

CENTRO UNIVERSITÁRIO FACENS
CURSOS TECNOLÓGICOS



Desenvolvimento Web Back-End

Próximas Datas

- **08/05:** Matéria + Exercício + avaliação do back-end do Projeto Final
- **15/05:** Matéria + Exercício + avaliação do back-end do Projeto Final
- **20/05:** Avaliação Geral (AG)
- **22/05:** Matéria e exercícios + avaliação do back-end do Projeto Final
- **29/05:** Matéria + **considerações finais para AF e AG**
- **05/06:** Apresentação do Projeto de Startup Final (não haverá aula)
- **12/06:** Avaliação Final (AF)
- **19/06:** Feriado
- **26/06:** Avaliação Substitutiva

AVALIAÇÃO DA CPA

ATENÇÃO, A AVALIAÇÃO DA CPA COMEÇOU!

**CHEGOU A HORA DE COLHER IDEIAS
PARA TRANSFORMAR EM RESULTADOS**


RESPONDA ATRAVÉS DO QR CODE:



The illustration shows a hand holding a pen over a bar chart with three bars of increasing height. To the right of the chart is a large QR code. Further right is a cartoon robot wearing a blue cap and holding a blue clipboard with the CPA logo. The background is dark blue with some faint geometric shapes.

 **CPA**
Comissão Própria de Avaliação

? ****Por que sua participação é importante?****

- ✓ Porque ****ajuda a identificar pontos de melhoria**** nos cursos, nas estruturas e nos serviços.
 - ✓ Porque é por meio da CPA que ****suas opiniões chegam diretamente à gestão acadêmica****.
 - ✓ Porque ****uma instituição só melhora com a escuta ativa dos seus estudantes****!
-  ****Sua voz é importante!**** Vamos construir juntos uma ****Facens ainda melhor****.

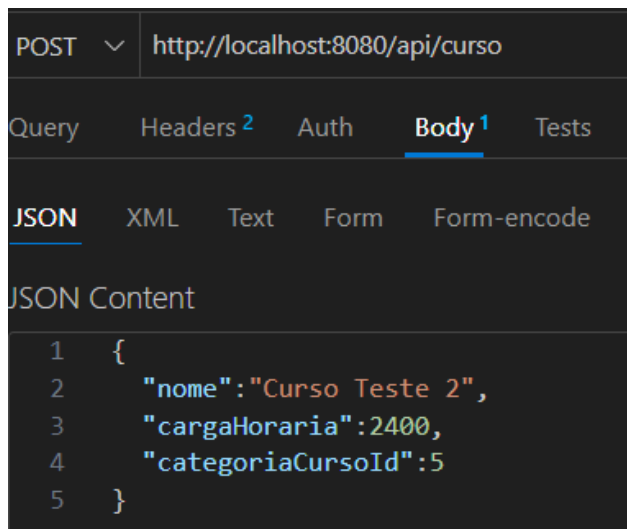
Arquitetura Web: Tratando Erros

Agenda da aula:

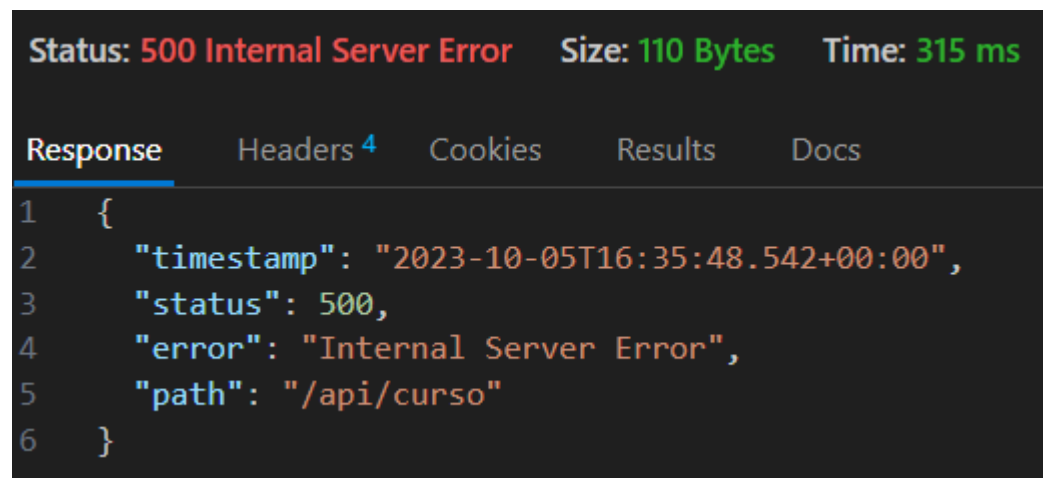
- ☐ Compreender o tratamento de erros na API;
- ☐ Compreender a anotação `@RestControllerAdvice`
- ☐ Implementar tratamento de erro com `@RestControllerAdvice`
- ☐ Compreender conceito de Builder
- ☐ Implementar Builder na construção de objetos
- ☐ Criar Método `ListarTodos()`
- ☐ Criar Método `ObterPorId()`
- ☐ Criar Método `Excluir()`
- ☐ Criar Método `Editar()`

Tratando erros na API

- Utilizando o método salvar da classe CursoService é possível verificar que quando o id da categoria não é encontrado é gerado uma exceção e o método é abortado.
- Ocorre que o erro que é apresentado na API fica incompreensível:



```
POST http://localhost:8080/api/curso
Query Headers 2 Auth Body 1 Tests
JSON XML Text Form Form-encode
JSON Content
1 {
2   "nome": "Curso Teste 2",
3   "cargaHoraria": 2400,
4   "categoriaCursoId": 5
5 }
```



```
Status: 500 Internal Server Error Size: 110 Bytes Time: 315 ms
Response Headers 4 Cookies Results Docs
1 {
2   "timestamp": "2023-10-05T16:35:48.542+00:00",
3   "status": 500,
4   "error": "Internal Server Error",
5   "path": "/api/curso"
6 }
```

Tratando erros na API

Para resolver o problema será interceptado as exceções geradas pelo sistema.

Nesse caso será verificado se o tipo da exceção é uma **RegraNegocioException** e tratado para não aparecer o erro 500 e sim um Array das exceções geradas.

@RestControllerAdvice

Para realizar a interceptação das exceções utilizaremos a anotação @RestControllerAdvice.

O @RestControllerAdvice indica que a classe é um **controlador de exceções que intercepta as exceções lançadas** por métodos anotados com @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping ou outros mapeamentos HTTP.

Passo 1: Criar a classe ApiErrorDTO

Na pasta DTOs, adicione a classe ApiErrorDTO :

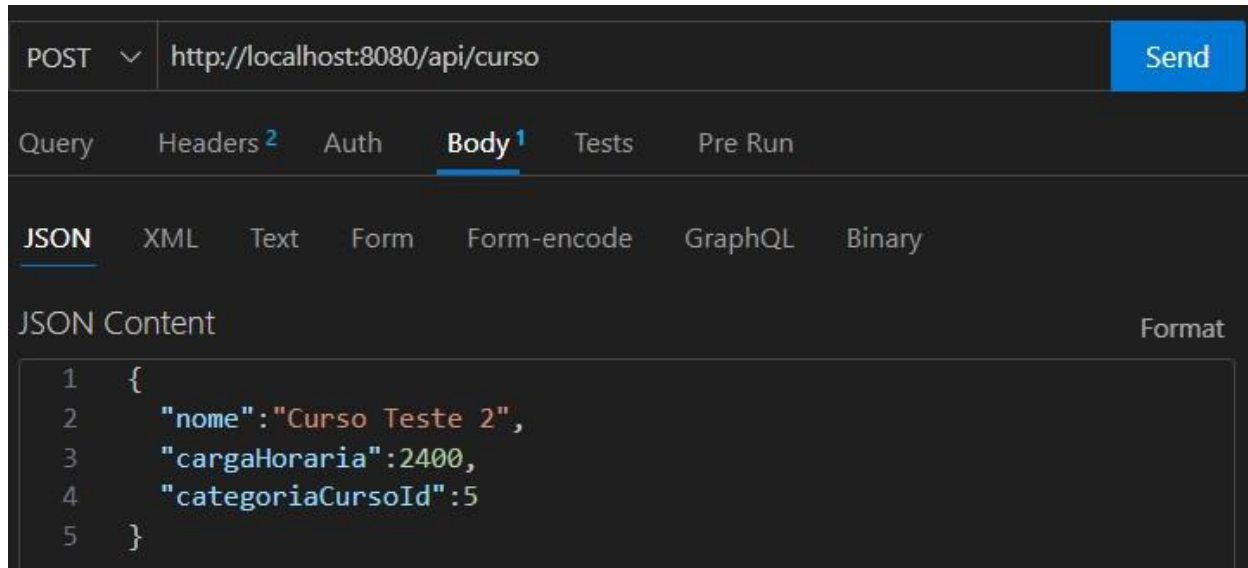
```
public class ApiErrorDTO {  
    @Getter  
    private List<String> errors;  
  
    public ApiErrorDTO(String mensagem){  
        this.errors = Arrays.asList(mensagem);  
    }  
}
```

Passo 2: Criar a classe ApplicationControllerAdvice

Crie dentro da pasta Controller **a classe: ApplicationControllerAdvice**

```
@RestControllerAdvice
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class ApplicationControllerAdvice {
    @ExceptionHandler(RegraNegocioException.class)
    public ApiErrorDTO handleRegraNegocioException(RegraNegocioException ex) {
        String msg = ex.getMessage();
        return new ApiErrorDTO(msg);
    }
}
```

Testando



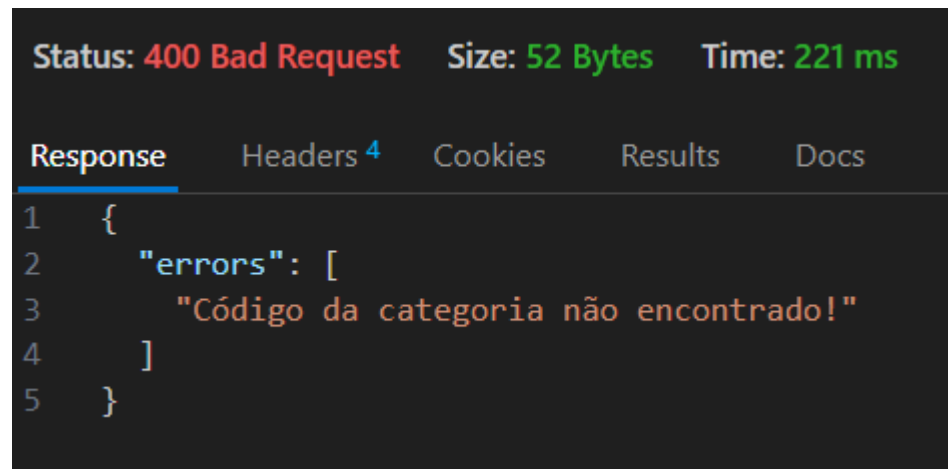
POST ▼ http://localhost:8080/api/curso Send

Query Headers ² Auth Body ¹ Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "nome": "Curso Teste 2",
3   "cargaHoraria": 2400,
4   "categoriaCursoId": 5
5 }
```



Status: 400 Bad Request Size: 52 Bytes Time: 221 ms

Response Headers ⁴ Cookies Results Docs

```
1 {
2   "errors": [
3     "Código da categoria não encontrado!"
4   ]
5 }
```

Builder

- O padrão de design Builder é **uma técnica que permite a criação de objetos complexos de maneira mais simples e intuitiva.**

Em vez de fornecer um construtor com muitos parâmetros ou vários construtores para lidar com diferentes cenários de inicialização, o padrão Builder fornece uma interface fluida para construir objetos em etapas.

A ideia central do padrão Builder é separar a construção de um objeto complexo de sua representação, de modo que o mesmo processo de construção possa criar diferentes representações do objeto. O padrão Builder utiliza um objeto Builder separado para a construção do objeto, e esse Builder tem métodos para configurar os atributos do objeto. Quando o objeto é construído, o Builder o retorna em sua representação final.

Builder

- O padrão Builder é implementado com uma classe Builder que tem o mesmo conjunto de atributos que a classe que está sendo construída, mas com métodos para definir cada um desses atributos de maneira separada.

A classe Builder retorna a instância da classe sendo construída, que pode ser criada em um único passo ou em vários passos, dependendo das configurações necessárias. O processo de construção é assim separado do processo de inicialização do objeto.

- **O uso do padrão Builder torna o código mais legível e fácil de entender, e permite a construção de objetos complexos sem a necessidade de fornecer muitos construtores ou métodos.** O padrão Builder é especialmente útil em casos em que há muitos atributos opcionais ou em que os atributos têm valores padrão diferentes.

Exemplo sem Builder e com Builder

Sem Builder

```
public List<CursoDTO> listarTodos() {  
    List<Curso> cursos = cursoRepository.findAll();  
    List<CursoDTO> cursosDTO = new ArrayList();  
    for (Curso c : cursos) {  
        cursosDTO.add(new CursoDTO(  
            c.getId(),  
            c.getNome(),  
            c.getCargaHoraria(),  
            c.getCategoriaCurso() != null ?  
            c.getCategoriaCurso().getId() : 0));  
    }  
    return cursosDTO;  
}
```

Com Builder

```
public List<CursoDTO> listarTodos() {  
    List<CursoDTO> cursos = cursoRepository.findAll().stream().map((Curso c) -> {  
        return CursoDTO.builder()  
            .id(c.getId())  
            .nome(c.getNome())  
            .cargaHoraria(c.getCargaHoraria())  
            .categoriaCursoId(c.getCategoriaCurso() != null ?  
            c.getCategoriaCurso().getId() : null)  
            .build();  
    }).collect(Collectors.toList());  
    return cursos;  
}
```

Implementando Builder com Lombok

- Na classe DadosCursoDTO e CategoriaCursoDTO acrescente a anotação **@Builder**:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class DadosCursoDTO {
    private Long id;
    private String nome;
    private Integer cargaHoraria;
    private CategoriaCursoDTO categoria;
}
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class CategoriaCursoDTO {
    private Integer id; private
    String nome;
}
```

Criar método ListarTodos

Criamos o método **listarTodos()** em uma API back-end, especialmente em aplicações Java com Spring Boot, para atender a uma **operação básica de leitura (READ)** no padrão **CRUD** (Create, Read, Update, Delete).

Passo 1: Crie uma classe DTO

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class DadosCursoDTO {
    private Long id;
    private String nome;
    private Integer cargaHoraria;
    private CategoriaCursoDTO categoria;
}
```

Passo 2: Altere a classe CursoService e CursoServiceImpl

- Adicione o código na classe CursoService
`List<DadosCursoDTO> listarTodos();`

- Adicione o código em CursoServiceImpl:

```
public List<DadosCursoDTO> listarTodos() {  
    return cursoRepository.findAll().stream().map((Curso c) -> {  
        return DadosCursoDTO.builder()  
            .id(c.getId())  
            .nome(c.getNome())  
            .cargaHoraria(c.getCargaHoraria())  
            .categoriaCurso(  
                CategoriaCursoDTO.builder()  
                    .id(c.getCategoriaCurso().getId())  
                    .nome(c.getCategoriaCurso().getNome())  
                    .build())  
            .build();  
    }).collect(Collectors.toList());
```

Passo 3: Adicionando método no Controller

```
@GetMapping("")  
public List<DadosCursoDTO> listarTodos() {  
    return cursoService.listarTodos();  
}
```

Criar método ObterPorId

Criamos o método **obterPorId()** para permitir que a API busque e retorne um **único registro** de uma entidade com base no seu identificador único (ID).

Passo 1: Altere a classe CursoService e CursoServiceImpl

- Adicione o código na classe CursoService

DadosCursoDTO ObterPorId(Long id);

- Adicione o código em CursoServiceImpl:

@Override

```
public DadosCursoDTO ObterPorId(Long id) {  
    return cursoRepository.findById(id).map((Curso c) -> {  
        return DadosCursoDTO  
            .builder()  
            .id(c.getId())  
            .nome(c.getNome())  
            .cargaHoraria(c.getCargaHoraria())  
            .categoria(CategoriaCursoDTO.builder()  
                .id(c.getCategoriaCurso().getId())  
                .nome(c.getCategoriaCurso().getNome())  
                .build())  
            .build();  
    }).orElseThrow(() -> new RegraNegocioException("Curso não encontrado com o ID fornecido"));  
}
```

Passo 2: Adicionando método no Controller

```
@GetMapping("{id}")  
public DadosCursoDTO ObterPorId(@PathVariable Long id) {  
    return cursoService.obterPorId(id);  
}
```

Criar método Excluir

Passo 1: Alterar CursoService e CursoServiceImpl

- Adicionar em CursoService:

```
void excluir(Long id);
```

- Adicionar em CursoServiceImpl:

```
@Override
```

```
@Transactional
```

```
public void excluir(Long id) {  
    cursoRepository.deleteById(id);  
}
```


Passo 2: Controller – Adicionar método Excluir

```
@DeleteMapping("{id}")  
    @ResponseStatus(HttpStatus.NO_CONTENT)  
    public void delete(@PathVariable Long id) {  
        cursoService.excluir(id);  
    }
```

Criar método Editar

Passo 1: Alterar CursoService e CursoServiceImpl

- Na classe CursoService adicione:

```
void editar(Long id, CursoDTO dto);
```

- Na classe CursoServiceImpl adicione:

```
@Override
```

```
public void editar(Long id, CursoDTO dto) {
```

```
    Curso curso = cursoRepository.findById(id)
```

```
        .orElseThrow(() -> new RegraNegocioException("Código  
usuário não encontrado."));
```

```
    CategoriaCurso categoriaCurso =  
    categoriaCursoRepository.findById(dto.getCategoriaCursoId())
```

```
        .orElseThrow(() -> new RegraNegocioException("Categoria  
não encontrado."));
```

```
    curso.setNome(dto.getNome());
```

```
    curso.setCargaHoraria(dto.getCargaHoraria());
```

```
    curso.setCategoriaCurso(categoriaCurso);
```

```
    cursoRepository.save(curso);
```

```
}
```

Passo 2: Controller – Adicionar método Edit

```
@PutMapping("{id}")  
    public void edit(@PathVariable Long id, @RequestBody CursoDTO  
dto) {  
        cursoService.editar(id, dto);  
    }
```

Exercício

1. Crie um `@RestControllerAdvice` para capturar exceções `IllegalArgumentException`.
2. Crie uma classe `ClienteDTO` com padrão `Builder` e use-a em um controller.
3. Implemente os métodos `ListarTodos`, `ObterPorId`, `Editar`, e `Excluir` em uma API de "Cliente".

O exercício deve apresentar a estrutura:

1. DTO

ClienteDTO.java : DTO com padrão Builder para transferência de dados.

2. Entidade

Cliente.java: Classe de entidade JPA representando a tabela cliente.

3. Repositório

ClienteRepository.java

4. Serviço

ClienteService.java

Interface com os métodos: listarTodos, obterPorId, editar, excluir.

ClienteServiceImpl.java

Implementação do serviço com as regras de negócio.

5. Controller

ClienteController.java

Define os endpoints da API REST para listar, obter, editar e excluir clientes.

6. Tratamento Global de Erros

GlobalExceptionHandler.java

Classe anotada com @RestControllerAdvice que captura IllegalArgumentException e retorna 400 Bad Request

Dúvidas?



MUITO OBRIGADA!!!!



cilene.real@facens.br