

## 8.1. Introdução ao tratamento e modelagem de erros

quinta-feira, 23 de fevereiro de 2023

17:54



Formas de tratamento de exceptions

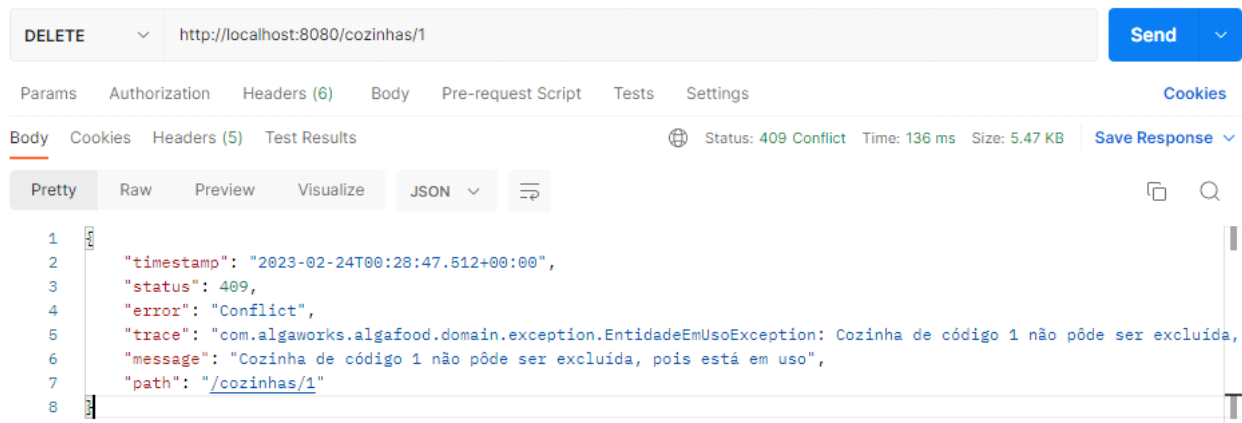
Customização de respostas

Modelar a resposta do erro de forma padronizada

## 8.2. Lançando exceções customizadas anotadas com @ResponseStatus

quinta-feira, 23 de fevereiro de 2023 19:19

Podemos tornar nosso código no controlador menos burocrático somente anotando nossas classes customizadas de exceção com @ResponseStatus. Mas não é a melhor maneira de tratar exceções pois **ficamos presos a resposta padrão do erro**:



Podemos anotar a classe passando no valor o código de status

```
12 usages
@ResponseStatus(value = HttpStatus.CONFLICT)
public class EntidadeEmUsoException extends RuntimeException{
    4 usages
    public EntidadeEmUsoException(String message) { super(message); }
}
```

e lançar quando o erro for capturado

```
public void deletar (Long cozinhaId){
    try {
        cozinhaRepository.deleteById(cozinhaId);
    } catch (DataIntegrityViolationException e){
        throw new EntidadeEmUsoException(String.format("Cozinha de código %d não pôde ser excluída, pois está em uso",
            cozinhaId));
    }
    catch (EmptyResultDataAccessException e){
        throw new EntidadeNaoEncontradaException(String.format("Cozinha de código %d não pôde ser encontrada",
            cozinhaId));
    }
}
```

Agora podemos tornar o código mais limpo e menos burocrático na requisição

```

/* @DeleteMapping("/{cozinhaId}")
public ResponseEntity<Cozinha> deletar (@PathVariable Long cozinhaId) {
    try {
        cozinhaService.deletar(cozinhaId);
        return ResponseEntity.noContent().build();

    } catch (EntidadeEmUsoException e) {
        return ResponseEntity.status(HttpStatus.CONFLICT).build();

    } catch (EntidadeNaoEncontradaException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
    }
}
} */

```

```

@DeleteMapping("/{cozinhaId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deletar (@PathVariable Long cozinhaId) {

    cozinhaService.deletar(cozinhaId);
}

```

Caso a requisição seja **concluída com sucesso**, retorna uma resposta No Content

DELETED <http://localhost:8080/cozinhas/5> Send

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

Body Cookies Headers (3) Test Results Status: 204 No Content Time: 31 ms Size: 112 B Save Response

Pretty Raw Preview Visualize Text

1

caso a **entidade esteja em uso**, a exceção com a anotação será lançada

DELETED <http://localhost:8080/cozinhas/2> Send

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

Body Cookies Headers (5) Test Results Status: 409 Conflict Time: 17 ms Size: 5.47 KB Save Response

Pretty Raw Preview Visualize JSON

```

1  {
2    "timestamp": "2023-02-24T00:39:39.804+00:00",
3    "status": 409,
4    "error": "Conflict",
5    "trace": "com.algaworks.algafood.domain.exception.EntidadeEmUsoException: Cozinha de código 2 não pôde ser excluída,
6    "message": "Cozinha de código 2 não pôde ser excluída, pois está em uso",
7    "path": "/cozinhas/2"
8  }

```

```

@ResponseStatus(value = HttpStatus.CONFLICT)
public class EntidadeEmUsoException extends RuntimeException{
    4 usages
    public EntidadeEmUsoException(String message) { super(message); }
}

```

## 8.3. Lançando exceções do tipo `ResponseStatusException`

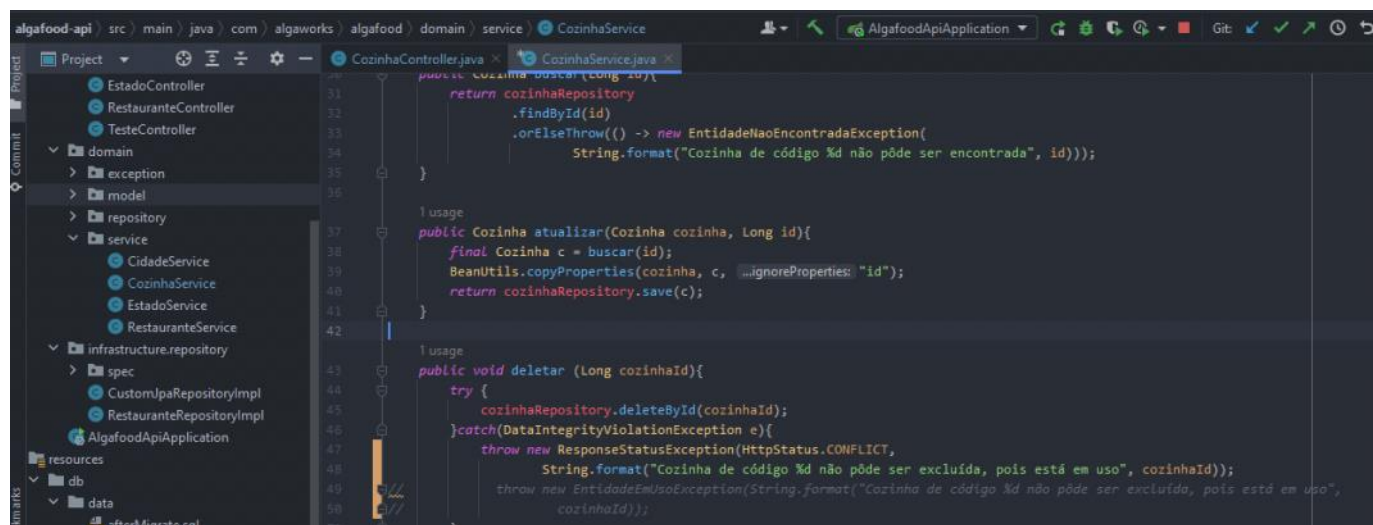
quinta-feira, 23 de fevereiro de 2023 22:05

A partir do Spring v5 foi introduzida uma nova classe de exception que serve como exception base (associada ao código HTTP e uma mensagem) que podemos lançar sem criar uma exceção específica. A `ResponseStatusException`

```
1 usage
public void deletar (Long cozinhaId){
    try {
        cozinhaRepository.deleteById(cozinhaId);
    } catch (DataIntegrityViolationException e){
        throw new ResponseStatusException(HttpStatus.CONFLICT,
            String.format("Cozinha de código %d não pôde ser excluída, pois está em uso", cozinhaId));
    }
}
```

Agora não precisamos de classes que implementam `RuntimeException`. Pois `ResponseStatusException` recebe como argumento o código de erro e a mensagem

Agora vamos observar a nossa estrutura de camadas da aplicação



Estamos em uma classe de domínio da aplicação, mais especificamente, uma classe de negócio, não deveríamos estar preocupados com classes e exceções a nível de controlador, a nível da web ou protocolo HTTP. Caso fosse exceções de negócio, poderíamos estar preocupados.

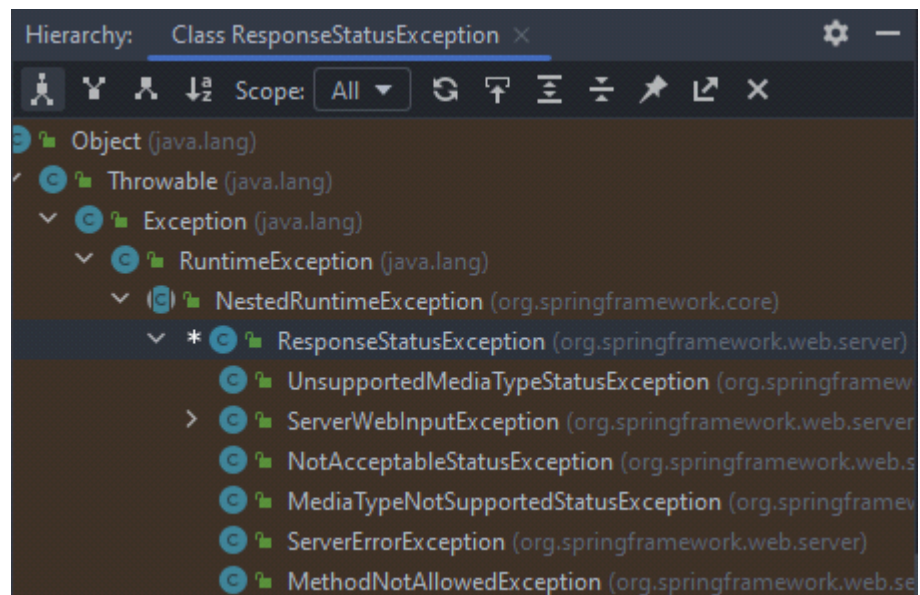
podemos voltar ao estado anterior do método

```
1 usage
public void deletar (Long cozinhaId){
    try {
        cozinhaRepository.deleteById(cozinhaId);
    } catch (DataIntegrityViolationException e){
        throw new EntidadeEmUsoException(String.format("Cozinha de código %d não pôde ser excluída, pois está em uso",
            cozinhaId));
    }
    catch (EmptyResultDataAccessException e){
        throw new EntidadeNaoEncontradaException(String.format("Cozinha de código %d não pôde ser encontrada",
            cozinhaId));
    }
}
```

capturar a exceção lançada por `EntidadeEmUsoException` e relançar com a `ResponseStatusException`

```
@DeleteMapping("/{cozinhaId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deletar(@PathVariable Long cozinhaId) {
    try {
        cozinhaService.deletar(cozinhaId);
    } catch (EntidadeEmUsoException e) {
        throw new ResponseStatusException(HttpStatus.CONFLICT,
            String.format("Cozinha de código %d não pôde ser excluída, pois está em uso", cozinhaId));
    }
}
```

Podemos utilizar subclasses de `ResponseStatusException` que tratam de forma mais específica alguns erros com códigos HTTP



O uso da `ResponseStatusException` é útil para projetos pequenos ou protótipos pois utiliza uma classe para customizar a resposta da requisição, uma alternativa a criação de subclasses de `RuntimeException` e anotá-las com anotações de resposta de status (`@ResponseStatus`), porém, ainda não conseguimos customizar o corpo da resposta por completo, principalmente se o intuito for a padronização de erros como resposta.

```
{
  "timestamp": "2023-02-24T17:03:00.553+00:00",
  "status": 409,
  "error": "Conflict",
  "trace": "org.springframework.web.server.ResponseStatusException: 409 CONFLICT \\",
  "message": "Cozinha de código 2 não pôde ser excluída, pois está em uso",
  "path": "/cozinhas/2"
}
```

## 8.4. Estendendo ResponseStatusException

sexta-feira, 24 de fevereiro de 2023 14:03

Como vimos antes, podemos anotar classes de exceção com `ResponseStatus` para definir os status ou podemos lançar diretamente exceções do tipo `ResponseStatusException`. Temos outra opção de estender `ResponseStatusException` e criarmos nossa própria exceção base com status pré-definidos

```
31 usages
public class EntidadeNaoEncontradaException extends ResponseStatusException {
    /**
     * status code 404 Not Found
     *
     * @param mensagem
     */
    8 usages
    public EntidadeNaoEncontradaException(String mensagem) {
        this(HttpStatus.NOT_FOUND, mensagem);
    }

    /**
     * Custom status code
     * @param status
     * @param reason
     */
    1 usage
    public EntidadeNaoEncontradaException(HttpStatus status, String reason) {
        super(status, reason);
    }
}
```

a ideia é instanciar `EntidadeNaoEncontradaException` utilizando o construtor com 1 argumento e utilizando o status code padrão ou utilizar o construtor sobrecarregado passando o `HttpStatus` e a mensagem

```
1 usage
public void deletar (Long cozinhaId){
    try {
        cozinhaRepository.deleteById(cozinhaId);
    } catch (DataIntegrityViolationException e){
        throw new EntidadeEmUsoException(String.format("Cozinha de código %d não pôde ser excluída, pois está em uso",
            cozinhaId));
    }
    catch (EmptyResultDataAccessException e){
        throw new EntidadeNaoEncontradaException(HttpStatus.NOT_FOUND, String.format("Cozinha de código %d não pôde ser encontrada",
            cozinhaId));
    }
}
```

porém, quebra o limite de responsabilidade da classe de domínio novamente como em [8.3. Lançando exceções do tipo ResponseStatusException](#)



## 8.5. Simplificando o código com o uso de @ResponseStatus em exceptions

sexta-feira, 24 de fevereiro de 2023 14:50

Apenas refatorando o código

Temos um método buscarOuFalhar que tem a responsabilidade de buscar uma entidade ou lançar uma exceção anotada com @ResponseStatus e passando uma mensagem no argumento do construtor

```
private Cozinha buscarOuFalhar(Long id) {  
    return cozinhaRepository  
        .findById(id)  
        .orElseThrow(() -> new EntidadeNaoEncontradaException(  
            String.format(ErrorMessage.ENTIDADE_NOT_FOUND.get(), id)));  
}
```

Um método atualizar que atualiza uma entidade que não possui uma outra entidade como propriedade

```
public Cozinha atualizar(Cozinha cozinha, Long id){  
    Cozinha cAtual = buscar(id);  
    BeanUtils.copyProperties(cozinha, cAtual, ...ignoreProperties: "id");  
    return cozinhaRepository.save(cAtual);  
}
```

e retorna a própria entidade, pois retorna por padrão um status code 200

```
@PutMapping("/{cozinhaId}")  
public Cozinha atualizar(@RequestBody Cozinha cozinha, @PathVariable Long cozinhaId)  
    return cozinhaService.atualizar(cozinha, cozinhaId);  
}
```

The screenshot shows a REST client interface. The top section displays a PUT request to `http://localhost:8080/cozinhas/10`. The request body is a JSON object: `{ "nome": "Norueeeegdduesa" }`. The bottom section shows the response, which is a 404 Not Found status. The response body is a JSON object: `{ "timestamp": "2023-02-24T18:52:16.958+00:00", "status": 404, "error": "Not Found", "trace": "com.algaworks.algafood.domain.exception.EntidadeNaoEncontradaException: A entidade Cozinha de código 10 não", "message": "A entidade Cozinha de código 10 não pôde ser encontrada", "path": "/cozinhas/10" }`.

## 8.6. Desafio refatorando os serviços REST

sexta-feira, 24 de fevereiro de 2023 15:53



## 8.7. Analisando os impactos da refatoração

sexta-feira, 24 de fevereiro de 2023 21:49

## 8.8. Criando a exception NegocioException

sábado, 25 de fevereiro de 2023

15:35

Não podemos tratar exceções da camada de serviço pensando no que o controlador pode receber ou pensando na resposta do erro na requisição.

O ideal é fazer alguma classe de exceção e tratar ela no controlador, ou relançar uma exceção com outra exceção que faça mais sentido e não lançar exceções da camada de serviço para fazer sentido na camada do controlador

```
@PutMapping("/{id}")
public Cidade atualizar(@RequestBody Cidade cidade, @PathVariable Long id) {

    try {
        Estado estado = estadoService.buscar(cidade.getEstado().getId());
        cidade.setEstado(estados);
    } catch (EntidadeNaoEncontradaException e) {
        throw new NegocioException(e.getMessage());
    }

    return cidadeService.atualizar(cidade, id);
}
```

Exemplo: A exceção NegocioException está sendo tratada dentro do controlador pois faz mais sentido nessa camada do que na camada de serviço, pois estamos buscando um recurso, e a exceção de busca de recurso lança um EntidadeNaoEncontradaException

tratar exceções de negócio com entidades que possuem outras entidades como propriedades

## 8.9. Desafio usando a exception `NegocioException`

sábado, 25 de fevereiro de 2023

18:42

## 8.10. Afinando a granularidade e definindo a hierarquia das exceptions de negócios

sábado, 25 de fevereiro de 2023

18:43

Do jeito que estávamos fazendo, tratando a exceção de uma forma genérica

```
@PostMapping("/{id}")
public Cidade atualizar(@RequestBody Cidade cidade, @PathVariable Long id) {

    try {
        Estado estado = estadoService.buscar(cidade.getEstado().getId());
        cidade.setEstado(estados);
    } catch (EntidadeNaoEncontradaException e) {
        throw new NegocioException(e.getMessage());
    }

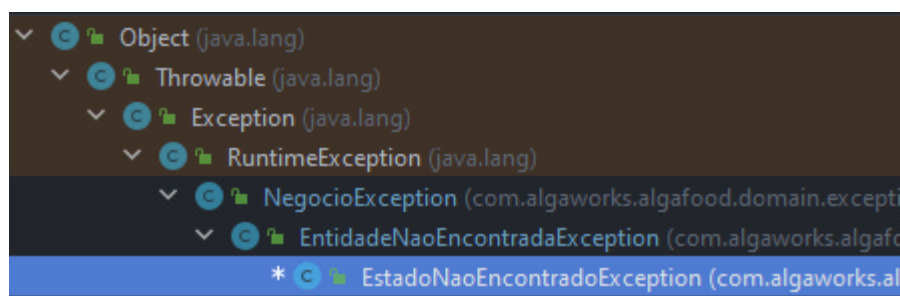
    return cidadeService.atualizar(cidade, id);
}
```

no qual o método `atualizar cidade` que verifica se o `Estado` que está na entidade existir, lança uma exceção, pode ser um convite a erros mais na frente no projeto, pois, ao evoluirmos o projeto, podemos fazer outros tratamentos e pode cair no catch `EntidadeNaoEncontradaException`.

Quando a granularidade da exceção é muito grossa, ou seja, muito genérica, não podemos tratar ela de forma específica. Uma recomendação é criar subclasses para que uma exceção possa ser tratada de forma específica e quando não houver necessidade, poderíamos tratar de forma mais genérica.

```
// @ResponseStatus(HttpStatus.NOT_FOUND)
public class EstadoNaoEncontradoException extends EntidadeNaoEncontradaException {
    public EstadoNaoEncontradoException(String msg) {
        super(msg);
    }
}
```

A anotação `@ResponseStatus` pode ser retirada pois a classe pai já está anotada



Agora podemos tratar especificamente uma entidade com sua exceção de negócio, que também podemos definir a hierarquia das exceptions.

```

@PutMapping("/{id}")
public Cidade atualizar(@RequestBody Cidade cidade, @PathVariable Long id) {

    try {
        Estado estado = estadoService.buscar(cidade.getEstado().getId());
        cidade.setEstado(estados);
    } catch (EstadoNaoEncontradoException e) {
        throw new NegocioException(e.getMessage(), e);
    }

    return cidadeService.atualizar(cidade, id);
}

```

Agora no endpoint que trata atualização de entidade, ao passar uma entidade que contém como propriedade uma entidade com id inválido (inexistente) não lançamos um 404 not found, pois o recurso existe, somente foi um erro de dados da requisição (400 bad request)

## 8.11. Desafio lançando exceptions de granularidade fina

sábado, 25 de fevereiro de 2023

22:13

## 8.12. Tratando exceções em nível de controlador com @ExceptionHandler

sábado, 25 de fevereiro de 2023 23:02

Somente com exceptions é trabalhoso e não recomendado customizar mensagens de erro de uma exception customizada

The screenshot shows a REST client interface. The top bar indicates a POST request to `http://localhost:8080/restaurantes`. The 'Body' tab is selected, showing a JSON payload: `{ "nome": "Colombiana", "taxaFrete": 5.99, "cozinha": 8, "id": 8 }`. The bottom section shows the response: Status: 400 Bad Request, Time: 22 ms, Size: 6.07 KB. The response body is displayed in JSON format: 

```
{
  "timestamp": "2023-02-26T16:03:31.394+00:00",
  "status": 400,
  "error": "Bad Request",
  "trace": "com.algaworks.algafood.domain.exception.NegocioException: A entidade Cozinha de código 8 não pôde ser encontrada",
  "message": "A entidade Cozinha de código 8 não pôde ser encontrada",
  "path": "/restaurantes"
}
```

```
@PostMapping
public ResponseEntity<> adicionar(@RequestBody Restaurante restaurante) {
    try {
        Restaurante restauranteSalvo = restauranteService.salvar(restaurante);
        return ResponseEntity.status(HttpStatus.CREATED).body(restauranteSalvo);
    } catch (CozinhaNaoEncontradaException e) {
        throw new NegocioException(e.getMessage(), e);
    }
}
```

Podemos ter acesso ao `ResponseEntity` que é enviado na requisição. Caso dê errado, podemos criar um método que trata exceções de dentro do controlador da entidade utilizando `@ExceptionHandler(Class<?>)`

```
@ExceptionHandler(EntidadeNaoEncontradaException.class) /*também subclasses*/
public ResponseEntity<> handleEntidadeNaoEncontradaException(
    EntidadeNaoEncontradaException e
){
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(e.getMessage());
}
```

No método `adicionar`, é lançada uma exceção do tipo `NegocioException`, mas no 2º argumento é passado a **causa** que é do tipo **Throwable** e o `ExceptionHandler` consegue interceptar se a causa for do tipo de exceção mapeada pela anotação.



Para customizar a mensagem de erro e deixar a exception mais compreensível, podemos criar uma classe própria de representação do erro

```
5 usages
@Getter
@Builder
public class Error {

    private LocalDateTime dataHora;
    private String mensagem;
}
```

```
@ExceptionHandler(EntidadeNaoEncontradaException.class) /*também subclasses*/
public ResponseEntity<?> handleEntidadeNaoEncontradaException(EntidadeNaoEncontradaException e) {
    Error error = Error.builder()
        .dataHora(LocalDateTime.now())
        .mensagem(e.getMessage())
        .build();

    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
}

@ExceptionHandler(NegocioException.class) /*também subclasses*/
public ResponseEntity<?> handleNegocioException(NegocioException e) {
    Error error = Error.builder()
        .dataHora(LocalDateTime.now())
        .mensagem(e.getMessage())
        .build();

    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(error);
}
```

Agora podemos criar um objeto representando o erro e atribuir as informações às propriedades e passar tal objeto no corpo da requisição.

The screenshot shows a REST client interface. At the top, a GET request is made to `http://localhost:8080/cidades/11`. Below the request bar, tabs for Params, Authorization, Headers (6), Body, Pre-request Script, Tests, and Settings are visible. The 'Query Params' section is empty. The response section shows a status of 404 Not Found, a time of 47 ms, and a size of 284 B. The response body is displayed in JSON format:

```
{
  "dataHora": "2023-02-26T13:51:46.2385965",
  "mensagem": "A entidade Cidade de código 11 não pôde ser encontrada"
}
```

Novamente, o método que intercepta as exceções lançadas está isolado na classe controladora pois está anotada a nível de método

## 8.13. Tratando exceções globais com @ExceptionHandler e @ControllerAdvice

domingo, 26 de fevereiro de 2023

14:28

Na aula [8.12. Tratando exceções em nível de controlador com @ExceptionHandler](#) estávamos tratando exceções com exception handler a nível de método - do próprio controlador, o que não seria viável para o projeto evoluir, pois iríamos duplicar o código para cada controlador.

Existe uma forma de criar exceptions handler a nível global utilizando `@ControllerAdvice`. Com essa anotação, dizemos ao Spring que a classe `ApiExceptionHandler` intercepta todas as exceções na qual os métodos estiverem assinados com `@ExceptionHandler` passando a exceção na propriedade da anotação

```
@ControllerAdvice
public class ApiExceptionHandler {

    @ExceptionHandler(EntidadeNaoEncontradaException.class) /*também subclasses*/
    public ResponseEntity<?> handleEntidadeNaoEncontradaException(EntidadeNaoEncontradaException e) {
        Error error = Error.builder()
            .dataHora(LocalDateTime.now())
            .mensagem(e.getMessage())
            .build();

        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
    }

    @ExceptionHandler(NegocioException.class) /*também subclasses*/
    public ResponseEntity<?> handleNegocioException(NegocioException e) {
        Error error = Error.builder()
            .dataHora(LocalDateTime.now())
            .mensagem(e.getMessage())
            .build();

        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(error);
    }
}
```

Especialização de `@Component` para classes que declaram os métodos `@ExceptionHandler`, `@InitBinder` ou `@ModelAttribute` a serem compartilhados entre várias classes `@Controller`. As classes anotadas com `@ControllerAdvice` podem ser declaradas explicitamente como Spring beans ou detectadas automaticamente por meio de varredura de caminho de classe. Todos esses beans são classificados com base na semântica `Ordered` ou nas declarações `@Order` `@Priority`, com a semântica `Ordered` tendo precedência sobre as declarações `@Order` `@Priority`. Os beans `@ControllerAdvice` são então aplicados nessa ordem no tempo de execução. Observe, no entanto, que os beans `@ControllerAdvice` que implementam `PriorityOrdered` não recebem prioridade sobre os beans `@ControllerAdvice` que implementam `Ordered`. Além disso, `Ordered` não é honrado para beans `@ControllerAdvice` com escopo — por exemplo, se tal bean tiver sido configurado como um bean com escopo de solicitação ou de sessão. Para lidar com exceções, um `@ExceptionHandler` será escolhido no primeiro aviso com um método manipulador de exceção correspondente. Para atributos de modelo e inicialização de ligação de dados, os métodos `@ModelAttribute` e `@InitBinder` seguirão a ordem `@ControllerAdvice`. Nota: Para métodos `@ExceptionHandler`, uma correspondência de exceção raiz será preferida a apenas corresponder a uma causa da exceção atual, entre os métodos manipuladores de um determinado bean de aviso. No entanto, uma correspondência de causa em um aviso de prioridade mais alta ainda terá preferência sobre qualquer correspondência (seja raiz ou nível de causa) em um bean de aviso de prioridade mais baixa. Como consequência, declare seus mapeamentos de exceção de raiz primária em um bean de aviso priorizado com uma ordem correspondente. Por padrão, os métodos em um `@ControllerAdvice` se aplicam globalmente a todos os controladores. Use seletores como anotações, `basePackageClasses` e `basePackages` (ou seu valor de alias) para definir um subconjunto mais restrito de controladores de destino. Se vários seletores forem declarados, a lógica OR booleana será aplicada, o que significa que os controladores selecionados devem corresponder a pelo menos um seletor. Observe que as verificações do seletor são realizadas em tempo de execução, portanto, adicionar muitos seletores pode afetar negativamente o desempenho e aumentar a complexidade.

Agora definimos um ponto central na nossa aplicação para tratamento de exception handlers.

Agora todos os controladores ou camadas de serviço que lançam

```
@ExceptionHandler(EntidadeNaoEncontradaException.class) /*também subclasses*/
```

ou

```
@ExceptionHandler(NegocioException.class) /*também subclasses*/
```

e também subclasses na hierarquia são capturados por pelos exceptions handlers da classe `ControllerAdvice`.

## 8.14. Desafio implementando exception handler

domingo, 26 de fevereiro de 2023

18:43

Primeiro tentamos deletar uma entidade que está sendo usada como foreign key, e lançamos como `EntidadeEmUsoException`

```
public void deletar(Long estadoId) {  
    try {  
        estadoRepository.deleteById(estadoId);  
    } catch (DataIntegrityViolationException e) {  
        throw new EntidadeEmUsoException(Estado.class.getSimpleName(), estadoId);  
    }  
}
```

O Spring lança uma exceção de acesso ao banco de dados e capturamos para relançar como `EntidadeNaoEncontradaException` passando o nome da entidade e o id da mesma.

```
11 usages  
public class EntidadeEmUsoException extends NegocioException{  
    4 usages  
    public EntidadeEmUsoException(String entidade, Long id) {  
        super(String.format(ErrorMessage.ENTIDADE_EM_USO.get(), entidade, id)  
    }  
}
```

A exception tem o construtor que repassar para o construtor da super classe a mensagem de erro

```
/**  
 * Constantes de String que representam a mensagem de erro de exceptions.  
 * O método get() retorna a string que é utilizada em um  
 * String.format().  
 * O 1º argumento recebe um valor em String,  
 * o 2º argumento recebe um valor numérico inteiro (Long, int, Integer).  
 * Ex: String.format(ENTIDADE_NOT_FOUND.get(), entidade, entidadeId)  
 * -> A entidade Entidade de código x não pôde ser encontrada.  
 */  
public enum ErrorMessage {  
    4 usages  
    ENTIDADE_NOT_FOUND( mensagem: "A entidade %s de código %d não pôde ser encontrada"),  
    1 usage  
    ENTIDADE_EM_USO( mensagem: "A entidade %s de código %d não pôde ser excluída, pois está em uso");  
}
```

Construção da mensagem de erro por constantes para `String.format()`.

Ao termos um único ponto de interceptação de exceptions, mapeamos na classe interceptadora um método que intercepta exceções do tipo `EntidadeEmUsoException` com `@ExceptionHandler`

```
@ControllerAdvice  
public class ApiExceptionHandler {
```

```
@ExceptionHandler(EntidadeEmUsoException.class)
public ResponseEntity<?> handleEntidadeEmUsoException(EntidadeEmUsoException e){
    Error error = Error.builder()
        .dataHora(LocalDateTime.now())
        .mensagem(e.getMessage())
        .build();
    return ResponseEntity.status(HttpStatus.CONFLICT).body(error);
}
```

## 8.15. Criando um exception handler global com ResponseEntityExceptionHandler

domingo, 26 de fevereiro de 2023

19:43

Temos várias exceções internas do SpringMVC como

```
@ExceptionHandler(HttpMediaTypeNotSupportedException.class)
public ResponseEntity<?> handleHttpMediaTypeNotSupportedException(){
    Error error = Error.builder()
        .dataHora(LocalDateTime.now())
        .mensagem("O tipo de mídia não é aceito.")
        .build();
    return ResponseEntity.status(HttpStatus.UNSUPPORTED_MEDIA_TYPE).body(error);
}
```

e não precisamos tratá-las pois podemos estender uma classe que faz isso **ResponseEntityExceptionHandler**. É uma classe de conveniência para exceções internas globais, como a `HttpMediaTypeNotSupportedException`.

```
@ControllerAdvice
public class ApiExceptionHandler extends ResponseEntityExceptionHandler {
```

e depois de estender a classe, temos um erro de ambiguidade pois a super classe já trata a exceção interna

A super classe possui um método que intercepta uma lista de exceções

```
@ExceptionHandler({
    HttpRequestMethodNotSupportedException.class,
    HttpMediaTypeNotSupportedException.class,
    HttpMediaTypeNotAcceptableException.class,
    MissingPathVariableException.class,
    MissingServletRequestParameterException.class,
    ServletRequestBindingException.class,
    ConversionNotSupportedException.class,
    TypeMismatchException.class,
    HttpMessageNotReadableException.class,
    HttpMessageNotWritableException.class,
    MethodArgumentNotValidException.class,
    MissingServletRequestPartException.class,
    BindException.class,
    NoHandlerFoundException.class,
    AsyncRequestTimeoutException.class
})
@Nullable
public final ResponseEntity<Object> handleException(Exception ex, WebRequest request) throws Exception {
    HttpHeaders headers = new HttpHeaders();
```

e relança para o método

```
else if (ex instanceof HttpMediaTypeNotSupportedException) {
    HttpStatus status = HttpStatus.UNSUPPORTED_MEDIA_TYPE;
    return handleHttpMediaTypeNotSupported((HttpMediaTypeNotSupportedException) ex, headers, status, request);
}
```

que relança para o método `handleExceptionInternal`



```

protected ResponseEntity<Object> handleHttpMediaTypeNotSupported(
    HttpMediaTypeNotSupportedException ex, HttpHeaders headers, HttpStatus status, WebRequest request) {

    List<MediaType> mediaTypes = ex.getSupportedMediaTypes();
    if (!CollectionUtils.isEmpty(mediaTypes)) {
        headers.setAccept(mediaTypes);
        if (request instanceof ServletWebRequest) {
            ServletWebRequest servletWebRequest = (ServletWebRequest) request;
            if (HttpMethod.PATCH.equals(servletWebRequest.getHttpMethod())) {
                headers.setAcceptPatch(mediaTypes);
            }
        }
    }

    return handleExceptionInternal(ex, body: null, headers, status, request);
}

```

nota: A maioria do corpo é nullo

```

protected ResponseEntity<Object> handleExceptionInternal(
    Exception ex, @Nullable Object body, HttpHeaders headers, HttpStatus status, WebRequest request) {

    if (HttpStatus.INTERNAL_SERVER_ERROR.equals(status)) {
        request.setAttribute(WebUtils.ERROR_EXCEPTION_ATTRIBUTE, ex, WebRequest.SCOPE_REQUEST);
    }

    return new ResponseEntity<>(body, headers, status);
}

```

que retorna uma instância de ResponseEntity

Podemos sobrescrever todos esses métodos e tratarmos da nossa maneira pois as classes são protected

## 8.16. Customizando o corpo da resposta padrão de ResponseEntityExceptionHandler

domingo, 26 de fevereiro de 2023 20:06

Como vimos na aula [8.15. Criando um exception handler global com ResponseEntityExceptionHandler](#) temos uma classe base conveniente para classes @ControllerAdvice que desejam fornecer manipulação de exceção centralizada em todos os métodos @RequestMapping por meio de métodos @ExceptionHandler. Essa classe base fornece um método @ExceptionHandler para lidar com exceções internas do Spring MVC.

Podemos sobrescrever quaisquer métodos da classe e tratar especificamente cada resposta, porém, a própria classe oferece um único ponto de manipulação de exceções depois de todas as classes tratadas

```
1 usage
protected ResponseEntity<Object> handleHttpMessageNotWritable(
    HttpMessageNotWritableException ex, HttpHeaders headers, HttpStatus status, WebRequest request) {

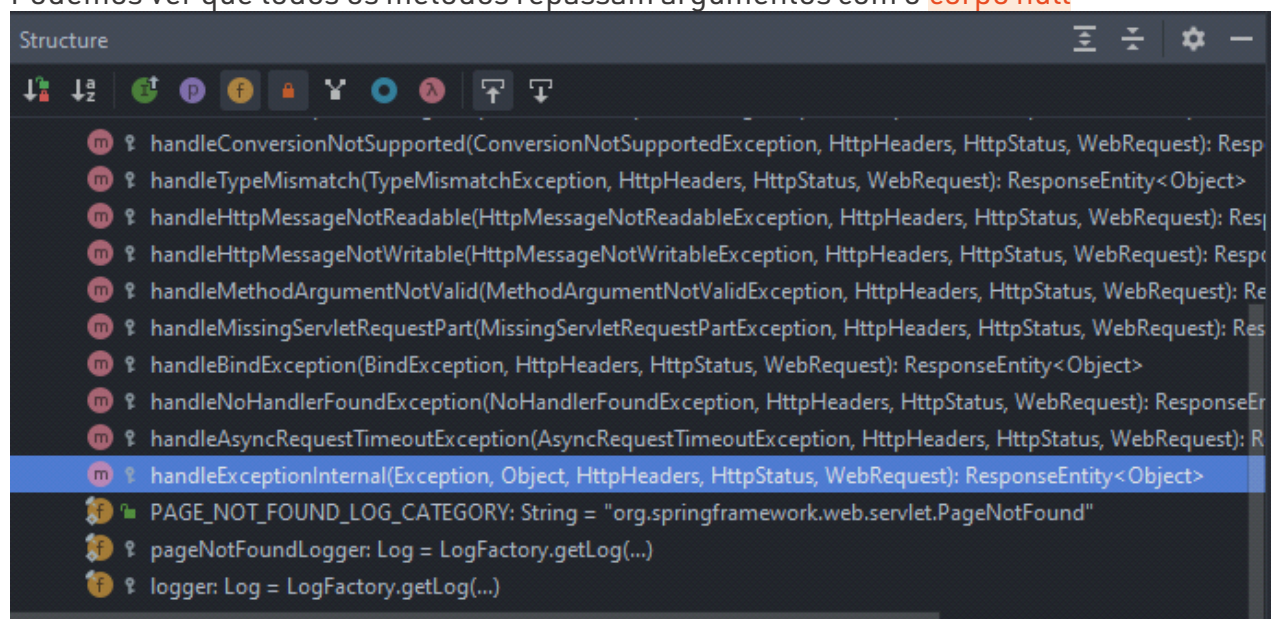
    return handleExceptionInternal(ex, body: null, headers, status, request);
}

Customize the response for MethodArgumentNotValidException.
This method delegates to handleExceptionInternal.
Params: ex – the exception
        headers – the headers to be written to the response
        status – the selected response status
        request – the current request
Returns: a ResponseEntity instance

1 usage
protected ResponseEntity<Object> handleMethodArgumentNotValid(
    MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request) {

    return handleExceptionInternal(ex, body: null, headers, status, request);
}
```

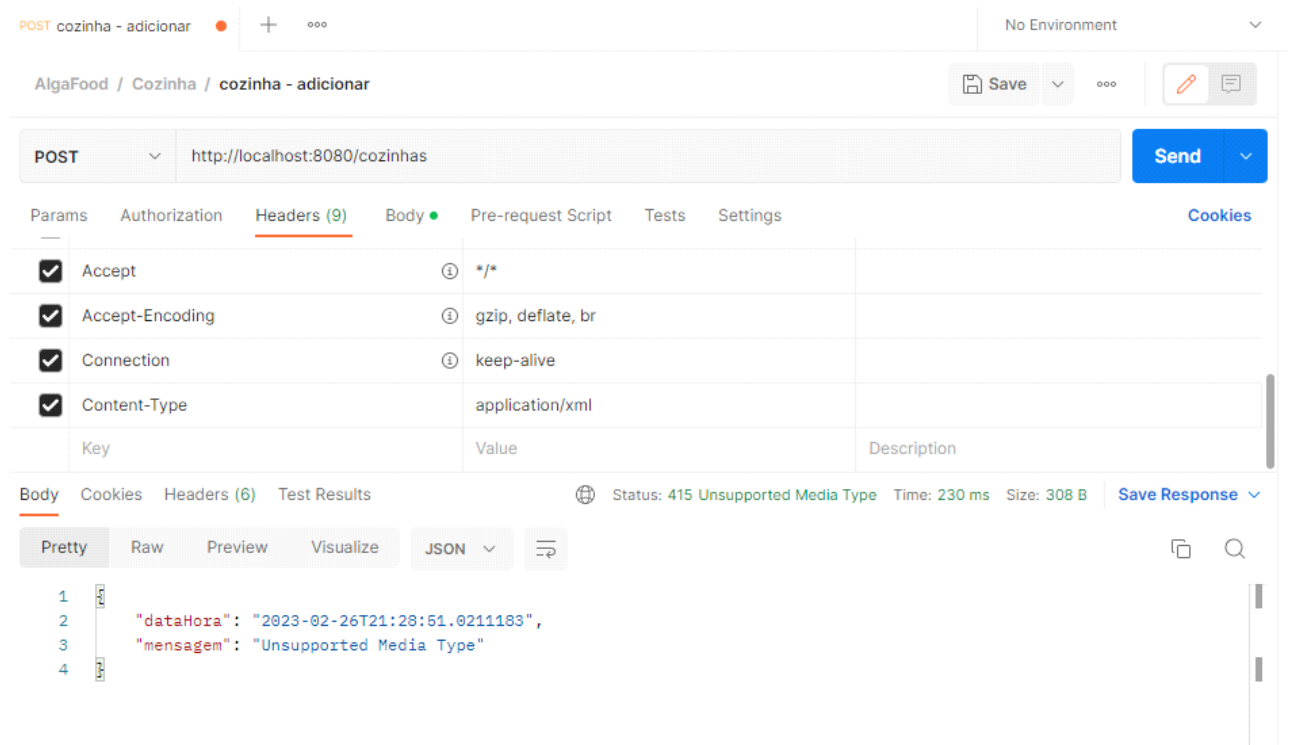
Podemos ver que todos os métodos repassam argumentos com o **corpo null**



e chega em handleExceptionInternal com corpo null, mas podemos sobrescrever **handleExceptionInternal** e adicionar um corpo para todas as exceptions que passam por

ela.

Vamos forçar um erro de Unsupported Media Type



AlgaFood / Cozinha / cozinha - adicionar

POST http://localhost:8080/cozinhas

Send

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

Key	Value	Description
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
Content-Type	application/xml	

Body Cookies Headers (6) Test Results

Status: 415 Unsupported Media Type Time: 230 ms Size: 308 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "dataHora": "2023-02-26T21:28:51.0211183",
3   "mensagem": "Unsupported Media Type"
4 }
```

```
ApiExceptionHandler.java
46     return ResponseEntity.status(HttpStatus.CONFLICT).body(error);
47 }
48
49 31 usages
50 @Override
51 protected ResponseEntity<Object> handleExceptionInternal(Exception ex, Object body, HttpHeaders headers,
52                                                         HttpStatus status, WebRequest request) {
53     body = Error.builder()
54         .dataHora(LocalDate.now())
55         .mensagem(status.getReasonPhrase())
56         .build();
57     return super.handleExceptionInternal(ex, body, headers, status, request);
58 }
59 }
60 }
```

Na classe `ApiExceptionHandler`, que está estendendo `ResponseEntityExceptionHandler` sobrescrevemos `handleExceptionInternal` e repassamos para o método da superclasse algumas modificações na representação (`body`) da requisição

Com isso, podemos utilizar o próprio método sobrescrito nos nossos métodos

```

@ControllerAdvice
public class ApiExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(EntidadeNaoEncontradaException.class) /*também subclasses*/
    public ResponseEntity<?> handleEntidadeNaoEncontradaException(EntidadeNaoEncontradaException e) {

        Error error = Error.builder()
            .dataHora(LocalDateTime.now())
            .mensagem(e.getMessage())
            .build();

        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
    }

    @ExceptionHandler(NegocioException.class) /*também subclasses*/
    public ResponseEntity<?> handleNegocioException(NegocioException e) {
        Error error = Error.builder()
            .dataHora(LocalDateTime.now())
            .mensagem(e.getMessage())
            .build();

        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(error);
    }

    @ExceptionHandler(EntidadeEmUsoException.class)
    public ResponseEntity<?> handleEntidadeEmUsoException(EntidadeEmUsoException e){
        Error error = Error.builder()
            .dataHora(LocalDateTime.now())
            .mensagem(e.getMessage())
            .build();

        return ResponseEntity.status(HttpStatus.CONFLICT).body(error);
    }
}

```

Por exemplo em EntidadeNaoEncontradaException

```

@ExceptionHandler(EntidadeNaoEncontradaException.class) /*também subclasses*/
public ResponseEntity<?> handleEntidadeNaoEncontradaException(EntidadeNaoEncontradaException ex, WebRequest request) {

    return handleExceptionInternal(ex, ex.getMessage(), new HttpHeaders(), HttpStatus.NOT_FOUND, request);
}

```

- 1º argumento - ex: classe de exceção: Exception:ex
- 2º argumento - ex.getMessage: Mensagem da exceção - Object:body
- 3º argumento - new HttpHeaders: cabeçalho da requisição - HttpHeaders:headers
- 4º argumento - HttpStatus.NOT\_FOUND: status da exceção - HttpStatus:status
- 5º argumento - request: requisição da web - WebRequest:request

chamamos `handleExceptionInternal` e passamos os argumentos. O argumento `request` conseguimos pois adicionamos no parâmetro do método uma instância de `WebRequest`, e o Spring se encarrega de atribuir a instância do `WebRequest` ao argumento do método.

```

6 @ControllerAdvice
7 public class ApiExceptionHandler extends ResponseEntityExceptionHandler {
8
9     @ExceptionHandler(EntidadeNaoEncontradaException.class) /*também subclasses*/
10    public ResponseEntity<> handleEntidadeNaoEncontradaException(EntidadeNaoEncontradaException ex, WebRequest request) {
11        return handleExceptionInternal(ex, ex.getMessage(), new HttpHeaders(), HttpStatus.NOT_FOUND, request);
12    }
13
14    @ExceptionHandler(NegocioException.class) /*também subclasses*/
15    public ResponseEntity<> handleNegocioException(NegocioException ex, WebRequest request) {
16        return handleExceptionInternal(ex, ex.getMessage(), new HttpHeaders(), HttpStatus.BAD_REQUEST, request);
17    }
18
19    @ExceptionHandler(EntidadeEmUsoException.class)
20    public ResponseEntity<> handleEntidadeEmUsoException(EntidadeEmUsoException ex, WebRequest request){
21        return handleExceptionInternal(ex, ex.getMessage(), new HttpHeaders(), HttpStatus.CONFLICT, request);
22    }
23
24    34 usages
25    @Override
26    protected ResponseEntity<Object> handleExceptionInternal(Exception ex, Object body, HttpHeaders headers,
27        HttpStatus status, WebRequest request) {
28
29        body = Error.builder()
30            .dataHora(LocalDateTime.now())
31            .mensagem(status.getReasonPhrase())
32            .build();
33        return super.handleExceptionInternal(ex, body, headers, status, request);
34    }

```

```

@Override
protected ResponseEntity<Object> handleExceptionInternal(Exception ex, Object body, HttpHeaders headers,
    HttpStatus status, WebRequest request) {

    body = Error.builder()
        .dataHora(LocalDateTime.now())
        .mensagem(body instanceof String ? (String) body: status.getReasonPhrase())
        .build();

    return super.handleExceptionInternal(ex, body, headers, status, request);
}

```

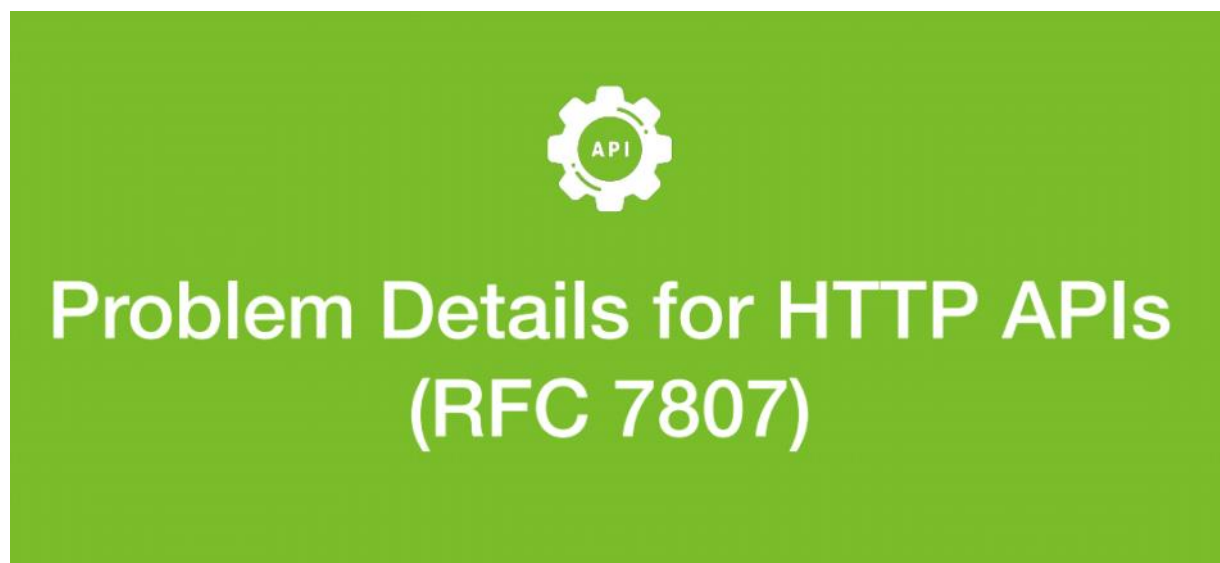
há o tratamento ao passar a mensagem no builder de Error pois se o body do argumento ser uma String que veio dos nossos próprios métodos, vai ser atribuído a string (ex.getMessage()) se a chamada for de algum método gerenciado pelo Spring que passa um corpo null, vai ser atribuído uma mensagem genérica que representa o status



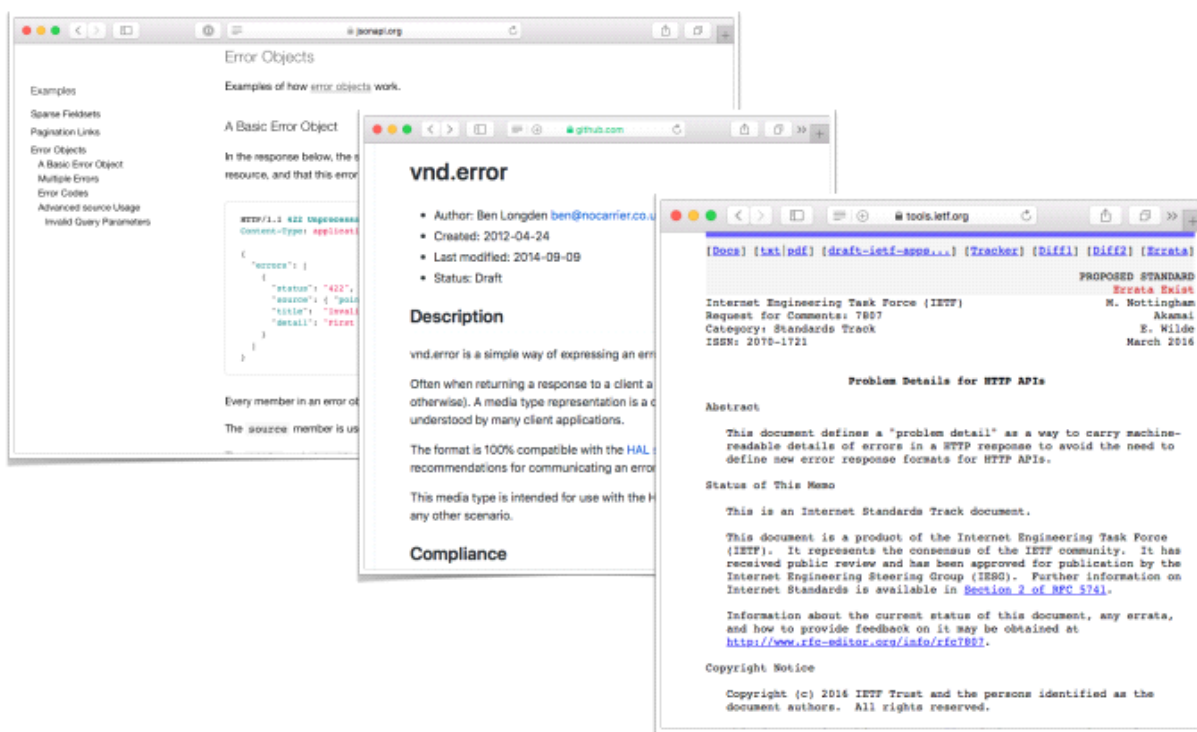
## 8.17. Conhecendo a RFC 7807 (Problem Details for HTTP APIs)

domingo, 26 de fevereiro de 2023

22:44



É necessário responder a requisição web uma mensagem de erro caso aconteça, com o status http mais adequado e consistente, mas muitas vezes não é o bastante para que o consumidor da api saiba de todas as informações sobre o problema.



Podemos adicionar um corpo e propriedades no corpo do erro indicando medidas para o que fazer ou informações detalhadas sobre o erro. Mas existem especificações que padronizam mensagens de erros como vnd.error, json error ou a Problem Details for HTTP APIs. a PDHAPIS é especificada pela RFC 7807 pela Internet Engineering Task Force (IETF)

## Exemplo de corpo de resposta com erro, usando Problem Details for HTTP APIs

```
{
  "status": 400,
  "type": "https://algafood.com.br/recurso-em-uso",
  "title": "Recurso em uso",
  "detail": "Não foi possível excluir a cozinha de código 8, porque ela está em uso",
  "instance": "/cozinhas/8/erros/98204983"
}
```

## Estendendo o formato do problema

```
{
  "status": 400,
  "type": "https://algafood.com.br/recurso-em-uso",
  "title": "Recurso em uso",
  "detail": "Não foi possível excluir a cozinha de código 8, porque ela está em uso",
  "timestamp": "2019-09-28T00:36:18",
  "cozinha": 8
}
```

**Todas as respostas com erros precisam ter um corpo descrevendo o problema?**

```
{
  "status": 415,
  "title": "Unsupported Media Type",
}
```



## 8.18. Padronizando o formato de problemas no corpo de respostas com a RFC 7807

domingo, 26 de fevereiro de 2023

23:08

Nossa classe que representa o corpo do erro na resposta de acordo com a RFC 7807

```
@Getter
@Builder
public class Problem {

    private Integer status;
    private String type;
    private String title;
    private String detail;
}
```

handleExceptionInternal atual

```
@Override
protected ResponseEntity<Object> handleExceptionInternal(Exception ex, Object body, HttpHeaders headers,
    HttpStatus status, WebRequest request) {

    body = Problem.builder()
        .title(body instanceof String ? (String) body: status.getReasonPhrase())
        .status(status.value())
        .build();

    return super.handleExceptionInternal(ex, body, headers, status, request);
}
```

PUT http://localhost:8080/restaurantes/15

Send

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

Key	Value	Description
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
Content-Type	application/xml	

Body Cookies Headers (6) Test Results

Status: 415 Unsupported Media Type Time: 217 ms Size: 303 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "status": 415,
3   "type": null,
4   "title": "Unsupported Media Type",
5   "detail": null
6 }
```

Ao passar apenas algumas propriedades para o corpo da mensagem de erro, a representação Json mostra propriedades nulas, podemos mudar isso anotando método por método ou toda a classe especificando que irá serializar apenas propriedades não nulas com `@JsonInclude(JsonInclude.Include.NON_NULL)`.

```

@Getter
@Builder
@JsonInclude(JsonInclude.Include.NON_NULL)
public class Problem {

    private Integer status;
    private String type;
    private String title;
    private String detail;
}

```

Para implementar a RFC 7807, temos que ter em mente que as exceções internas não possuem informações extras como detail e type (a menos que sobrescrevemos os métodos) mas nossas exceções possuem e podemos construir a mensagem de erro, como:

```

@ExceptionHandler(EntidadeNaoEncontradaException.class) /*também subclasses*/
public ResponseEntity<> handleEntidadeNaoEncontradaException(EntidadeNaoEncontradaException ex, WebRequest request) {

    HttpStatus statusNotFound = HttpStatus.NOT_FOUND;
    Problem problem = Problem.builder()
        .status(statusNotFound.value())
        .type("http://localhost:8080/entidade-nao-encontrada")
        .title("Entidade não encontrada")
        .detail(ex.getMessage())
        .build();

    return handleExceptionInternal(ex, problem, new HttpHeaders(), statusNotFound, request);
}

```

e repassamos o objeto **problem** com a mensagem construída baseada na RFC 7807

AlgaFood / Restaurante / Restaurante - atualizar

PUT http://localhost:8080/restaurantes/15 Send

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
5   "id": 1
6 }

```

Body Cookies Headers (5) Test Results Status: 404 Not Found Time: 22 ms Size: 349 B Save Response

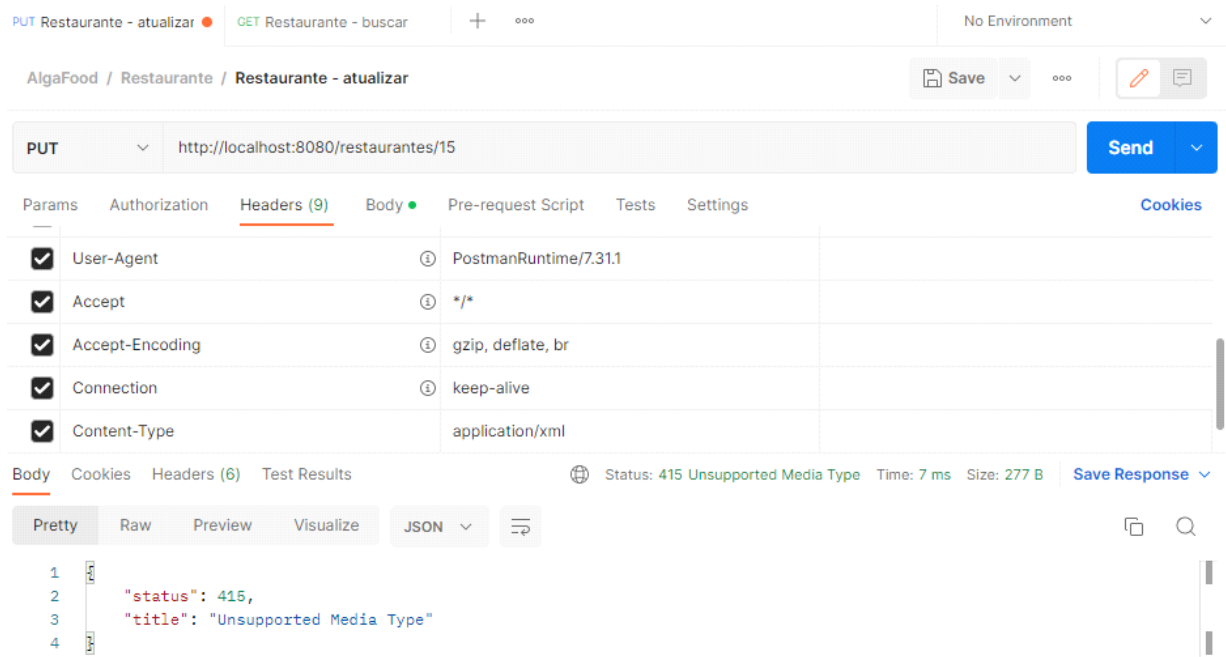
Pretty Raw Preview Visualize JSON Copy

```

1 {
2   "status": 404,
3   "type": "http://localhost:8080/entidade-nao-encontrada",
4   "title": "Entidade não encontrada",
5   "detail": "A entidade Restaurante de código 15 não pôde ser encontrada"
6 }

```

Ou com exceções internas



Nós podemos melhorar o código criando um método que retorna um builder de **Problem**, pois o código está muito exposto.

```

4 @Usage
5 private Problem.ProblemBuilder createProblemType (HttpStatus status, ProblemType problemType, String detail){
6
7     return Problem.builder()
8         .status(status.value())
9         .type(problemType.getUri())
10        .title(problemType.getTitle())
11        .detail(detail);
12 }

```

Ao final da classe, criamos um método para encapsular a criação do Problem nos métodos **handle**

método criado sendo chamado

```

@ExceptionHandler(EntidadeNaoEncontradaException.class) /*também subclasses*/
public ResponseEntity<> handleEntidadeNaoEncontradaException(EntidadeNaoEncontradaException ex, WebRequest request) {

    HttpStatus statusNotFound = HttpStatus.NOT_FOUND;
    ProblemType problemType = ProblemType.ENTIDADE_NAO_ENCONTRADA;
    final Problem problem2 = createProblemType(statusNotFound, problemType, ex.getMessage()).build();
    Problem problem = Problem.builder()
        .status(statusNotFound.value())
        .type("http://localhost:8080/entidade-nao-encontrada")
        .title("Entidade não encontrada")
        .detail(ex.getMessage())
        .build();

    return handleExceptionInternal(ex, problem2, new HttpHeaders(), statusNotFound, request);
}

```

a classe `handleExceptionInternal` sempre é chamada por todas as exceções internas e personalizadas por nós

```

@Override
protected ResponseEntity<Object> handleExceptionInternal(Exception ex, Object body, HttpHeaders headers,
                                                         HttpStatus status, WebRequest request) {

    /**
     * se o (Object) body não for nullo, ele é do tipo Problem (Especificação RFC 7807) e não é necessário
     * criar um novo objeto.
     * Se ele for nullo, a exceção foi tratada pelo SpringMVC interno, o corpo da exceção vem nulla.
     * Se ele for apenas uma String também um problem vai ser instanciado e construído a partir da RFC 7807
     */
    if(body == null || body instanceof String) {
        body = Problem.builder()
            .title(body instanceof String ? (String) body : status.getReasonPhrase())
            .status(status.value())
            .build();
    }
    return super.handleExceptionInternal(ex, body, headers, status, request);
}

```

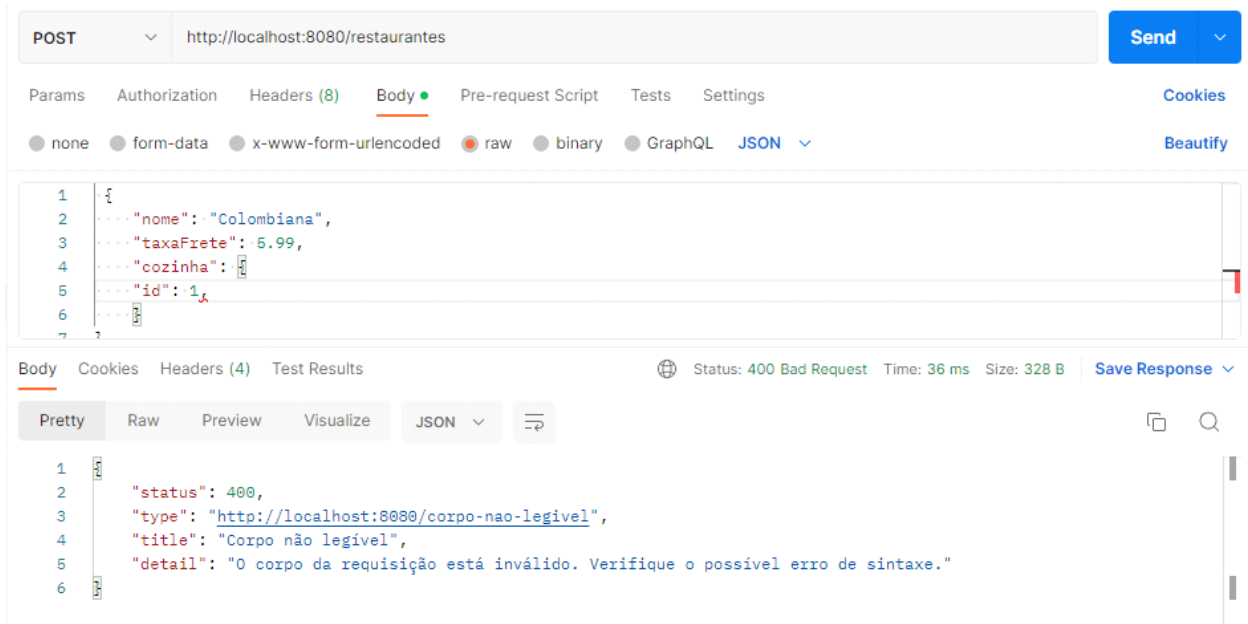
## 8.19. Desafio usando o formato de problemas no corpo de respostas

segunda-feira, 27 de fevereiro de 2023 15:58

## 8.20. Customizando exception handlers de ResponseEntityExceptionHandler

segunda-feira, 27 de fevereiro de 2023 16:43

A classe `ResponseEntityExceptionHandler` intercepta exceções internas do SpringMVC e trata internamente implementando métodos para vários tipos de exceção, mas podemos customizar um desses métodos, o `handleHttpMessageNotReadableException` que trata exceções do nome do método. Podemos simular a exceção passando um corpo não legível no corpo da requisição:



e retornando uma mensagem da exceção com o corpo do erro no padrão RFC 7807. Primeiros sobrescrevemos o método da superclasse

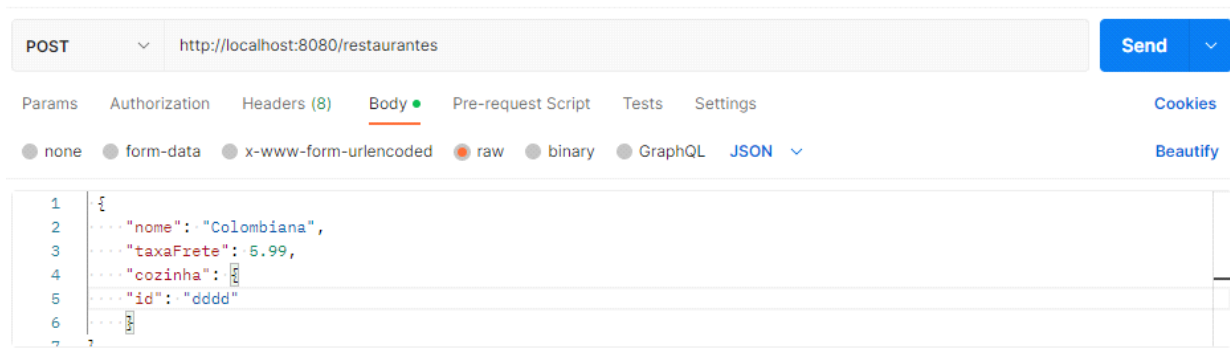
```
/**
 * Algumas variáveis do parâmetro não precisam ser definidas pois a superclasse já atribuiu valores
 * @param ex the exception
 * @param headers the headers to be written to the response
 * @param status the selected response status.
 * @param request the current request
 * @return
 */
@Override
protected ResponseEntity<Object> handleHttpMessageNotReadable(HttpMessageNotReadableException ex, HttpHeaders headers,
                                                                HttpStatus status, WebRequest request) {

    ProblemType problemType = ProblemType.CORPO_NAO_LEGIVEL;
    String detail = "O corpo da requisição está inválido. Verifique o possível erro de sintaxe.";
    final Problem problem = createProblemType(status, problemType, detail).build();

    return handleExceptionInternal(ex, problem, headers, status, request);
}
```

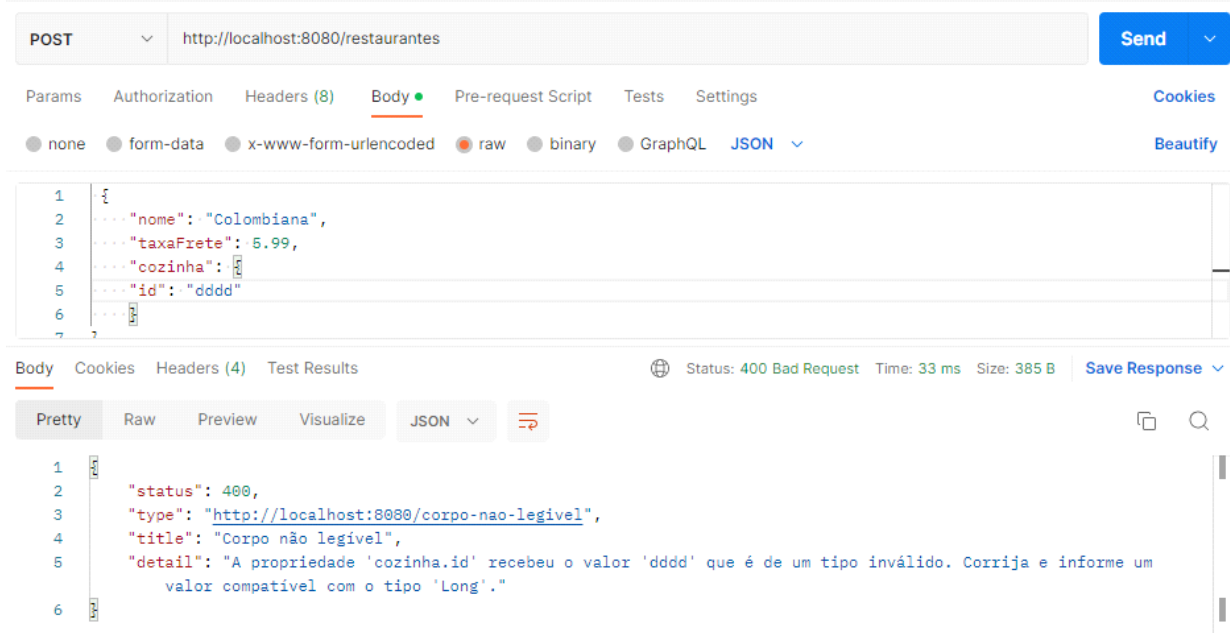
## 8.21. Tratando a exception InvalidFormatException na desserialização

terça-feira, 28 de fevereiro de 2023 23:57



A ideia é detalhar a resposta de erro quando acontece a `InvalidFormatException`, que acontece porque passamos valores inválidos a propriedades no corpo JSON e o jackson não consegue desserializar para o formato apropriado.

como a exemplo:



Primeiro é necessário adicionar uma biblioteca externa que ajuda a tratar as propriedades da exceção, pois o facilita a captura da causa raiz da exceção, isso se faz necessário porque `HttpMessageNotReadableException` pode ter como causa raiz a `InvalidFormatException`, pois cai no caso de valores passados incorretamente precisamos detalhar esses valores caso seja o erro de sintaxe dos valores.

biblioteca lang3

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

A biblioteca lang3 oferece a classe `ExceptionUtil` para trabalhar com exceções e tem



disponível o método `getRootCause` para retornar a causa raiz:

```
final Throwable rootCause = ExceptionUtils.getRootCause(ex);

if (rootCause instanceof InvalidFormatException) {
    return handleInvalidFormatException((InvalidFormatException) rootCause, headers, status, request);
}

ProblemType problemType = ProblemType.CORPO_NAO_LEGIVEL;
```

1 usage

```
private ResponseEntity<Object> handleInvalidFormatException(InvalidFormatException ex, HttpHeaders headers,
    HttpStatus status, WebRequest request) {

    ProblemType problemType = ProblemType.CORPO_NAO_LEGIVEL;

    final String path = ex.getPath().list().stream()
        .stream()
        .map(JsonMappingException.Reference::getFieldName)
        .collect(Collectors.joining(" "));

    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());

    final Problem problem = createProblemType(status, problemType, detail).build();
    return handleExceptionInternal(ex, problem, headers, status, request);
}
```

## 8.22. Habilitando erros na desserialização de propriedades inexistentes ou ignoradas

quarta-feira, 1 de março de 2023

14:24

Em `application.properties`, podemos habilitar a propriedade do jackson para lançar exceções quando o cliente enviar uma requisição com propriedades e valores desconhecidos do objeto ou quando a propriedade da entidade esteja sendo ignorada

```
32 #Habilitando erros na desserialização de propriedades inexistentes ou ignoradas
33 spring.jackson.deserialization.fail-on-unknown-properties=true
34 spring.jackson.deserialization.fail-on-ignored-properties=true
```

*#Habilitando erros na desserialização de propriedades inexistentes ou ignoradas*  
`spring.jackson.deserialization.fail-on-unknown-properties=true`  
`spring.jackson.deserialization.fail-on-ignored-properties=true`

```
@Override
protected ResponseEntity<Object> handleHttpMessageNotReadable(HttpMessageNotReadableException ex, HttpHeaders headers,
    HttpStatus status, WebRequest request) {

    final Throwable rootCause = ExceptionUtils.getRootCause(ex);

    if (rootCause instanceof InvalidFormatException) {
        return handleInvalidFormatException((InvalidFormatException) rootCause, headers, status, request);
    } else if (rootCause instanceof PropertyBindingException) {
        return handlePropertyBindingException((PropertyBindingException) rootCause, headers, status, request);
    }

    ProblemType problemType = ProblemType.CORPO_NAO_LEGIVEL;
    String detail = "O corpo da requisição está inválido. Verifique o possível erro de sintaxe.";
    final Problem problem = createProblemType(status, problemType, detail).build();

    return handleExceptionInternal(ex, problem, headers, status, request);
}
```

## 8.23. Desafio tratando a PropertyBindingException na desserialização

quarta-feira, 1 de março de 2023 20:55

## 8.24. Lançando exception de desserialização na atualização parcial (PATCH)

quarta-feira, 1 de março de 2023

22:04

Em uma atualização parcial, nós configuramos um ObjectMapper para trabalhar com a desserialização das propriedades manualmente e não o Spring Boot, logo, os efeitos da ativação das propriedades

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

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

não surtem efeitos, mas podemos fazer programaticamente.

```
ObjectMapper objectMapper = new ObjectMapper();
objectMapper.configure(DeserializationFeature.FAIL_ON_IGNORED_PROPERTIES, state: true);
objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, state: true);
```

após fazer uma requisição, o esperado seria ser capturado pelo método sobrescrito `handleHttpRequestNotReadable` na classe `ApiExceptionHandler` como foi feito em [8.22. Habilitando erros na desserialização de propriedades inexistentes ou ignoradas](#) pois entra na característica de formato do corpo json inválido, mas não é capturada pois quem lança a exceção é `IllegalArgumentException` que tem como causa `IgnoredPropertyException`.

Podemos relançar a exceção para ser capturada por `handleHttpRequestNotReadable` e ser tratada pela sua causa raiz (`IgnoredPropertyException` ou `UnrecognizedPropertyException`)

```
@Override
protected ResponseEntity<Object> handleHttpRequestNotReadable(HttpMessageNotReadableException ex, HttpHeaders headers,
                                                                HttpStatus status, WebRequest request) {

    final Throwable rootCause = ExceptionUtils.getRootCause(ex);

    if (rootCause instanceof InvalidFormatException) {
        return handleInvalidFormatException((InvalidFormatException) rootCause, headers, status, request);
    } else if (rootCause instanceof PropertyBindingException) {
        return handlePropertyBindingException((PropertyBindingException) rootCause, headers, status, request);
    }

    ProblemType problemType = ProblemType.CORPO_NAO_LEGIVEL;
    String detail = "O corpo da requisição está inválido. Verifique o possível erro de sintaxe.";
    final Problem problem = createProblemType(status, problemType, detail).build();

    return handleExceptionInternal(ex, problem, headers, status, request);
}
```

```

private static void merge(Map<String, Object> dadosOrigem, Restaurante restauranteDestino, HttpServletRequest servl
try {
    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.configure(DeserializationFeature.FAIL_ON_IGNORED_PROPERTIES, state: true);
    objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, state: true);

    final Restaurante restauranteOrigem = objectMapper.convertValue(dadosOrigem, Restaurante.class);

    dadosOrigem.forEach((nomePropriedade, valorPropriedade) -> {
        Field field = ReflectionUtils.findField(Restaurante.class, nomePropriedade);
        field.setAccessible(true);

        final Object novoValorConvertido = ReflectionUtils.getField(field, restauranteOrigem);

        ReflectionUtils.setField(field, restauranteDestino, novoValorConvertido);
    });
} catch (IllegalArgumentException ex){
    final Throwable rootCause = ExceptionUtils.getRootCause(ex);

    final ServletServerHttpRequest servletServerHttpRequest = new ServletServerHttpRequest(servletRequest);
    throw new HttpMessageNotReadableException(ex.getMessage(), rootCause, servletServerHttpRequest );
}

```

Capturamos a exceção `IllegalArgumentException` e relançamos a exceção `HttpMessageNotReadableException` com seus argumentos.

A exceção recebe a mensagem de erro, a causa raiz e um `ServletServerHttpRequest`

A causa raiz da exceção `IllegalArgumentException` foi `IgnoredPropertyException` (subclasse de `PropertyBindingException`) e para ser tratada como um erro de sintaxe, uma exceção do tipo `HttpMessageNotReadableException` com a causa raiz `IgnoredPropertyException` ou `UnrecognizedPropertyException` deve ser lançada (as duas classes são subclasses de `PropertyBindingException`)

## 8.25. Desafio tratando exception de parâmetro de URL inválido

quarta-feira, 1 de março de 2023 23:53

```
ProblemType problemType = ProblemType.PARAMETRO_INVALIDO;
```

```
O parâmetro de URL '%s' recebeu o valor '%s', que é de um tipo inválido.  
Corrija e informe um valor compatível com o tipo %s.
```

## 8.26. Desafio tratando a exceção NoHandlerFoundException

quinta-feira, 2 de março de 2023 09:46

Desafio: Tratar exceções para recursos inexistentes na aplicação passado na URI ex: /fornecedores/2 no qual não existe mapeamento para fornecedores.

edit: é diferente de um erro de sintaxe pois não estamos passando nada pelo corpo.

AlgaFood / Restaurante / Restaurante - buscar

GET http://localhost:8080/restaurante/z Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Key	Value	Description
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
Content-Type	application/xml	

Body Cookies Headers (8) Test Results 404 Not Found 42 ms 395 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "timestamp": "2023-03-02T13:22:35.780+00:00",
3   "status": 404,
4   "error": "Not Found",
5   "message": "No message available",
6   "path": "/restaurante/z"
7 }
```

a exceção em particular não é capturada e referenciada na pilha de exceção por conta, pois a configuração automática do Spring Boot adicionará automaticamente um `ResourceHttpRequestHandler` recurso estático para lidar com o serviço. Por padrão, esse manipulador é mapeado `/**` e é o último item na cadeia do manipulador.

O que isso significa é que `DispatcherServlet` não lançará um `NoHandlerFoundException` (que é lançado mas não mostrado na stacktrace) porque encontrou o manipulador de recursos. O manipulador de recursos processa a solicitação e chama `response.sendError(HttpServletResponse.SC_NOT_FOUND)` para retornar um 404. Para contornar esse comportamento, basta adicionar propriedades no `application.properties`:

```
#8.26. Desafio tratando a exceção NoHandlerFoundException
#Habilitar o lançamento da exceção e e informar ao spring boot p/ não registrar o manipulador
spring.mvc.throw-exception-if-no-handler-found=true
spring.web.resources.add-mappings=false
```

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

`spring.web.resources.add-mappings=false`

<https://stackoverflow.com/questions/51048707/spring-boot-handling-nohandlerfoundexception>

```

@Override
protected ResponseEntity<Object> handleNoHandlerFoundException(NoHandlerFoundException ex, HttpHeaders headers,
                                                                HttpStatus status, WebRequest request) {

    final ProblemType problemType = ProblemType.RECURSO_NAO_ENCONTRADO;
    final String requestURL = ex.getRequestURL();
    String detail = String.format("O recurso '%s' é inexistente", requestURL);
    final Problem problema = createProblemType(status, problemType, detail).build();

    return handleExceptionInternal(ex, problema, headers, status, request);
}

```

GET Restaurante - buscar + ... No Environment

AlgaFood / Restaurante / Restaurante - buscar Save ...

GET http://localhost:8080/restaurante/1 Send

Params	Authorization	Headers (7)	Body	Pre-request Script	Tests	Settings	Cookies
<input checked="" type="checkbox"/>	Accept		*/*				
<input checked="" type="checkbox"/>	Accept-Encoding		gzip, deflate, br				
<input checked="" type="checkbox"/>	Connection		keep-alive				
<input type="checkbox"/>	Content-Type		application/xml				
	Key		Value				Description

Body Cookies Headers (5) Test Results 404 Not Found 20 ms 325 B Save Response

Pretty Raw Preview Visualize JSON

```

1 {
2   "status": 404,
3   "type": "http://localhost:8080/recuso-nao-encontrado",
4   "title": "Recurso não encontrado",
5   "detail": "O recurso '/restaurante/1' é inexistente"
6 }

```



## 8.27. Desafio tratando outras exceções não capturadas

quinta-feira, 2 de março de 2023 13:36

Desafio: Tratar as exceções não capturada da aplicação (fora do contexto SpringMVC) e deixar no formato da RFC 7807

```
| ProblemType.ERRO_DE_SISTEMA
```

```
I
```

```
"Ocorreu um erro interno inesperado no sistema. Tente novamente e se o "  
+ "problema persistir, entre em contato com o administrador do sistema."
```

## 8.28. Estendendo o formato do problema para adicionar novas propriedades

quinta-feira, 2 de março de 2023

14:31

Podemos adicionar mais atributos no corpo da resposta da exceção para facilitar o consumidor da api na informação e interpretação do problema.

```
12 usages
@Getter
@Builder
@JsonInclude(JsonInclude.Include.NON_NULL)
public class Problem {

    private Integer status;
    private String type;
    private String title;
    private String detail;
    private String userMessage;
}
```

A vantagem em utilizar o padrão Builder na classe é pela possibilidade de extensão de atributos sem quebrar o código, podendo permanecer os mesmos atributos ou adicionar o atributo em diferentes partes do sistema.

AlgaFood / Restaurante / Restaurante - atualizar

PUT http://localhost:8080/restaurantes/11

Send

Params Auth Headers (9) Body Pre-req. Tests Settings

raw JSON

```
1 {
2   .... "nome" : "Norddesa",
3   .... "taxaFrete" : 99.90,
4   .... "cozinha" : {
5     .... "id" : "AA"
6   }
7 }
```

Body 400 Bad Request 208 ms 538 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.id' recebeu o valor 'AA' que é de um tipo
6     inválido. Corrija e informe um valor compatível com o tipo 'Long'.",
7   "userMessage": "Ocorreu um erro interno inesperado no sistema. Tente novamente e
8     se o proplema persistir, entre em contato com o administrador do sistema."
9 }
```

## 8.29. Desafio estendendo o formato do problema

quinta-feira, 2 de março de 2023

14:59

Como foi visto em [8.28. Estendendo o formato do problema para adicionar novas propriedades](#) podemos estender a especificação, e o desafio da aula é adicionar mais propriedades

The screenshot shows a REST client interface with a PUT request to `http://localhost:8080/restaurantes/11`. The request body is a JSON object with the following properties:

```
{
  "nome": "Noirdesa",
  "taxaFrete": 99.90,
  "cozinha": 9,
  "id": 9
}
```

The response is a 400 Bad Request with the following JSON body:

```
{
  "status": 400,
  "type": "http://localhost:8080/erro-negocio",
  "title": "Violação de regra de negócio",
  "detail": "A entidade Cozinha de código 9 não pôde ser encontrada",
  "userMessage": "A entidade Cozinha de código 9 não pôde ser encontrada",
  "localDateTime": "2023-03-02T17:34:28.3417825"
}
```

The response status is 400 Bad Request, with a time of 69 ms and a size of 433 B. The response is displayed in the 'Body' tab, formatted as JSON.