Spring Boot

1 – Introdução

O Spring Boot é um framework MVC, Framework é um termo inglês que, em sua tradução direta, significa estrutura. De maneira geral, essa estrutura é feita para resolver um problema específico.

Na programação, um framework é um conjunto de códigos genéricos capaz de unir trechos de um projeto de desenvolvimento.

Ele funciona como uma peça de quebra cabeça capaz de se encaixar nos mais diversos lugares e conectar todos as linhas de código de uma maneira quase perfeita.

Por mais que seu funcionamento e aplicação pareçam simples, o dev precisa entender bem o tipo de framework que ele está usando ou usará em seu projeto.

2 – Nosso Projeto

Para entendermos o funcionamento do Spring Boot. Iremos criar um projeto, que vai permitir cadastrar alunos, disciplinas e depois definir notas para os alunos nestas disciplinas.

Em nosso backend teremos um projeto Spring Boot, que estará pronto para receber requisições POST, PUT, DELETE e GET.

Desta forma será possível realizar inserções, atualizações, exclusões e realizar buscas das informações em nosso banco de dados. Lembrando que estaremos preparando o nosso server e depois iremos implementar o Front deste projeto com AngularJS.

3 – Criando Projeto

Iremos criar o Projeto Alunos com o Maven.

Nosso primeiro trabalho será indicar em nosso projeto, que estamos utilizando o Spring Boot Framework, para isso utilizaremos a tag Parent do Maven. Esta TAG serve para indicar qual projeto

pai será utilizado, no caso informaremos o projeto Spring Boot. Para isso iremos adicionar a dependência no MAVEN.

Spring Boot Starter » 2.2.6.RELEASE

Core starter, including auto-configuration support, logging and YAML

License	Apache 2.0
Organization	Pivotal Software, Inc.
HomePage	https://projects.spring.io/spring-boot/#/spring-boot-parent/ ...
Date	(Mar 26, 2020)
Files	jar (397 bytes) View All
Repositories	Central
Used By	5,142 artifacts

Note: There is a new version for this artifact

New Version

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.2.6.RELEASE</version>
</dependency>
```

Com esta informação iremos adicionar no POM.xml, porém com a TAG Parent.

Desta forma as próximas dependências adicionadas do Spring, não precisam da informação da versão, pois será considerada a versão definida no Parent 2.2.6.RELEASE.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.6.Release</version>
</parent>
```

Então iremos adicionar as dependências de nosso projeto Spring Boot. Como vimos anteriormente o Spring, possui vários Starters que são adicionados conforme a necessidade dos devs. Vamos então adicionar as dependências do starter do Spring Boot e do Stater Web.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Ainda precisamos adicionar o plugin do Spring Boot ao Build da Aplicação. Neste caso este plugin serve para realizar a compilação de um projeto Spring Boot e gerar o arquivo da nossa aplicação. Para isso iremos adicionar o Spring boot plugin na tag build, abaixo das dependências.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Para concluirmos a nossa configuração inicial, precisaremos criar a classe principal de nossa aplicação Spring Boot, aonde teremos o main desta aplicação.

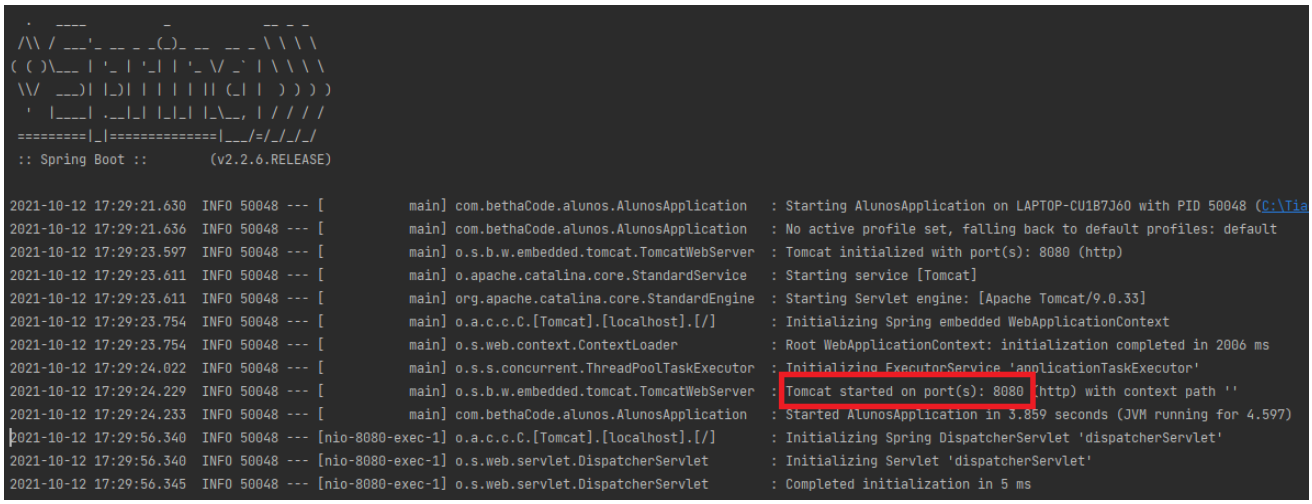
Então iremos criar no pacote com.bethaCode.alunos a classe AlunosApplication está é uma nomenclatura utilizada para indicar a classe principal do Projeto.

Para o Spring Boot identificar que está é a classe inicial, ou seja, o ponto de partida da aplicação, precisamos adicionar a anotação @SpringBootApplication. Nesta classe precisamos adicionar o main com o código resposnável pela execução do servidor Spring Boot, que será SpringApplication.run().

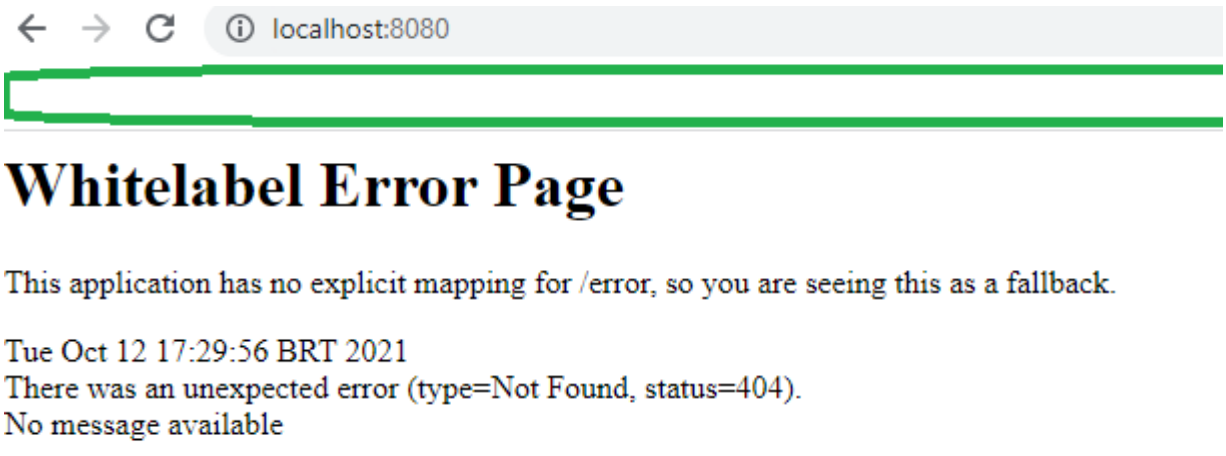
Neste método passamos como parâmetro a classe principal do Projeto e os argumentos necessários para a sua execução.

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AlunosApplication {

    public static void main(String[] args) {
        SpringApplication.run(AlunosApplication.class, args);
    }
}
```

Executando este projeto será demonstrado que o mesmo utiliza o Spring Boot e a porta onde o mesmo estará sendo executado.



Executando no browser, será demonstrado que temos um projeto em execução na porta 8080, mas ainda não temos urls definidas para o projeto.



4 – Conexão com Banco de Dados

Para utilizarmos o Banco de Dados, precisaremos adicionar as dependências referentes a JPA do Spring Boot e do nosso banco de dados PostGres.

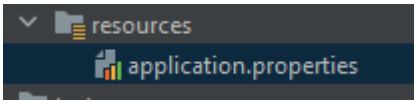
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
```

Porém ao executar nossa aplicação novamente teremos como resultado.

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class


Action:

Consider the following:
    If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
    If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).
```

Este problema ocorre, pois indicamos para o Spring Boot que iremos utilizar o módulo de banco de dados, porém não temos uma configuração de acesso a banco de dados em nossa aplicação.

Para isso precisamos criar o arquivos application.properties em resources, este é o arquivo que o Spring vai procurar em nossa aplicação para buscar a configuração de acesso ao banco de dados.



Antes de realizar a configuração do arquivo acessar o pgAdmin do Postgres e criar o banco de dados alunosBethaCode. No arquivo teremos basicamente as mesmas informações do Persistense.xml, quando configuramos a nossa aplicação JPA.

Teremos a informação da URL do banco de dados, usuário e senha. Além das informações do dialeto de banco de dados e as configurações para demonstrar os comandos sqls aplicados no banco e a criação da estrutura de banco a cada nova execução do projeto.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/alunosBethaCode
spring.datasource.username=postgres
spring.datasource.password=postgres

spring.jpa.database-plataform=org.hibernate.dialect.PostgreSQLDialect

spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.hbm2ddl.auto=create-drop
```

5 – Modelo de Alunos

Nossa configuração de projeto Spring Boot ficou pronta. Iremos implementar o nosso modelo para Aluno utilizando o mapeamento JPA.

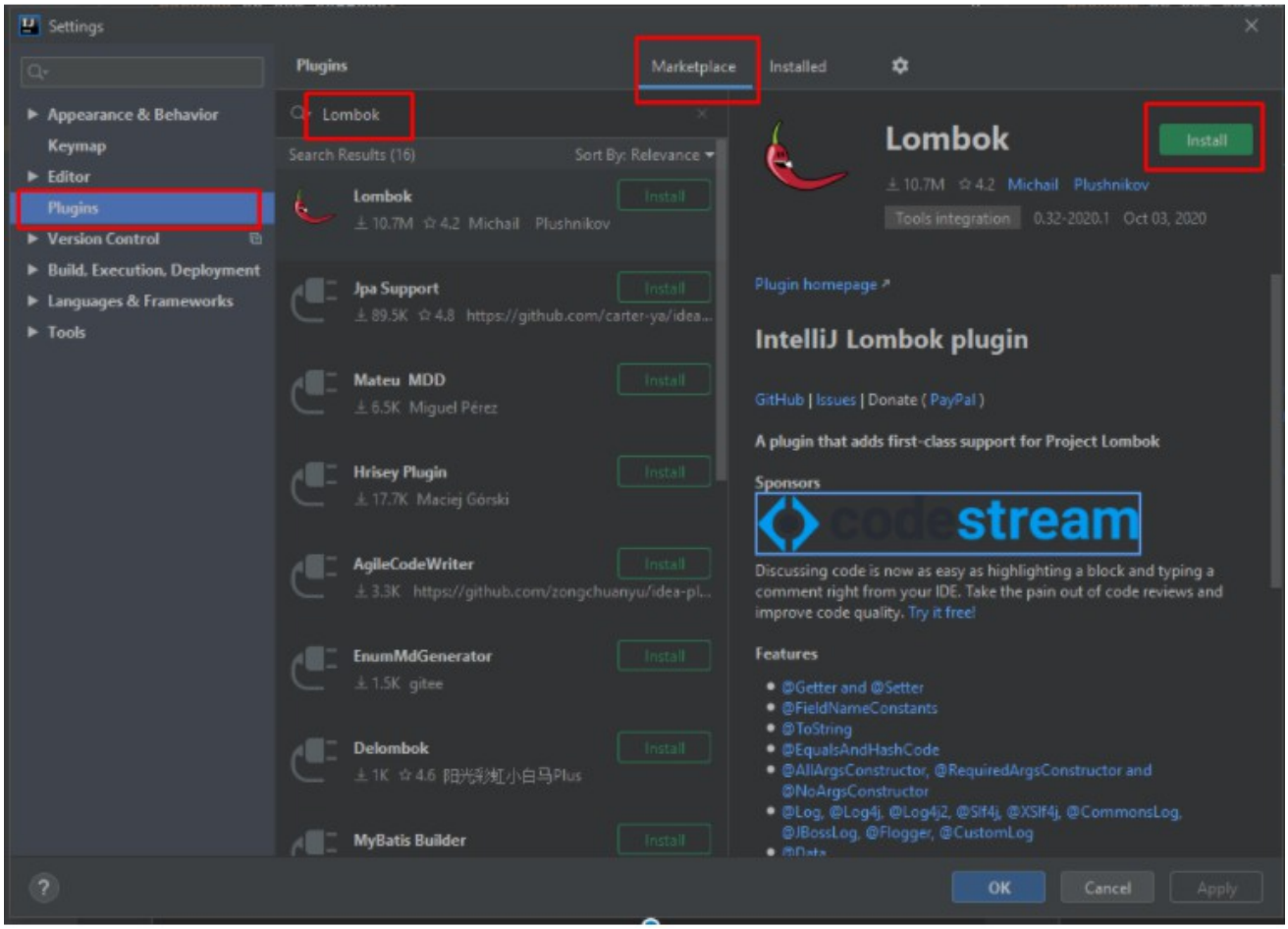
Então iremos criar a classe Aluno, com id, nome, idade, data de matricula, número do endereço, rua, cep, bairro, cidade e estado.

Porém antes de modelarmos, iremos adicionar o plugin Lombok. Este plugin serve para adicionar anotações as classes de modelo, não sendo necessário a criação dos getters e setters, pois o Lombok fará a sua criação em tempo de execução.

Para isso será necessário adicionar o Lombok nas dependências do projeto.

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

Também será necessário adicionar o plugin no IntellJ (File >> Settings), opção Plugins, digitar Lombok e depois instalar. Será necessário reiniciar o IntellJ.



Com o Lombok instalado, vamos criar o nosso Aluno modelo. Para isso iremos ciar no pacote model.entity a classe Aluno. Esta classe terá o @Entity para indicar que esta classe é um mapeamento JPA. Ainda será adicionado as anotações @Getter e @Setter do Lombok, estas anotações permitem que sejam criados em tempo de execução os getters e setters da classe Aluno.

Deve ser adicionado o atributo id com as anotações do @id e @Generation que estarão indicando que a coluna é a chave primária da tabela e que deve ser gerada de forma automática na inserção de registros no Banco de Dados.

Ainda teremos a atributo nome, com a indicação que a informação não poderá ser nula e o seu tamanho definido como 100 caracteres.

Os demais atributos terão apenas a indicação de tamanho, quando serem do do tipo String, porém todos poderão receber valor nulo ao inserir no banco de dados. Além do nome, teremos idade, data de matricula, número do endereço, rua, cep, bairro, cidade e UF.

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

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

    @Column
    private Integer idade;

    @Column(name = "data_matricula")
    @JsonFormat(pattern = "dd/MM/yyyy")
    private LocalDate dataMatricula;

    @Column(length = 10)
    private String numero;

    @Column(length = 100)
    private String rua;

    @Column(length = 8)
    private String cep;

    @Column(length = 60)
    private String bairro;

    @Column(length = 60)
    private String cidade;

    @Column(length = 2)
    private String uf;
}
```

Executando novamente a aplicação, pode ser verificado que a tabela Aluno, será criada em nosso banco de dados.

```
create table aluno (
   id int4 not null,
    bairro varchar(60),
    cep varchar(8),
    cidade varchar(60),
    data_matricula date,
    idade int4,
    nome varchar(100) not null,
    numero varchar(10),
    rua varchar(100),
    uf varchar(2),
    primary key (id)
)
```

Porém no banco de dados, podemos visualizar que será adicionado a coluna id e as demais colunas serão criadas em ordem alfabetica, diferente do definido no model. Está é uma situação do JPA, que para ser contornada, apenas criando diretamente no banco de dados esta tabela.

```
1  select * from aluno
```

a Output Explain Messages Notifications

id [PK] integer	bairro character varying (60)	cep character varying (8)	cidade character varying (60)	data_matricula date	idade integer	nome character

Para finalizar nosso modelo iremos adicionar um evento, que será executado antes do objeto ser salvo no banco de dados e neste evento iremos adicionar a data de matrícula, a data atual que o cadastro está sendo salvo. Para isso iremos adicionar o @PrePersist, indicando ao JPA que este evento deve ser executado antes de persistir a entidade no banco de dados.

```
@PrePersist
public void prePersist(){
    setDataMatricula(LocalDate.now());
}
```

6 – Repository

Um Repository(Repositório) é um objeto que isola os objetos ou entidades do domínio do código que acessa o banco de dados. Temos que um repositório implementa parte das regras de negócio no que se refere à composição das entidades.

Criamos o objetos DAO, onde tinhamos o método para inserir, atualizar, excluir, buscarPorId e buscarTodos. Estas classes recebiam uma conexão e então iteravamos informações com o banco de dados e retornavamos a nossa aplicação.

O repository é uma interface pronta, com os métodos de iteração com o banco de dados, onde apenas indicamos ao Spring Boot, que estamos implementando a Interface de repository do JPA.

Desta forma iremos criar a interface AlunoRepository em model.repository. Esta interface vai herdar de JpaRepository e será necessário passar duas informações: A classe que se refere o repository Aluno e o tipo da chave desta Entidade, no caso id é Integer.

```
import com.bethaCode.alunos.model.entity.Aluno;
import org.springframework.data.jpa.repository.JpaRepository;

public interface AlunoRepository extends JpaRepository<Aluno, Integer> {
}
```

7 – Controller

No Framework Spring, um Controller é uma classe responsável pela preparação de um modelo de Map com dados a serem exibidos pela view e pela escolha da view correta. Basicamente ele é o responsável por controlar as requisições indicando quem deve receber as requisições para quem deve responde-las.

Nossa ideia é criar um controller que vai receber requisições pela url de alunos. Vai gravar as informações em nosso banco de dados e responder de alguma forma a nossa requisição.

Então iremos criar no pacote rest a classe AlunoController. Precisamos indicar para o Spring Boot que está 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/alunos. Como nossa aplicação está sendo executada na porta 8080, teremos http://localhost:8080/api/alunos.

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 AlunoRespository, que será passada no construtor, onde temos a anotação @AutoWired. Esta anotação indica que o alunoRespository 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 um novo aluno 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 um 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 Aluno, porém no parâmetro temos as anotações @Valid e @RequestBody. O @Valid vai permitir validações em nossa entidade JPA Aluno, antes de persistir o objeto no banco de dados, enquanto o @RequestBody está indicado que o aluno será recebido no JSON no corpo da requisição Post.

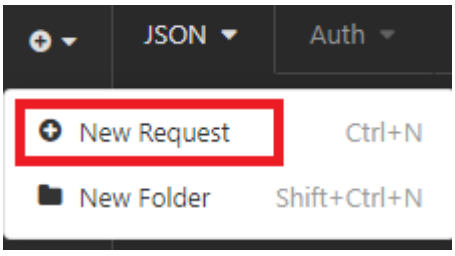
Na execução do método, apenas utilizamos o método salvar da repository, passando o aluno como argumento. O retorno do método salvar é o próprio objeto passado como parâmetro, já com as informações persistidas no banco de dados como o ID.

Assim temos a nossa aplicação Spring Boot, pronta para adicionar alunos em nosso banco de dados. Iremos utilizar o aplicativo para simular requisições de inserção e depois concluirmos nosso projeto, implementando os demais verbos de requisição (PUT, DELETE e GET).

```java
import javax.validation.Valid;

@RestController
@RequestMapping("/api/alunos")
public class AlunoController {

    private final AlunoRepository repository;

    @Autowired
    public AlunoController(AlunoRepository repository){
        this.repository = repository;
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Aluno salvar(@Valid @RequestBody Aluno aluno){
        return repository.save(aluno);
    }
}
```

8 – Insomnia

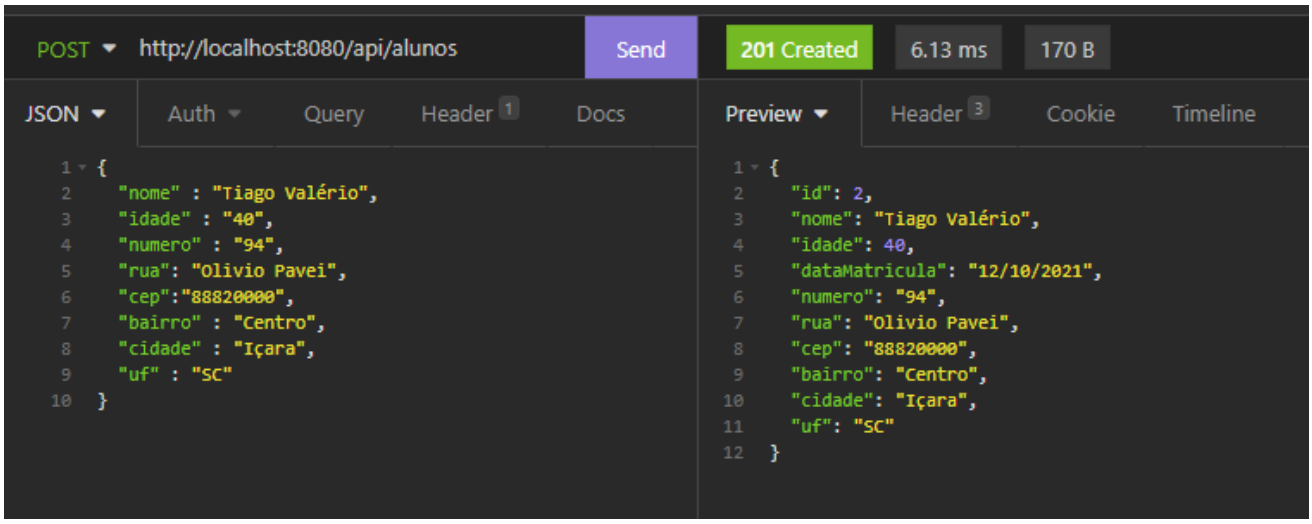
Temos à nossa disposição ferramentas que nos ajudam nos testes de APIs REST. Uma dessas ferramenta é o Insomnia. Ela é muito simples, mas muito poderosa. Iremos utiliza-la para testar nossa api de Alunos. Iniciar o Insomnia e clicar em <+> e selecionar New Request.



Na próxima etapa precisamos dar um nome a requisição, indicar que a mesma será do tipo Post e utilizaremos JSON.

Agora 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 Aluno, sem as informações do ID e dataMatricula, que serão geradas automaticamente pelo nosso server.

Será enviado o JSON e receberemos o retorno uma entidade de aluno, porém com a informação do id e também da data de matricula, que forma geradas no banco de dados e o retorno do status é o 201 created, conforme configurado em nosso método executado pelo POST.



9 – Buscar Aluno

A próxima etapa de nosso servidor rest, será permitir a busca por alunos cadastrados em nosso banco de dados, para isso utilizamos o verbo GET, será informado a URL base e o código do aluno que desejamos a informação.

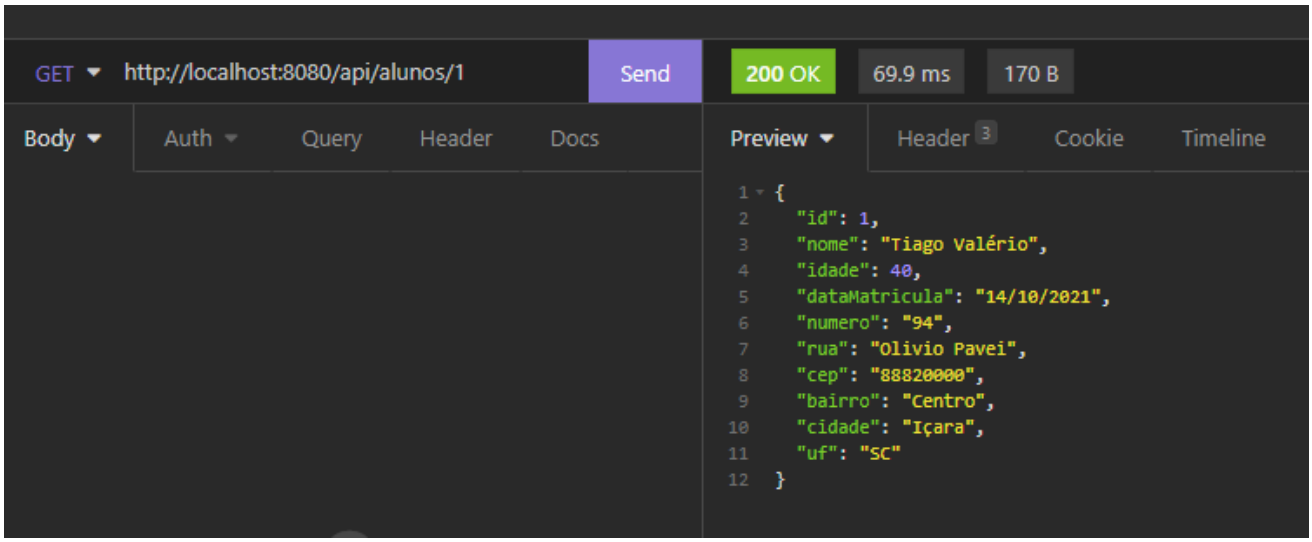
Então temos o método acharPorId que vai retornar um Aluno. A anotação @GetMapping está indicando que o método será executado, quando recebido o verbo GET nesta URL. Sendo indicando Id como parâmetro a ser recebido na requisição.

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 do Aluno, este metódo retorna um objeto do tipo Optional que pode ser um Aluno ou vazio. Desta forma será retornado o Aluno caso exista, senão será executada a função orElseThrow onde estamos lançando uma exceção NOT_FOUND, indicando que o Aluno não existe.

```
@GetMapping("{id}")
public Aluno acharPorId(@PathVariable Integer id){
    return repository
            .findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
}
```

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



10 – Deletar Aluno

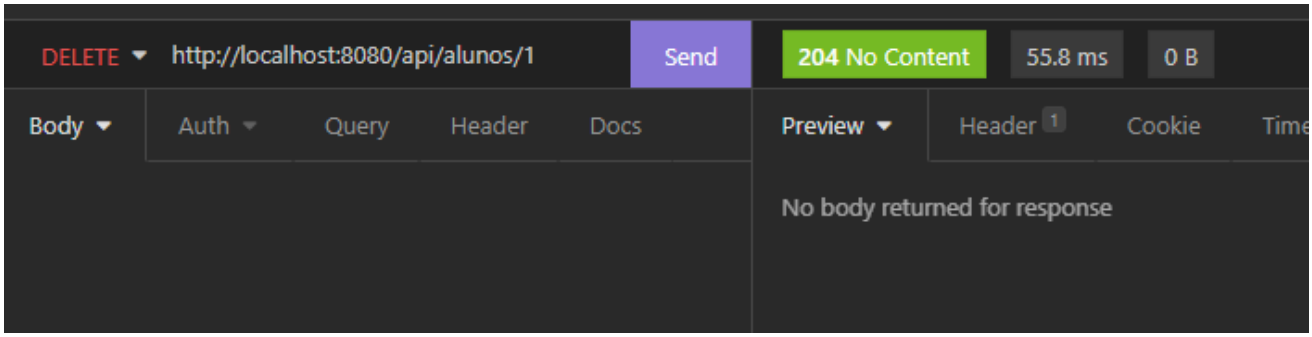
O processo de exclusão será basicamente o mesmo que a busca do Aluno, porém iremos utilizar o método para exclusão do repositório.

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 o aluno retornado, excluimos e adicionamos o retorno void. Quando não localizado o aluno, retornamos o status NOT_FOUND para a requisição.

```
@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 ResponseStatusException(HttpStatus.NOT_FOUND));
}
```

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



11 – Atualizar Aluno

Nosso último método será o atualizar, este será parecido com o Adicionar, pois precisaremos passar o JSON com as alterações realizadas no Aluno 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 do Aluno, por isso nosso parâmetro possui a anotação @RequestBody, indicando que a informação do aluno será recebida no corpo da requisição.

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

```java
@PutMapping("{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void atualizar(@PathVariable Integer id, @Valid @RequestBody Aluno alunoAtualizado){
    repository
            .findById(id)
            .map(aluno -> {
                aluno.setNome(alunoAtualizado.getNome());
                aluno.setNumero(alunoAtualizado.getNumero());
                aluno.setIdade(alunoAtualizado.getIdade());
                aluno.setRua(alunoAtualizado.getRua());
                aluno.setCep(alunoAtualizado.getCep());
                aluno.setBairro(aluno.getBairro());
                aluno.setCidade(alunoAtualizado.getCidade());
                aluno.setUf(alunoAtualizado.getUf());
                return repository.save(aluno);
            })
            .orElseThrow(()-> new ResponseStatusException(HttpStatus.NOT_FOUND));
}
```

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.