedade estar causando os erros.

Repare neste trecho de código onde utilizamos a interface BindingResult para capturar os atributos causadores dos erros.

@Override  
protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,  
 HttpHeaders headers, HttpStatus status,  
 WebRequest request) {  
 ProblemType problemType = ProblemType.*DADOS\_INVALIDOS*;  
 String detail = "Um ou mais campos estão inválidos. Faça o preenchimento correto e tente novamente.";  
  
 BindingResult bindingResult = ex.getBindingResult();  
  
 List<Problem.Field> problemFields = bindingResult.getFieldErrors().stream()  
 .map(fieldError -> Problem.Field.*builder*()  
 .name(fieldError.getField())  
 .userMessage(fieldError.getDefaultMessage())  
 .build()).collect(Collectors.*toList*());  
  
 Problem problem = createProblemBuilder(status, problemType, detail)  
 .userMessage(detail)  
 .fields(problemFields)  
 .build();  
  
 return handleExceptionInternal(ex, problem, headers, status, request);  
}

Repare que instanciamos um BindingResult e atribuímos ele à exception e utilizamos o método getBindingResult para capturar todas as informações de ocorrência de erros. Com o auxilio de uma classe que possui atributos de Lista veja abaixo:

@JsonInclude(JsonInclude.Include.*NON\_NULL*)  
@Getter  
@Builder  
public class Problem {  
  
 private Integer status;  
 private String type;  
 private String title;  
 private String detail;  
 private String userMessage;  
 private LocalDateTime timestamp;  
 private List<Field> fields;  
  
 @Getter  
 @Builder  
 public static class Field {  
 private String name;  
 private String userMessage;  
 }  
}

Podemos criar uma variável que guarda uma lista dos resultados das ocorrências, utilizamos um stream para mapear esses resultados e no final coletar para a lista. E por fim adicionamos essa variável nos atributos de Problem para que ela seja exibida na exception.

## Conhecendo mais constraints de validação

Existem várias constraints de validação, que podemos utilizar para fazer validação dos nossos atributos em uma classe, elas vào além do @NotNull, podemos validar números positivos ou zero com @PositiveOrZero, podemos validar que não queremos espaço, nulo ou branco com @NotBlank, existem diversas, dê uma olhada na documentação aqui neste link: <https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#section-builtin-constraints>

## Validando em Cascata

Para validarmos propriedades em cascata utilizamos a annotation @Valid na propriedade que contém relacionamento com outra classe, e com isso o bean validation irá executar e vai na outra classe em busca das validations, isso se torna uma operação de validação em cascata.

@Valid  
@NotNull  
@ManyToOne  
@JoinColumn(nullable = false)  
private Cozinha cozinha;

Repare que utilizamos o @Valid na propriedade cozinha que é um relacionamento muitos para um com a entidade Cozinha, com isso o bean validation irá até a classe cozinha e fará a verificação de todas as validations da classe Cozinha que foi referenciada aqui.

Mas ao utilizar esta abordagem ficamos com alguns problemas em algumas rotas da aplicação, quebra muitas rotinas de adição, atualização, veja no próximo tópico como resolver:

## Criando grupos de validação

Quando utilizamos annotations de validação do bean validation podemos também criar grupos, para especificar uma ordem de checagem de validação, isso é útil para não haver conflitos de validação ocorrendo assim interferência quando utilizamos uma rota para adicionar, atualizar e essa rota passa por validações bean validation defaults, no tópico anterior fizemos exatamente isso, mas aquela abordagem vai atrapalhar as outras rotas. Para criar grupos apenas adicionamos alguns atributos nas annotations de bean validation, o atributo groups, veja como fica:

@NotNull(groups = Groups.CadastroRestaurante.class)  
@EqualsAndHashCode.Include  
@Id  
@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
private Long id;

Veja que é necessário criar uma classe ou Interface apenas para referenciar o grupo, neste caso foi criada uma simples interface com outras interface dentro, veja:

public interface Groups {  
  
 public interface CadastroRestaurante {}  
  
}

Repare que temos uma interface chamada CadastroRestaurante é com ela que estamos referenciando o grupo de cadastro de restaurante, se quisermos podemos criar mais referencias dentro desta interface para criarmos outros grupos.

Tendo isso em mente podemos organizar o fluxo de validação conforme o designio da rota, ou seja, dependendo do comportamento da rota da api podemos trilhar um caminho de validação para que as validações não extrapolem o que seja determinado e não atrapalhe quando não for necessário em outra rota, com isso restringimos a validação a situações especificas.

Mas para que isso funcione de forma eficaz precisamos substituir a annotations @Valid nos controllers para a annotation @Validated do Spring, com essa anotação podemos especificar grupos, o que a @Valid não nos permite. Veja o nosso exemplo:

@PostMapping  
@ResponseStatus(HttpStatus.*CREATED*)  
public Restaurante adicionar(@RequestBody @Validated(Groups.CadastroRestaurante.class) Restaurante restaurante) {  
 try {  
 return cadastroRestaurante.salvar(restaurante);  
 } catch (CozinhaNaoEncontradaException e) {  
 throw new NegocioException(e.getMessage());  
 }  
}

Portanto agora toda vez que utilizarmos essa rota a validação vai ocorrer nos grupos marcados, podemos referenciar mais de um grupo, basta utilizar { } e separar por virgula.

Por padrão se não especificarmos grupos nos bean validations eles são do tipo Default.class.

## Convertendo grupos de constraints para @ConvertGroup

Mais uma vez podemos simplificar ainda mais a criação de grupos, a abordagem vista no tópico acima com o tempo pode ficar verbosa e é muito trabalhoso adicionar atribuições em todas as validations dos atributos da classe, podemos simplificar isso com a annotation @ConvertGroup, ela consiste em converter de um grupo para outro em determinada situação, no nosso caso o de relacionamento com outra classe, veja o exemplo:

@Valid  
@ConvertGroup(from = Default.class, to = Groups.CozinhaId.class)  
@NotNull  
@ManyToOne  
@JoinColumn(nullable = false)  
private Cozinha cozinha;

Repare que estamos convertendo o grupo default para o grupo especificado acima, o bean validation vai entrar na classe Cozinha e procurar pelo validation do grupo especificado e irá ignorar os defaults ou grupos diferentes do especificado. Isso permite remover os grupos dos atributos adjacentes que tínhamos anotado antes, sendo necessário apenas adicionar especificação na annotation de bean validation necessária, aqui neste caso só mantemos no atributo id da classe cozinha, veja:

@NotNull(groups = Groups.CozinhaId.class)  
@EqualsAndHashCode.Include  
@Id  
@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
private Long id;

E também podemos remover a annotation @Validated e voltar a utilizar a @Valid quando utilizamos essa abordagem.

## Formas de customizar mensagens de constraints

Vamos ver algumas formas de customizar as mensagens das constraints do bean validation, a primeira consiste em fazer isso:

@NotBlank(message = "Mensagem personalizada")  
@Column(nullable = false)  
private String nome;

Precisamos somente adicionar o parâmetro message na annotation.

Outra forma de fazer isso sem precisar adicionar essa message em cada propriedade é criar um arquivo chamado messages.properties e neste arquivo nós inserimos as mensagens customizadas. Veja um exemplo dessas mensagens:

#Quanto maior a procedencia de uma especificação ela toma a preferencia, é parecido com a especificidade do css.  
NotBlank.restaurante.nome=Nome do restaurante é obrigatório  
  
#Podemos utilizar os placeholders em {} ele vai pegar o nome da propriedade da entidade.  
NotBlank={0} é obrigatório  
NotBlank.nome=Informe um nome, porque é obrigatório  
  
#Podemos especificar um nome exclusivo para propriedades das entidades.  
nome=Nome  
restaurante.nome=Nome do Restaurante

E no método onde estamos tratando as exceptions referentes à esse erro precisamos incluir alguns itens. Precisamos injetar uma dependência:

@Autowired  
private MessageSource messageSource;

E no método onde tratamos essa exception fazemos o método capturar as mensagens desse arquivo que criamos, veja:

@Override  
protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,  
 HttpHeaders headers, HttpStatus status,  
 WebRequest request) {  
 ProblemType problemType = ProblemType.*DADOS\_INVALIDOS*;  
 String detail = "Um ou mais campos estão inválidos. Faça o preenchimento correto e tente novamente.";  
  
 BindingResult bindingResult = ex.getBindingResult();  
  
 List<Problem.Field> problemFields = bindingResult.getFieldErrors().stream()  
 .map(fieldError -> {  
 String message = messageSource.getMessage(fieldError, LocaleContextHolder.*getLocale*());  
  
 return Problem.Field.*builder*()  
 .name(fieldError.getField())  
 .userMessage(message)  
 .build();  
 }).collect(Collectors.*toList*());  
  
 Problem problem = createProblemBuilder(status, problemType, detail)  
 .userMessage(detail)  
 .fields(problemFields)  
 .build();  
  
 return handleExceptionInternal(ex, problem, headers, status, request);  
}

Fazemos isso com essa linha:

String message = messageSource.getMessage(fieldError, LocaleContextHolder.*getLocale*());

O método getMessage da interface MessageSource recebe um parâmetro do tipo MessageSourceResolvable que é o nosso fieldError, e também passamos um Locale que se refere a localização geográfica do local, uma curiosidade é que em um projeto onde precisaríamos utilizar diferentes línguas para o retorno de mensagens de erro, precisaríamos ter vários arquivos messages.properties, um para cada língua, por exemplo, um para inglês, messages\_en.properties, um para espanhol messages\_es.properties e assim por diante.

Existe outra forma de customizar as mensagens, esse forma descrita acima é a forma que o Spring trata as dependências, existe outra que é pelo bind do Hibernate validation, se criarmos um arquivo chamado ValidationMessages.properties todas as regras que inserimos nesse arquivo serão tratadas pelo bind do Hibernate Validation, mas ele possui uma precedência menos que o bind validations do Spring, o recomendado é utilizar o messages.properties mesmo.

Mas podemos unificar esses dois arquivos e fazer o bind do hibernate funcionar no do Spring, para isso é necessário criar uma classe de Configurations do Spring e criar um Bean, veja abaixo:

@Configuration  
public class ValidationConfig {  
  
 @Bean  
 public LocalValidatorFactoryBean validator(MessageSource messageSource) {  
 LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();  
 bean.setValidationMessageSource(messageSource);  
  
 return bean;  
 }  
}

LocalValidatorFactoryBean é uma classe que faz a integração e configuração do bean validation com o Spring. O MessageSource é responsável por ler os arquivos messages.properties e fazer resolução de mensagens. O que fizemos acima foi customizar uma instancia de LocalValidatorFactoryBean indicando que o ValidationMessageSource é um MessageSource.

## Criando constraints de validação customizadas usando composição

Podemos criar annotations para efetuar validações, para isso primeiro criamos uma annotation, veja o exemplo abaixo:

package com.algaworks.algafoodapi.core.validation;  
  
import javax.validation.Constraint;  
import javax.validation.OverridesAttribute;  
import javax.validation.Payload;  
import javax.validation.constraints.PositiveOrZero;  
  
import static java.lang.annotation.ElementType.*ANNOTATION\_TYPE*;  
import static java.lang.annotation.ElementType.*CONSTRUCTOR*;  
import static java.lang.annotation.ElementType.*FIELD*;  
import static java.lang.annotation.ElementType.*METHOD*;  
import static java.lang.annotation.ElementType.*PARAMETER*;  
import static java.lang.annotation.ElementType.*TYPE\_USE*;  
import static java.lang.annotation.RetentionPolicy.*RUNTIME*;  
  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
  
@Target({ *METHOD*, *FIELD*, *ANNOTATION\_TYPE*, *CONSTRUCTOR*, *PARAMETER*, *TYPE\_USE*})  
@Retention(*RUNTIME*)  
@Constraint(validatedBy = { })  
@PositiveOrZero  
public @interface TaxaFrete {  
  
 @OverridesAttribute(constraint = PositiveOrZero.class, name = "message")  
 String message() default "{TaxaFrete.invalida}";  
  
 Class<?>[] groups() default { };  
  
 Class<? extends Payload>[] payload() default { };  
}

Repare que uma annotation contém outras annotations em seu escopo, ela necessita de outras annotations para poder funcionar. Explico cada annotation abaixo:

@Target – É uma annotation comum do Java, responsável por dar permissão de utilização da annotation, ou seja pode ser utilizada em um METHOD (método), FIELD (propriedade), CONSTRUCTOR(construtor) e etc.

@Retention – E’responsável por setar como a annotation vai ser lida e interpretada, no caso acima a nossa annotation será executada em tempo de execução.

@Constraint – É uma annotation do bean validation responsável por restringir a validação a determinadas condições, com a utilização de outras annotations, classes, atributos, files etc. É utilizada para limitar o range de forma especifica.

Nesta linha:

@PositiveOrZero  
public @interface TaxaFrete {

Utilizamos a annotation @PositiveOrZero para que a nossa nova annotation receba os atributos, propriedades de validação dessa annotation, uma forma de herança.

E para utilizarmos a nova annotation personalizada inserimos ela na classe especifica:

@NotNull  
//@PositiveOrZero  
@TaxaFrete  
@Column(nullable = false)  
private BigDecimal taxaFrete;

E inserimos a mensagem que desejamos no arquivo messages.properties:

#Propriedades de Annotations customizadas  
TaxaFrete.invalida={0} está inválido

Do mesmo modo como fazemos com todas as classes do projeto.

### Conflito entre Bean Validation do Javax entre o Spring

Ao criarmos uma annotation personalizada e utilizarmos ela em uma propriedade, pode ocorrer conflitos entre o Spring e o bean validation do Java, por exemplo:

#Annotations globais  
PositiveOrZero={0} deve ser um valor maior ou igual a zero  
  
  
#Propriedades de Annotations customizadas  
TaxaFrete.invalida={0} está inválido

Se mantermos as duas messages acima a precedência sempre será a do bean do Spring, o Spring sobrepõe o validation do Java, para contornar isso sem precisar apagar as messages relacionadas ao spring validation criamos uma constraint no atributo da mesage, veja:

public @interface TaxaFrete {  
  
 @OverridesAttribute(constraint = PositiveOrZero.class, name = "message")  
 String message() default "{TaxaFrete.invalida}";  
  
 Class<?>[] groups() default { };  
  
 Class<? extends Payload>[] payload() default { };  
}

Utilizamos a annotation @OverridesAttribute para setar uma constraint que faz o parse da annotation @PositiveOrZero utilizar a message que foi setada na nova annotation.

## Constraints de validação customizadas com implementação de ConstraintValidator

Ao invés de criarmos uma annotation composta ou seja que utiliza métodos de outras annotations, podemos criar uma totalmente customizada na unha do zero, com nossos métodos e lógica próprios.

Para isso partimos da criação de uma annotation normalmente e o diferencial agora é que criamos também uma classe para implementar a interface ConstraintValidator, que possui dois métodos onde inserimos os nossos códigos de acordo com a necessidade.

Neste exemplo vamos criar uma Annotation que é responsável por validar se o número recebido é múltiplo do número que foi setado na annotation como comparação, veja abaixo a Annotation criada:

package com.algaworks.algafoodapi.core.validation;  
  
import javax.validation.Constraint;  
import javax.validation.Payload;  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
  
import static java.lang.annotation.ElementType.\*;  
import static java.lang.annotation.ElementType.*TYPE\_USE*;  
import static java.lang.annotation.RetentionPolicy.*RUNTIME*;  
  
@Target({ *METHOD*, *FIELD*, *ANNOTATION\_TYPE*, *CONSTRUCTOR*, *PARAMETER*, *TYPE\_USE*})  
@Retention(*RUNTIME*)  
@Constraint(validatedBy = { MultiploValidator.class})  
public @interface Multiplo {  
  
 String message() default "Multiplo inválido";  
  
 Class<?>[] groups() default { };  
  
 Class<? extends Payload>[] payload() default { };  
  
 int numero();  
}

Repare que desta vez não sobrescrevemos com uma Annotation que possui métodos de lógica próprios como anteriormente fizemos com @PositiveOrZero, desta vez as regras de lógica dos métodos serão criadas do zero na unha por nós. Para isso criamos agora uma classe para inserirmos nosso código, veja abaixo:

package com.algaworks.algafoodapi.core.validation;  
  
import javax.validation.ConstraintValidator;  
import javax.validation.ConstraintValidatorContext;  
import java.math.BigDecimal;  
  
public class MultiploValidator implements ConstraintValidator<Multiplo, Number> {  
  
 private int multipleNumber;  
  
 @Override  
 public void initialize(Multiplo constraintAnnotation) {  
 this.multipleNumber = constraintAnnotation.numero();  
 }  
  
 @Override  
 public boolean isValid(Number value, ConstraintValidatorContext constraintValidatorContext) {  
 boolean valido = true;  
  
 if (value != null){  
 var valorDecimal = BigDecimal.*valueOf*(value.doubleValue());  
 var multiploDecimal = BigDecimal.*valueOf*(this.multipleNumber);  
 var resto = valorDecimal.remainder(multiploDecimal);  
  
 valido = BigDecimal.*ZERO*.compareTo(resto) == 0;  
 }  
  
 return valido;  
 }  
}

Essa classe implementa os métods da interface ConstraintValidator que possui 2 métodos:

Initialize – Responsável por inicializar a classe.

isValid – Responsável por inferir as condições impostas pelo programador.

ConstraintValidator recebe a classe da nossa Annotation, e o parametro de Tipo que neste caso utilizamos Number para ser o mais genérico possível permitindo assim um trabalho com os tipos de números mais abrangente por Number ser o maior na hierarquia dos tipos de números.

E na classe onde desejamos ativar esse comportamento anotamos ela com nossa nova Annotation:

@NotNull  
//@PositiveOrZero  
@Multiplo(numero = 10)  
@Column(nullable = false)  
private BigDecimal taxaFrete;

E podemos customizar a mensagem com sempre fazemos no arquivo message.properties:

#Propriedades de Annotations customizadas  
Multiplo={0} deve ser um valor múltiplo de {1}

## Constraints de validação customizadas em nível de classe

Podemos criar uma Annotation em nível de classe, que pode validar todas as propriedades de uma classe completa ao invés de apenas uma propriedade como estavámos fazendo nos métodos anteriores. Para isso criamos uma Annotation do modo com fizemos anteriormente mas limitamos o @Target para apenas TYPE, veja abaixo:

package com.algaworks.algafoodapi.core.validation;  
  
import javax.validation.Constraint;  
import javax.validation.Payload;  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
  
import static java.lang.annotation.RetentionPolicy.*RUNTIME*;  
  
@Target({ ElementType.*TYPE* })  
@Retention(*RUNTIME*)  
@Constraint(validatedBy = { ValorZeroIncluiDescricaoValidator.class })  
public @interface ValorZeroIncluiDescricao {  
  
 String message() default "Descrição obrigatória inválida";  
  
 Class<?>[] groups() default { };  
  
 Class<? extends Payload>[] payload() default { };  
  
 String valorField();  
  
 String descricaoField();  
  
 String descricaoObrigatoria();  
}

Com isso limitamos o uso dessa annotation apenas em nível de classe, se tentarmos utilizar essa Annotation em apenas um atributo da classe ocorrerá um erro. Chamamos essa Annotation no cabeçalho da classe:

@Data  
@ValorZeroIncluiDescricao(valorField = "taxaFrete", descricaoField = "nome", descricaoObrigatoria = "Frete Grátis")  
@EqualsAndHashCode(onlyExplicitlyIncluded = true)  
@Entity  
public class Restaurante {  
  
 @EqualsAndHashCode.Include  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;

Apenas esse detalhe mudou na criação da Annotation de nível de classe, e veja abaixo também a classe de implementação dos métodos:

package com.algaworks.algafoodapi.core.validation;  
  
import org.springframework.beans.BeanUtils;  
  
import javax.validation.ConstraintValidator;  
import javax.validation.ConstraintValidatorContext;  
import javax.validation.ValidationException;  
import java.math.BigDecimal;  
  
public class ValorZeroIncluiDescricaoValidator implements ConstraintValidator<ValorZeroIncluiDescricao, Object> {  
  
 private String valorField;  
 private String descricaoField;  
 private String descricaoObrigatoria;  
  
 @Override  
 public void initialize(ValorZeroIncluiDescricao constraint) {  
 this.valorField = constraint.valorField();  
 this.descricaoField = constraint.descricaoField();  
 this.descricaoObrigatoria = constraint.descricaoObrigatoria();  
 }  
  
 @Override  
 public boolean isValid(Object objetoValidacao, ConstraintValidatorContext constraintValidatorContext) {  
 boolean valido = true;  
  
 try {BigDecimal valor = (BigDecimal) BeanUtils.*getPropertyDescriptor*(objetoValidacao.getClass(), valorField)  
.getReadMethod().invoke(objetoValidacao);  
  
 String descricao = (String) BeanUtils.*getPropertyDescriptor*(objetoValidacao.getClass(), descricaoField).getReadMethod().invoke(objetoValidacao);  
  
 if (valor != null && BigDecimal.*ZERO*.compareTo(valor) == 0 && descricao != null){  
 valido = descricao.toLowerCase().contains(this.descricaoObrigatoria.toLowerCase());  
 }  
  
 } catch (Exception e) {  
 throw new ValidationException(e);  
 }  
  
 return valido;  
 }  
}

Também tratamos a mensagem no arquivo message.properties:

#Propriedades de Annotations customizadas  
ValorZeroIncluiDescricao={1} deve conter {2}

Quando criamos ou temos uma Anotation que trabalha via Objeto, ou seja que tem o escopo de trabalho no objeto completo e não apenas nas propriedades precisamos mudar a forma como trabalhamos as exceptions nos métodos de exceptionHandler, se utilizarmos o método

bindingResult.getFieldErrors()

getFieldErrors() é pegar os erros apenas dos atributos da classe e não da classe propriamente dita em si. Vai ocorrer que as exceptions dos métodos de annotations globais de classe não serão lançadas corretamente na requisição. Para podermos trabalhar com exceptions em annotations de classe podemos utilizar:

bindingResult.getAllErrors()

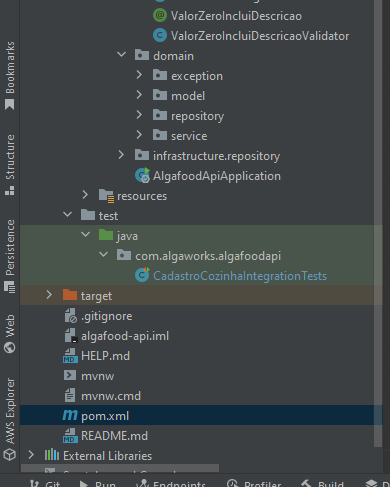
O .getAllErrors() resolve o problema para que todos os tipos de erros sejam capturados, incluindo propriedades e objetos de classe.

# TESTES DE INTEGRAÇÃO

Testes de integração são responsáveis por testar a nossa aplicação como um todo, utilizando todos os serviços responsáveis pela funcionalidade de toda estrutura do projeto, ao contrário do teste unitário que testa apenas uma funcionalidade independente do projeto.

O Spring boot já adiciona automaticamente nos projetos o JUnit, com a dependência spring-boot-satrter-test, versões do Spring Boot acima da 2.4 utilizam o JUnit 5 ao invés do antigo JUnit 4, que traz pequenas alterações na sintaxe de escrita dos testes.

Ao criar um projeto o Spring Boot se encarrega de já criar um diretório de testes, veja na imagem abaixo:



## Nomenclatura dos métodos de testes

Não existe uma regra fixa para nomeação de métodos de testes, o que é consenso entre os profissionais é que exista um padrão a ser repetido no projeto, por toda a equipe de desenvolvimento, para evitar que cada desenvolvedor utilize um modo pessoal para nomear os métodos. Nomear métodos de testes é mais flexível do que nomear métodos fora do escopo de testes, mas deve atentar-se para a criação de nomes que sejam fáceis de ler e compreender do que se trata o teste, precisam ser nomes descritivos e legíveis.

Veja um exemplo de algumas formas de nomear métodos de testes:

@Test  
public void deve\_falhar\_quando\_excluir\_cozinha\_em\_uso() {  
   
}

@Test  
public void deveFalharQquandoExcluirCozinhaEmUso() {  
   
}

@Test  
public void whenExcluirCozinhaEmUsoSholdFalhar() {  
   
}

Abaixo veja uma classe de testes completa com alguns métodos de testes realizados.

package com.algaworks.algafoodapi;  
  
  
import com.algaworks.algafoodapi.domain.exception.EntidadeEmUsoException;  
import com.algaworks.algafoodapi.domain.exception.EntidadeNaoEncontradaException;  
import com.algaworks.algafoodapi.domain.model.Cozinha;  
import com.algaworks.algafoodapi.domain.service.CadastroCozinhaService;  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.Assertions;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
  
import javax.validation.ConstraintViolationException;  
  
@SpringBootTest  
class CadastroCozinhaIntegrationTests {  
 @Autowired  
 private CadastroCozinhaService cadastroCozinha;  
 private Cozinha novaCozinha = new Cozinha();  
  
 @Test  
 public void deve\_cadastrar\_cozinha\_com\_sucesso() {  
 // Cenário  
 novaCozinha.setNome("Chinesa");  
  
 // Ação  
 novaCozinha = cadastroCozinha.salvar(novaCozinha);  
  
 // Validação  
 Assertions.*assertNotNull*(novaCozinha);  
 Assertions.*assertNotNull*(novaCozinha.getId());  
 }  
  
 @Test  
 public void deve\_falhar\_quando\_cadastrar\_cozinha\_sem\_nome() {  
 novaCozinha.setNome(null);  
  
 ConstraintViolationException erroEsperado =  
 Assertions.*assertThrows*(ConstraintViolationException.class, () -> {  
 cadastroCozinha.salvar(novaCozinha);  
 });  
  
 Assertions.*assertNotNull*(erroEsperado);  
 }  
  
 @Test  
 public void deve\_falhar\_quando\_excluir\_cozinha\_em\_uso() {  
 EntidadeEmUsoException erroEsperado =  
 Assertions.*assertThrows*(EntidadeEmUsoException.class, () -> {  
 cadastroCozinha.excluir(1L);  
 });  
  
 Assertions.*assertNotNull*(erroEsperado);  
 }  
  
 @Test  
 public void deve\_falhar\_quando\_excluir\_cozinha\_Inexistente() {  
 EntidadeNaoEncontradaException erroEsperado =  
 Assertions.*assertThrows*(EntidadeNaoEncontradaException.class, () -> {  
 cadastroCozinha.excluir(100L);  
 });  
  
 Assertions.*assertNotNull*(erroEsperado);  
 }  
}

A utilização do JUnit é simples, só se atentar para a sintaxe que pode mudar de uma versão para outra, o ideal é sempre consultar a documentação.

## Executando testes com o Maven

Podemos realizar os testes pelo terminal do maven sem precisar utilizar a API do JUnit, para isso basta entrarmos no diretório do projeto e executarmos o seguinte comando:

seu-projeto$ ./mvnw test

Com isso o Maven vai rodar todos os testes. É preciso se atentar no entanto que ao fazermos um package do projeto, criar um build no Maven, os testes serão executados no final do empacotamento, e isso pode demorar muito, levar horas em projetos muito grandes.

Também acontece de o build não ser completo caso haja um ou mais testes com falha. Há uma forma de contornar essa situação para isso fazemos o seguinte:

Adicionamos esta dependência no nosso pom.xml:

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-failsafe-plugin</artifactId>  
 <version>3.0.0-M6</version>  
 </plugin>  
 </plugins>  
</build>

De uma olhada no site da Maven para ver a versão mais atual. Após adicionar essa dependência você precisa renomear as suas classes de testes de integração para a nomenclatura final ficar com as duas letras: IT, veja um exemplo:

@SpringBootTest  
class CadastroCozinhaIT {

Isso só afeta os testes de integração, os testes unitários continuam sendo executados normalmente, e todos os testes na IDE também são executados normalmente, só que agora quando executado o comando: seu-projeto$ ./mvnw test os testes não serão executados, e não também não serão executados quando for realizado um build do projeto.

Para podermos executar os testes de integração após ter feito essa modificação basta utilizar esse comando:

seu-projeto$ ./mvnw verify

S