



Universidade Federal de Uberlândia

Faculdade de Computação

12º Trabalho de Programação para Internet – Prof. Daniel A. Furtado

Trabalho Individual – Introdução ao Spring Boot

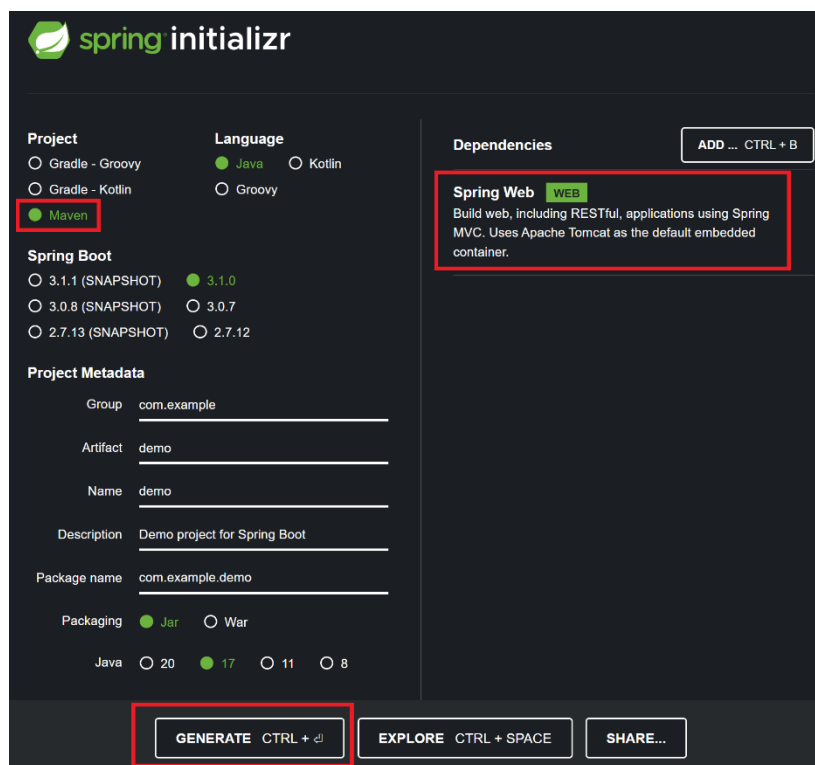
## Instruções Gerais

---

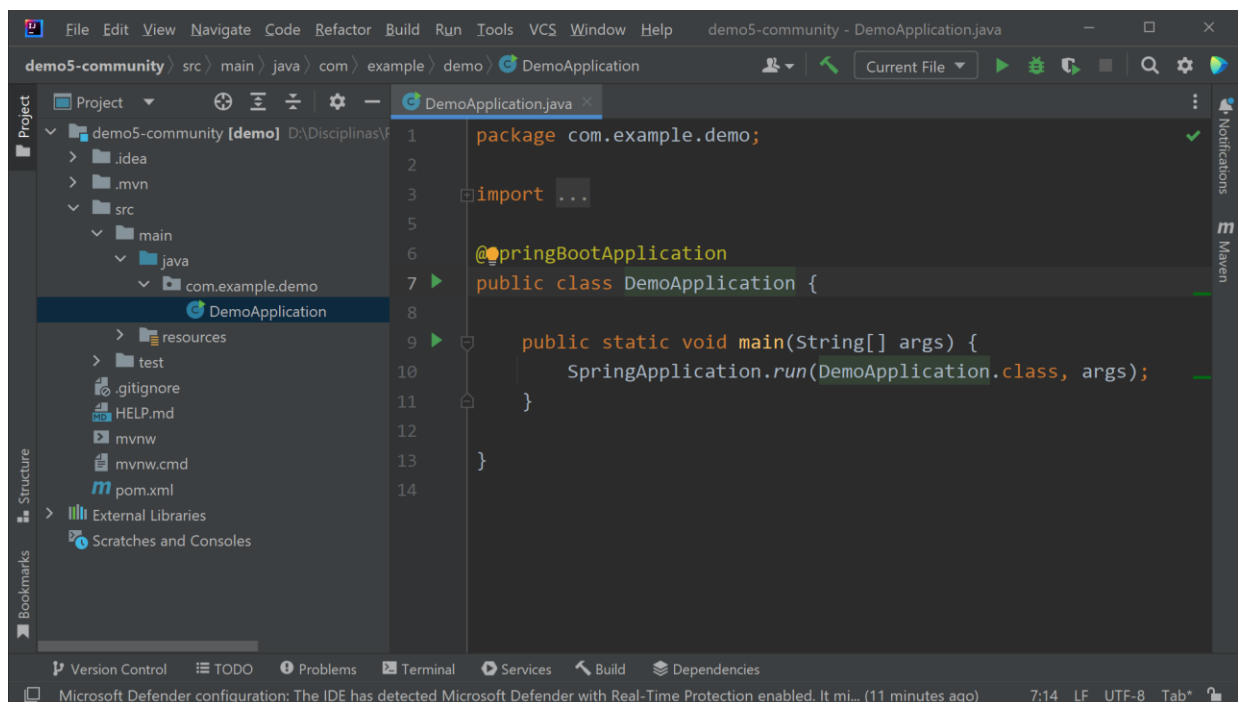
- Esta atividade deve ser realizada individualmente;
- Tecnologias permitidas: HTML5, CSS, JavaScript, API Fetch, IntelliJ IDEA, Spring Boot.
- Esteja atento às **observações sobre plágio** apresentadas no final deste documento;
- Trabalhos com implementações utilizando trechos de códigos retirados de sites da Internet ou de trabalhos de semestres anteriores serão anulados;
- As resoluções dos exercícios não devem conter qualquer conteúdo de caráter imoral, desrespeitoso, pornográfico, discurso de ódio, desacato etc.;
- O trabalho deve ser entregue até a data/hora definida pelo professor. Não deixe para enviar o trabalho nos últimos instantes, pois eventuais problemas relacionados a eventos adversos como instabilidade de conexão, congestionamento de rede, etc., não serão aceitos como motivos para entrega da atividade por outras formas ou em outras datas;
- Este trabalho deve ser feito **mantendo os trabalhos anteriores intactos**, ou seja, os trabalhos anteriores devem permanecer online conforme foram entregues;
- Trabalhos enviados por e-mail ou pelo MS Teams **não serão considerados**.


Este trabalho tem como objetivo fazer uma introdução ao framework Spring, utilizando o Spring Boot para criar um web service RESTful simplificado que permita a criação, o acesso e a exclusão de dados de endereço (CEP, rua, bairro e cidade). Faça uma leitura dos slides disponibilizados em <http://www.furtado.prof.ufu.br/site/teaching/PPI/PPI-Modulo9-Intro-Web-Services-Spring.pdf> e siga os passos a seguir:

- 1) Baixe e instale o **IntelliJ IDEA Community Edition**. Caso tenha acesso à versão **Ultimate**, ela também pode ser utilizada (neste caso os passos 2 e 3 a seguir não serão necessários, pois um novo projeto Web utilizando o Maven pode ser criado diretamente dentro do IntelliJ IDEA Ultimate);
- 2) Caso esteja utilizando o **IntelliJ IDEA Community** será necessário acessar a ferramenta disponível em [start.spring.io](http://start.spring.io) para criação de um projeto pré-inicializado, o qual será aberto posteriormente no IntelliJ IDEA Community. Após entrar em [start.spring.io](http://start.spring.io), no esquerdo da página, escolha o **Maven** como ferramenta de automação de compilação e deixe as demais opções como estão. No lado direito, adicione a dependência **Spring Web** e posteriormente clique no botão **Generate** para exportar os dados do projeto e salvar o arquivo compactado (veja figura a seguir);




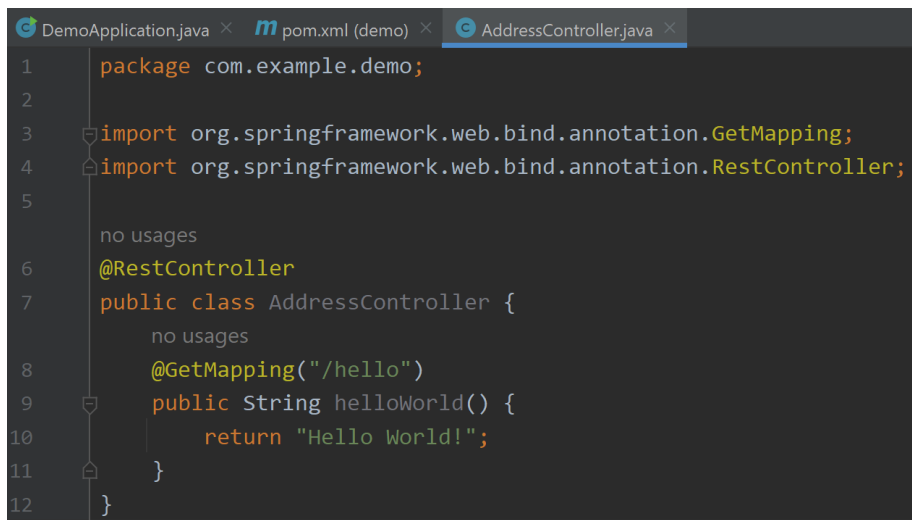
- 3) Descompacte o arquivo zip para um local de sua escolha e observe o conteúdo da pasta gerada. Abra o IntelliJ IDEA e escolha a opção **Open** para abrir a pasta do projeto;
- 4) No painel esquerdo, expanda as pastas e observe o conteúdo da classe **DemoApplication**, cujo método **main** será o ponto de entrada da nossa aplicação:



- 5) Execute a aplicação clicando no botão  e observe as mensagens apresentadas no painel inferior. Repare que as dependências do projeto são automaticamente baixadas e que o servidor web **Apache Tomcat** é automaticamente embutido e inicializado, aguardando por requisições HTTP na porta 8080. Abra o navegador de internet e digite **localhost:8080**. A seguinte página deve ser apresentada:



- 6) Pare a execução da aplicação clicando em . Em seguida, no painel à esquerda, procure pelo arquivo **pom.xml**. Esse é o arquivo de configuração do **Maven** contendo dados importantes sobre o projeto, como nome de identificação, versão do Java e dependências. Observe que a dependência selecionada no início desse roteiro aparece com a identificação **spring-boot-starter-web**;
- 7) **Criação de classe do tipo Controller**. No painel à esquerda, clique com o botão direito sobre **com.example.demo**, escolha **New → Java Class** e informe o nome **AddressController**. Crie um método de nome **helloWorld** que retorne a string "Hello World!". Adicione a *annotation* **@RestController** na classe **AddressController** e a *annotation* **@GetMapping("/hello")** no método **helloWorld**, conforme apresentado na figura a seguir. Em web services RESTful criados com o Spring as requisições HTTP são tratadas por um controlador, identificado por **@RestController**. Isso permitirá o mapeamento dos endereços e métodos das requisições HTTP aos métodos internos dessa classe. **@GetMapping("/hello")** especifica que uma requisição HTTP ao endereço /hello utilizando o método GET deve executar o método da classe associado (neste caso, o método **helloWorld()**);



```
1 package com.example.demo;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 no usages
7 @RestController
8 public class AddressController {
9     no usages
10    @GetMapping("/hello")
11    public String helloWorld() {
12        return "Hello World!";
13    }
14 }
```

- 8) Clique com o botão direito sobre **DemoApplication** e execute novamente aplicação. Abra o navegador e digite **localhost:8080/hello** e observe o resultado;
- 9) **Criação da classe Address**. No painel à esquerda, clique com o botão direito sobre **com.example.demo** e adicione uma nova classe de nome **Address**. Adicione os atributos **cep**, **rua**, **bairro** e **cidade**, como mostrado a seguir:

```

public class Address {
    no usages
    private String cep;
    no usages
    private String rua;
    no usages
    private String bairro;
    no usages
    private String cidade;
}

```

- 10) Gere um construtor para a classe: **botão direito** → **Generate** → **Constructor** e selecione todos os atributos:

```

public class Address {
    1 usage
    private String cep;
    1 usage
    private String rua;
    1 usage
    private String bairro;
    1 usage
    private String cidade;

    no usages
    public Address(String cep, String rua, String bairro, String cidade) {
        this.cep = cep;
        this.rua = rua;
        this.bairro = bairro;
        this.cidade = cidade;
    }
}

```

- 11) Gere os métodos *getters* e *setters*: **botão direito** → **Generate** → **Getter and Setter** e selecione todos os atributos:

```

public class Address {
    3 usages
    private String cep;
    3 usages
    private String rua;
    3 usages
    private String bairro;
    3 usages
    private String cidade;

    no usages
    public Address(String cep, String rua, String bairro, String cidade) {
        this.cep = cep;
        this.rua = rua;
        this.bairro = bairro;
        this.cidade = cidade;
    }

    no usages
    public String getCep() { return cep; }

    no usages
    public void setCep(String cep) { this.cep = cep; }

    no usages
    public String getRua() { return rua; }

    no usages
    public void setRua(String rua) { this.rua = rua; }

    no usages
    public String getBairro() { return bairro; }

    no usages
    public void setBairro(String bairro) { this.bairro = bairro; }

    no usages
    public String getCidade() { return cidade; }

    no usages
    public void setCidade(String cidade) { this.cidade = cidade; }
}

```

- 12) Volte à classe **AddressController** e crie uma lista de endereços como um atributo, identificado pelo nome **addresses**, conforme figura a seguir. Essa lista representará nossa base de dados de endereços (**OBS:** neste exemplo introdutório não será criada uma classe de serviço, não haverá acesso a banco de dados e não envolverá os conceitos de Injeção de Dependência e Inversão de Controle (IoC)).

```

@RestController
public class AddressController {

    no usages
    private final List<Address> addresses = new ArrayList<>(
        Arrays.asList(
            new Address(cep: "38400100", rua: "Floriano Peixoto", bairro: "Centro", cidade: "Uberlândia"),
            new Address(cep: "38400200", rua: "Tiradentes", bairro: "Fundinho", cidade: "Uberlândia"),
            new Address(cep: "38400300", rua: "Lions Clube", bairro: "Osvaldo Rezende", cidade: "Uberlândia")
        )
    );

    no usages
    @GetMapping("/hello")
    public String helloWorld() {
        return "Hello World!";
    }
}

```

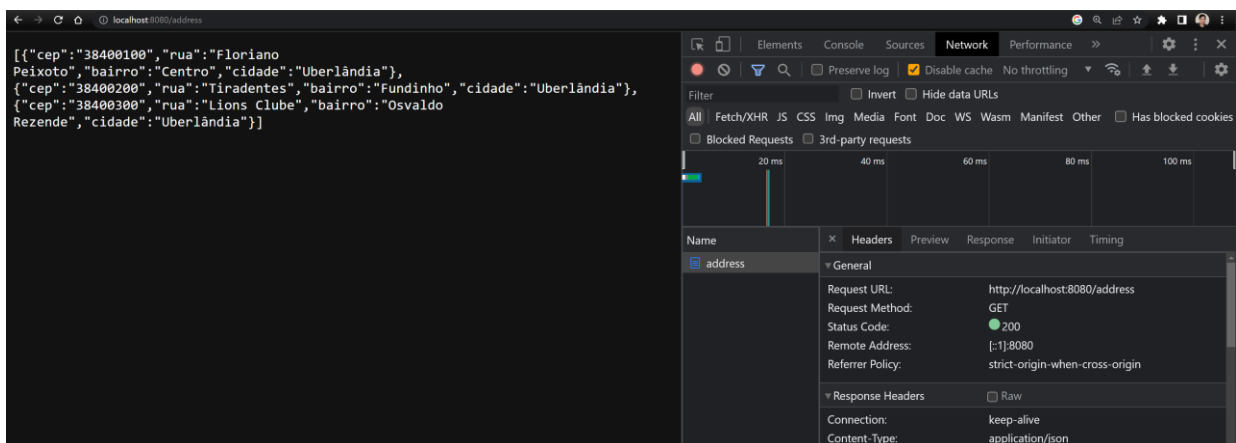
- 13) **Retornando todos os endereços.** Adicione um método de nome **getAddresses** para retornar toda a lista de endereços. Acrescente também a *annotation* **@GetMapping("/address")** para associar as requisições GET utilizando o endereço **"/address"** à chamada do método:

```

no usages
@GetMapping("/address")
public List<Address> getAddresses() {
    return this.addresses;
}

```

- 14) Execute novamente a aplicação, abra o navegador e digite **localhost:8080/address**. Observe que a aplicação retorna uma resposta HTTP contendo os dados de todos os endereços no formato JSON (a conversão do atributo **addresses**, do tipo **List<Address>**, para JSON é feita automaticamente pela aplicação/framework):



The screenshot shows a web browser at `localhost:8080/address` displaying the following JSON response:

```
[{"cep": "38400100", "rua": "Floriano Peixoto", "bairro": "Centro", "cidade": "Uberlândia"}, {"cep": "38400200", "rua": "Tiradentes", "bairro": "Fundinho", "cidade": "Uberlândia"}, {"cep": "38400300", "rua": "Lions Clube", "bairro": "Osvaldo Rezende", "cidade": "Uberlândia"}]
```

The Chrome DevTools Network tab shows a GET request to `http://localhost:8080/address` with a status code of 200. The response headers indicate the content type is `application/json`.

- 15) **Retornando um endereço específico dado o CEP.** Acrescente um método de nome **getAddress** que receba o CEP por parâmetro, busque por ele na lista de endereços e retorne uma resposta HTTP contendo os respectivos dados:

```

@GetMapping("/address/{cep}")
public ResponseEntity<Address> getAddress(@PathVariable String cep) {
    for (Address address : this.addresses)
        if (address.getCep().equals(cep))
            return ResponseEntity.ok(address);

    return ResponseEntity.notFound().build();
}

```

Neste caso é necessário resgatar o CEP direto da URL. Observe que o endereço inserido em **@GetMapping** contém uma variável **{cep}**. A *annotation* **@PathVariable** antes do

parâmetro **cep** do método possibilita que o valor da variável **cep** retirado da URL seja repassado automaticamente ao parâmetro de mesmo nome do método. Observe também que nesse método foi utilizado a classe **ResponseEntity** para possibilitar a construção de uma resposta HTTP em situação de sucesso ou erro. Caso o cep seja localizado, será retornado uma resposta HTTP com código de status 200 e o endereço como conteúdo (body). Caso contrário será retornado uma resposta com código de status 404 (not found);

- 16) Execute novamente a aplicação, abra o navegador e digite **localhost:8080/address/38400-100**. Observe que a aplicação retorna uma string JSON contendo os dados do endereço. Repita a requisição utilizando um cep inexistente e observe o resultado;
- 17) **Adicionando um novo endereço**. Acrescente o método `addAddress`, conforme figura a seguir, para permitir que requisições POST no endereço `/address` contendo um objeto JSON como *payload* possam adicionar novos endereços na lista de endereços:

```
no usages
@PostMapping("/address")
public void addAddress(@RequestBody Address address) {
    this.addresses.add(address);
}
```

Observe que agora foi utilizado a *annotation* **@PostMapping**, uma vez que o objetivo é mapear requisições que utilizarem o método POST. Repare também que foi utilizado a *annotation* **@RequestBody** antes do parâmetro **address** do método. Ela permitirá que a string JSON enviada no corpo da requisição HTTP (payload) seja automaticamente carregada, convertida em um objeto `Address` e repassada ao parâmetro do método;

- 18) Para testar rapidamente o método **addAddress** recomenda-se a utilização de um Cliente HTTP como o Postman, o cliente HTTP do próprio IntelliJ IDEA Ultimate ou o plugin Talend API Tester. Para utilizar o plugin, basta procurar pelo respectivo nome utilizando a opção Extensões do navegador. Veja a seguir um exemplo de requisição POST utilizando o Talend API Tester:

The screenshot displays the Talend API Tester interface for configuring a POST request. At the top, the 'METHOD' is set to 'POST' and the 'URL' is 'http://localhost:8080/address'. Below the URL, the 'QUERY PARAMETERS' section is collapsed. The 'HEADERS' section is expanded, showing a header 'Content-Type' with the value 'application/json'. The 'BODY' section is also expanded, showing a JSON payload: 

```
{
  "cep": "38400-400",
  "rua": "Elisa de Freitas",
  "bairro": "Osvaldo Rezende",
  "cidade": "Uberlândia"
}
```

- 19) Adicione um método na classe para permitir que requisições utilizando o método DELETE possam excluir os dados de um endereço específico dado o seu cep (por exemplo, utilizando uma requisição DELETE ao endereço `/address/38400-100`);
- 20) Para testar os recursos implementados, adicione um arquivo de nome **index.html** na pasta **src/main/resources/static**. O arquivo deve ter o formulário apresentado a seguir. As funcionalidades associadas aos botões devem ser implementadas utilizando requisições assíncronas que acessem as funcionalidades implementadas nos passos anteriores (deve-se utilizar a API Fetch com `async/await`):

## Testando API Restful

CEP

Rua:

Bairro:

Cidade:

Buscar endereço pelo CEP (GET)

Criar novo (POST)

Apagar pelo CEP (DELETE)

Listar todos os endereços

## Entrega

---

Compacte a pasta raiz do projeto e envie pelo Sistema Acadêmico de Aplicação de Testes (SAAT) até a data limite indicada pelo professor em sala de aula.

## Sobre Eventuais Plágios

---

Este é um trabalho individual. Os alunos envolvidos em qualquer tipo de plágio, total ou parcial, seja entre equipes ou de trabalhos de semestres anteriores ou de materiais disponíveis na Internet (exceto os materiais de aula disponibilizados pelo professor), serão duramente penalizados (art. 196 do Regimento Geral da UFU). Todos os alunos envolvidos terão seus **trabalhos anulados** e receberão **nota zero**.