

11.1. Analisando e definindo melhor o escopo das transações

sábado, 11 de março de 2023 14:02

Métodos da classe SimpleJpaRepository utilizam a anotação @Transactional para utilizar transações nos métodos que manipulam dados no banco de dados.

```
Default implementation of the org.springframework.data.repository.CrudRepository interface. This will offer you a more sophisticated interface than the plain EntityManager.

Author: Oliver Gierke, Eberhard Wolff, Thomas Darimont, Mark Paluch, Christoph Strobl, Stefan Fussenegger, Jens Schauder, David Madden, Moritz Becker, Sander Krabbenborg, Jesse Wouters, Greg Turnquist, Yanming Zhou, Ernst-Jan van der Laan, Diego Krupitza

Type parameters: <T> – the type of the entity to handle
                 <ID> – the type of the entity's identifier

9 usages 2 inheritors
@Repository
@Transactional(readOnly = true)
public class SimpleJpaRepository<T, ID> implements JpaRepositoryImplementation<T, ID> {
```

Tais métodos funcionam apenas quando houver uma transação, e com a anotação o próprio Spring se encarrega de gerenciar as transações sempre que o método se inicia e encerra, iniciando e encerrando a transação.

```
/*
 * (non-Javadoc)
 * @see org.springframework.data.repository.CrudRepository#delete(java.io.Serializable)
 */
@Transactional
@Override
public void deleteById(ID id) {

    Assert.notNull(id, ID_MUST_NOT_BE_NULL);

    delete(findById(id).orElseThrow(() -> new EmptyResultDataAccessException(
        String.format("No %s entity with id %s exists!", entityInformation.getJavaType(), id), 1)));
}
```

Dado momento que estamos manipulando dados, podemos ter métodos que manipulam mais de uma transação no banco de dados e caso haja falha ao decorrer da execução do método, há riscos inserirmos dados inconsistentes no banco de dados e a API defeituosa.

```
@Transactional
public void deletar(Long restauranteId) {

    try{
        restauranteRepository.deleteById(restauranteId);
    } catch(DataIntegrityViolationException e){
        throw new EntidadeEmUsoException(Restaurante.class.getSimpleName(), restauranteId);
    } catch(EmptyResultDataAccessException e){
        throw new RestauranteNaoEncontradoException(restauranteId);
    }
}
```

Iniciaremos uma transação e os métodos que contiverem uma anotação de transação usarão a mesma transação já aberta. Caso haja falha no método, a transação gerenciada fará um rollback.

11.2. Refinando o payload de cadastro com @JsonIgnoreProperties

sábado, 11 de março de 2023 15:24

Ao tentarmos atualizar uma entidade com outra entidade dentro, podemos passar apenas o id da subentidade para que o JPA faça a persistência e adicione a foreign key no campo. Mas quando precisamos ignorar todas as outras propriedades e deixar somente o id para a serialização de JSON para Objeto Java, podemos utilizar propriedades para ignorar os atributos.

The screenshot shows a REST client interface. The top bar indicates a PUT request to `http://localhost:8080/restaurantes/1`. The 'Body' tab is selected, showing a JSON payload:

```
1 {
2   "nome": "Norddesa",
3   "taxaFrete": 99.90,
4   "cozinha": {
5     "id": 1,
6     "nome": "tentando atualizar cozinha"
7   }
8 }
```

Below the request, the 'Body' tab of the response is shown, displaying the updated JSON:

```
1 {
2   "id": 1,
3   "nome": "Norddesa",
4   "taxaFrete": 99.90,
5   "cozinha": {
6     "id": 1,
7     "nome": "Tailandesa"
8   }
9 }
```

```
@ConvertGroup(to = Groups.CozinhaId.class)
@NotNull
@Valid
@ManyToOne
@JoinColumn(name = "cozinha_id", nullable = false)
private Cozinha cozinha;
```

dentro de Restaurante há uma Cozinha

Ao fazer uma atualização, o JPA precisa apenas do id da entidade para adicionar a FK no banco de dados, ignorando todo o resto dos atributos. Mas estamos ignorando a intenção do usuário, precisamos indicar que o atributo "nome" em "cozinha" ao atualizar e adicionar um Restaurante é inexistente nesses casos. Como:

PUT http://localhost:8080/restaurantes/1

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "nome": "Norddesa",
3   "taxaFrete": 99.90,
4   "cozinha": {
5     "id": 1,
6     "nome": "tentando atualizar cozinha"
7   }
8 }
```

Body Cookies Headers (4) Test Results Status: 400 Bad Request Time: 25 ms Size: 495 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "status": 400,
3   "type": "http://localhost:8080/corpo-nao-legivel",
4   "title": "Corpo não legível",
5   "detail": "A propriedade 'cozinha.nome' está indisponível.",
6   "userMessage": "Ocorreu um erro interno inesperado no sistema. Tente novamente e se o problema persistir, entre em contato com o administrador do sistema.",
7   "timeStamp": "2023-03-11T15:56:29.9362827"
8 }
```

```
@JsonIgnore
@NotBlank
@Column(name = "nome", length = 30, nullable = false)
private String nome;
```

Mas ao adicionar ou buscar uma cozinha, a propriedade não serializa

GET http://localhost:8080/cozinhas/1

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

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (5) Test Results Status: 200 OK Time: 19 ms Size: 172 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1
3 }
```

Podemos ignorar uma propriedade específica dentro de Restaurante com @JsonIgnoreProperties

```

@JsonIgnoreProperties(value = "nome", allowGetters = true)
@ConvertGroup(to = Groups.CozinhaId.class)
@NotNull
@Valid
@ManyToOne
@JoinColumn(name = "cozinha_id", nullable = false)
private Cozinha cozinha;

```

A propriedade `allowGetters = true` habilita a serialização para os métodos `get` de dentro de `Restaurante`. Pois ao fazer uma busca em `Restaurante/1` o nome da cozinha também é necessário, mas ao adicionar um `Restaurante` com o nome da cozinha não é necessário.

The screenshot shows a REST client interface. The top section displays a PUT request to `http://localhost:8080/restaurantes/1`. The request body is a JSON object:

```

{
  "nome": "Norddesa",
  "taxaFrete": 99.90,
  "cozinha": {
    "id": 1,
    "nome": "tentando atualizar cozinha"
  }
}

```

The bottom section shows the response, which is a 400 Bad Request. The response body is a JSON object with the following details:

```

{
  "status": 400,
  "type": "http://localhost:8080/corpo-nao-legivel",
  "title": "Corpo não legível",
  "detail": "A propriedade 'cozinha.nome' está indisponível.",
  "userMessage": "Ocorreu um erro interno inesperado no sistema. Tente novamente e se o problema persistir, entre em contato com o administrador do sistema.",
  "timeStamp": "2023-03-11T16:01:32.0545891"
}

```

11.3. Criando classes de mixin para usar as anotações do Jackson

sábado, 11 de março de 2023 16:05

Ao decorrer do projeto, à medida que a aplicação vai evoluindo, utilizamos anotações nos atributos das classes de domínio para representar os recursos, ou seja, utilizamos o modelo de domínio na representação da api e utilizamos as anotações do Jackson para customizar essas representações.

Porém, ao observar, estamos usando essas anotações para customizar a representação de recursos dentro das classes de domínio, ou seja, estamos misturando as coisas tornando menos coesas ao utilizar anotações para api dentro de classes de domínio.

Podemos utilizar uma classe Mixin para separar os atributos anotados pelas anotações do Jackson da classe.

A ideia é separar a classe que possui vários atributos anotados em duas estruturas a fim de melhorar a coesão nas anotações Jackson. A classe original conterá todos os atributos originais sem adição ou diminuição, mas apenas deletando as anotações do Jackson, e a classe Mixin conterá os atributos anotados com anotações Jackson da classe original, somente atributos que estão anotados e somente anotações Jackson.

Classe original

```

25 public class Restaurante {
26     @EqualsAndHashCode.Include
27     @Id
28     @GeneratedValue(strategy = GenerationType.IDENTITY) //provedor de persistência
29     private Long id;
30
31     @NotBlank
32     @Column(nullable = false)
33     private String nome;
34
35     @NotNull
36     @PositiveOrZero
37     @Column(name = "taxa_frete", nullable = false)
38     private BigDecimal taxaFrete;
39
40     @ConvertGroup(to = Groups.CozinhaId.class)
41     @NotNull
42     @Valid
43     @ManyToOne
44     @JoinColumn(name = "cozinha_id", nullable = false)
45     private Cozinha cozinha;
46
47     @CreationTimestamp
48     @Column(nullable = false, columnDefinition = "datetime")
49     private LocalDateTime dataCadastro;
50
51     @UpdateTimestamp
52     @Column(nullable = false, columnDefinition = "datetime")
53     private LocalDateTime dataAtualizacao;
54
55     @ManyToMany
56     @JoinTable(name = "restaurante_forma_pagamento",
57         joinColumns = @JoinColumn(name = "restaurante_id"),
58         inverseJoinColumns = @JoinColumn(name = "forma_pagamento_id"))
59     private List<FormaPagamento> formasPagamento = new ArrayList<>();

```

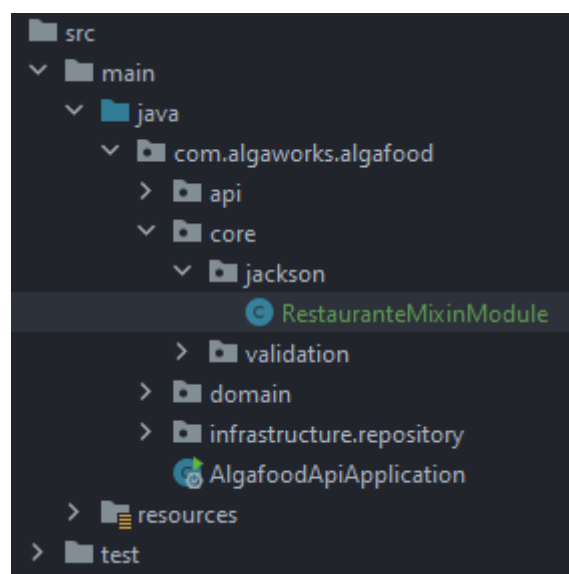
Classe Mixin

```

2 usages
15 public class RestauranteMixin {
16
17     @JsonIgnoreProperties(value = "nome", allowGetters = true)
18     private Cozinha cozinha;
19
20     @JsonIgnore
21     private LocalDateTime dataCadastro;
22
23     @JsonIgnore
24     private LocalDateTime dataAtualizacao;
25
26     @JsonIgnore
27     private List<FormaPagamento> formasPagamento = new ArrayList<>();
28
29     @JsonIgnore
30     @Embedded
31     private Endereco endereco;
32
33     @JsonIgnore
34     private List<Produto> produtos = new ArrayList<>();
35 }
36

```

As duas classes ainda não possuem conexão, temos que criar uma classe intermediária que fará como ponte



```

8 @Component
9 public class RestauranteMixinModule extends SimpleModule {
10     public RestauranteMixinModule() {
11         setMixInAnnotation(Restaurante.class, RestauranteMixin.class);
12     }
13 }

```

Estendemos a classe SimpleModule do Jackson e chamamos um método da superclasse setMixInAnnotation.

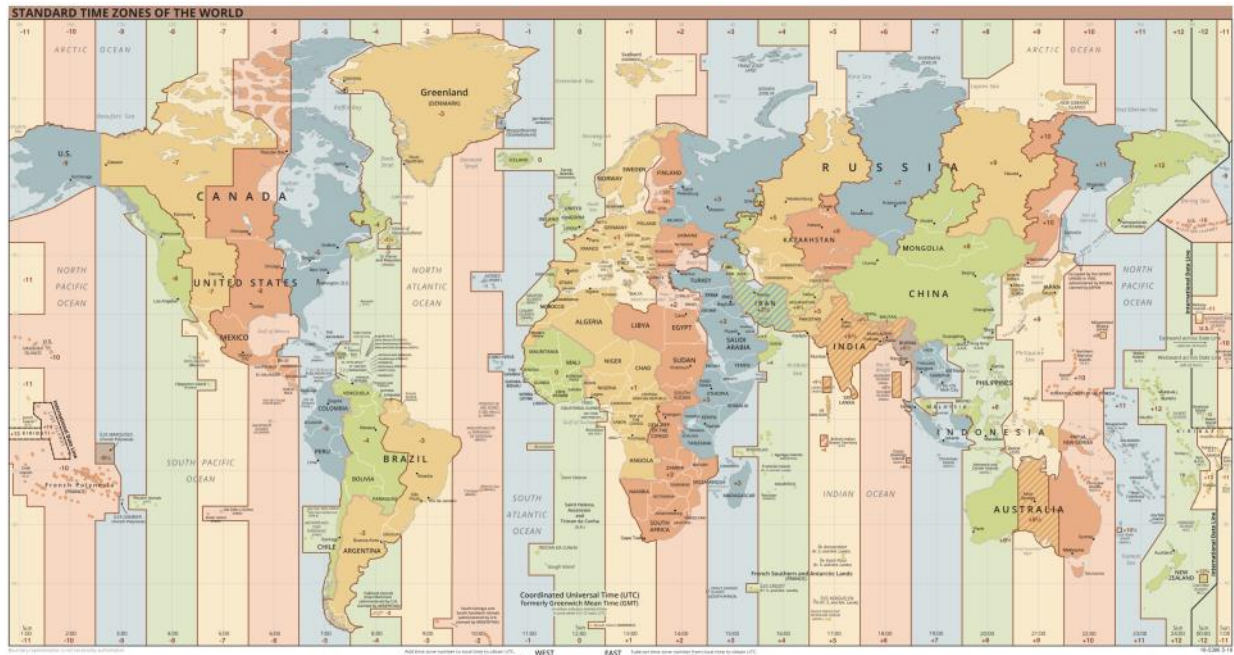
11.4. Desafio usando @JsonIgnoreProperties e Jackson Mixin

domingo, 12 de março de 2023

22:46

11.5. Antes de estudar sobre datahora relembrando as aulas de geografia e entendendo os fusos horários

segunda-feira, 13 de março de 2023 09:34



Horário Médio de Greenwich - Greenwich Mean Time - GMT: Baseado no UTC

Tempo Universal Coordenado - Coordinated Universal Time - UTC: 00:00

Offset: diferença de horas em relação ao UTC

Timezone: fuso horário

BRT: zona de Brasília

```
{  
  "lastLoginDate": "2019-10-12T14:15:38-03:00"  
}
```

Padrão ISO-8601: ano-mês-diaTIMEHH:mm:ss-offset

11.6. Boas práticas para trabalhar com data e hora em REST APIs

segunda-feira, 13 de março de 2023 10:03

5 Leis para se trabalhar com datas e horas

1 ISO-8601

- Padrão Internacional de Data e Hora. O objetivo é definir um método bem definido de representação para evitar interpretações confusas nos fusos horários de diferentes fusos horários.

Padrão ISO-8601: ano-mês-diaTIMEHH:mm:ss-offset

```
{
  "lastLoginDate": "2019-10-12T14:15:38-03:00"
}
```

Padrão ISO-8601: sem offset

```
{
  "lastLoginDate": "2019-10-12T14:15:38Z"
}
```

2 Aceitar qualquer fuso horário

- A aplicação tem a responsabilidade de converter os horários

```
{
  ...
  "servico": "Limpeza de bancos"
  "dataAgendamento": "2019-11-20T15:00:00-05:00"
}
```

Caso a API estiver no offset -03:00 por padrão. a diferença entre -3 e -5 é 02:00 horas, chegando no horário T17:00:00

3 Armazenar as datas no formato UTC

4 Retornar em UTC

5 Não incluir horário, se não for necessário

11.7. Configurando e refatorando o projeto para usar UTC

segunda-feira, 13 de março de 2023 10:25

Para entrar em conformidade com as regras e boas práticas de fuso horários, temos que refatorar o código.

- Os registros no banco de dados deverão ser armazenados no horário UTC.

```
(1, 'Thai Gourmet', 10, 1, utc_timestamp, utc_timestamp, 1,
```

- A aplicação também gerencia códigos referentes aos horários em UTC. Pois como foi visto na aula 11.6, a aplicação tem que retornar em UTC.

```
@JsonIgnore
private OffsetDateTime dataCadastro;

@JsonIgnore
private OffsetDateTime dataAtualizacao;
```

- A classe OffsetDateTime formata a data vinda do banco para um formato que tenha o offset. Mostrará a data formatada para o padrão da máquina onde a JVM está instalada

```
"dataCadastro": "2023-03-13T11:25:47-03:00",
"dataAtualizacao": "2023-03-13T11:25:47-03:00"
```

- Agora temos que configurar nossa aplicação

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

    public static void main(String[] args) {

        TimeZone.setDefault(TimeZone.getTimeZone( ID: "UTC"));
        SpringApplication.run(AlgafoodApiApplication.class, args);
    }
}
```

The screenshot displays a REST client interface. At the top, a GET request is configured for the URL `http://localhost:8080/restaurantes/1`. The 'Body' tab is selected, showing a message: 'This request does not have a body'. Below the request configuration, the response is shown in the 'Body' tab. The response status is 200 OK, with a time of 50 ms and a size of 330 B. The response body is displayed in JSON format, showing details for a restaurant with ID 1, named 'Thai Gourmet', with a shipping tax of 10.00, and a kitchen named 'Tailandesa'. The response also includes timestamps for 'dataCadastro' and 'dataAtualizacao'.

```
1  {
2    "id": 1,
3    "nome": "Thai Gourmet",
4    "taxaFrete": 10.00,
5    "cozinha": {
6      "id": 1,
7      "nome": "Tailandesa"
8    },
9    "dataCadastro": "2023-03-13T14:34:13Z",
10   "dataAtualizacao": "2023-03-13T14:34:13Z"
11 }
```

Agora nosso sistema tem um padrão UTC para representar datas e horas e a responsabilidade de formatar para o usuário fica a cargo do consumidor da api.

11.8. Desafio refatorando o código para usar `OffsetDateTime`

segunda-feira, 13 de março de 2023 11:36

- Revisar endpoints para ter certeza para seguir todas as regras sobre UTC
- Refatorar trechos de código que usam `LocalDateTime` para `OffsetDateTime`

11.9. Isolando o Domain Model do Representation Model com o padrão DTO

segunda-feira, 13 de março de 2023 12:39

Classe de transferência de dados - Data Transfer Object - DTO

Agrupa um conjunto de propriedades de uma ou mais classes a partir dos modelos de domínio da aplicação a fim de desacoplar as entidades dos modelos de representação de recursos

11.10. Implementando a conversão de entidade para DTO

segunda-feira, 13 de março de 2023 13:59

Vamos refatorar a entidade Restaurante para utilizar DTO como representação de um recurso Restaurante.

```
8  @Getter
9  @Setter
10 public class RestauranteDTO {
11     private Long id;
12     private String nome;
13     private BigDecimal taxaFrete;
14     private CozinhaDTO cozinha;
15 }
16
```

Método que converte um Restaurante para RestauranteDTO.

```
private static RestauranteDTO toDTO(Restaurante restaurante) {
    CozinhaDTO cozinhaDTO = new CozinhaDTO();
    cozinhaDTO.setId(restaurante.getCozinha().getId());
    cozinhaDTO.setNome(restaurante.getCozinha().getNome());

    RestauranteDTO restauranteDTO = new RestauranteDTO();
    restauranteDTO.setId(restaurante.getId());
    restauranteDTO.setNome(restaurante.getNome());
    restauranteDTO.setTaxaFrete(restaurante.getTaxaFrete());
    restauranteDTO.setCozinha(cozinhaDTO);
    return restauranteDTO;
}
```

Transformar lista de Restaurante em Lista de RestauranteDTO usando Method Reference

```
@GetMapping
public List<RestauranteDTO> listar() {
    final List<Restaurante> listaRestaurante = restauranteService.listar();

    return listaRestaurante.stream().map(RestauranteController::toDTO).toList();
}
```

11.11. Criando DTOs para entrada de dados na API

segunda-feira, 13 de março de 2023 15:34

Até o momento estamos usando um modelo de domínio, uma entidade, como entrada de recursos da api, ou seja, recebemos um JSON e ele é serializado para um modelo de representação de domínio.

Não existe uma forma correta, apenas formas de abordagens diferentes.

Nem sempre um recurso como é representado na saída contém os mesmos atributos na entrada, pois existem recursos somente leitura que aceitam diferentes atributos na entrada.

Poderemos ter mais de um DTO para modelar representações diferentes de um mesmo recurso.

DTO para entrada de informação da requisição.

```
/*DTO criado para a serialização de objetos Restaurante vindos de uma requisição que só necessitam
* dos atributos declarados abaixo*/
6 usages
@Getter
@Setter
public class RestauranteInputDTO {

    @NotBlank
    private String nome;

    @NotNull
    @PositiveOrZero
    private BigDecimal taxaFrete;

    @NotNull
    @Valid
    private CozinhaRefDto cozinha;
}
```

Alterando o controller

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public RestauranteDTO adicionar
    (@RequestBody @Valid RestauranteInputDTO restauranteInput) {
    try {
        return toDTO(restauranteService.salvar(toDomainModel(restauranteInput)));
    } catch (CozinhaNaoEncontradaException e) {
        throw new NegocioException(e.getMessage(), e);
    }
}
```


as outras camadas da aplicação não deverão entrar em contato com os DTO's, ou seja, não temos que alterar as assinaturas de métodos das outras camadas, por exemplo a **service**. Uma alternativa é criar um método dentro da classe que está utilizando o DTO para converter em outro objeto do domínio, ou ter um método dentro do DTO para converter para a sua classe originária.

```
1 usage
private Restaurante toDomainModel(RestauranteInputDTO restauranteInput) {

    Restaurante restaurante = new Restaurante();
    restaurante.setNome(restauranteInput.getNome());
    restaurante.setTaxaFrete(restauranteInput.getTaxaFrete());

    Cozinha cozinha = new Cozinha();
    cozinha.setId(restauranteInput.getCozinha().getId());
    restaurante.setCozinha(cozinha);

    return restaurante;
}
```

Discussão:

Temos agora uma entrada de dados DTO para representar um recurso adicionado utilizando a API. Tanto o DTO quanto sua classe originária estão anotados com anotações do Bean Validation para validar na entrada do controlador. A questão é: como na entrada do controlador quem está sendo validado é o DTO, podemos retirar as anotações Bean Validation das classes originárias? depende.

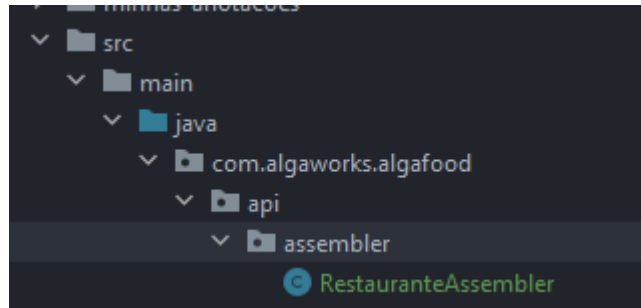
Devemos nos perguntar se a única entrada de dados da api é pelo endpoint do controlador e nos certificar que sim, podemos sim retirar as validações da classe originária, senão, é recomendado deixar as anotações nas duas classes.

Para futuras alterações na representação e entrada de dados, temos o dever de verificar as das classes com anotações do Bean Validation e alterá-las.

11.12. Refatorando e criando um assembler de DTO

segunda-feira, 13 de março de 2023 23:07

- Deletar o Mixin de Restaurante pois temos um DTO de Restaurante.
- Refatorar os métodos toDTO e toListDTO do Controller para uma classe assembler no pacote api.assembler



O nome Assembler quer dizer montador, mas poderia ser chamado de converter, transformer ou coisas relacionadas a converter um objeto a outro.

```
5 usages
10 @Component
11 public class RestauranteAssembler {
12     6 usages
13     @
14     public static RestauranteDTO toDTO(Restaurante restaurante) {
15         CozinhaDTO cozinhaDTO = new CozinhaDTO();
16         cozinhaDTO.setId(restaurante.getCozinha().getId());
17         cozinhaDTO.setNome(restaurante.getCozinha().getNome());
18
19         RestauranteDTO restauranteDTO = new RestauranteDTO();
20         restauranteDTO.setId(restaurante.getId());
21         restauranteDTO.setNome(restaurante.getNome());
22         restauranteDTO.setTaxaFrete(restaurante.getTaxaFrete());
23         restauranteDTO.setCozinha(cozinhaDTO);
24         return restauranteDTO;
25     }
26
27     @
28     public static List<RestauranteDTO> toListDTO(List<Restaurante> list){
29         return list.stream().map(RestauranteAssembler::toDTO).toList();
30     }
31 }
```

- Deletar classes Mixin referentes a Restaurante, pois já temos um DTO do mesmo.

Chamada do método adicionar no Controller

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public RestauranteDTO adicionar
    (@RequestBody @Valid RestauranteInputDTO restauranteInput) {
    try {
        return RestauranteAssembler.toDTO(restauranteService.salvar(toDomainModel(restauranteInput)));
    } catch (CozinhaNaoEncontradaException e) {
        throw new NegocioException(e.getMessage(), e);
    }
}
```

Foi escolhido uma classe para fazer as conversões mas poderia existir um método dentro do próprio DTO para que essa conversão seja feita.

11.13. Desafio Refatorando e criando um diasmembler do DTO

terça-feira, 14 de março de 2023

14:28

- Refatorar o método que converte um RestauranteInputDto vindo de uma requisição para uma entidade do domínio.

```
7      public class RestauranteInputDisassembler {
8
9      @
10         1 usage
11         public static Restaurante toDomainModel(RestauranteInputDTO restauranteInput) {
12             Restaurante restaurante = new Restaurante();
13             restaurante.setNome(restauranteInput.getNome());
14             restaurante.setTaxaFrete(restauranteInput.getTaxaFrete());
15
16             Cozinha cozinha = new Cozinha();
17             cozinha.setId(restauranteInput.getCozinha().getId());
18             restaurante.setCozinha(cozinha);
19
20             return restaurante;
21         }
22     }
```

11.14. Adicionando e usando o ModelMapper

terça-feira, 14 de março de 2023

15:19

Podemos refatorar o código que convertia objetos da entidade em Dtos e vice-versa utilizando o ModelMapper

<!--<https://mvnrepository.com/artifact/org.modelmapper/modelmapper>-->

<dependency>

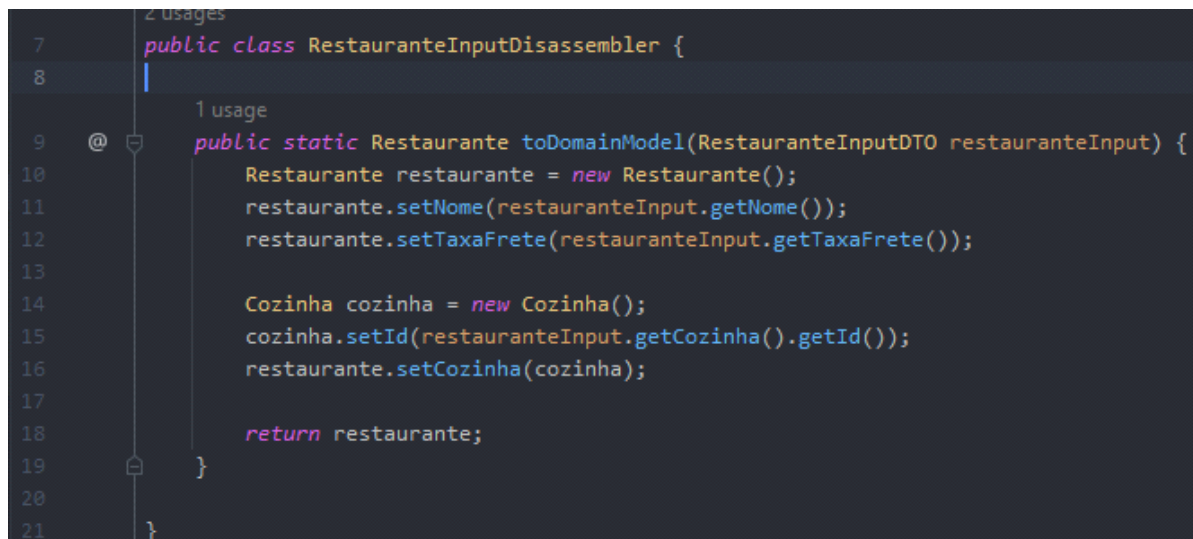
<groupId>org.modelmapper</groupId>

<artifactId>modelmapper</artifactId>

<version>3.1.1</version>

</dependency>

- Antes:



```
7 public class RestauranteInputDisassembler {
8
9     @
10     public static Restaurante toDomainModel(RestauranteInputDTO restauranteInput) {
11         Restaurante restaurante = new Restaurante();
12         restaurante.setNome(restauranteInput.getNome());
13         restaurante.setTaxaFrete(restauranteInput.getTaxaFrete());
14
15         Cozinha cozinha = new Cozinha();
16         cozinha.setId(restauranteInput.getCozinha().getId());
17         restaurante.setCozinha(cozinha);
18
19         return restaurante;
20     }
21 }
```

- Depois:



```
11 @Component
12 public class RestauranteAssembler {
13
14     @Autowired
15     ModelMapper modelMapper;
16
17     public RestauranteDTO toDTO(Restaurante restaurante) {
18         return modelMapper.map(restaurante, RestauranteDTO.class);
19     }
20 }
```

11.15. Entendendo a estratégia padrão do ModelMapper para correspondência de propriedades

terça-feira, 14 de março de 2023

16:45

ModelMapper possui estratégias de correspondências de propriedades (matching)

separa grupos de propriedades, tanto da classe origem quanto classe destino em tokens (grupo de propriedades).

Podemos utilizar diferentes nomes com correspondências semelhantes . Por exemplo, agora podemos facilmente trocar o nome dos atributos no DTO de resposta para serializar de uma forma diferente do que está na classe da entidade:

```
@Getter
@Setter
public class RestauranteDTO {
    private Long id;
    private String nome;
    private BigDecimal frete; // taxaFrete
    private CozinhaDTO cozinha;
}
```

representação json

```
1  {
2    "id": 1,
3    "nome": "Thai Gourmet",
4    "frete": 10.00,
5    "cozinha": {
6      "id": 1,
7      "nome": "Tailandesa"
8    }
9  }
```

inclusive em atributos em objetos alinhados a outros objetos, no caso de Cozinha dentro de Restaurante.

```
@Getter
@Setter
public class CozinhaDTO {
    private Long id;
    private String nomeCozinha;
}
```

No entanto, não possuímos um modelMapper para Cozinha

```
1  @Component
2  public class RestauranteAssembler {
3
4      1 usage
5      @Autowired
6      ModelMapper modelMapper;
7      5 usages
8      public RestauranteDTO toDTO(Restaurante restaurante) {
9
10         return modelMapper.map(restaurante, RestauranteDTO.class);
11     }
12 }
```

11.16. Customizando o mapeamento de propriedades com ModelMapper

terça-feira, 14 de março de 2023 18:59

Quando mudamos uma propriedade cuja a estratégia do modelMapper falha em atribuir à sua correspondência na classe de destino, podemos explicitar nas configurações as propriedades destino e propriedades origem.

Ao mudar taxaFrete para precoFrete do DTO, o modelMapper não conseguirá atribuir por não encontrar correspondências seguras o suficiente.

```
@Getter
@Setter
public class RestauranteDTO {
    private Long id;
    private String nome;
    private BigDecimal precoFrete;
    private CozinhaDTO cozinha;
}
```

GET: restaurantes/1

```
1
2      "id": 1,
3      "nome": "Thai Gourmet",
4      "precoFrete": null,
5      "cozinha": {
6        "id": 1,
7        "nome": "Tailandesa"
8      }
9
```

O banco traz um objeto tipo Restaurante e o método RestauranteAssembler.toDTO converte para um objeto DTO, mas o modelMapper.map dentro de toDTO não consegue obter a correspondência de precoFrete para fazer a atribuição taxaFrete-precoFrete, mas podemos customizar manualmente a correspondência de atributos ao configurar o modelMapper.

```
9
10
11
12
13
14
15
16
17
18
19
20
21
@Configuration
public class ModelMapperConfig {

    @Bean
    public ModelMapper modelMapper(){

        ModelMapper modelMapper = new ModelMapper();
        modelMapper.createTypeMap(Restaurante.class, RestauranteDTO.class)
            .addMapping(Restaurante::getTaxaFrete, RestauranteDTO::setPrecoFrete);

        return modelMapper;
    }
}
```

Primeiros mapeamos o tipo de classes para a correspondência e adicionamos o mapeamento do método getTaxaFrete para setPrecoFrete.

11.17. Mapeando para uma instância destino (e não um tipo) com ModelMapper

terça-feira, 14 de março de 2023 20:48

Vamos refatorar o método atualizar na camada de serviço no qual copiamos atributos vindos da requisição e atribuímos a uma entidade filtrando atributos que não queremos alterar.

```
82     @PutMapping("/{id}")
83     public RestauranteDTO atualizar(@RequestBody @Valid RestauranteInputDTO restauranteInput, @PathVariable Long id) {
84         final Restaurante restaurante = rInputDisassembler.toDomainModel(restauranteInput);
85         try {
86             Cozinha cozinha = cozinhaService.buscar(restaurante.getCozinha().getId());
87             restaurante.setCozinha(cozinha);
88         } catch (CozinhaNaoEncontradaException e) {
89             throw new NegocioException(e.getMessage(), e);
90         }
91
92         return rAssembler.toDTO(restauranteService.atualizar(restaurante, id));
93     }
```

Esse tipo de código "hardcode" é um pouco dificultoso pois sempre precisamos especificar de forma muito explícita e exposta atributos que não queremos alterar com BeanUtils.copyProperties.

```
@Transactional
public Restaurante atualizar(Restaurante restaurante, Long id) {

    Restaurante r = this.buscar(id);

    BeanUtils.copyProperties(restaurante, r, ...ignoreProperties: "id",
        "formasPagamento", "endereco", "dataCadastro", "produtos");

    return restauranteRepository.save(r);
}
```

Para atualizar usando modelMapper, podemos criar um método que copia atributos de um DTO vindo de uma requisição e atribui a uma instância do modelo de domínio Restaurante, vinda do banco de dados e com atributos atuais.

```
9     @Component
10     public class RestauranteInputDTODisassembler {
11
12         2 usages
13         @Autowired
14         Mapper modelMapper;
15
16         2 usages
17         public Restaurante toDomainModel(RestauranteInputDTO restauranteInput) {
18             return modelMapper.map(restauranteInput, Restaurante.class);
19         }
20
21         public void copyToDomainObject(RestauranteInputDTO restauranteInputDTO, Restaurante restaurante){
22             modelMapper.map(restauranteInputDTO, restaurante);
23         }
24     }
```

copyToDomainObject recebe no 1º argumento um input vindo da requisição e 2º um restaurante atual vindo do banco de dados, o método map do modelMap copia atributos do DTO do argumento 1 e atribui a um objeto originário do 2º argumento, automaticamente transformando-o em um novo objeto.

Método atualizar do controlador refatorado:

```
@PutMapping("/{id}")
public RestauranteDTO atualizar(@RequestBody @Valid RestauranteInputDTO restauranteInput, @PathVariable Long id) {

    final Restaurante restauranteAtual = restauranteService.buscar(id);
    rInputDisassembler.copyToDomainObject(restauranteInput, restauranteAtual);

    try {
        Cozinha cozinha = cozinhaService.buscar(restauranteAtual.getCozinha().getId());
        restauranteAtual.setCozinha(cozinha);
    } catch (CozinhaNaoEncontradaException e) {
        throw new NegocioException(e.getMessage(), e);
    }

    return rAssembler.toDTO(restauranteService.atualizar(restauranteAtual, id));
}
```

método atualizar da camada de serviço:

```
2 usages
@Transactional
public Restaurante atualizar(Restaurante restaurante, Long id) {

    return restauranteRepository.save(restaurante);
}
```

2ª parâmetro inutilizável.

Ao copiar atributos, o Hibernate lança uma exceção pois de certa forma estamos alterando o Id da cozinha

```
20 @
21     public void copyToDomainObject(RestauranteInputDTO restauranteInputDTO, Restaurante restaurante){
22
23         //evitar Caused by: org.hibernate.HibernateException: identifier of an instance of
24         // com.algaworks.algafood.domain.model.Cozinha was altered from 1 to 4
25         restaurante.setCozinha(new Cozinha());
26
27         modelMapper.map(restauranteInputDTO, restaurante);
28     }
29 }
```

11.18. Revisando e ajustando as mensagens de validação com o uso de DTOs

terça-feira, 14 de março de 2023

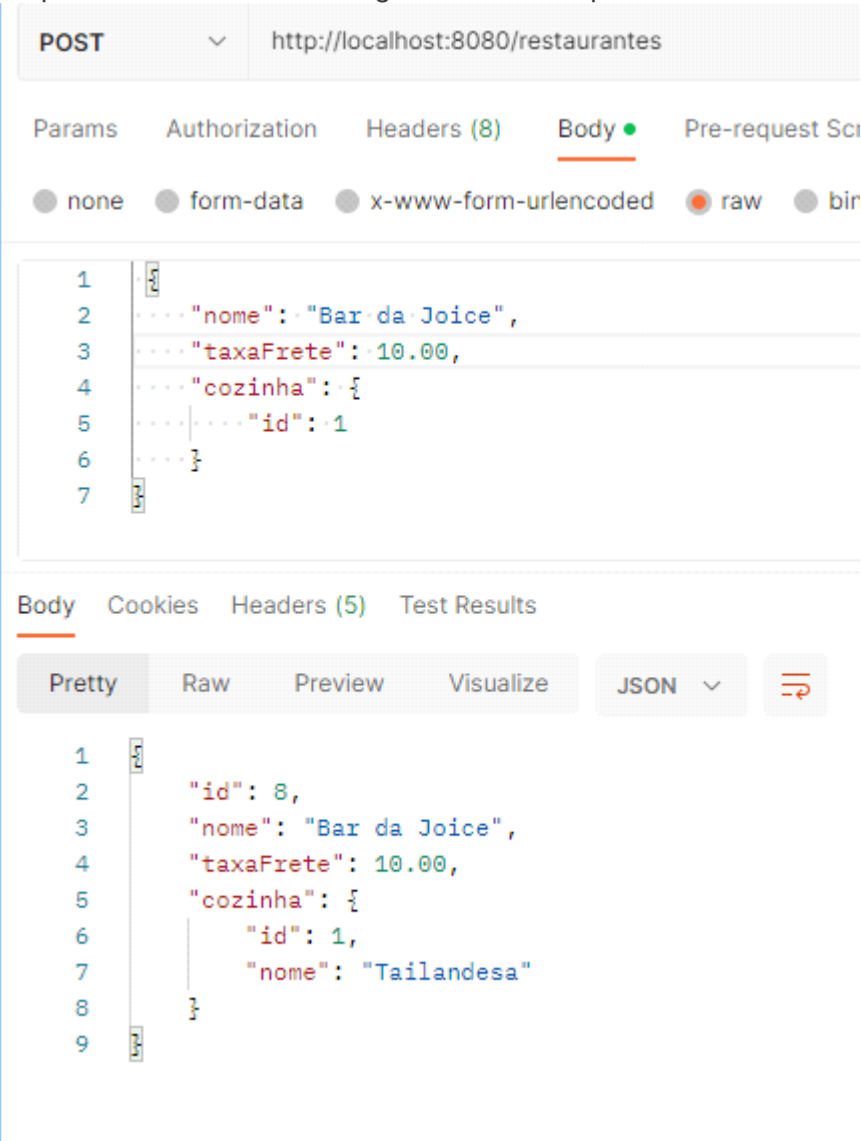
23:17

```
messages.properties x
19  #9.12. Desafio: customizando mensagens de validação
20
21  # Cozinha
22  cozinha.nome=Nome da cozinha
23  cozinha.id=Código da cozinha
24
25  # Restaurante
26  NotNull.restauranteInputDTO.taxaFrete={0} é obrigatória
27  NotNull.restauranteInputDTO.cozinha={0} é obrigatória
28  restauranteInputDTO.nome=Nome do restaurante
29  restauranteInputDTO.cozinha=Cozinha do restaurante
30  restauranteInputDTO.taxaFrete=Taxa de frete do restaurante
31
32  # Estado
33  estado.nome=Nome do estadozzzzzz
34  estado.id=Código do estado
35
36  # Cidade
37  cidade.nome=Nome da cidade
38  cidade.estado=Estado da cidade
39  #javax.validation.constraints.PositiveOrZero.message = deve
40  taxaFrete.invalida={0} está inválido
41
```

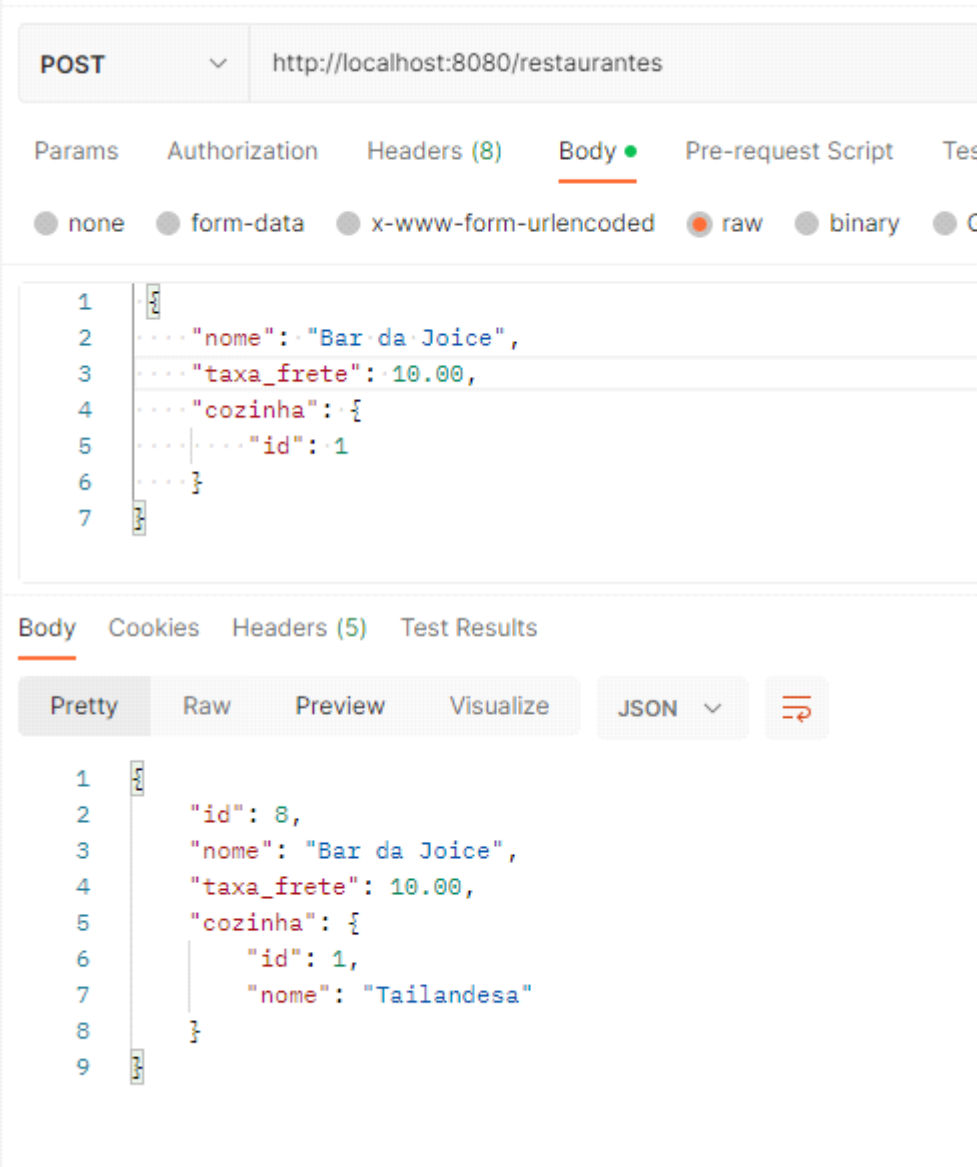
11.19. Estratégias de nomes de propriedades-snake case vs lower camel case

terça-feira, 14 de março de 2023 23:45

Por padrão, o Jackson serializa os nomes em JSON no formato lower camel case, que são iniciadas por letra minúscula e seguidas de cada palavra de letras maiúsculas.



Podemos alterar a estratégia de nome JSON para Snake Case, no caso são nomes de propriedades iniciadas por minúsculas e separadas por underscore.



11.20. Desafio usando DTOs como representation model

terça-feira, 14 de março de 2023 23:59

- Criar DTO's para não expor entidades Cozinha, Cidade e Estado.
- Deletar Mixin's restantes.
- Criar Assemblers e Disassemblers para cada DTO's.
- Ajustar controladores para que não mude externamente a API.
- Definir anotações em atributos dos DTO's para validações.

- ☒ DTO estado
- ☒ DTO input estado
 - ☒ DTOinput com anotações Bean Validation da entidade de origem
- ☒ Assembler e disassembler Estado
 - ☒ Assembler: entidade to DTO: lista de entidade para lista de DTO
 - ☒ Disassembler: DTO para Entidade: atributos DTOinput para objeto da entidade
- ☐ Ajustes nas mensagens de exceção do Bean Validation
- ☐ Apagar anotações Bean Validation da entidade de origem mantendo apenas anotação JPA

- Erro ao enviar cidade com id de estado nullo

11.21. Corrigindo bug de tratamento de exception de integridade de dados com flush do JPA

quarta-feira, 15 de março de 2023

15:37

Na aula [11.1. Analisando e definindo melhor o escopo das transações](#) adicionamos anotações de transações nos método de deleção, mas acabamos adicionando as exclusões no contexto do JPA para que ao finalizar o método, o JPA realizasse o commit. Ao realizar o commit, um erro é lançado mas não conseguimos capturar por conta da realização do commit depois dos métodos trycat finalizando o método anotado com @Transactional.

Podemos afirmar uma transação com o método flush logo após realizações de alteração no banco.

```
@Transactional
public void deletar(Long cidadeId) {
    try {
        cidadeRepository.deleteById(cidadeId);
        cidadeRepository.flush();
    } catch (DataIntegrityViolationException e) {
        throw new EntidadeEmUsoException(Cidade.class.getSimpleName(), cidadeId);
    } catch (EmptyResultDataAccessException e) {
        throw new CidadeNaoEncontradaException(cidadeId);
    }
}
```