



INSTITUTO FEDERAL  
Triângulo Mineiro  
Campus Uberlândia Centro

# Projeto Backend

# Microserviços e NoSQL

## Aplicação com Spring Boot

Prof. Lucas Montanheiro  
[lucasmontanheiro@iftm.edu.br](mailto:lucasmontanheiro@iftm.edu.br)

# Cenário Atual de Desenvolvimento

- Atualmente, a maioria dos sistemas está sendo desenvolvida utilizando APIs e a arquiteturas de microsserviços.
- Com as APIs, o back-end é totalmente isolado do front-end e, com os microsserviços, os projetos são muito menores, não passando de algumas dezenas ou centenas de arquivos.
- Além disso, evita-se a duplicação de código, já que cada serviço implementa uma funcionalidade específica e pode ser reutilizado por diversas aplicações.

# Cenário Atual de Desenvolvimento

- O Java continua sendo uma das linguagens mais utilizadas para o desenvolvimento de aplicações back-end.
- Isso se deve ao grande grau de maturidade da linguagem e da sua máquina virtual.
- Existem diversas formas de desenvolver microserviços em Java, como utilizar bibliotecas nativas ou alguns frameworks, como o Quarkus e, principalmente, o Spring.

# Cenário Atual de Desenvolvimento

- O Java continua sendo uma das linguagens mais utilizadas para o desenvolvimento de aplicações back-end.
- Isso se deve ao grande grau de maturidade da linguagem e da sua máquina virtual.
- Existem diversas formas de desenvolver microserviços em Java, como utilizar bibliotecas nativas ou alguns frameworks, como o Quarkus e, principalmente, o Spring.

# Aplicação a ser desenvolvida

- Durante a disciplina vamos usar o Spring Boot, com diversas funcionalidades do Spring Web, Spring Data e o Spring Cloud;
- Começaremos pelo uso do Spring Boot e seguiremos no desenvolvimento de três microsserviços;

# Aplicação a ser desenvolvida

- A aplicação consiste em:
  - Um serviço para cadastro de cliente;
  - Um para cadastro de produtos;
  - Um serviço para compras.
- Para a execução das compras, os clientes e os produtos devem ser validados, o que requer a comunicação entre os serviços.

# Framework Spring

- O Spring é um framework Java que possui uma grande quantidade de projetos, como o Spring Boot, o Spring Data e o Spring Cloud, que podem ser utilizados em conjunto ou não.
- É também possível utilizar outros frameworks com o Spring e até mesmo as bibliotecas do Enterprise Java Beans (EJB).
- Spring é bastante antigo — sua primeira versão foi publicada em 2002 — e é um projeto robusto e estável.

# Spring Boot

- O Spring Boot é uma forma de criar aplicações baseadas no framework Spring de forma simples e rápida.
- Nelas, já existe um contêiner web, que pode ser o Tomcat ou o Jetty, e a aplicação é executada com apenas um **run**, diferentemente de quando é necessário primeiro instalar e configurar um contêiner, gerar um arquivo WAR (*Web Application Resource*) e, por fim, implantá-lo no contêiner.



# Spring Data

- O Spring Data é um projeto do Spring para facilitar a criação da camada de persistência de dados.
- Esse projeto tem abstrações para diferentes modelos de dados, como banco de dados relacionais e não relacionais, como o MongoDB e o Redis.

# Spring Cloud

- O Spring Cloud é um projeto que disponibiliza diversas funcionalidades para a construção de sistemas distribuídos, como gerenciamento de configuração, serviços de descoberta, proxys, eleição de líderes etc.
- É bastante simples usá-lo junto a projetos Spring Boot.

# Spring Cloud

- Durante nosso projeto, vamos usá-lo para a criação de um ***api-gateway*** que será uma aplicação que fica na frente de todos os microsserviços recebendo todas as requisições da aplicação.
- Com isso, podemos centralizar essas requisições, não sendo necessário que um cliente dos serviços conheça o endereço e a porta onde estarão instaladas todas as aplicações.

# Lombok

- O Lombok é uma biblioteca Java para a geração de trechos de código que normalmente são iguais para todas as classes, como os construtores e os métodos **get** e **set** .
- Ela não adiciona nenhuma nova funcionalidade, mas ela aumenta bastante a produtividade dos programadores e facilita a manutenção das aplicações.
- Veremos que apenas adicionando algumas anotações, como **@Getter** e **@Setter**, o Lombok gera uma grande quantidade de código.

# Maven

- Utilizaremos o Maven para a gerência de dependências e também para a construção das imagens do Docker.
- Instalar o Maven é simples nos três SOs:
  - No Linux, basta executar o comando **sudo apt install maven**
  - MacOS o comando: **brew install maven**
  - No Windows, é necessário baixar no site oficial da ferramenta a última versão e descompactar o arquivo, o que criará uma pasta com os arquivos do Maven. Depois, basta adicionar na variável PATH do SO o caminho para a pasta do Maven.

# Docker

- O Docker é uma ferramenta para criar e executar contêineres.
- Durante a disciplina o Docker será usado para criar os contêineres de cada um dos microsserviços e também para executar o banco de dados.
- No Linux, a instalação é um pouco mais complexa e deve ser feita via linha de comando; já no Windows e no Mac, existe a versão Docker for Desktop.

# Criando os primeiros serviços

- Vamos criar uma aplicação que simula um e-commerce com três microsserviços:
  - um para o cadastro de usuários (**user-api**),
  - um para o cadastro de produtos (**product-api**)
  - um serviço de compras (**shopping-api**).
- Inicialmente eles funcionarão de forma independente e, depois, trabalharemos na comunicação entre os microsserviços.

# Estrutura das APIs

- A **user-api** será responsável por manter os dados dos usuários da aplicação. Alguns serviços que estarão disponíveis nela serão a criação e exclusão de usuários e a validação da existência de um usuário.
- A **product-api** conterá todos os produtos cadastrados em nossa aplicação e terá serviços como cadastrar produtos e recuperar informações de um produto, como preço e descrição.
- A **shopping-api** será utilizada para o cadastro de compras na aplicação. Assim, para realizar uma compra, a **shopping-api** receberá informações sobre o usuário que está fazendo a compra e uma lista de produtos, e todos esses dados precisarão ser validados na **user-api** e na **product-api**.

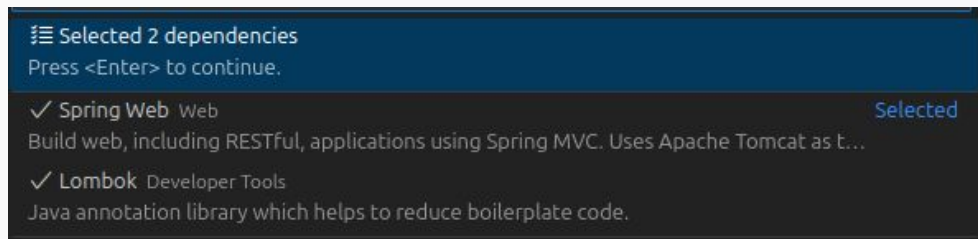


# Estrutura das APIs

- O desenvolvimento dos três microserviços terá o mesmo padrão, que são as camadas ***Controller***, ***Service*** e ***Repository***.
- Teremos as entidades que representam os dados dos banco de dados e os *Data Transfer Objects* (DTO), que são as classes utilizadas para receber e enviar informações entre os microserviços e também para o front-end.

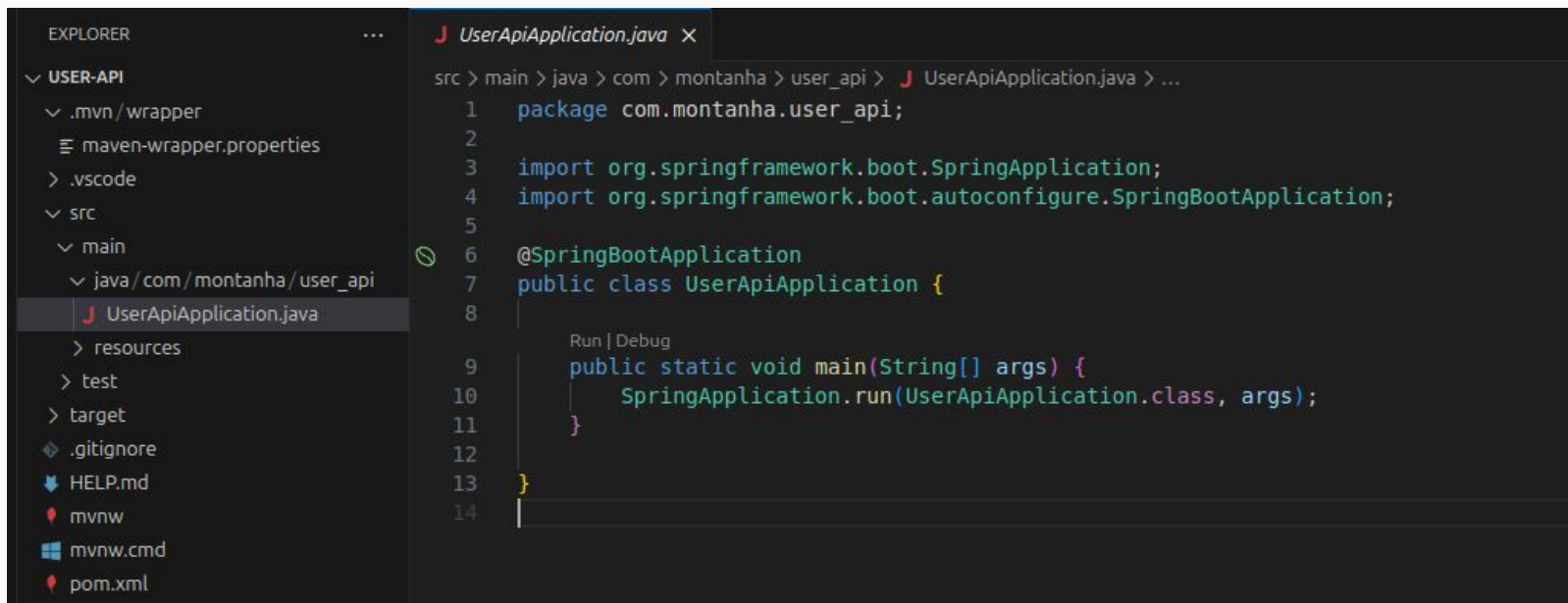
# Hello World com Spring Boot

- Vamos iniciar um novo projeto Java no Visual Studio Code do tipo Spring Boot na versão 3.3.4;
- Utilizando Maven;
- Serviço: user-api
- Java 17;
- Adicione as dependências:



# Hello World com Spring Boot

- A IDE deverá criar um projeto com uma estrutura dessa forma:



The screenshot shows an IDE with a project structure on the left and a code editor on the right. The project structure in the EXPLORER view is as follows:

- USER-API
  - .mvn/wrapper
    - maven-wrapper.properties
  - .vscode
  - src
    - main
      - java/com/montanha/user\_api
        - UserApiApplication.java
    - resources
    - test
    - target
    - .gitignore
    - HELP.md
    - mvnw
    - mvnw.cmd
    - pom.xml

The code editor displays the content of `UserApiApplication.java`:

```
src > main > java > com > montanha > user_api > UserApiApplication.java > ...
1  package com.montanha.user_api;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6  @SpringBootApplication
7  public class UserApiApplication {
8
9      Run | Debug
10     public static void main(String[] args) {
11         SpringApplication.run(UserApiApplication.class, args);
12     }
13 }
14
```

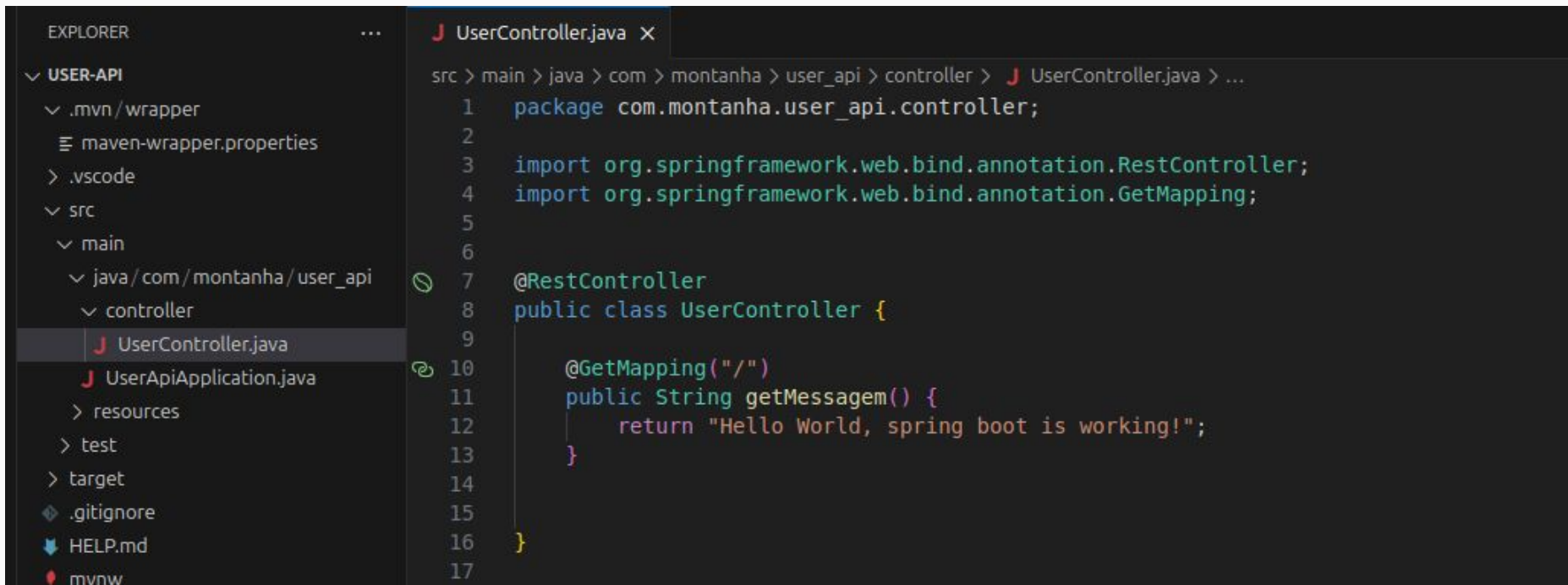
# Hello World com Spring Boot

- Para criar o primeiro serviço, será criada uma nova classe **UserController.java** e ela será criada dentro da pasta **controller**.
- A anotação **@RestController** vai permitir a criação de métodos que serão chamados via web utilizando o protocolo HTTP.
- Uma rota back-end é um caminho no servidor que recebe uma requisição HTTP de um usuário. Essa rota pode receber informações e retornar uma resposta.

# Hello World com Spring Boot

- Podemos criar um método simples que, quando chamado pelo browser, retornará uma mensagem para o usuário.
- Esse método deve ser anotado com **@GetMapping** e, como parâmetro, deve ser passado o caminho para acessar esse método.
- Vamos fazer um método chamado `getMensagem` que retornará uma mensagem simples para o usuário.

# Hello World com Spring Boot

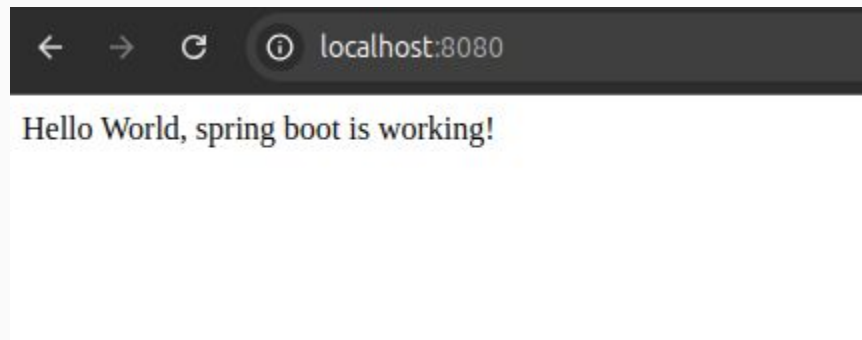


The image shows a VS Code editor interface. On the left, the 'EXPLORER' sidebar displays the project structure for 'USER-API'. The file tree includes: .mvn/wrapper, maven-wrapper.properties, .vscode, src, main, java/com/montanha/user\_api, and controller. The file 'UserController.java' is selected and highlighted. The main editor area shows the code for 'UserController.java'. The code defines a REST controller with a single GET endpoint that returns a 'Hello World' message.

```
src > main > java > com > montanha > user_api > controller > J UserController.java > ...  
1  package com.montanha.user_api.controller;  
2  
3  import org.springframework.web.bind.annotation.RestController;  
4  import org.springframework.web.bind.annotation.GetMapping;  
5  
6  
7  @RestController  
8  public class UserController {  
9  
10     @GetMapping("/")  
11     public String getMessagem() {  
12         return "Hello World, spring boot is working!";  
13     }  
14  
15  
16 }  
17
```

# Hello World com Spring Boot

- Execute o projeto e verifique o resultado no navegador:



# Padrão JSON

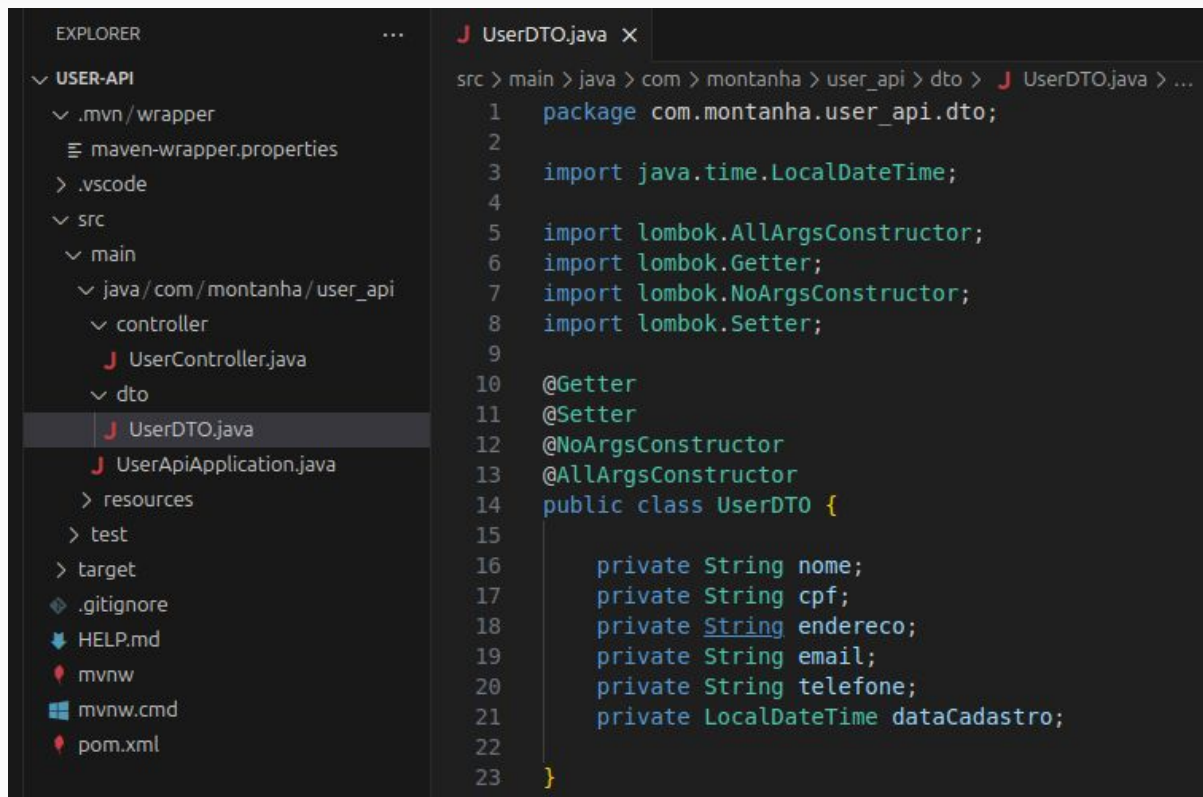
- A maioria dos serviços retorna dados mais complexos, principalmente no formato JSON (*JavaScript Object Notation*), que hoje é o principal padrão para a troca de mensagem entre aplicações.
- Esse formato é utilizado por praticamente todas as linguagens e serve tanto para a comunicação entre os serviços do back-end quanto para a integração do back-end com o front-end.
- A tradução para esse formato é feita automaticamente pelo Spring, bastando que o método do Controller retorne um objeto Java.



# Criando UserDTO

- Vamos criar a classe **UserDTO.java** dentro da pasta **dto**.
- Usaremos também as anotações **@Getter** , **@Setter** , **@NoArgsConstructor** e **@AllArgsConstructor** .
- Essas anotações são do Lombok e indicam que devem ser gerados respectivamente os métodos **get** , os **set** , um construtor vazio e um construtor com todos os argumentos da classe.
- Com isso, não precisamos escrever nenhum desses métodos.

# Criando UserDTO



```
src > main > java > com > montanha > user_api > dto > J UserDTO.java > ...
1  package com.montanha.user_api.dto;
2
3  import java.time.LocalDateTime;
4
5  import lombok.AllArgsConstructor;
6  import lombok.Getter;
7  import lombok.NoArgsConstructor;
8  import lombok.Setter;
9
10 @Getter
11 @Setter
12 @NoArgsConstructor
13 @AllArgsConstructor
14 public class UserDTO {
15
16     private String nome;
17     private String cpf;
18     private String endereco;
19     private String email;
20     private String telefone;
21     private LocalDateTime dataCadastro;
22
23 }
```

# Classes de DTO

- Todas as classes que tiverem essa sigla DTO (*Data Transfer Objects*) no nome serão utilizadas como o retorno dos métodos da camada **Controller**.
- Elas possuem apenas os atributos da classe que estamos criando, como nome, CPF e endereço dos usuários da aplicação, e os métodos **get** e **set** (que, nesse caso, foram gerados com o Lombok).

# Ajustando a UserController

- Antes de iniciar a implementação das rotas, vamos criar um método que cria e retorna uma lista com objetos do tipo **UserDTO**.
- Essa lista será utilizada em mais de um método, por isso é estática e, para inicializá-la apenas uma vez, vamos criar um método chamado **initiateList** que insere três usuários na lista.
- Vamos usar a notação **@PostConstruct**, que faz com que ele seja executado logo depois que o contêiner inicializa a classe **UserController**.
- Essa anotação pode ser utilizada em todas as classes gerenciadas pelo Spring, como **Controllers** e **Services**.

# Ajustando a UserController

- Vamos também adicionar a anotação **@RequestMapping("/user")** na classe.
- Essa anotação indica que todas as rotas que serão implementadas neste controlador possuirão o path iniciado com **/user**.
- Lembre-se de remover a classe **getMessage()** e a anotação **@GetMapping("/")**.
- Vamos adicionar também um novo **@GetMapping** para retornar os dados.

# Ajustando a UserController

```
package com.montanha.user_api.controller;

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

import com.montanha.user_api.dto.UserDTO;

import jakarta.annotation.PostConstruct;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

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

@RestController
@RequestMapping("/user")
public class UserController {

    public static List<UserDTO> usuarios = new ArrayList<>();

    @PostConstruct
    public void initiateList() {

        UserDTO userDTO = new UserDTO();
        userDTO.setNome("Eduardo");
        userDTO.setCpf("123");
        userDTO.setEndereco("Rua A");
        userDTO.setEmail("eduardo@email.com");
        userDTO.setTelefone("1234-5678");
        userDTO.setDataCadastro(LocalDate.now());

        UserDTO userDTO2 = new UserDTO();
        userDTO2.setNome("Luiz");
        userDTO2.setCpf("456");
        userDTO2.setEndereco("Rua B");
        userDTO2.setEmail("luiz@email.com");
        userDTO2.setTelefone("1234-5678");
        userDTO2.setDataCadastro(LocalDate.now());

        UserDTO userDTO3 = new UserDTO();
        userDTO3.setNome("Bruna");
        userDTO3.setCpf("678");
        userDTO3.setEndereco("Rua C");
        userDTO3.setEmail("bruna@email.com");
        userDTO3.setTelefone("1234-5678");
        userDTO3.setDataCadastro(LocalDate.now());

        usuarios.add(userDTO);
        usuarios.add(userDTO2);
        usuarios.add(userDTO3);

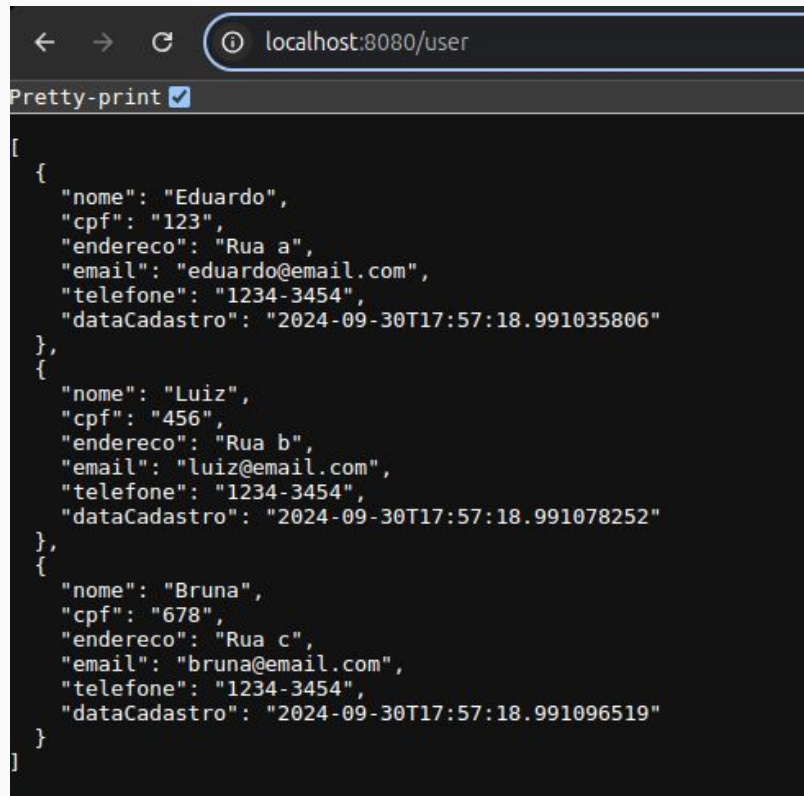
    }

    @GetMapping
    public List<UserDTO> getUsers() {
        return usuarios;
    }
}
```

# Ajustando a UserController

- Agora, um método do **Controller** pode retornar a lista de usuários que foi criada no método **initiateList** .
- Ele se chama **getUsers** e recebe a anotação **@GetMapping**, que indica que essa rota retornará dados do servidor.
- O endereço dessa rota será **http://localhost:8080/user** , pois o **/user** foi definido na classe com a anotação **@RequestMapping** .

# Ajustando a UserController



```
localhost:8080/user
Pretty-print ☒

[
  {
    "nome": "Eduardo",
    "cpf": "123",
    "endereco": "Rua a",
    "email": "eduardo@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2024-09-30T17:57:18.991035806"
  },
  {
    "nome": "Luiz",
    "cpf": "456",
    "endereco": "Rua b",
    "email": "luiz@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2024-09-30T17:57:18.991078252"
  },
  {
    "nome": "Bruna",
    "cpf": "678",
    "endereco": "Rua c",
    "email": "bruna@email.com",
    "telefone": "1234-3454",
    "dataCadastro": "2024-09-30T17:57:18.991096519"
  }
]
```



# Ajustando a UserController

- Note que o JSON tem exatamente os mesmos valores dos objetos da classe **UserDTO**.
- Com isso aprendemos as principais ideias de como desenvolver uma API com o Spring Boot, que são a criação de um **Controller** e os métodos que retornam os dados no formato JSON.
- Agora vamos desenvolver os serviços para manipular a lista de usuários.