Por que aprender e usar Spring

Com o Spring é possível ter maior domínio do projeto que está sendo desenvolvido, tendo como maior característica o suporte à infraestrutura direto na aplicação, permitindo assim que os times de desenvolvimento possam se concentrar na parte lógica da aplicação, sem precisar se preocupar desnecessariamente com questões de configuração de ambiente

Pool de conexões e Flyway

**Criando migração a partir de DDL gerado por schema generation**

Essa criação utiliza uma geração automática utilizando as classes mapeadas no spring para geração das tabelas. A baixo esta o código gerador que dever ser incluso no aplication.properties ou no properties que estiver utilizando no momento da geração do seu banco de dados

spring.jpa.properties.javax.persistence.schemageneration.scripts.action=create

spring.jpa.properties.javax.persistence.schema-generation.scripts.create

target=src/main/resources/ddl.sql

O primeiro comando gera os scripts e o segundo armazena no documento criado (antes deve ser criado um documento .sql para receber os scripts)

**Adicionando dados de testes com callback do Flyway**

Dentro dos arquivos de migração do flyway não é recomendável cadastrar dados mas o flyway permite fazer uma chamada de call-back em um arquivo separado chamado de **afterMigrate.sql** nesse arquivo é possível adicionar dados de teste. E recomendado adicionar esse arquivo em uma pasta de teste ou dev para não ser enviado para produção. A baixo esta o comando que lê esse arquivo em uma pasta separada do migration

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

Nesse comando e possível notar dois caminhos informado para busca do flyway

É dentro destas duas pastas que ele faz a busca dos arquivo

Obs: quando for fazer um insert dentro de um migrate e aconselhado seguir o passo a baixo desabilitar as foreign Keys, dropar habilitar as foreign Keys e alterar auto\_increment = 1; e um ignore no inser se quiser um banco limpo apenas com base de testes sempre que iniciar seu banco de dados

set foreign\_key\_checks = 0;

delete from cidade;

set foreign\_key\_checks = 1;

alter table cidade auto\_increment = 1;

insert ignore into cozinha (id, nome) values (1 ,'tailandesa');

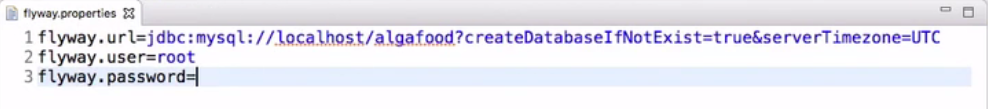
**Reparando migrações com erros**

Vamos aprender duas formas de resolver migrações com erro no flyway, uma das formas que é a mais simples é fazendo a correção manualmente no banco de dados, basta dar um select \* from flyway\_schema\_history no banco de dados para ver a migração que esta com success = 0 e deletar (esse método funciona melhor quando trata-se da ultima atualização)

A outra forma e deixar o próprio flyway executar o reparo



Usando o comando acima no **prompt de comando** e possível que o flyway faça o reparo mas ele irá pedir um login e senha então deve ser criado um properties para não ter que digitar sempre o esses dados



Feito o arquivo properties basta executar o comando a baixo no **Prompt de Comando** que a correção será executada



Esse comando busca o banco, login e senha para executar o reparo

Tratamento e modelagem de erros da API

Agora vamos aprender a como tratar erros de forma mais eficiente separando o tratamento de erros das camadas de serviço e controller. Nessa primeira parte vamos mostrar como utilizar o @ResponseStatus direto na classe responsável por enviar a exceção

Metodo da camada Serviçe

**public** **void** remover(Long id) {

**try** {

cozinhaRepository.deleteById(id);

} **catch** (DataIntegrityViolationException e) {

**throw** **new** EntidadeEmUsoException(

String.*format*("Cozinha de codigo %d não pode ser removido", id));

} **catch**(EmptyResultDataAccessException e) {

**throw** **new** EntidadeNaoEncontradaException(

String.*format*("Não existe um cadastro de cozinha com o codigo %d ", id));

}

Nesse caso ainda estamos utilizando o try catch na camada de serviço como podemos notar pois esse e remove e da camada service mas agora podemos enviar o status direto para uma das duas classe

EmptyResultDataAccessException

EntidadeEmUsoException

@ResponseStatus(code = HttpStatus.***CONFLICT***, reason = "Entidade em uso")

**public** **class** EntidadeEmUsoException **extends** RuntimeException {

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

**public** EntidadeEmUsoException(String message) {

**super**(message);

}

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

**public** **class** EntidadeNaoEncontradaException **extends** RuntimeException {

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

**public** EntidadeNaoEncontradaException(String message) {

**super**(message);

// **TODO** Auto-generated constructor stub

}

Como podemos ver as classes responsáveis pelas exceções agora estão com a anotação **@ResponseStatus** com o respectivo status de envio informado pelo **code** que e responsável por informar qual será o código http e temos também o **reason** que é uma descrição do motivo do status gerado

**Lançando exceções do tipo ResponseStatusException**

Trata-se de um construtor que permite enviar status http e uma notificação do tipo do status personalizado

@DeleteMapping("/{cozinhaId}")

@ResponseStatus(value = HttpStatus.***NO\_CONTENT***)

**public** **void** delete(@PathVariable Long cozinhaId) {

**try** {

cadastroCozinha.remover(cozinhaId);

} **catch** (EntidadeNaoEncontradaException e) {

**throw** **new** ResponseStatusException(HttpStatus.***NOT\_FOUND***,

e.getMessage());

}

}

Esse construtor e responsável por enviar o status deixar uma msg no caso estamos usando a msg que já vem no escopo do erro mas se quiser personalizar basta alterar aonde esta o campo e.getMessage

**Estendendo ResponseStatusException**

Outra forma de utilizarmos esse construtor e estender ele direto na exception neste caso podemos criar um código mais limpo

**public** **class** EntidadeEmUsoException **extends** ResponseStatusException {

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

**public** EntidadeEmUsoException(HttpStatus status, String message) {

**super**(status, message);

// **TODO** Auto-generated constructor stub

}

**public** EntidadeEmUsoException(String message) {

**this**(HttpStatus.***NOT\_FOUND*** , message);

}

.............................................................................

Camada Controller

@DeleteMapping("/{cozinhaId}")

@ResponseStatus(value = HttpStatus.***NO\_CONTENT***)

**public** **void** delete(@PathVariable Long cozinhaId) {

cadastroCozinha.remover(cozinhaId);

}

Veja utilizando como extensão foi possível remover o try catch da camada de controle

**Simplificando o código com o uso de @ResponseStatus em exceptions**

Podemos melhorar a chamada de exceções em nossas camadas de controller e service para isso vamos utilizar um método que existe no JpaRepository que e o orElseThrow esse método permite lançar uma exceção se não receber o objeto solicitado.

**public** Cozinha findById(Long id) {

**return** cozinhaRepository.findById(id).orElseThrow(() -> **new** EntidadeNaoEncontradaException(

String.*format*("Não existe um cadastro de cozinha com o codigo %d ", id)));}

Como podemos notar a busca por id na camada servisse não esta utilizando try catch e sim uma expressão lambida para realizar o retorno da exceção. Desse forma é possível realizar o retorno do status direto da exception. Observe como algumas buscas podem ficar mais limpas no controller

@GetMapping("/{cozinhaId}")

**public** Cozinha findById(@PathVariable Long cozinhaId) {

**return** cadastroCozinha.findById(cozinhaId);

}

E esse e a exception chamada, que utiliza a anotação @ResponseStatus com seu retorno e mensagem de erro

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

**public** **class** EntidadeNaoEncontradaException **extends** RuntimeException {

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

**public** EntidadeNaoEncontradaException(String message) {

**super**(message);

// **TODO** Auto-generated constructor stub

}

}

**Criando a exception NegocioException**

Nessa aula foi ensinado a criar uma exception para o lançamento de uma **bad\_request** visando evitar logica em classe de domínio serviçe a finalidade dessa exceção era lançar a exceção quando o estado não existe em e foi cadastrado em cidade. A exceção é **NegocioException** ela foi aplicada em cidadeController

**Afinando a granularidade e definindo a hierarquia das exceptions de negócios**

Foi ensinado como deixar mais explicito de onde veio o erro, nos tratamentos anteriores estávamos tratando o erro de forma genérica sem especificar de qual classe ele partia. Agora foi criado o EstadoNaoEncontradoException estendendo EntidadeNaoEncontradoException que por sua vez passou a estender NegocioException para mantermos uma hierarquia de exceções com essa mudança foi possível desenvolver um método em EstadoNaoEncontradoException que já retornava a msg de erro personalizada para estado bastando apenas instancia-lo no lugar de EntidadeNaoEncontradoException em EstadoService para termos o mesmo efeito com um código mais enxuto na nossa camada de serviço. Essa forma de lançar a exceção será atribuída no próximo desafio em todos os services

**Tratando exceções em nível de controlador com @ExceptionHandler**

Ao usar o **@ExceptionHandler** nos podemos capturar as exceções enviadas a nível de controlador basta informar qual exceção ele deve capturar. Utilizando ele com o ResponseEntity podemos também personalizar as mensagens enviadas segue exemplos de código

@ExceptionHandler(EntidadeNaoEncontradaException.**class**)

**public** ResponseEntity<?> tratarEntidadeNaoEncontradaException(EntidadeNaoEncontradaException e){

Problema problema = Problema.*builder*()

.data(LocalDateTime.*now*())

.message(e.getMessage()).build();

**return** ResponseEntity.*status*(HttpStatus.***NOT\_FOUND***).body(problema);

}

Nesse método podemos notar que o **@ExceptionHandler** esta capturando **EntidadeNaoEncontradaException** e utilizando a classe problema para personalizar o envio de erro

@Getter

@Builder

**public** **class** Problema {

**private** LocalDateTime data;

**private** String message;

}

A classe problema foi desenvolvida apenas com essa função. Podemos notar que ela vem assinada com anotações do lombook

Observação: o método apresentado acima **tratarEntidadeNaoEncontradaException** foi criado na camada de controller