

# SpringBoot

## Introdução



Professores:  
André Flores dos Santos.  
Cássio Gamarra.

## Mas, o que é Spring Boot?

O **Spring Boot é um framework Java open source** que tem como objetivo facilitar o processo de criação e configuração em aplicações Java. Consequentemente, ele traz mais agilidade para o processo de desenvolvimento, uma vez que devs conseguem **reduzir o tempo gasto com as configurações iniciais**.

Com o Spring Boot conseguimos abstrair e facilitar a configuração de, por exemplo:

- Servidores;
- Gerenciamento de dependências;
- Configurações de bibliotecas;
- Métricas & health checks (testes);
- Entre outros!

## Como o Spring Boot funciona?

Para realizar todo esse processo o Spring Boot utiliza um conceito chamado **convenção sobre configuração**.

Mas o que isso significa? Significa que é uma ferramenta que decide para você a melhor forma de se fazer algo. É o que chamamos de ferramenta opinativa, ela toma as decisões no nosso lugar baseado em convenções, aplicando configurações padrões e facilitando o trabalho.

No entanto ela **não é inflexível** e ainda permite uma configuração diferente da *default* caso o usuário assim deseje.

Por exemplo, você pode alterar para que ele utilize o Jetty como servidor ao invés do Tomcat que é a configuração padrão.

Uma das maiores vantagens que o Spring Boot trouxe ao desenvolvimento é que toda essa configuração não necessita mais ser realizada pelos temidos XMLs, embora ele ainda suporte esse tipo de configuração. A maior parte da **configuração pode ser feita de forma programática** via anotações.

O Spring Boot é composto por vários módulos que ajudam nesse processo. Alguns deles são:

## Spring Boot

É o módulo principal que ajuda na configuração e integração dos outros módulos.

## Spring Boot Starters

Os starters são dependências que agrupam outras dependências com um propósito em comum. Dessa forma, somente uma configuração é realizada no seu gerenciador de dependências.

Por exemplo, o spring-boot-starter-amqp, é um starter que permite a construção de soluções de mensageria baseadas em AMQP e RabbitMQ.

Ao realizar a configuração no meu gerenciador de dependência se define somente o starter:

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

O Spring Boot é composto por vários módulos que ajudam nesse processo. Alguns deles são:

Alguns exemplos de starters disponíveis:

- Spring Boot Starter Web:** Auxilia na construção de aplicações web trazendo já disponíveis para uso Spring MVC, Rest e o Tomcat como servidor.

- Spring Boot Starter Test:** Contém a maioria das dependências necessárias para realizar testes da sua aplicação: Junit, AssertJ, Hamcrest, Mockito, entre outros

- Spring Boot Starter Data JPA:** Facilita a construção da nossa camada de persistência, ajudando na abstração do nosso banco de dados provendo uma série de facilidades para criação de repositories, escrita de queries, entre outros. Como podem ver, reduzem o número de dependências adicionadas, deixando meu arquivo muito mais limpo.

## Spring Boot Autoconfigure

Como dito anteriormente o **Spring Boot** trabalha de forma **opinativa, tomando decisões para você.**

Mas baseado em que? Essas decisões padrões são baseadas através do conteúdo do seu *classpath*.

O *Autoconfigure* é responsável por ler este conteúdo e realizar as configurações necessárias para que a aplicação funcione. É ele quem gerencia todo o processo de configuração da aplicação.

## Spring Boot Actuator

O Spring Boot Actuator é uma ferramenta que permite monitorar e gerenciar as aplicações implantadas. Dentre os recursos disponibilizados temos:

- **Métricas:** Obtém e disponibiliza diversos dados da nossa aplicação, como por exemplo, espaço em disco, memória, tempo de resposta etc.
- **Logging:** Facilita o acesso ao arquivo de log da aplicação por meio de um *endpoint* específico.
- **HealthChecks:** Disponibiliza *endpoints* de *health checks*.
- **Informações da Aplicação:** Permite a disponibilização de informações da aplicação. Por exemplo, versão, informações do git etc.



## Spring Boot Test

O Spring Boot Test contém funcionalidades úteis e anotações que facilitam e ajudam a testar sua aplicação.

## Spring Boot Devtools

**Spring Boot Devtools** é um conjunto de funcionalidades que ajuda o trabalho de qualquer dev. Como, por exemplo, restart automático da aplicação quando ocorre alguma mudança no código.



## Spring Tool Suite

O Spring nos fornece uma IDE totalmente customizada para o desenvolvimento de aplicações do ecossistema spring:

- o **Spring Tool Suite (STS)**.

O STS é uma IDE baseada em Eclipse que já vem com algumas funcionalidades facilitadoras para projetos Spring.

**[download no site oficial.](#)**

Além da IDE baseada em Eclipse, o STS já está disponível como plugin para VSCode.

# Spring Initializr

E para facilitar a criação de aplicações utilizando outras IDEs a Spring disponibilizou o **Spring Initializr**.  
Permite a criação de projetos Spring Boot de forma facilitada.

The screenshot displays the Spring Initializr web interface. At the top, there is a hamburger menu icon, the Spring Initializr logo, and a settings icon. The main content area is divided into several sections:   
1. **Project**: Includes radio buttons for 'Maven Project' (selected) and 'Gradle Project'.   
2. **Language**: Includes radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'.   
3. **Spring Boot**: Includes radio buttons for versions '2.5.0 (SNAPSHOT)', '2.5.0 (RC1)', '2.4.6 (SNAPSHOT)', '2.4.5' (selected), '2.3.11 (SNAPSHOT)', and '2.3.10'.   
4. **Project Metadata**: Contains input fields for 'Group' (filled with 'com.example'), 'Artifact' (filled with 'demo'), and 'Name' (filled with 'demo').   
5. **Dependencies**: A section with a button 'ADD DEPENDENCIES... CTRL + B' and the text 'No dependency selected'.   
At the bottom, there is a 'Description' field filled with 'Demo project for Spring Boot' and three large buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'. Social media icons for GitHub and Twitter are visible on the left side of the bottom bar.

Através dele definimos nome do projeto, pacotes, dependências (starters do spring e outros projetos), linguagem (Java, Groovy ou Kotlin).

Uma vez definido é só clicar no botão *Generate* e o projeto será criado, gerando um zip pronto para ser importado na IDE de sua preferência.

## **Como Instalar o Spring Boot no Eclipse**

<https://pt.wikihow.com/Instalar-o-Spring-Boot-no-Eclipse>

<https://www.youtube.com/watch?v=hytIn5-Wws4>

Site do SpringBoot:

<https://spring.io/projects>

<https://spring.io/>

<https://spring.io/quickstart>

## Exercício 01:

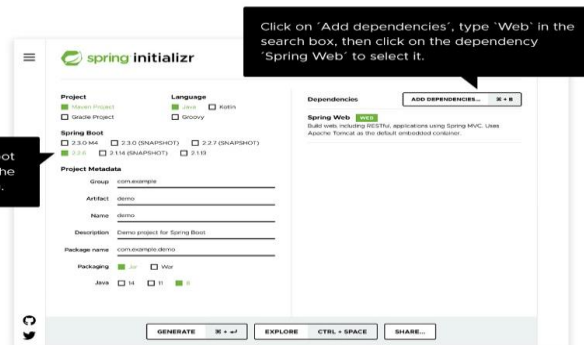
Hello world springboot

<https://spring.io/quickstart>

Atenção! Quando for rodar programas com SpringBoot que utiliza o Apache Tomcat como servidorweb desabilitar o servidor apache do pacote xampp se estiver ativado para não dar conflitos na porta 8080.

### Etapa 1: iniciar um novo projeto Spring Boot

Usar [start.spring.io](https://start.spring.io) para criar um projeto "web". Na caixa de diálogo "Dependencies", procure e adicione a dependência "web" conforme mostrado na captura de tela. Clique no botão "Gerar", baixe o zip e descompacte-o em uma pasta no seu computador.



## Etapa 2: adicione seu código

Abra o projeto em seu IDE e localize o `DemoApplication.java` arquivo na `src/main/java/com/example/demo` pasta.

Agora altere o conteúdo do arquivo adicionando o método extra e as anotações mostradas no código abaixo.

Você pode copiar e colar o código ou apenas digitá-lo.

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {

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

    @GetMapping("/hello")
    public String hello(@RequestParam(value = "name", defaultValue = "World") String name) {
        return String.format("Hello %s!", name);
    }
}
```

CÓPIA DE

O `hello()` método que adicionamos foi projetado para receber um parâmetro `String` chamado `name` e, em seguida, combinar esse parâmetro com a palavra "Hello" no código. Isso significa que, se você definir seu nome "Amy" na solicitação, a resposta será "Hello Amy". A `@RestController` anotação informa ao Spring que este código descreve um endpoint que deve ser disponibilizado na web. O `@GetMapping("/hello")` diz ao Spring para usar nosso `hello()` método para responder a solicitações que são enviadas para o `http://localhost:8080/hello` endereço. Finalmente, `@RequestParam` está dizendo ao Spring para esperar um `name` valor na solicitação, mas se não estiver lá, ele usará a palavra "World" por padrão.

## Passo 3: Experimente

Vamos construir e executar o programa. Abra uma linha de comando (ou terminal) e navegue até a pasta onde você tem os arquivos do projeto. Podemos construir e executar o aplicativo emitindo o seguinte comando:

**Mac OS/Linux:**

```
./mvnw spring-boot:run
```

**Janelas:**

```
mvnw spring-boot:run
```

Obs: Podemos executar o aplicativo direto no Eclipse dando um run(botão de executar), a mesma tela do próximo slide irá aparecer no console mostrando se deu tudo certo. Porém sempre devemos rodar em um local ou em outro, pois se rodar os dois ao mesmo tempo irá dar problemas nas requisições.



Você deve ver uma saída muito semelhante a esta:

```

demo ./mvnw spring-boot:run --quiet

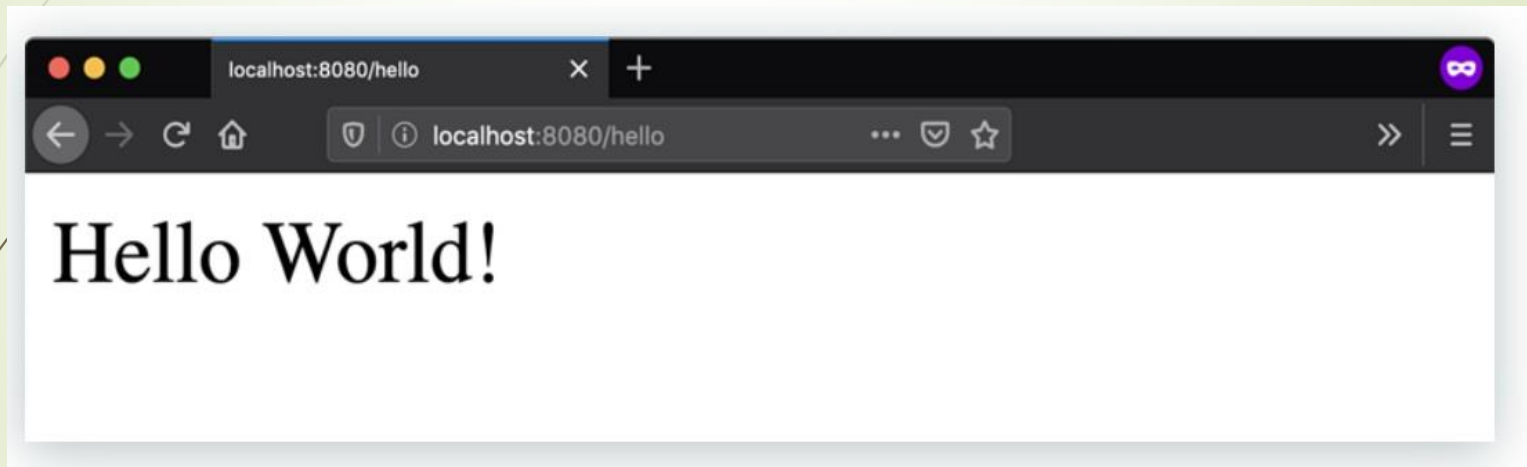
:: Spring Boot :: (v2.2.4.RELEASE)

2020-02-14 16:16:47.746 INFO 4838 --- [main] com.example.demo.DemoApplication : Starting DemoApplication
on Brians-MacBook-Pro.local with PID 4838 (/Users/bclozel/workspace/tmp/demo/target/classes started by bclozel in /Users/bclozel/workspace/tmp/demo)
2020-02-14 16:16:47.748 INFO 4838 --- [main] com.example.demo.DemoApplication : No active profile set, falling back to default profiles: default
2020-02-14 16:16:48.272 INFO 4838 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-02-14 16:16:48.279 INFO 4838 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-02-14 16:16:48.279 INFO 4838 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.30]
2020-02-14 16:16:48.323 INFO 4838 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-02-14 16:16:48.324 INFO 4838 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 532 ms
2020-02-14 16:16:48.438 INFO 4838 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-02-14 16:16:48.533 INFO 4838 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s) 8080 (http) with context path ''
2020-02-14 16:16:48.535 INFO 4838 --- [main] com.example.demo.DemoApplication : Started DemoApplication in 1.006 seconds (JVM running for 1.248)

```

As últimas linhas aqui nos dizem que a spring começou. O servidor Apache Tomcat incorporado do Spring Boot está agindo como um servidor web e está escutando solicitações na [localhost](http://localhost:8080) porta 8080.

Abra seu navegador e na barra de endereços na parte superior, digite <http://localhost:8080/hello>. Você deve obter uma boa resposta amigável como esta:



Agora vamos mandar um nome para trocar pela palavra padrão 'world'

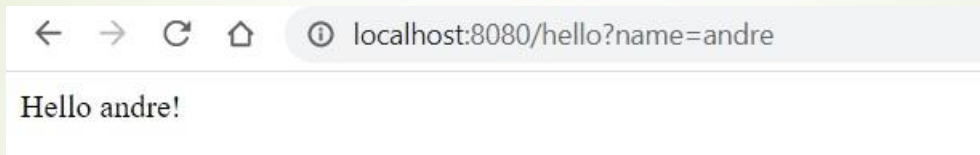
```

3 import org.springframework.boot.SpringApplication;
9
0 @SpringBootApplication
1 @RestController
2 public class DemoApplication {
3
4     public static void main(String[] args) {
5         SpringApplication.run(DemoApplication.class, args);
6         //System.out.println("Hello");
7     }
8
9     @GetMapping("/hello")
0     public String hello(@RequestParam(value = "name", defaultValue = "World") String name) {
1         return String.format("Hello %s!", name);
2     }
3
4 }

```

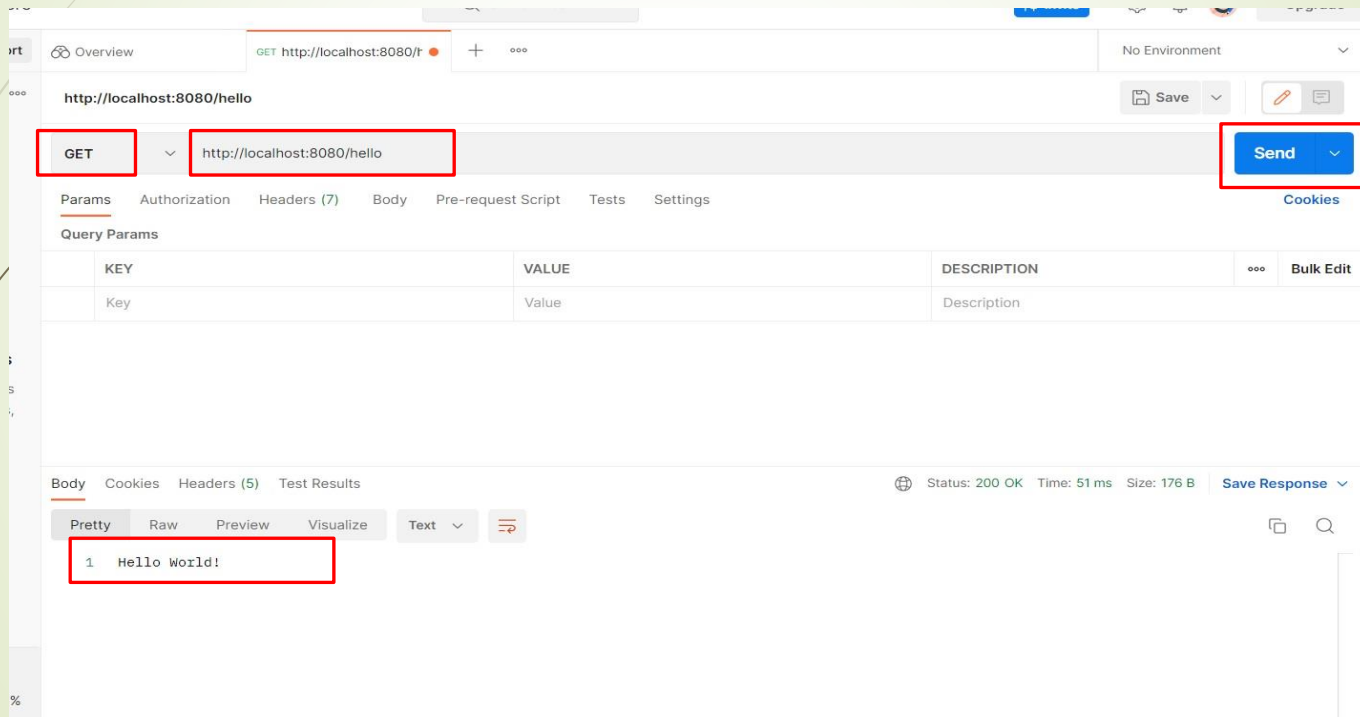
Digitar <http://localhost:8080/hello?name=Andre>

A resposta foi:



Digitar <http://localhost:8080/hello?name=Andre>

Vamos usar a ferramenta Postman: <https://web.postman.co/>



Vamos usar a ferramenta Postman: <https://web.postman.co/>

The screenshot displays the Postman web interface. On the left sidebar, the 'My Workspace' section is active, showing a 'My first collection' with two folders: 'First folder inside collection' and 'Second folder inside collection'. The main area shows a GET request to 'http://localhost:8080/hello?name=Andre'. The 'GET' method and the URL are highlighted with red boxes. A 'Send' button is also highlighted. Below the URL bar, the 'Query Params' tab is selected, showing a table with one parameter: 'name' with the value 'Andre'. The 'name' checkbox is checked, and both the parameter name and value are highlighted with red boxes. At the bottom, the 'Body' tab shows the response 'Hello Andre!', which is also highlighted with a red box. The status bar at the bottom indicates 'Status: 200 OK', 'Time: 4 ms', and 'Size: 176 B'.

Home Workspaces ▾ API Network ▾ Explore

Search Postman

My Workspace New Import Overview GET http://localhost:8080/hello?name=Andre No Environment ▾

Save ▾

GET http://localhost:8080/hello?name=Andre Send ▾

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	Bulk Edit
<input checked="" type="checkbox"/> name	Andre		
Key	Value	Description	

Create a collection for your requests

A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.

Create collection

Body Cookies Headers (5) Test Results

Status: 200 OK Time: 4 ms Size: 176 B Save Response ▾

Pretty Raw Preview Visualize Text ▾

Hello Andre!

# Parte II

## Utilizando JPA+SpringBoot + bd mysql



Vamos acessar o link de explicação direto no site do  
SpringBoot

Link: [https://spring-io.translate.google.com/guides/gs/accessing-data-mysql/?\\_x\\_tr\\_sl=en&\\_x\\_tr\\_tl=pt&\\_x\\_tr\\_hl=pt-BR&\\_x\\_tr\\_pto=sc](https://spring-io.translate.google.com/guides/gs/accessing-data-mysql/?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt-BR&_x_tr_pto=sc)

O que você vai construir?

-Você criará um banco de dados MySQL, construirá um aplicativo Spring e o conectará ao banco de dados recém-criado.

Temos duas formas de iniciar o projeto:

- Usar o `springInitializr`
- Ou Clonar o projeto se estiver disponível (aqui podemos usar)
- Para clonar o projeto `git clone` <https://github.com/spring-guides/gs-accessing-data-mysql.git>
- Após baixar e importar o projeto teremos a pasta '`gs-accessing-data-mysql/complete`' disponível no topo do projeto.

- Se preferir usar o [springInitializr](#).

1. Navegue até <https://start.spring.io> . Este serviço extrai todas as dependências que você precisa para um aplicativo e faz a maior parte da configuração para você.

2. Escolha Gradle ou Maven e o idioma que você deseja usar. Este guia pressupõe que você escolheu Java.

3. Clique em **Dependências** e selecione **Spring Web** , **Spring Data JPA** e **MySQL Driver** .

4. Clique em **Gerar** .

5. Baixe o arquivo ZIP resultante, que é um arquivo de um aplicativo da web configurado com suas escolhas.

## Passo 2

Criar o banco de dados com o nome **db\_example**



Servidor: 127.0.0.1

Base de Dados SQL Estado Contas de utilizador Exportar

### Base de Dados

 Criar base de dados 

db\_example utf8mb4\_general\_ci 

**Criar**

Crie o `application.properties` arquivo

O Spring Boot fornece padrões em todas as coisas. Por exemplo, o banco de dados padrão é `H2`. Conseqüentemente, quando você deseja usar qualquer outro banco de dados, deve definir os atributos de conexão no `application.properties` arquivo.

Crie um arquivo de recurso chamado `src/main/resources/application.properties`, como mostra a listagem a seguir:

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/db_example
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#spring.jpa.show-sql: true
```

## Passo 4

### Criar a entidade 'User' dentro do pacote com.example.accessingdatamysql

Você precisa criar o modelo de entidade, como

`src/main/java/com/example/accessingdatamysql/User.java` mostra a listagem a seguir (em ):

```
package com.example.accessingdatamysql;
import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import
javax.persistence.Id;
@Entity // This tells Hibernate to make a table out of this class
public class User {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;
    private String name;
    private String email;
    public Integer getId() {
        return id; }

    public void setId(Integer id) {
        this.id = id; }
    public String getName() {
        return name; }
    public void setName(String name) {
        this.name = name; }
    public String getEmail() {
        return email; }
    public void setEmail(String email) {
        this.email = email; }
}
```

## Passo 5

### Criar a entidade ‘UserRepository’

#### Crie o Repositório

Você precisa criar o repositório que contém os registros do usuário, como

`src/main/java/com/example/accessingdatamysql/UserRepository.java` mostra a listagem a seguir (em ):

```
package com.example.accessingdatamysql;

import org.springframework.data.repository.CrudRepository;

import com.example.accessingdatamysql.User;

// This will be AUTO IMPLEMENTED by Spring into a Bean called userRepository
// CRUD refers Create, Read, Update, Delete

public interface UserRepository extends CrudRepository<User, Integer> {

}
```

CÓPIA DE

O Spring implementa automaticamente esta interface de repositório em um bean que tem o mesmo nome (com uma mudança no caso — é chamado `userRepository` ).



## Passo 6

### Criar a entidade 'MainController'

#### Criar um controlador

Você precisa criar um controlador para lidar com solicitações HTTP para seu aplicativo, como

`src/main/java/com/example/accessingdatamysql/MainController.java` mostra a listagem a seguir (em ):

```
package com.example.accessingdatamysql;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller // This means that this class is a Controller
@RequestMapping(path="/demo") // This means URL's start with /demo (after Application
path)
public class MainController {
    @Autowired // This means to get the bean called userRepository
    // Which is auto-generated by Spring, we will use it to handle the data
    private UserRepository userRepository;

    @PostMapping(path="/add") // Map ONLY POST Requests
    public @ResponseBody String addNewUser (@RequestParam String name
    , @RequestParam String email) {
        // @ResponseBody means the returned String is the response, not a view name
        // @RequestParam means it is a parameter from the GET or POST request

        User n = new User();
        n.setName(name);
        n.setEmail(email);
        userRepository.save(n);
        return "Saved";
    }

    @GetMapping(path="/all")
    public @ResponseBody Iterable<User> getAllUsers() {
        // This returns a JSON or XML with the users
        return userRepository.findAll();
    }
}
```

CÓPIA DE

## Passo 7

### Criar a entidade 'AccessingDataMysqlApplication'

#### Criar uma classe de aplicativo

Spring Initializr cria uma classe simples para o aplicativo. A listagem a seguir mostra a classe que Initializr criou para este exemplo (em

`src/main/java/com/example/accessingdatamysql/AccessingDataMysqlApplication.java`):

```
package com.example.accessingdatamysql;

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

@SpringBootApplication
public class AccessingDataMysqlApplication {

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

}
```

CÓPIA DE

Para este exemplo, você não precisa modificar a `AccessingDataMysqlApplication` classe.

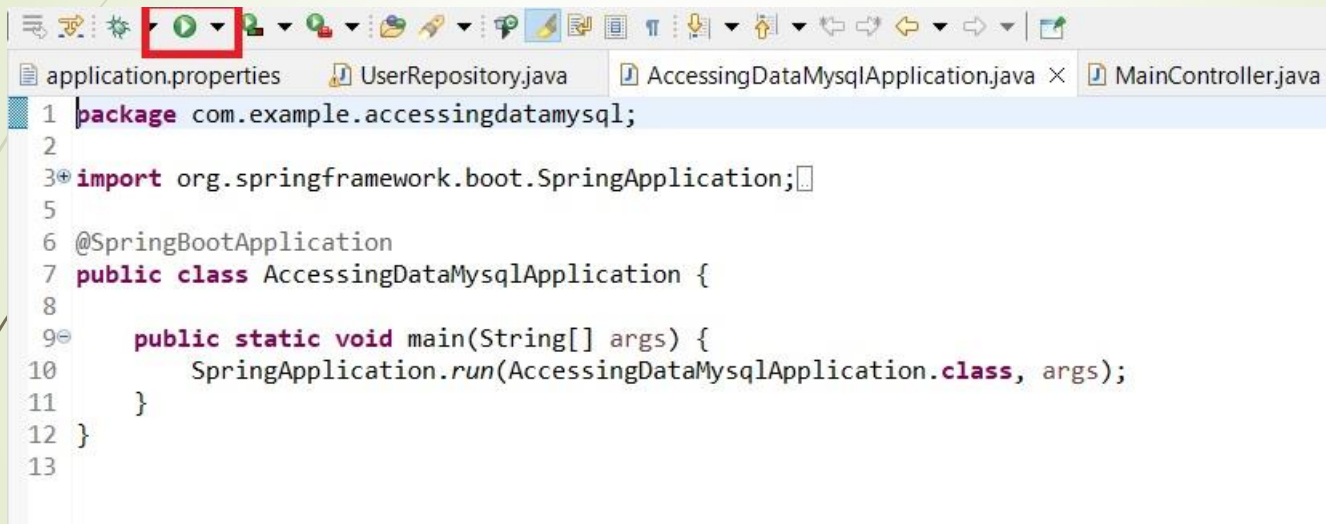
`@SpringBootApplication` é uma anotação de conveniência que adiciona todos os itens a seguir:

- `@Configuration`: marca a classe como uma fonte de definições de bean para o contexto do aplicativo.
- `@EnableAutoConfiguration`: diz ao Spring Boot para começar a adicionar beans com base nas configurações do caminho de classe, outros beans e várias configurações de propriedade. Por exemplo, se `spring-webmvc` estiver no caminho de classe, essa anotação sinaliza o aplicativo como um aplicativo da Web e ativa os principais comportamentos, como configurar um arquivo `DispatcherServlet`.
- `@ComponentScan`: diz ao Spring para procurar outros componentes, configurações e serviços no `com/example` pacote, deixando-o encontrar os controladores.

O `main()` método usa o método do Spring Boot `SpringApplication.run()` para iniciar um aplicativo. Você notou que não havia uma única linha de XML? Também não há `web.xml` arquivo. Este aplicativo da web é 100% Java puro e você não teve que lidar com a configuração de nenhum encanamento ou infraestrutura.

## Passo 8

- Primeiro rodar a aplicação e verificar se não houve erros.
- Verificar se a base de dados foi criada corretamente.



```

1 package com.example.accessingdatamysql;
2
3 import org.springframework.boot.SpringApplication;
4
5
6 @SpringBootApplication
7 public class AccessingDataMysqlApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(AccessingDataMysqlApplication.class, args);
11     }
12 }
13
  
```

## Passo 9

Vamos testar nossa aplicação através de requisições POST e GET com os endereços configurados na aplicação.

-Buscar todos os dados cadastrados:

`localhost:8080/demo/all`

GET localhost:8080/demo/all

localhost:8080/demo/all

GET localhost:8080/demo/all

Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (5) Test Results

Status: 200 OK Time: 123 ms Size: 166 B Save Response

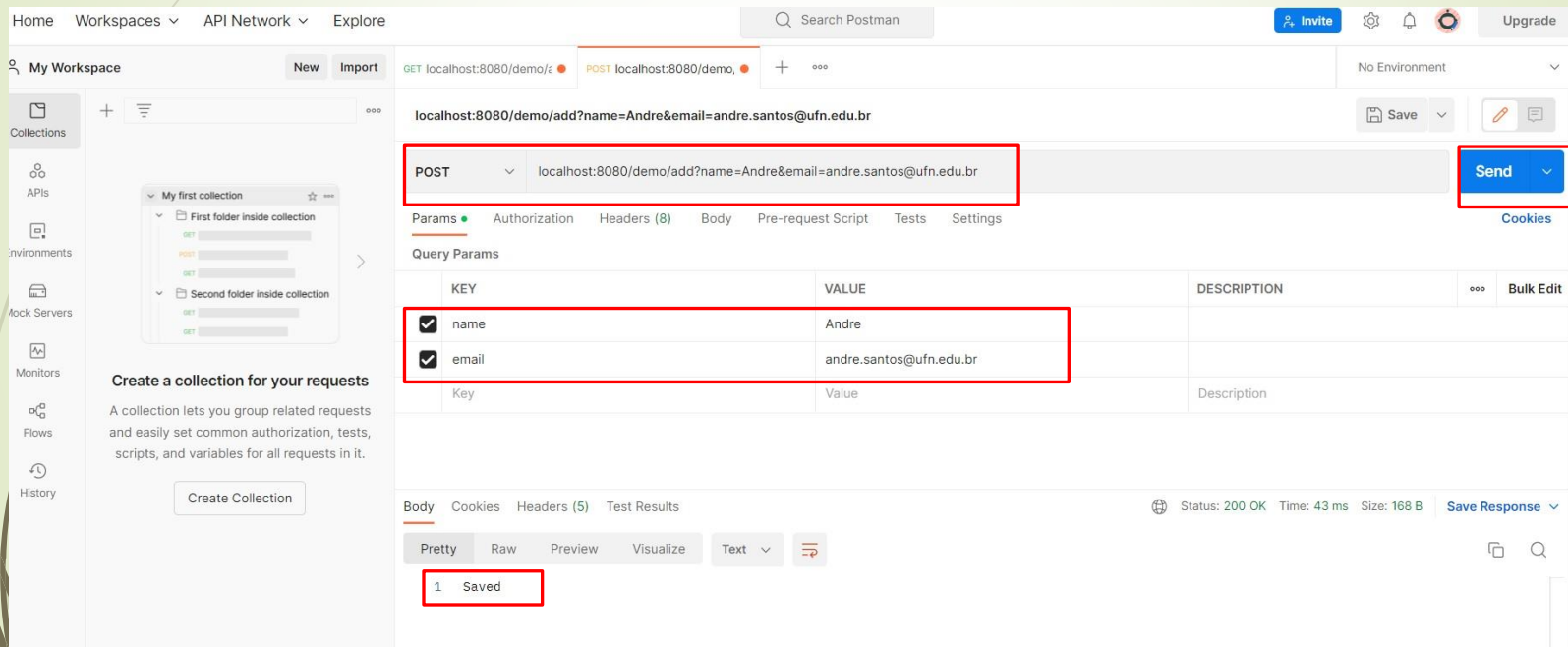
Pretty Raw Preview Visualize JSON

1

## Passo 10

Vamos testar nossa aplicação através de requisições POST e GET com os endereços configurados na aplicação.

- Como não foram cadastrados dados não deve ter retornado nada além de '[ ]', vamos fazer o cadastro de 1 usuário (se preferir faça mais):



The screenshot shows the Postman interface with a POST request configured. The URL is `localhost:8080/demo/add?name=Andre&email=andre.santos@ufn.edu.br`. The request body is empty. The 'Query Params' section shows two parameters: 'name' with value 'Andre' and 'email' with value 'andre.santos@ufn.edu.br'. The 'Send' button is highlighted. The response status is 200 OK.

**Request Configuration:**

- Method: POST
- URL: `localhost:8080/demo/add?name=Andre&email=andre.santos@ufn.edu.br`
- Query Params:
 

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> name	Andre	
<input checked="" type="checkbox"/> email	andre.santos@ufn.edu.br	

**Response:**

Status: 200 OK Time: 43 ms Size: 168 B

Body: Pretty Raw Preview Visualize Text

1 Saved

Atenção com o nome das colunas devem ser igual ao cadastrado no bd para não dar erro!!

Vamos fazer o mesmo processo de consultar os dados no bd novamente, agora com os dados já cadastrados.

The screenshot displays the Postman API client interface. On the left, a sidebar shows a workspace named 'My Workspace' with a 'My first collection' containing two folders: 'First folder inside collection' and 'Second folder inside collection'. A 'Create a collection for your requests' dialog is open, explaining that a collection groups related requests and allows setting common authorization, tests, scripts, and variables. The main area shows a GET request to 'localhost:8080/demo/all'. The 'Send' button is highlighted. Below the request, the 'Query Params' table is empty. The 'Body' tab is selected, showing the response in 'Pretty' format. The response is a JSON object with the following structure:

```
1 {
2   "id": 1,
3   "name": "Andre",
4   "email": "andre.santos@ufn.edu.br"
5 }
```

The status bar at the bottom indicates 'Status: 200 OK', 'Time: 20 ms', and 'Size: 275 B'.



Exercício de aula: implementar agora o exemplo do SpringBoot + JPA com o bando de dados padrão H2, executar o projeto e testar se tudo deu ok conforme o tutorial encontrado no link abaixo.

<https://spring-io.translate.goog/guides/gs/accessing-data-jpa/? x tr sl=en& x tr tl=pt& x tr hl=pt-BR& x tr pto=sc>



## Referências:

<https://www.zup.com.br/blog/spring-boot>

<https://spring.io/tools>

<https://start.spring.io/>