

## Spring Boot – Parte II

### 12 – Validando Informações

Como em nossas requisições de POST e PUT, adicionamos a anotação `@Validation`, podemos realizar validações antes do objeto ser persistido no Banco de Dados.

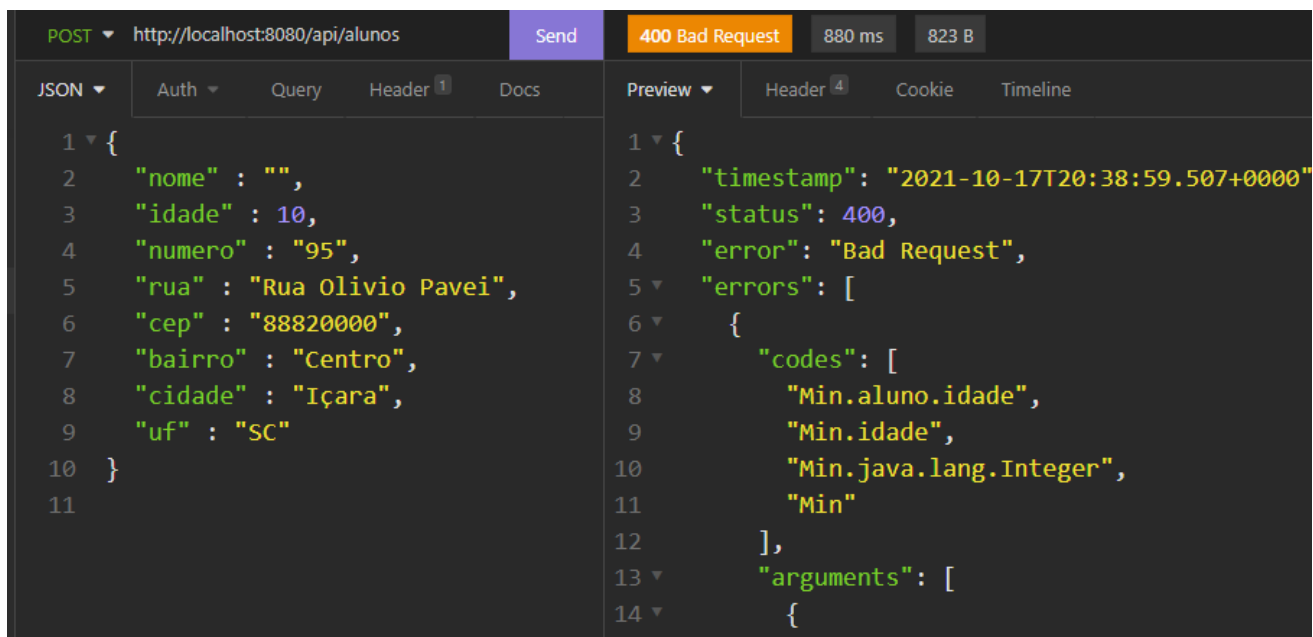
Este procedimento é possível adicionando anotações do pacote `javax.validation`, estas validações são aplicadas no objeto da entidade, antes de realizar a persistência no banco de dados. Seriam pré-validações executadas em objetos deste tipo.

Em nosso exemplo iremos utilizar a validação `@NotEmpty` no atributo `nome`, esta anotação vai validar se o campo `nome` tem a informação diferente de nulo e vazio. No atributo `idade` iremos colocar a anotação `@Min` com o valor de 15, indicando que esta deve ser a idade mínima permitida. Estas alterações devem ser realizadas na classe `Aluno` em `model.entity`.

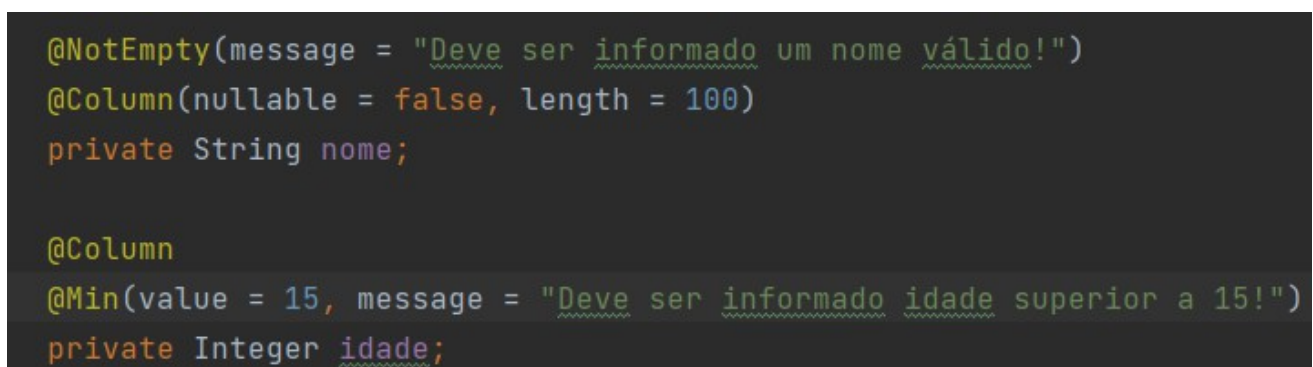
```
@NotEmpty
@Column(nullable = false, length = 100)
private String nome;

@Column
@Min(15)
private Integer idade;
```

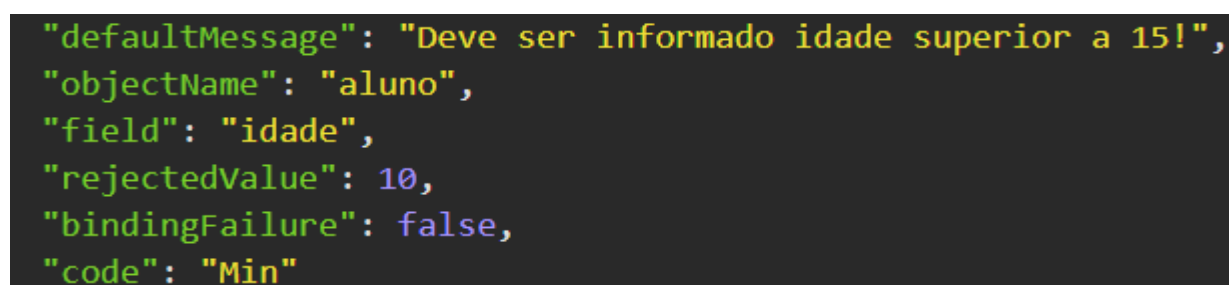
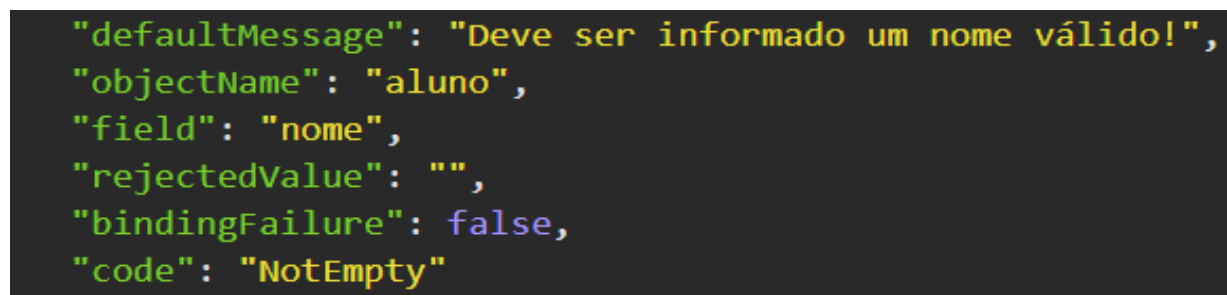
Utilizando o Insomnia, podemos deixar o campo `nome` vazio e a `idade` como 10, ao executar a requisição POST, iremos receber os erros referente a `idade` e ao campo `nome`. Porém a mensagem é retornada conforme o padrão utilizado pelo Spring Boot.



Podemos ainda definir mensagens a serem retornadas por estas validações em nossos serviços, para isso precisamos adicionar o message com o texto desejado em nossas validações.



Desta forma teremos o retorno com as mensagens, ao utilizar o Insomnia.



Atualmente temos um retorno destes erros, extremamente complicado para realizarmos algum tratamento com este retorno em nosso Front por exemplo. Nesta etapa iremos melhorar este retorno de erro do nosso backEnd, nossa ideia é retornar apenas uma lista de erros com as mensagens do BackEnd.

O Spring Boot permite alterar o retorno de erros de nossa aplicação utilizando o `@RestControllerAdvice`. No caso as controllers de nossa aplicação são executadas conforme a definição das urls em nossas requisições. Porém uma classe com a anotação `@RestControllerAdvice` é executada quando alguma requisição de nossa aplicação tem alguma exceção.

Desta forma podemos criar um controller que recebe as requisições com erro, para tratarmos este retorno antes de ser apresentado ao nosso cliente que está consumindo nosso serviço.

Então iremos criar um modelo de como estes erros serão retornado em nossos serviços. Para isso deve ser criado no pacote `rest.exception` a classe `ApiErrors`.

Nosso JSON de retorno com os erros, será gerado com base nesta classe, então teremos erros como base e uma lista com as mensagens de erro.

Esta classe tem um `List` de `String` que vai receber as mensagens de erro. Teremos dois construtores, pois podemos criar o objeto com base em uma lista ou se tiver uma única mensagem, será gerada lista com esta `String`

```
public class ApiErrors {  
    @Getter  
    private List<String> erros;  
  
    public ApiErrors(List<String> erros){  
        this.erros = erros;  
    }  
  
    public ApiErrors(String message){  
        this.erros = Arrays.asList(message);  
    }  
}
```

Com o nosso modelo pronto, iremos criar a controller que vai interceptar os erros das requisições, iremos criar a classe `ApplicationControllerAdvice` no pacote `rest`.

Iremos adicionar na classe a anotação `@RestControllerAdvice`, que estará configurando nossa aplicação Spring Boot, para utilizar esta classe quando localizar um erro em requisições nos nossos serviços.

Criamos o método `tratarRetornoDeErro`, que retorna o objeto de nossa classe `ApiErrors` e recebe `MethodArgumentNotValidException`, que é o objeto com a exceção criada pelos nossos serviços. Adicionando a anotação `@ExceptionHandler` indicando para que exceção este método deve ser executado e o `@ResponseStatus` com o retorno do serviço para este tipo de problema.

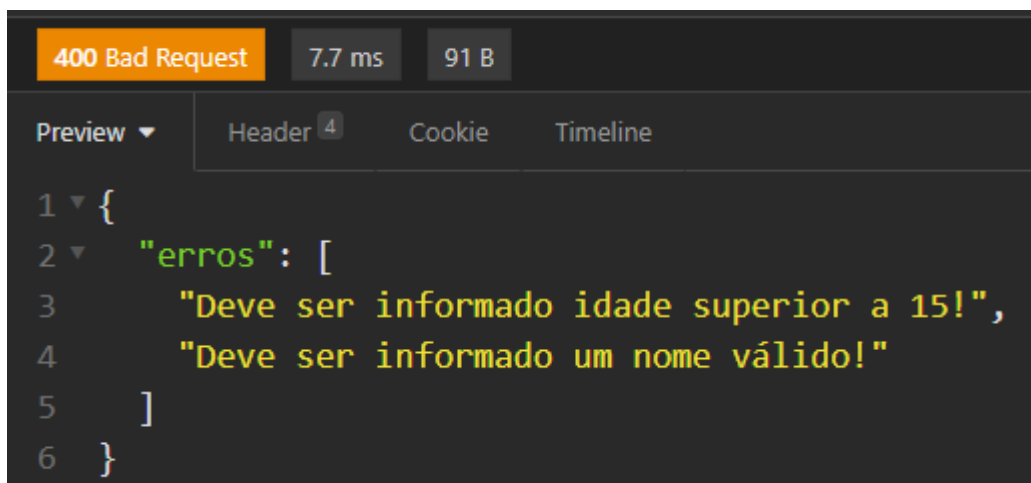
No corpo do método utilizamos `BindingResult` que é a representação da exceção obtida por nosso serviço, desta forma criamos uma expressão Lambda, que vai pegar os erros retornados, filtrando apenas a informação do `DefaultMessage`, que possui as mensagens adicionadas em nossa entidade. No fim retornamos o objeto `ApiErrors` passando a lista das mensagens obtidas em nosso serviço.

```
@RestControllerAdvice
public class ApplicationControllerAdvice {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ApiErrors retornoErroEntidade(MethodArgumentNotValidException ex){
        BindingResult bindingResult = ex.getBindingResult();
        List<String> mensagensDeErro = bindingResult
            .getAllErrors() List<ObjectError>
            .stream() Stream<ObjectError>
            .map(objetocomErro -> objetocomErro.getDefaultMessage()) Stream<String>
            .collect(Collectors.toList());

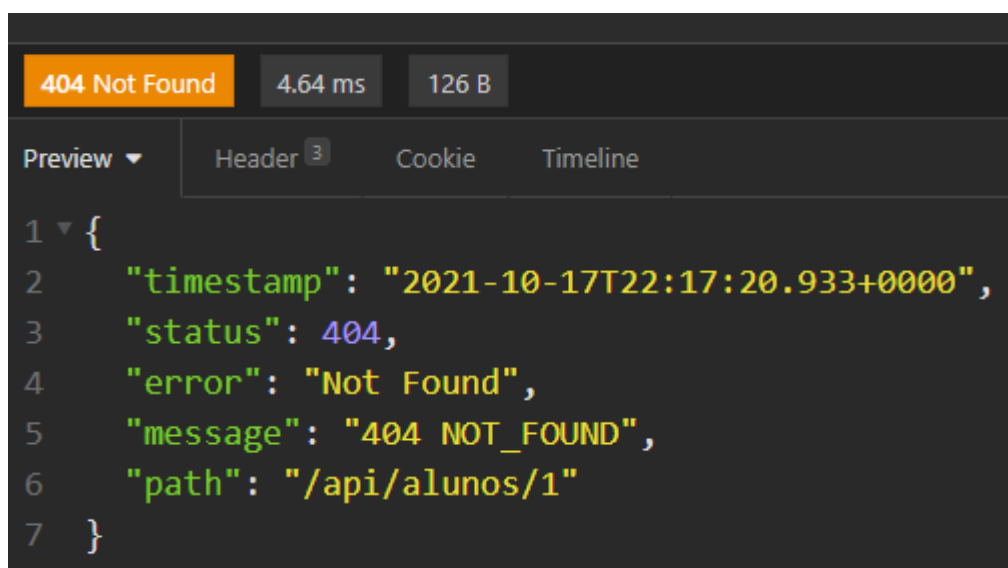
        return new ApiErrors(mensagensDeErro);
    }
}
```

Utilizando novamente o Insomnia, teremos apenas a informação de erros e as mensagens retornadas por nossa entidade.



```
400 Bad Request 7.7 ms 91 B
Preview ▾ Header 4 Cookie Timeline
1 {
2   "erros": [
3     "Deve ser informado idade superior a 15!",
4     "Deve ser informado um nome válido!"
5   ]
6 }
```

Em nosso tratamento de erros, estamos preparados para retornar quando temos algum problema relacionado as validações das entidades. Iremos adicionar um tratamento para quando retornarmos o status not found para os nossos métodos PUT, DELETE e GET. Pois atualmente temos como retorno a mensagem padrão do Spring e iremos utilizar a nossa Api de erros.



```
404 Not Found 4.64 ms 126 B
Preview ▾ Header 3 Cookie Timeline
1 {
2   "timestamp": "2021-10-17T22:17:20.933+0000",
3   "status": 404,
4   "error": "Not Found",
5   "message": "404 NOT_FOUND",
6   "path": "/api/alunos/1"
7 }
```

Iremos alterar nos métodos GET, PUT e DELETE de AlunoController, para quando retornamos a exceção indicando que o aluno não existe, adicionar uma mensagem, que será posteriormente capturada ao adaptarmos o retorno do serviço ao nosso cliente final. Para isso precisamos no `ResponseStatusException`, além de retornar o tipo de Status, adicionarmos uma mensagem.

```

@GetMapping("/{id}")
public Aluno acharPorId(@PathVariable Integer id){
    return repository
        .findById(id)
        .orElseThrow(() -> new RuntimeException(HttpStatus.NOT_FOUND, "Aluno " + id + " não cadastrado!"));
}

```

```

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deletar(@PathVariable Integer id){
    repository
        .findById(id)
        .map(aluno -> {
            repository.delete(aluno);
            return Void.TYPE;
        })
        .orElseThrow(() -> new RuntimeException(HttpStatus.NOT_FOUND, "Aluno " + id + " não cadastrado!"));
}

```

Com a mensagem adicionada, iremos preparar o nosso ApplicationControllerAdvice para tratar este tipo de retorno.

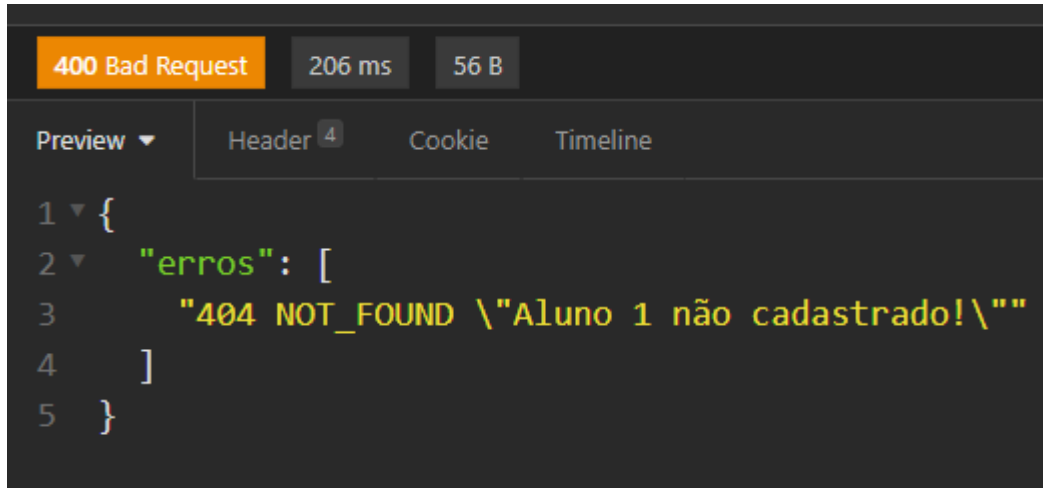
Teremos o método retornoErroNotFound, que vai receber um RuntimeException e na anotação do @ExceptionHandler, será indicado que este método deve ser executado, quando nossa aplicação possuir uma exceção do tipo RuntimeException. Como esta exceção não é uma lista, podemos retornar o ApiErrors apenas com a informação da getMessage.

```

@ExceptionHandler(RuntimeException.class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
public ApiErrors retornoErroNotFound(RuntimeException ex){
    return new ApiErrors(ex.getMessage());
}

```

Executando o Insomnia, teremos o retorno adicionado na exceção dos nossos serviços GET, DELETE e PUT.



### 13 – Api de Disciplinas

Nesta etapa iremos criar o serviço para as disciplinas a serem cadastradas em nossa base de dados, basicamente vamos passar novamente todo o caminho necessário para disponibilização de um serviço rest no Spring Boot para nossos usuários.

A primeira classe é o nosso modelo de disciplinas, deve ser criado no pacote model.entity a classe Disciplina. Neste classe teremos o `@Entity`, indicando que está classe será utilizada pelo JPA/Hibernate e as anotações do Lombok(`@Getter` e `@Setter`) para criar os métodos em tempo de execução.

Utilizaremos as informações de id, descrição e número de horas da disciplina. O id será Integer com as anotações que o mesmo é o id da entidade e a sua estratégia de geração deste sequencial.

Na descrição será informada que a mesma não pode ser nula e terá o tamanho de 100 caracteres, ainda estamos adicionando o `@NotEmpty` que vai validar a descrição antes de persistir no banco de dados, com a mensagem que deve ser retornada, quando não informada a descrição.



O número de horas é Integer, temos o name em @Column para definirmos como a coluna deve se chamar no banco de dados e adicionamos as validações para o valor mínimo e máximo para esta coluna.

```
@Entity
@Getter@Setter
public class Disciplina {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    @NotEmpty(message = "Deve ser informada a descrição da disciplina!")
    @Column(nullable = false, length = 100)
    private String descricao;

    @Min(value = 1, message = "Deve ser informado o número de horas!")
    @Max(value = 10, message = "A disciplina não pode possuir mais de 10 horas!")
    @Column(name = "numeros_horas")
    private Integer numeroHoras;
}
```

Com o modelo pronto, precisamos criar o repository de Disciplina, que terá as operações de CRUD. Deve ser criado em model.repository a Interface DisciplinaRepository que vai herdar a JpaRepository e vai passar a informação da classe de modelo (Disciplina) e o tipo de dado do Id de Disciplina nesta caso Integer.

```
public interface DisciplinaRepository extends JpaRepository<Disciplina, Integer> {
}
```

Então iremos criar no pacote rest a classe DisciplinaController. Precisamos indicar para o Spring Boot que esta classe vai receber requisições do nosso navegador e qual url será informada para executá-la.

Para isso precisamos adicionar a anotação @RestController, desta forma estamos indicando ao Spring Boot que esta classe receberá requisições rest e com a anotação @RequestMapping estaremos indicando o nome da url, neste caso caso será /api/disciplinas. Como nossa aplicação está sendo executada na porta 8080, teremos <http://localhost:8080/api/disciplinas>.



Isso resolve a parte da requisição chegar nesta controller, mas ainda precisamos configurar o acesso ao nosso banco de dados, que vai ocorrer pela repository.

Para isso iremos criar uma variável de `DisciplinaRepository`, que será passada no construtor, onde temos a anotação `@Autowired`. Esta anotação indica que o `disciplinaRepository` será adicionado pela injeção de dependência, ou seja, no momento que iniciamos a aplicação, será criada a instância deste objeto e cada vez que executarmos o serviço, será executado este objeto criado.

Para finalizar precisamos indicar o método e o verbo. Neste caso queremos criar uma nova disciplina em nosso banco, então devemos executar o método `POST`, desta forma adicionamos a anotação `@PostMapping`, que indica ao controller que este método será executado por uma requisição do tipo `POST`. Ainda temos a anotação `@ResponseStatus`, que vai indicar o tipo de retorno que será passado a requisição, quando executado com sucesso.

No método `salvar`, retornamos e recebemos um objeto `Disciplina`, porém no parâmetro temos as anotações `@Valid` e `@RequestBody`. O `@Valid` vai permitir validações em nossa entidade JPA `Disciplina`, antes de persistir o objeto no banco de dados, enquanto o `@RequestBody` está indicado que a disciplina será recebida no `JSON` no corpo da requisição `Post`.

Na execução do método, apenas utilizamos o método `salvar` da `repository`, passando a disciplina como argumento. O retorno do método `salvar` é o próprio objeto passado como parâmetro, já com as informações persistidas no banco de dados como o `ID`. Assim temos a nossa aplicação `Spring Boot`, pronta para adicionar disciplinas em nosso banco de dados.

```

@RestController
@RequestMapping("/api/disciplinas")
public class DisciplinaController {

    private final DisciplinaRepository repository;

    @Autowired
    public DisciplinaController(DisciplinaRepository repository){
        this.repository = repository;
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Disciplina salvar(@Valid @RequestBody Disciplina disciplina){
        return repository.save(disciplina);
    }
}

```

No Insomnia precisamos criar uma requisição POST para disciplinas, para isso precisamos apenas indicar a URL da requisição e montarmos o nosso JSON para envio na requisição. No caso o JSON são as informações dos atributos da classe Disciplina, sem a informação do ID, que será gerada automaticamente pelo nosso server.

Será enviado o JSON e receberemos como retorno, uma entidade de disciplina, porém com a informação do id, que foi gerado no banco de dados e o retorno do status é o 201 created, conforme configurado em nosso método executado pelo POST.

Method	URL	Status	Time	Size
POST	http://localhost:8080/api/disciplinas	201 Created	738 ms	64 B

Request Body (JSON)	Response Body (JSON)
<pre>{   "descricao": "Lógica de Programação ",   "numeroHoras": 3 }</pre>	<pre>{   "id": 1,   "descricao": "Lógica de Programação ",   "numeroHoras": 3 }</pre>

A próxima etapa de nosso servidor rest, será permitir a busca por disciplinas cadastradas em nosso banco de dados, para isso utilizamos o verbo GET, será informado a URL base e o código da disciplina que desejamos a informação.

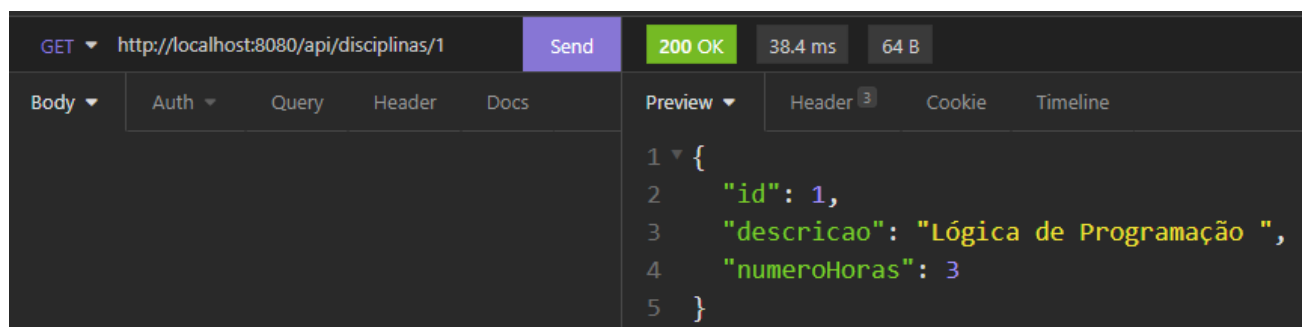
Então temos o método `acharPorId` que vai retornar uma `Disciplina`. A anotação `@GetMapping` está indicando que o método será executado, quando recebido o verbo `GET` nesta URL. Sendo indicando `Id` como parâmetro a ser recebido na requisição.

Como parâmetro temos a anotação `@PathVariable` indicando que será recebido um argumento pela requisição que será atribuído a variável `id`.

Para finalizar, nosso método utiliza o repositório executando a função `findById` passando o código da `Disciplina`, este método retorna um objeto do tipo `Optional` que pode ser uma `Disciplina` ou vazio. Desta forma será retornado a `Disciplina` caso exista, senão será executada a função `orElseThrow` onde estamos lançando uma exceção `NOT_FOUND`, indicando que a `Disciplina` não existe.

```
@GetMapping("/{id}")
public Disciplina acharPorId(@PathVariable Integer id){
    return repository
        .findById(id)
        .orElseThrow(() -> new RuntimeException(HttpStatus.NOT_FOUND, "Disciplina " + id + " não cadastrada!"));
}
```

Para testar o método `GET`, precisamos criar uma nova requisição no Insomnia com o verbo `GET` e na url de disciplinas adicionar `/` e o código da disciplina que desejamos selecionar. Neste caso será retornado a `Disciplina` ou `Not Found` quando o mesmo não existir em nosso banco de dados.



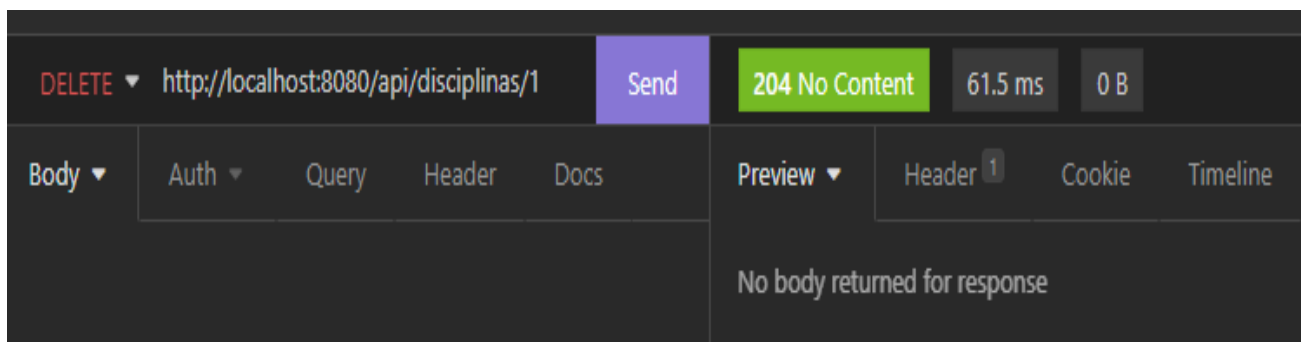
O processo de exclusão será basicamente o mesmo que a busca da Disciplina, porém iremos utilizar o método para exclusão do repositório.

Temos o método deletar sem retorno, que vai receber o parâmetro passado na URL com o nome de id. Utilizamos a anotação `@DeleteMapping` para indicar que este método deve ser executado ao receber uma requisição do método DELETE.

A anotação `@ResponseStatus` vai retornar `NO_CONTENT`, quando a exclusão for executada com sucesso, pois este é o retorno a ser utilizado de forma padrão para este tipo de operação. No corpo do método utilizamos o `findById` quando localizado temos o retorno em map, onde pegamos a disciplina retornada, excluimos e adicionamos o retorno void. Quando não localizado a disciplina, retornamos o status `NOT_FOUND` para a requisição.

```
@DeleteMapping("{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deletar(@PathVariable Integer id){
    repository
        .findById(id)
        .map(disciplina -> {
            repository.delete(disciplina);
            return Void.TYPE;
        })
        .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Disciplina " + id + " não cadastrada!"));
}
```

Para testar o método DELETE, precisamos criar uma nova requisição no Insomnia com o verbo DELETE e na url de disciplinas adicionar / e o código da disciplina que desejamos selecionar. Neste caso será retornado No Content quando excluído ou Not Found quando o mesmo não existir em nosso banco de dados.



Nosso último método será o atualizar, este será parecido com o Adicionar, pois precisaremos passar o JSON com as alterações realizadas na Disciplina para o método.

A anotação `@PutMapping` indica que o método será executado quando realizada uma requisição com o verbo PUT, sendo que teremos dois parâmetros. A informação do Id que queremos atualizar e um JSON com as informações da Disciplina, por isso nosso parâmetro possui a anotação `@RequestBody`, indicando que a informação da disciplina será recebida no corpo da requisição.

No corpo do método utilizamos o `findById`, quando localizado a disciplina, setamos nele as informações recebidas na requisição e salvamos o cadastro, quando não existir será retornado o Status `NOT_FOUND`.

```
@PutMapping("{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void atualizar(@PathVariable Integer id, @Valid @RequestBody Disciplina dadoAtualizado){
    repository
        .findById(id)
        .map(disciplina -> {
            disciplina.setDescricao(dadoAtualizado.getDescricao());
            disciplina.setNumeroHoras(dadoAtualizado.getNumeroHoras());
            return repository.save(disciplina);
        })
        .orElseThrow(() -> new RuntimeException(HttpStatus.NOT_FOUND, "Disciplina " + id + " não cadastrada!"));
}
```

Para testar nosso PUT, deve ser criada uma nova requisição do tipo PUT, indicando que no corpo da requisição teremos um JSON.

Será necessário indicar na URL o id que queremos atualizar e informar o JSON com os dados que queremos alterar, não é necessário informar o ID, pois é o código informado na requisição.

