BandTec
DIGITAL SCHOOL

Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Técnicas de Programação Web

Introdução ao Spring Boot

Sumário

Spring Boot

O **Spring Boot** é uma tecnologia que facilita a criação de **REST APIs** na plataforma Java (caso não esteja familiarizado com esse conceito, pode recorrer a apostila **Introdução a REST**). Neste material, será usada a versão **2.2.9** do Spring Boot.

Podemos citar como os maiores benefícios do uso do Spring Boot:

- Menor volume de código devido ao grande grau de abstração que a ferramenta oferece. Por exemplo, não nos preocupamos em converter JSON para objetos Java ou vice-versa;
- Agrupa um conjunto de outras tecnologias muito consolidadas (JPA, Hibernate, JUnit etc) de forma bem simples;
- Alta performance. Projetos consomem poucos recursos do sistema operacional;
- É fácil criar testes automatizados para projetos criados com Spring Boot;
- Muito, muito material (inclusive gratuitos) em inglês, claro, e em português também;

A família de bibliotecas Spring é a mais usada na plataforma Java para aplicações corporativas (Vide Figura 1).

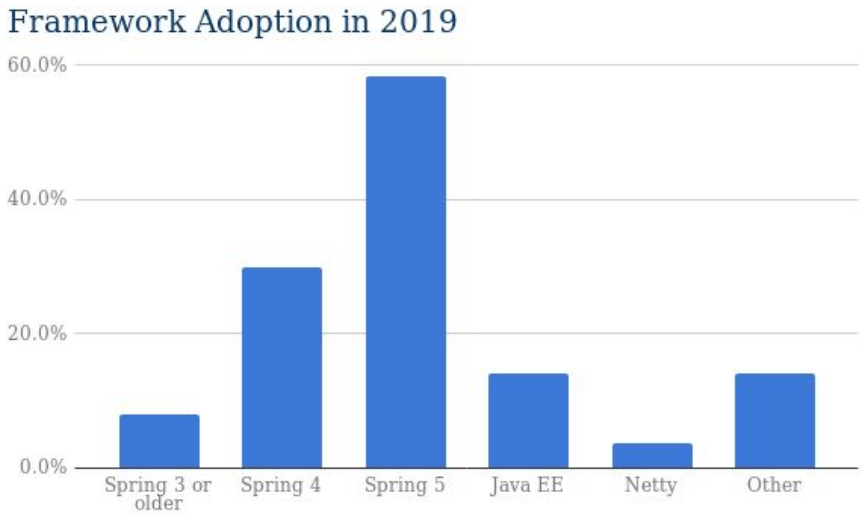

Figura 1: Adoção dos frameworks Spring x JEE
Fonte: http://www.baeldung.com/java-in-2019

Criando um projeto Spring Boot

Um projeto Spring Boot nada mais é que um projeto **Maven**[1]. Como projetos Maven são compatíveis com todas as principais IDEs da plataforma Java, um mesmo projeto Spring Boot pode ser aberto no IntelliJ, IntelliJ e Netbeans.

Podemos criar um projeto Spring Boot com os mesmo passos usados no livro *"Programação em Java"* ou usar a ferramenta online **Spring Initializr** (disponível em **https://start.spring.io/**). A sugestão aqui é usar essa ferramenta. Na Figura 2 é possível ver como criar um projeto usando-a.

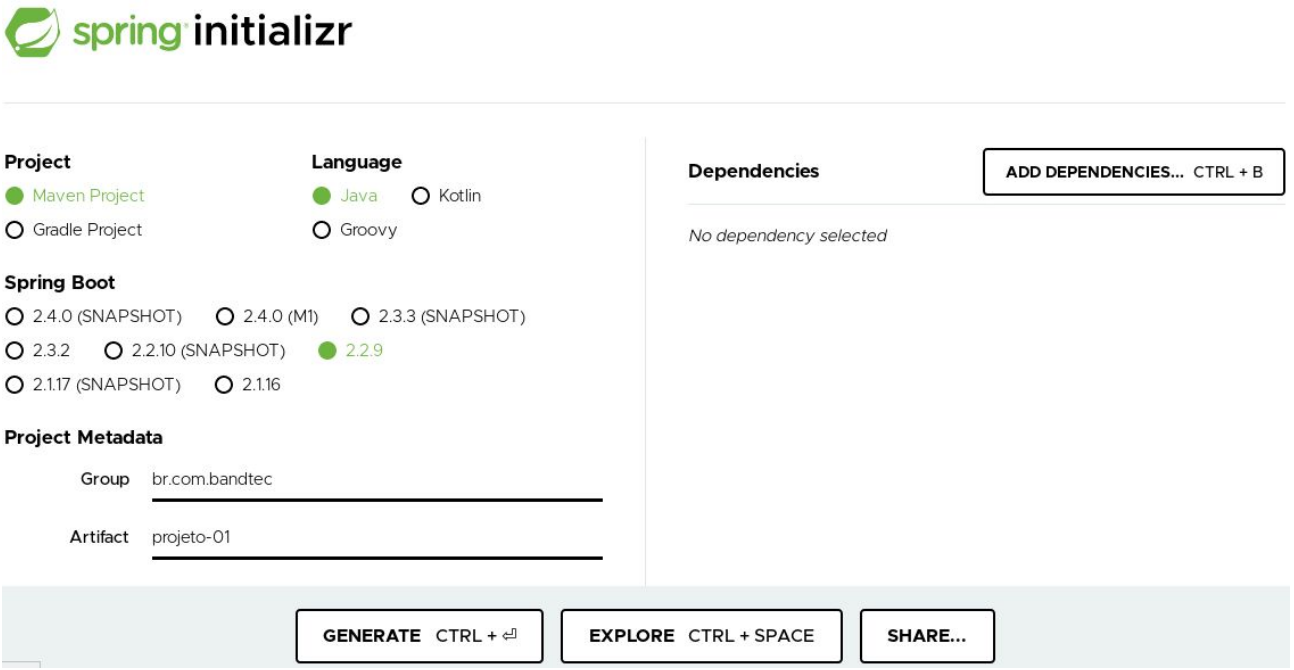

Figura 2: Página de criação de projeto Spring Boot do Spring Initializr

A sugestão é usarmos o *Group* **br.com.bandtec** e o *Artifact* **projeto-01**. A versão do Spring Boot, é a **2.2.9**.

Adicione 2 dependências clicando em **ADD DEPENDENCIES**, que são: **Spring Web** e **Spring Boot DevTools** (vide Figura 3).

[1] O **Apache Maven** é uma ferramenta desenvolvida que serve para gerenciar as dependências e automatizar seus *builds* em projetos para a plataforma Java. Site oficial: **http://maven.apache.org/**

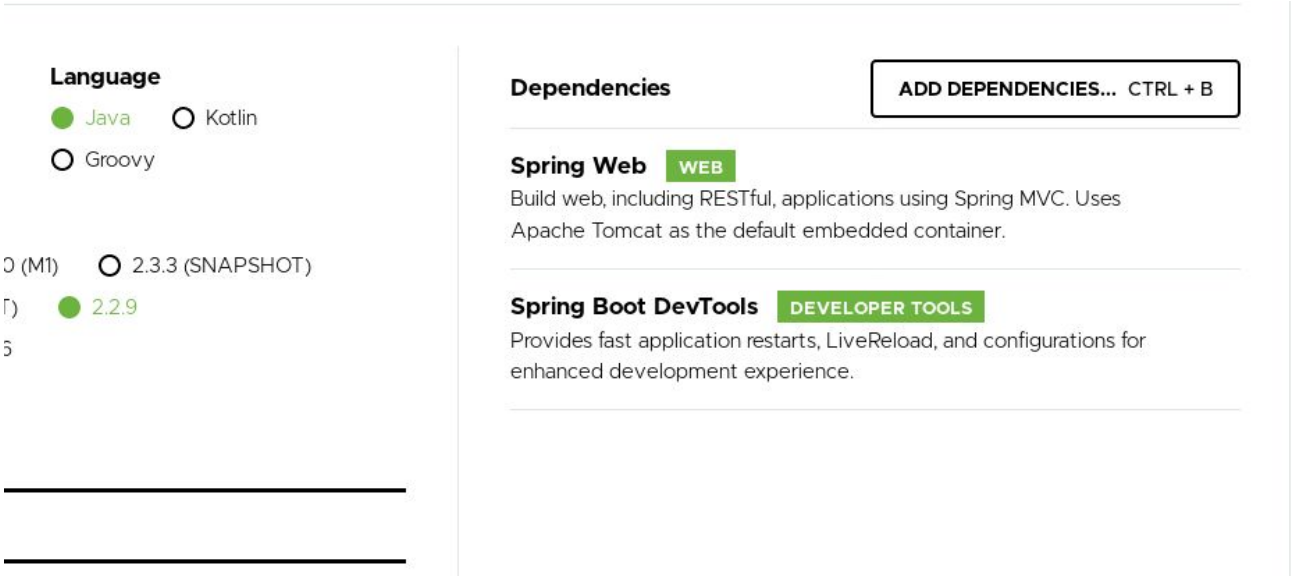

Figura 3: Dependências do projeto Spring Boot do Spring Initializr

Por fim, basta clicar em "**Generate Project**" e o projeto chegará no formato **.zip**. O navegador fará o download desse arquivo. Salve e descompacte onde preferir.

Importando um projeto Spring Boot no IntelliJ

Para importar o projeto Maven Spring Boot que criamos online, primeiro descompactamos o arquivo .zip num diretório de nossa preferência. Depois, apenas pedimos para abrir o projeto no IntelliJ. Para usá-lo, vamos no menu **File -> Open**, ou na opção **Open or Import**, caso estejamos na telinha de *Welcome* do Intellij. Em ambos os casos, aponte para a pasta na qual descompactou o .zip.

Após abrir projeto (que pode *demorar um pouco*, pois o Maven fará o download do Spring Boot e das bibliotecas que ele precisa), o projeto, que já contém algumas classes e arquivos, ficará como na Figura 4.

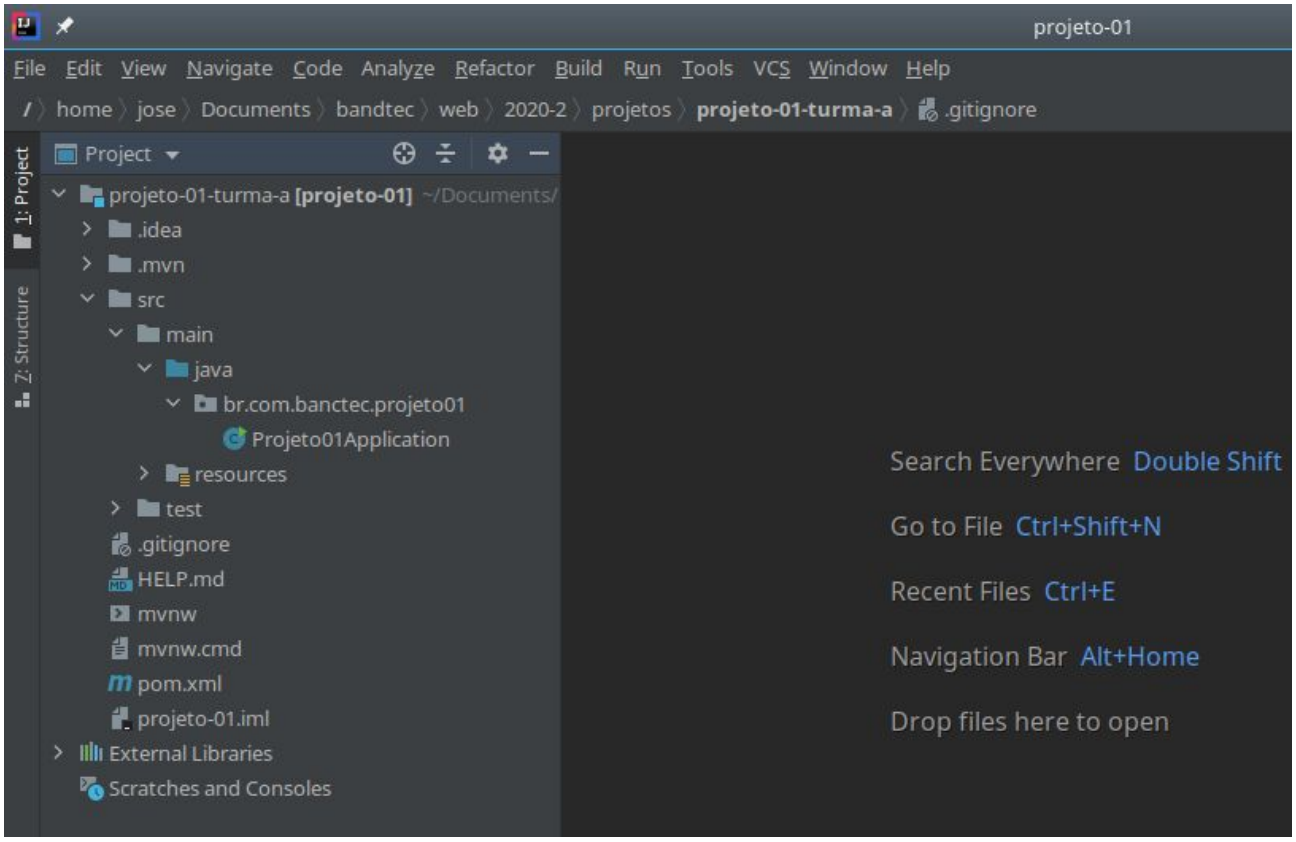

Figura 4: Estrutura do Maven Spring Boot recém importado

Vá abrindo (expandindo) os itens até chegar a uma classe Java **Projeto01Application**. Se tudo deu certo, ela fica marcada com uma seta verde, como na figura.

O arquivo **pom.xml**

O arquivo que é o "coração", ou seja, o principal arquivo num projeto Maven chama-se **pom.xml**, que fica no diretório raiz do projeto. Abra ele e observe as várias dependências e configurações que o *Spring Initializr* deixou nele.

Se quiser, altere o conteúdo da tag **<description>**. Não é algo que altere o funcionamento do projeto, pois é uma mera descrição.

O conteúdo da tag **<parent>** define a versão do Spring Boot do projeto.

A tag **<java.version>** caso exista, define a versão do Java a ser usado. sugerimos usar o **1.8**, mesmo que tenha uma versão mais recente do JDK em sua máquina.

Dentro da tag **<dependencies>** há várias tags **<dependency>**. Cada uma delas define no Maven o que chamamos de **dependência**. Quando incluímos uma dependência, nosso projeto incorpora uma **biblioteca**, que nada mais é que um projeto Java disponível em algum repositório público ou privado na internet ou numa rede local ou mesmo local no computador. No caso, como não definimos manualmente nenhum repositório, o Maven seu repositório padrão da internet que é o

Maven Central. As dependências iniciais de um projeto Spring Boot são as 2 no **pom.xml** de exemplo.

A tag **<build>** contém a indicação de um plugin do Spring Boot que "ensina" como o Maven deve gerar um "executável" do projeto.

A classe **Projeto01Application**

Outro arquivo que terá vindo criado no projeto é a classe **Projeto01Application** no pacote **br.com.bandtec.projeto01**. Seu conteúdo deve estar como do código fonte a seguir.

```
package br.com.bandtec.projeto01;


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


@SpringBootApplication
public class Projeto01Application {


    public static void main(String[] args) {
        SpringApplication.run(Projeto01Application.class, args);
    }
}
```

Essa classe é que iniciar a REST API do projeto. Note que ela possui um método main comum. Como já estudamos antes, é o método que permite a uma classe Java ser executável. Assim, basta simplesmente executar essa classe que a REST API do projeto fica disponível.

O arquivo **application.properties**

Outro importante arquivo que terá vindo criado no projeto fica no diretório **src/main/resources** e se chama **application.properties**. Ele contém configurações diversas que o Spring Boot aceita. São configurações de todos os tipos, desde configurações de log, passando por configurações de cache, internacionalização,

acesso ao banco de dados etc. **São mais de 500 configurações possíveis**. Existe uma página que contém todas configurações disponíveis para esse arquivo, disponível em **https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html**.

Testando e manipulando o projeto Spring Boot

A seguir, vamos fazer nossos primeiros testes e manipulações no projeto.

Teste inicial

Para verificar se o projeto está corretamente configurado, basta pedir que a classe **Projeto01Application** seja executada. Se tudo estiver correto, verá várias informações do Spring Boot no painel **Run** (ou seja, é o log de inicialização do projeto), como um "desenho" do próprio Spring, como na Figura 5.

Figura 5: "Desenho" do Spring Boot no log da inicialização do projeto

Após esse "desenho", várias informações vão sendo exibidas no log. Se o projeto iniciar corretamente, a parte final do log é algo como:

2020-08-13 14:16:54.772 INFO 32295 --- [restartedMain] b.c.b.projeto01.Projeto01Application : Started Projeto01Application in 4.402 seconds (JVM running for 6.001)

Fique atento para o seguinte detalhe: A execução da classe não será encerrada. No painel **Run** a classe constará como em execução ainda. Isso porque a API está em execução aguardando requisições.

Para testar a REST API, abra qualquer navegador e acesse o endereço **localhost:8080**. Ocorre que o Spring Boot, por padrão, tenta usar a porta TCP **8080**

como a que escuta as requisições para a API. Ao acessar esse endereço você verá uma página de erro como a da Figura 6.

Figura 6: Teste inicial da API no navegador

Apesar de ser uma página de erro, note que não é aquela que aparece quando o site não existe. Essa página foi devolvida pelo Spring Boot.

Caso o teste não funcione

Caso a última mensagem do log não for como a indicada aqui, provavelmente você verá uma mensagem de erro. A causa de erro mais provável é que a porta **8080** já esteja sendo ocupada por outro processo do sistema operacional. Use as ferramentas que tiver conhecimento para investigar isso e pare o outro processo que está usando a porta 8080 antes de solicitar novamente a execução da **Projeto01Application**.

Não posso parar o que está na porta 8080

Vamos supor que ou você não conseguiu identificar o que está na porta 8080 ou é algo que você não pode parar. Nesse caso, temos que configurar o projeto para tentar usar outra porta TCP. Isso é bem simples: Basta abrir o arquivo **application.properties** que fica no diretório **src/main/resources** e incluir nele a linha:

server.port=8888

Feito isso, apenas solicite a inicialização da **Projeto01Application** novamente. Supondo que informou a porta **8888** como no exemplo, teria que testar a API pelo endereço **localhost:8888** no navegador.

Programando Endpoints

Para criar **Endpoints** em nossa API REST, precisamos criar um **REST Controller**. Um REST Controller é uma classe Java que possui algumas anotações e objetos de classes disponibilizadas pela biblioteca do Spring Boot.

Criando o primeiro REST Controller

Vamos criar nosso primeiro REST Controller para termos Endpoints em nossa API seguindo o passo-a-passo:

a) Vamos criar um novo pacote **br.com.bandtec.projeto01.controllers**;

b) Nesse pacote, vamos criar a classe **FrasesController**;

c) Vamos colocar sobre a definição da classe as anotações **@RestController** e **@RequestMapping** (ambas do mesmo pacote do SpringBoot, o **org.springframework.web.bind.annotation**), como no código fonte a seguir:

```
@RestController
@RequestMapping("/frases")
public class FrasesController { ...
```

d) Vamos criar o método público **getFrase()**, que retorna um objeto do tipo **ResponseEntity** (pacote **org.springframework.http**). Este método deve estar anotado com @GetMapping (pacote **org.springframework.web.bind.annotation**) como no código fonte a seguir:

```
@GetMapping
public ResponseEntity getFrase() {
    return ResponseEntity.ok("Não sou dono do mundo mas sou filho do dono");
}
```

Assim, a classe **FrasesController** deve ficar com o código fonte como a seguir:

```
package br.com.bandtec.projeto01.controllers;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
@RequestMapping("/frases")
public class FrasesController {

	@GetMapping
	public ResponseEntity getFrase() {
		return ResponseEntity.ok("Não sou dono do mundo mas sou filho do dono");
	}
}
```

Antes de sabermos o que cada parte do código faz, que tal testarmos nosso primeiro Endpoint? Basta para a aplicação solicitando o encerramento da execução atual da **Projeto01Application**. Depois é só executá-la novamente. Ao terminar a inicialização, acesse o seguinte endereço no navegador: **localhost:8080/frases**. Você deve simplesmente ver a frase *"Não sou dono do mundo mas sou filho do dono"*, como na Figura 7.

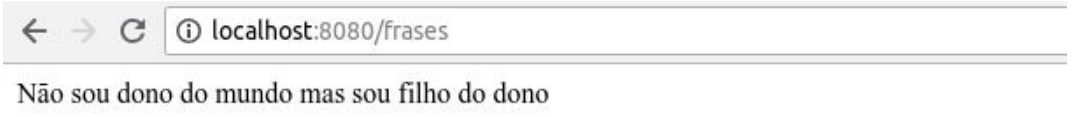

Figura 7: Teste do Endpoint recém criado no navegador

Entendendo o primeiro REST Controller

A classe **FrasesController** tornou-se um REST Controller pelo fato de estar anotado com **@RestController**. Essa anotação foi complementada pela **@RequestMapping**, na qual é definida a URI **/frases** para todos os Endpoints programados nessa classe.

O método **getFrase()** programou um Endpoint por ser público e estar anotado com **@GetMapping**. Essa anotação, como o nome sugere, faz com que o Endpoint seja acessível via método **GET** do HTTP.

Assim, o Endpoint que criamos foi o **GET /frases**. Por isso que funciona via navegador, pois toda vez que entramos com uma URL num navegador, ele faz um **GET** para ela. O termo **localhost** indica que estamos acessando algo via HTTP no próprio computador.

No método, usamos o como retorno esperado o tipo **ResponseEntity** e, no retorno usamos **ResponseEntity.*ok*("...")**. Essa classe serve para definir uma resposta HTTP válida, com status, corpo, cabeçalhos etc. Seu método estático **ok(...)** retorna um

objeto do com status **200** (resposta *"Ok"*) e o que estiver como argumento será usado como **corpo** da resposta. Assim como o **ok()**, existem outros métodos estáticos usados para criar as respostas mais comuns em REST APIs.

Criando outros Endpoints no REST Controller

Vamos simular um CRUD de frases na **FrasesController**, criando Endpoints para consulta de todas as frases, de apenas uma frase, para a criação, alteração e exclusão de frases. Vamos apagar o método anterior para, em seguida criar...

Um "banco de dados" de frases

Primeiro vamos criar uma **List** de **String** para que seja nosso "banco de dados" onde ficarão nossas frases. Basta criá-la como um atributo de instância na classe, como o código a seguir.

```
private List<String> frases = new ArrayList<>();
```

Endpoint "GET /frases"

Vamos criar um Endpoint que, ao ser invocado, retorna uma lista com todas as frases cadastradas. Criaremos um método na classe, como o código a seguir.

```
@GetMapping
public ResponseEntity recuperar() {
    return frases.isEmpty() ? ResponseEntity.noContent().build() : ResponseEntity.ok(frases);
}
```

Esse Endpoint é acessível via **GET** e retorna uma resposta com o status **204** ("sem conteúdo") caso a lista de frases estiver vazia ou **200** ("ok") e com a lista de frases não vazia como corpo.

Endpoint "GET /frases/{id}"

Vamos criar um Endpoint que, ao ser invocado, retorna uma das as frases cadastradas, conforme o **id** informado. Criaremos um método na classe, como o código a seguir.

```
@GetMapping("/{id}")
public ResponseEntity recuperar(@PathVariable("id") int id) {
    try {
        return ResponseEntity.ok(frases.get(id));
    } catch (IndexOutOfBoundsException e) {
        return ResponseEntity.notFound().build();
    }
}
```

Esse Endpoint é acessível via **GET** e exige uma barra e um valor logo após **/frases**. Esse valor (**id**) será usado como índice na lista de frases para tentar recuperar uma frase dela. Para que tudo funcione, o valor entre chaves na **@GetMapping()** deve ser o mesmo da anotação **@PathVariable()**.

Ele retorna um resposta com o status **404** ("não encontrado") caso a posição informada em **id** não seja um índice válido na lista ou **200** ("ok") e com a frase encontrada como corpo.

Endpoint "DELETE /frases/{id}"

Vamos criar um Endpoint que, ao ser invocado, exclui uma das as frases cadastradas, conforme o **id** informado. Criaremos um método na classe, como o código a seguir.

```
@DeleteMapping("/{id}")
public ResponseEntity excluir(@PathVariable("id") int id) {
    try {
        frases.remove(id);
        return ResponseEntity.ok().build();
    } catch (IndexOutOfBoundsException e) {
        return ResponseEntity.notFound().build();
    }
}
```

Esse Endpoint é acessível via **DELETE** e exige uma barra e um valor logo após **/frases**. Esse valor será usado como índice na lista de frases para tentar excluir uma frase dela.

Ele retorna um resposta com o status **404** (“não encontrado”) caso a posição informada em **id** não seja um índice válido na lista ou **200** (“ok”) após a exclusão da frase no índice informado.

Endpoint “POST /frases {JSON}”

Vamos criar um Endpoint que, ao ser invocado, cria uma nova frase na API, conforme um conteúdo em formato JSON enviado no corpo da requisição. Criaremos um método na classe, como o código a seguir.

```
@PostMapping
public ResponseEntity criar(@RequestBody String novaFrase) {
    frases.add(novaFrase);
    return ResponseEntity.status(HttpStatus.CREATED).body(novaFrase);
}
```

Esse Endpoint é acessível via **POST** (devido a anotação **@PostMapping**) e exige o envio de um corpo na requisição que, por padrão, deve estar no formato **JSON**. Esse JSON deve ter uma estrutura compatível com o tipo usado no argumento do método e sucedido pela anotação **@RequestBody**. No caso, o JSON precisa apenas ser um texto simples, dado que o argumento é uma String.

Ele adiciona a **novaFrase** na lista de frases e retorna um resposta com o status **201** (“recurso criado”) com a nova frase no corpo da resposta.

Endpoint “PUT /frases/{id} {JSON}”

Vamos criar um Endpoint que, ao ser invocado, atualiza uma frase já existente na API, conforme um conteúdo em formato JSON enviado no corpo da requisição. Criaremos um método na classe, como o código a seguir.

```
@PutMapping("/{id}")
public ResponseEntity atualizar(@PathVariable("id") int id,
                                @RequestBody String fraseAlterada) {
```

```
        try {
            frases.remove(id);
            frases.add(id, fraseAlterada);
            return ResponseEntity.ok(fraseAlterada);
        } catch (IndexOutOfBoundsException e) {
            return ResponseEntity.notFound().build();
        }
}
```

Esse Endpoint é acessível via **PUT** (devido a anotação **@PutMapping**) e exige um **id** o envio de um corpo na requisição que, por padrão, deve estar no formato **JSON**. Esse JSON deve ter uma estrutura compatível com o tipo usado no argumento do método e sucedido pela anotação **@RequestBody**. No caso, o JSON precisa apenas ser um texto simples, dado que o argumento é uma String.

Ele retorna um resposta com o status **404** ("não encontrado") caso a posição informada em **id** não seja um índice válido na lista ou **200** ("ok") após a alteração da frase no índice informado com o conteúdo de **fraseAlterada**.

Para testar os Endpoints com métodos que não são GET, você pode usar programas como o **Postman** (https://www.getpostman.com/) ou extensões de navegadores, como o **Advanced REST client** do Chrome (https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo).

Prática: Criando mais Endpoints

Tente criar um novo REST Controller que possua alguns Endpoints para diferentes métodos HTTP. Pode fazer algo parecido com o **FrasesController**.

Sugestão: Um **AmostrasController**, que vai armazenando amostras (números reais) e que possui um Endpoint que retorna a quantidade de amostras e outro que retorne o valor médio delas.

Apêndice A - Metaprogramação em Java: anotações

Na plataforma Java, metaprogramação pode ser feita por meio de *annotations* (comumente traduzido como **anotações**). Esse recurso foi introduzido na versão 5 da linguagem Java (lançada em 2004). Desde então, passou a ser cada vez mais usado tanto pelas mais diversas especificações oficiais da plataforma (por exemplo: JPA, Servlets API) quanto pelos inúmeros frameworks e bibliotecas compatíveis com a JVM (como soluções Spring Boot e GSON, por exemplo).

Metaprogramação

Segundo o livro Generative Programming: Methods, Tools, and Applications, escrito por Krzysztof Czarnecki e Ulrich Eisenecker, metaprogramação é a técnica de programação na qual algumas partes de um programa funcionam como configurações. Exemplos disso são pré-processadores e compiladores, ou quando um programa manipula outro como se fosse um repositório de dados.

No livro da Caso do Código Componentes reutilizáveis em Java com Reflexão e Anotações, escrito por Eduardo Guerra, metaprogramação é descrita como um código que trabalha com o próprio código. Com ela, um programa pode realizar ações computacionais a respeito dele mesmo.

Em Java, anotações podem estar presentes em diferentes partes do código, tais como:

- **Na definição da classe**. Nesse caso, definimos uma metaprogramação que influenciará toda a classe. Alguns exemplos vistos aqui são:

```
@SpringBootApplication
public class MinhaApiAppication { ...

@RestController
@RequestMapping("/frases")
public class FrasesController { ...
```

- **Na definição de um atributo**. Aqui, definimos uma metaprogramação que influenciará somente o atributo anotado. Alguns exemplos vistos aqui são:

```
@Autowired
private TipoEstabelecimentoRepository repository;

@Id
@Column(name = "id_tipo_estabelecimento")
private Integer id;
```

- **Na definição de um método**. Nesse caso, definimos uma metaprogramação que influenciará somente o método anotado. Alguns exemplos vistos aqui são:

```
@GetMapping
public ResponseEntity getFrase() {...

@PutMapping("/{id}")
public ResponseEntity atualizar(...
```

- **Nos argumentos de um método**. Nesse caso, definimos uma metaprogramação que influenciará somente os argumentos anotados em um método. Alguns exemplos vistos aqui são:

```
public ResponseEntity criar(@RequestBody TipoEstabelecimento tipoEstabelecimento) { ...

public ResponseEntity atualizar(@PathVariable("id") int id, @RequestBody TipoEstabelecimento tipoEstabelecimento) { ...
```

Bibliografia

BOAGLIO, Fernando. Spring Boot Acelere o desenvolvimento de microsserviços. São Paulo: Casa do Código, 2017.

GUERRA, Eduardo. Componentes Reutilizáveis em Java com Reflexão e Anotações. São Paulo: Casa do Código, 2014.

GUTIERREZ, Felipe. Pro Spring Boot. New York (EUA): Apress, 2016.

REMANI, Abdelmonaim. The Art of Metaprogramming in Java. Disponível em: https://cdn.oreillystatic.com/en/assets/1/event/80/The%20Art%20of%20Metaprogramming%20in%20Java%20%20Presentation.pdf. Acesso em: 20 de novembro de 2018.