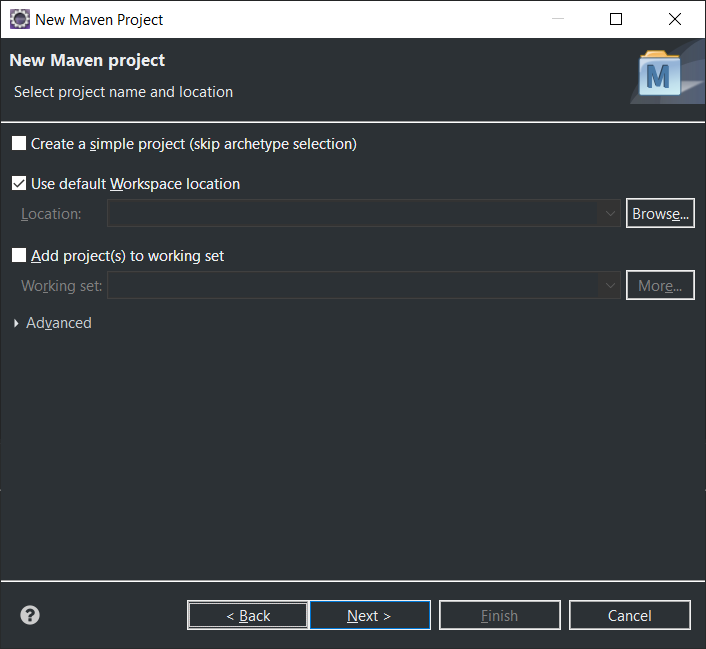
**Spring Boot REST API**

Vamos construir uma API com Spring Boot e testar diversos recursos como Hibernate, Postman, Jquery, JavaScript, HTML/CSS/Bootstrap e Ajax.

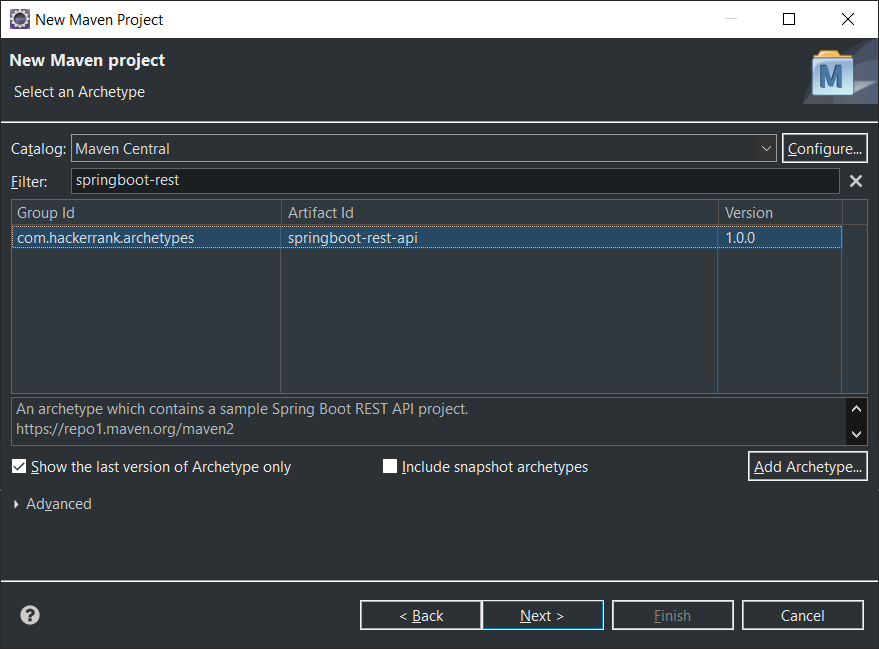
Vamos começar construindo um projeto Maven com Spring Boot assim:

File -> New -> Maven Project

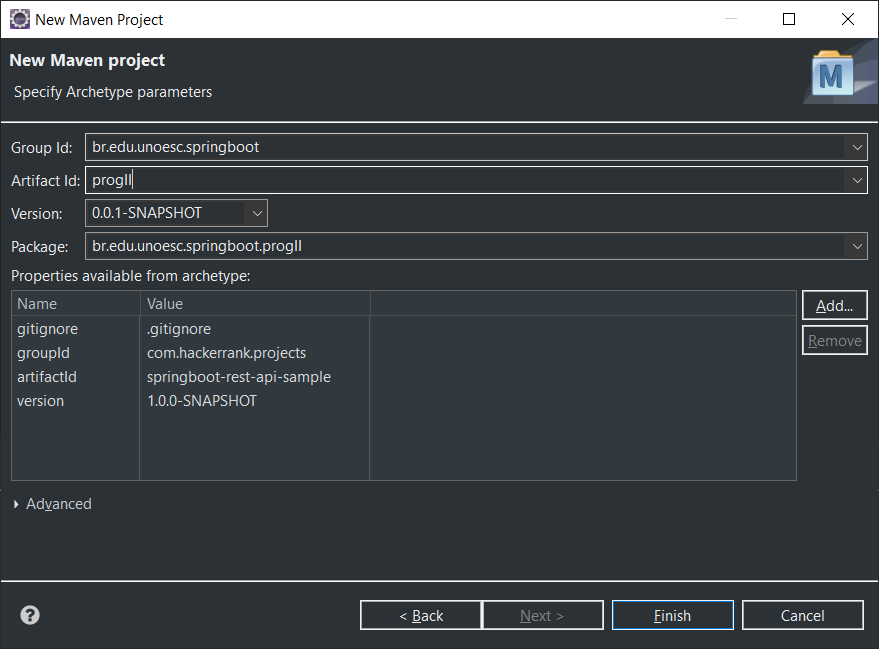
A tela inicial de configuração será aberta e vamos apenas escolher Next



Em seguida teremos a segunda tela onde escolheremos em filter: **springboot-rest**



Na tela seguinte informaremos Group Id e Artifact Id assim:

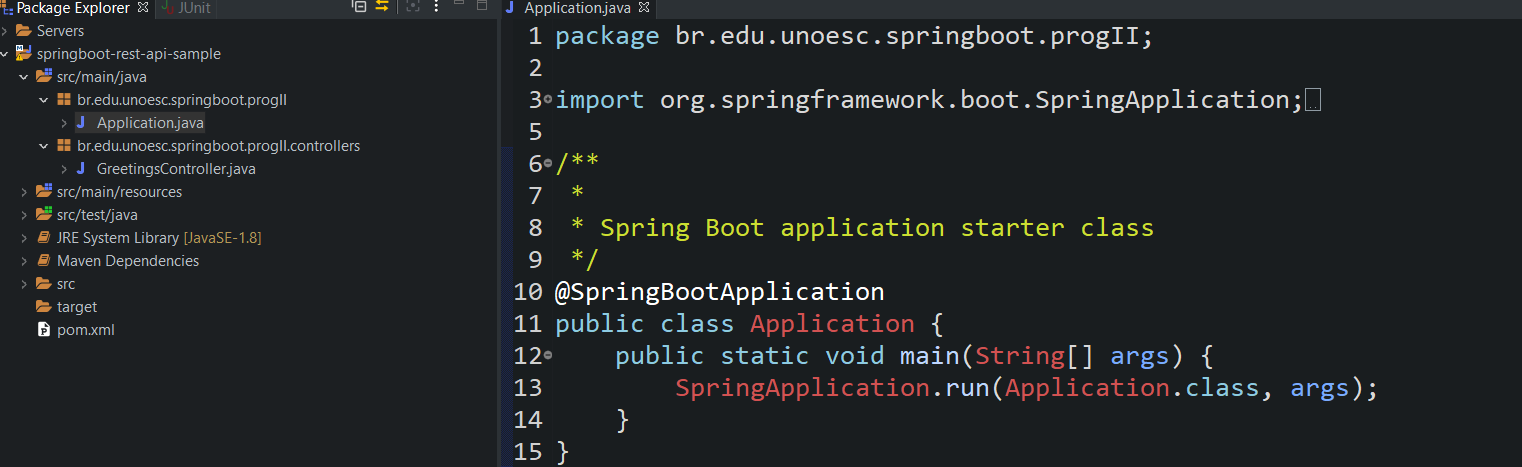


Ao finalizar, um projeto Maven com o Spring Boot será criado e as bibliotecas necessárias serão carregadas.

Ao abrir o projeto teremos um arquivo chamado Application.java e dentro dele teremos a seguinte anotação:

@SpringBootApplication

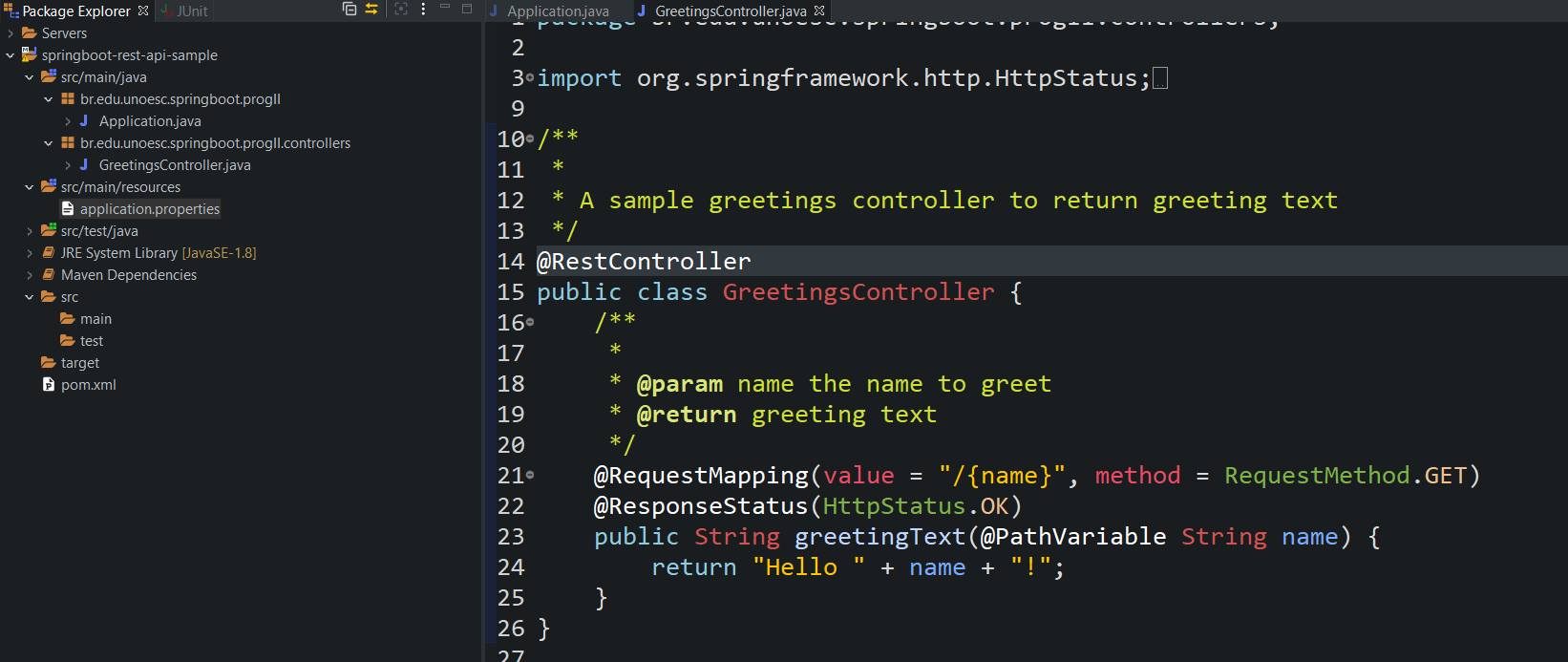
Essa anotação informa a partir de que classe e método a aplicação irá rodar e inicializará o projeto com as configurações necessárias.



Teremos também o arquivo GreetingsController.java e nele a anotação

@RestController

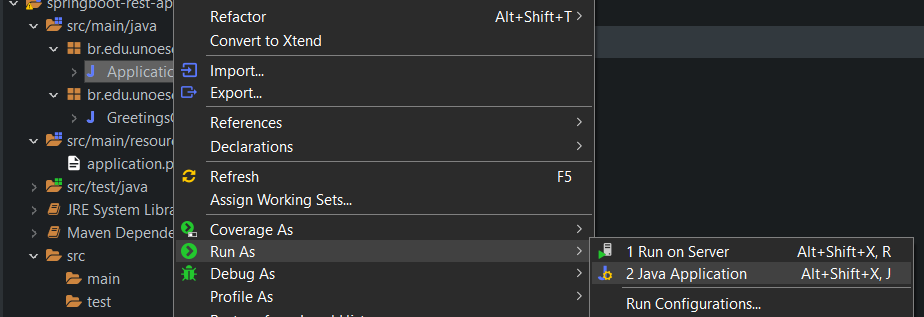
que interceptará todas as requisições que forem mapeadas e vierem pelo navegador, tanto com verbos POST, GET, DELETE, etc



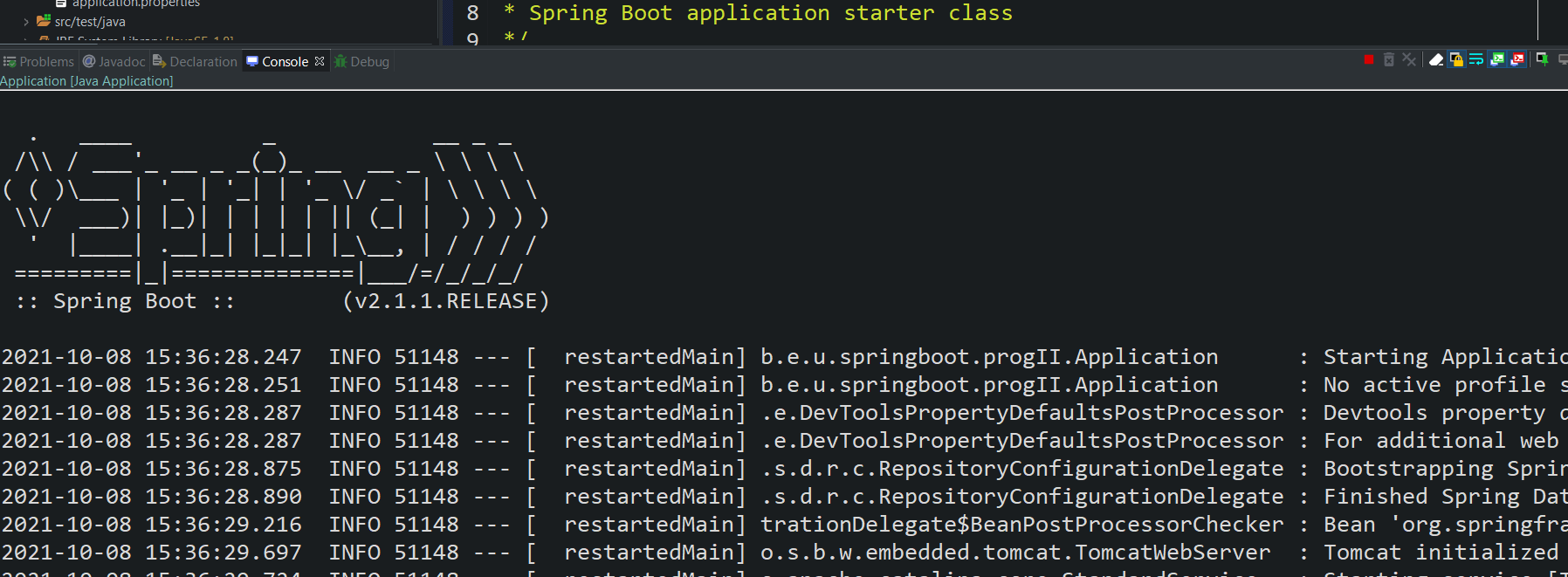
O controller, portanto, interceptará as requisições da aplicação e a partir daí é que os demais métodos serão executados.

O primeiro método que testaremos já está construído e podemos utilizar para testar.

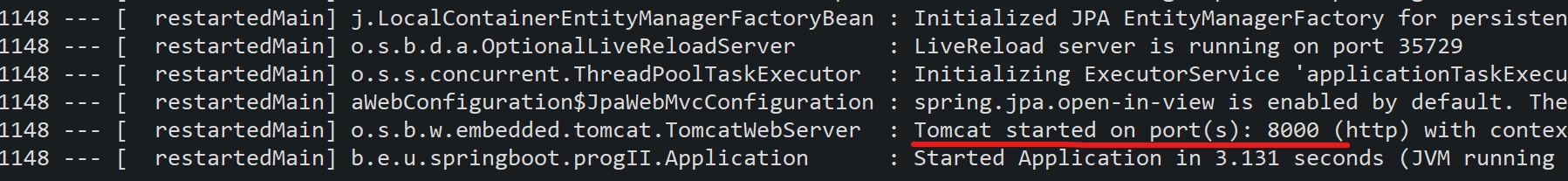
Para rodar o primeiro teste vamos selecionar **Application.java** e com o botão da direita vamos executar como um Java Application.



Em seguida o Spring vai rodar e vai subir a aplicação.



Podemos conferir em que porta está rodando

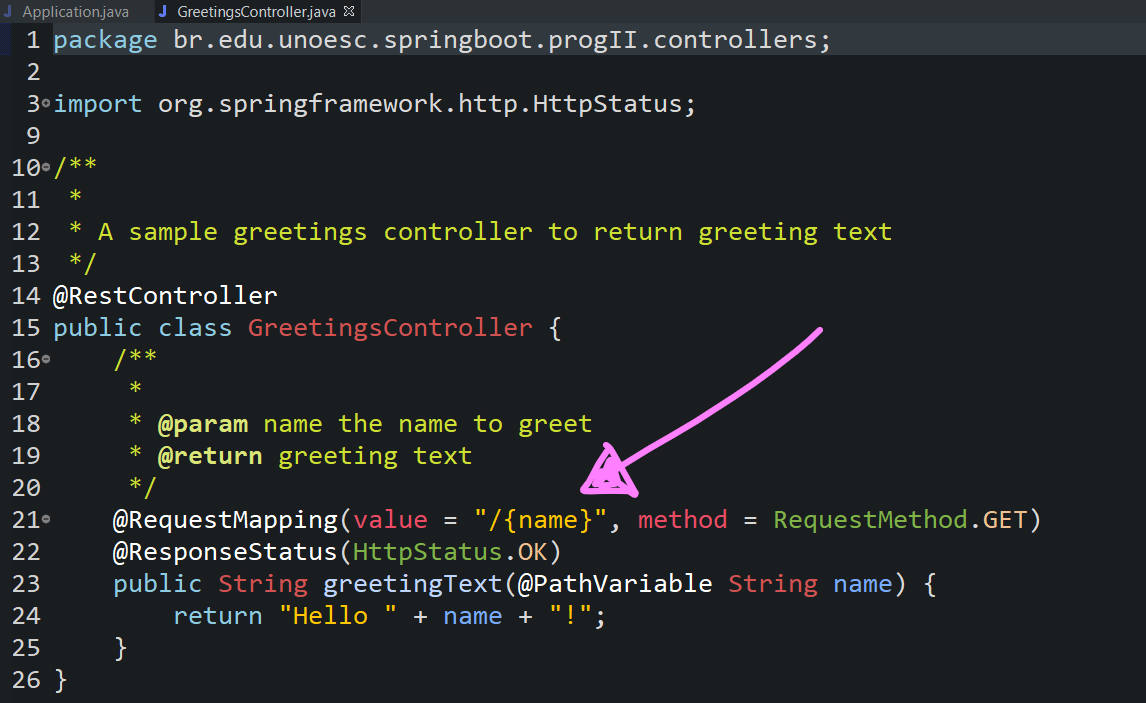


Nesse caso foi 8000, mas pode ser 8080, 8081 e depois poderemos modificar também.

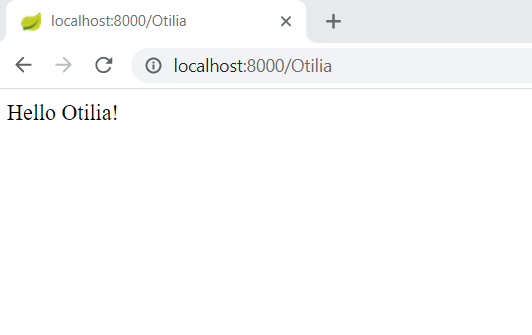
Para rodar vamos ao browser e vamos digitar:

**http://localhost:8000/Otilia**

Isso para que a requisição seja interceptada pelo controller seguindo o mapeamento sugerido inicialmente.



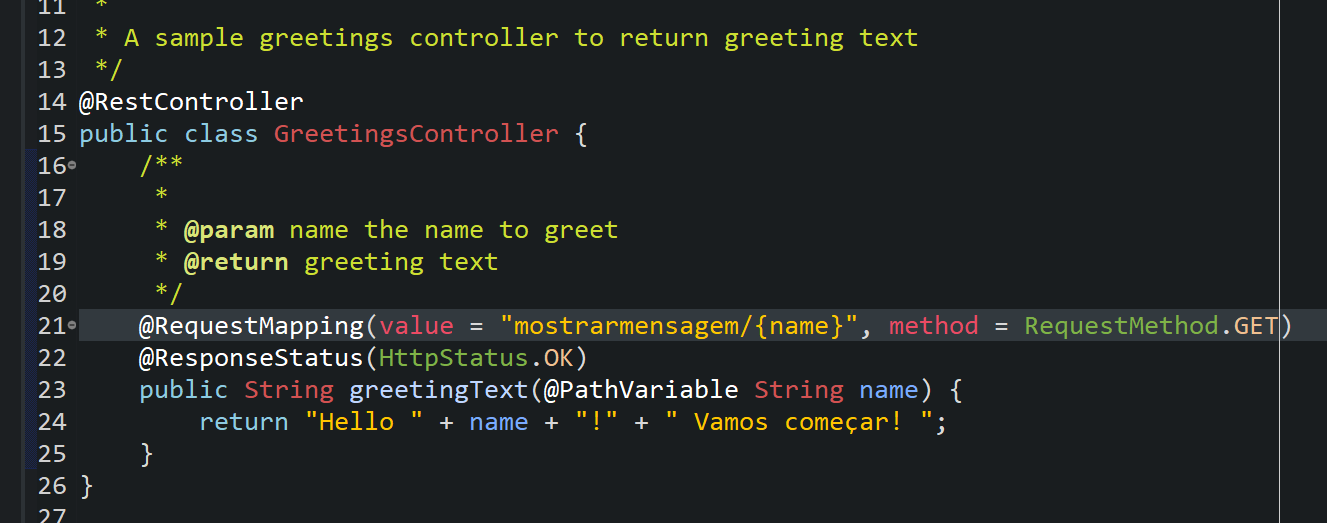
Esta será a resposta:



Podemos parar o servidor e modificar esse mapeamento (endpoint) para fazer outros testes como esses do exemplo a seguir.

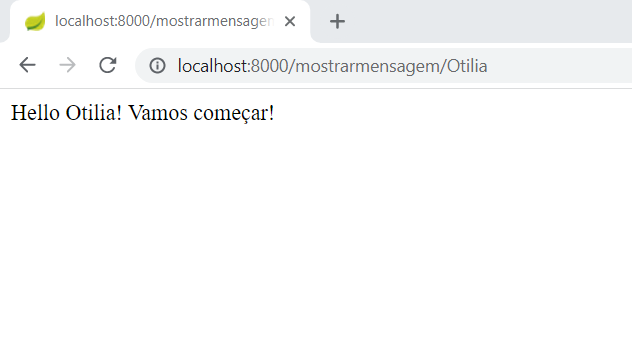
O valor passado como parâmetro {name} será usado e o método GET é usado quando quisermos utilizar a URL para passar os parâmetros.

O contexto pode ser alterado para incluir mostrarmensagem e também a mensagem que deve ser modificada assim:



Agora vamos novamente rodar a aplicação e vamos usar

**localhost:8000/mostrarmensagem/Otilia** para ter o retorno assim:



Podemos ter vários métodos mapeados. Vamos construir um segundo método, vamos mapeá-lo e definir o parâmetro texto:

@RequestMapping(value = "segundo/{texto}", method = RequestMethod.GET)

@ResponseStatus(HttpStatus.OK)

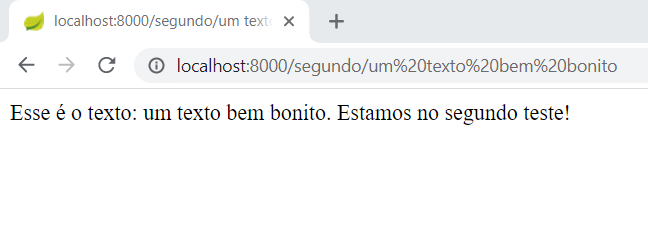
public String mostraSegundo(@PathVariable String texto){

return "Esse é o texto: " + texto + ". Estamos no segundo teste!";

}

Vamos testar usando a seguinte URL: **http://localhost:8000/segundo/um texto bem bonito**

e teremos como resposta



Vamos agora incluir a dependência do PostgreSQL e iniciar a conexão com o banco de dados

As dependências que precisarmos vamos encontrar na página do Maven: http://mvnrepository.com

Para facilitar, podemos usar o seguinte código:

<dependencies>

<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->

<dependency>

<groupId>org.postgresql</groupId>

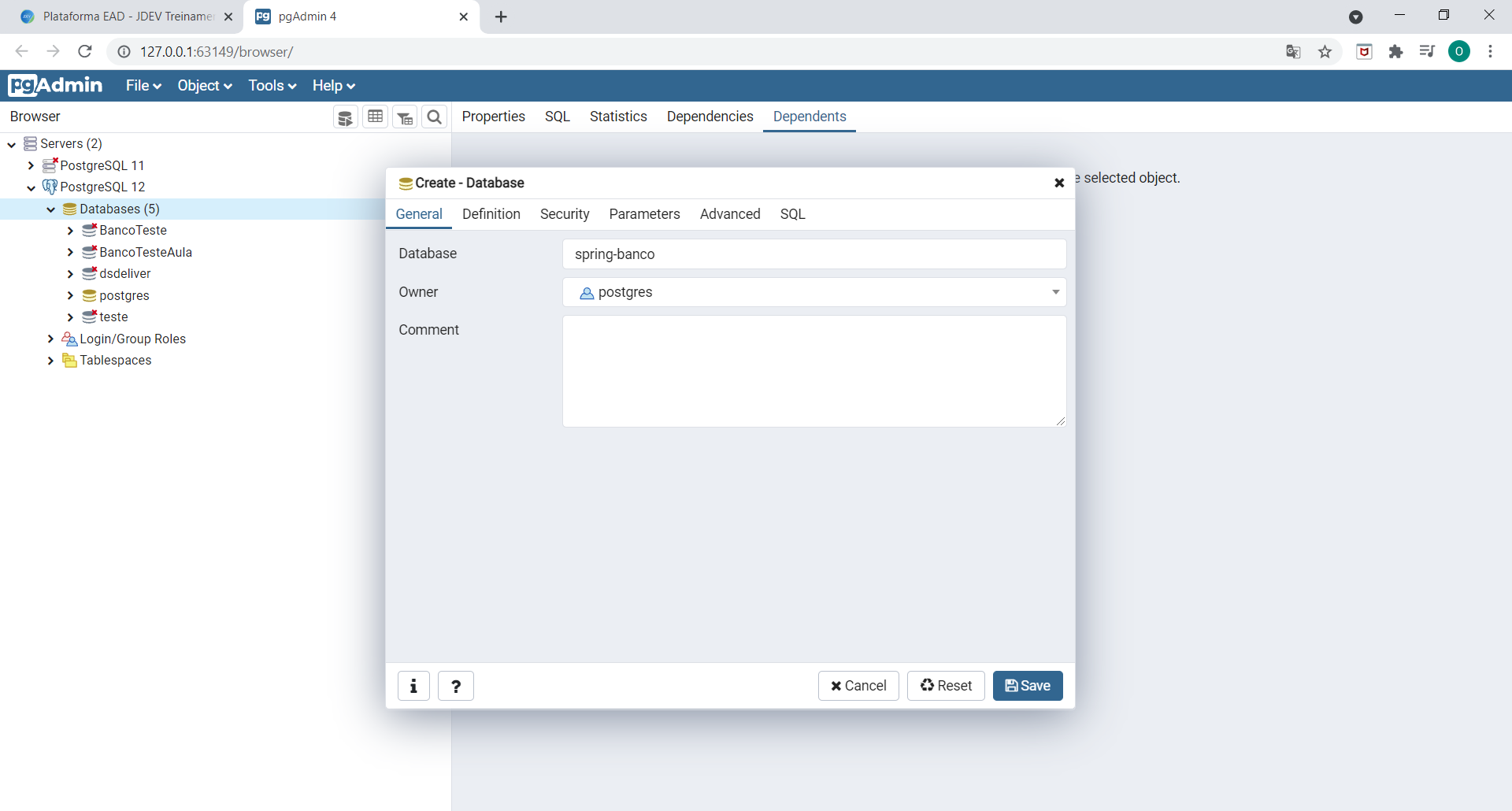
<artifactId>postgresql</artifactId>

</dependency>

</dependencies>

NOTE que a dependência deve ser **acrescentada** ao arquivo **POM.xml**, junto com as demais dependências já existentes (entre <dependencies>).

Vamos criar um banco de dados chamado spring-banco no pgAdmin



Agora vamos configurar a conexão com o banco de dados no arquivo **application.properties** informando a porta em que o postgresql está rodando (nesse caso é 5433, mas pode ser 5432) e o nome do banco de dados que será usado (nesse caso spring-banco), além disso é preciso informar o username e a senha que foram definidas durante a instalação do postgreSQL:

spring.datasource.url= jdbc:postgresql://localhost:5433/spring-banco

spring.datasource.username=postgres

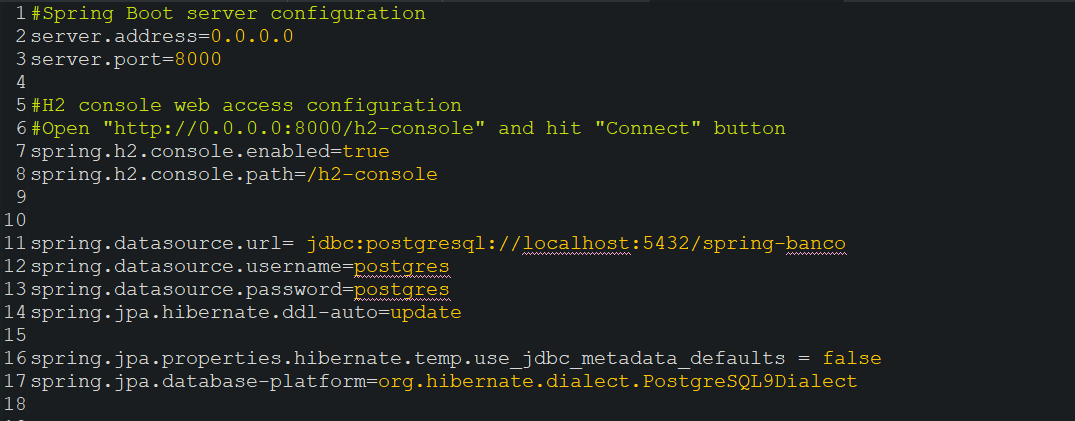
spring.datasource.password=postgres

spring.jpa.hibernate.ddl-auto=update

spring.jpa.properties.hibernate.temp.use\_jdbc\_metadata\_defaults = false

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL9Dialect

O application.properties ficará assim:



Podemos rodar o projeto e mesmo que não tenhamos ainda o banco de dados, a aplicação deverá continuar funcionando.

Precisamos construir uma classe que servirá de model para a construção do banco de dados. Vamos criar o package

**br.edu.unoesc.springboot.progII.model**

e a classe **Usuario** da seguinte forma, utilizando as anotações:

@Entity - para gerar a tabela no banco de dados

@SequenceGenerator - para gerar a sequência para o id

@Id - para gerar a chave na tabela

package br.edu.unoesc.springboot.progII.model;

import java.io.Serializable;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.SequenceGenerator;

@Entity

@SequenceGenerator(name = "seq\_usuario", sequenceName = "seq\_usuario", allocationSize = 1, initialValue = 1)

public class Usuario implements Serializable{

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq\_usuario")

private Long id;

private String nome;

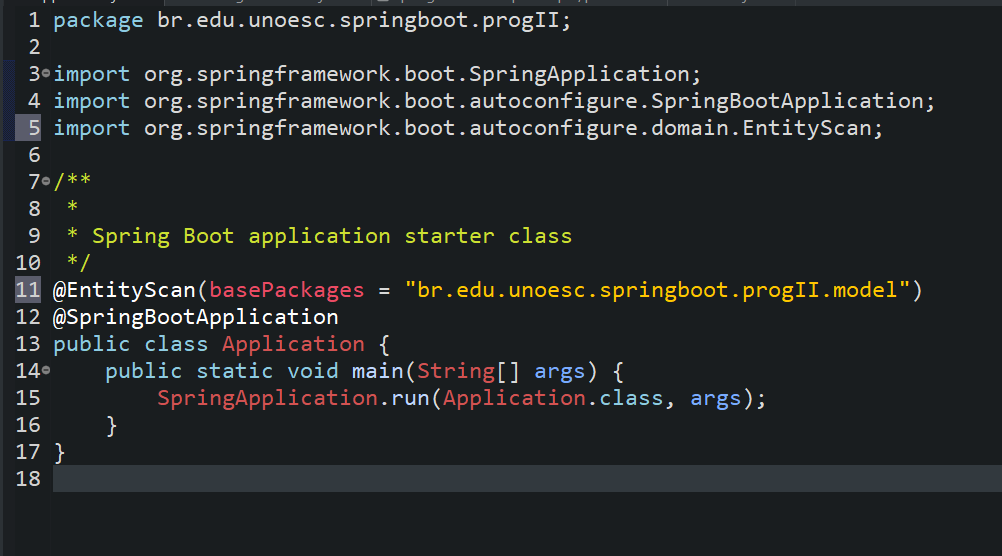
private int idade;

//getters e setters

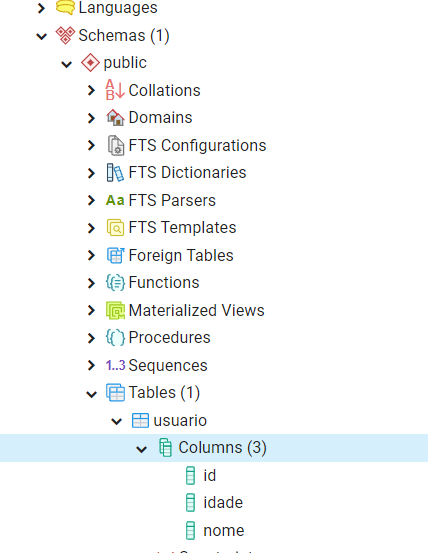
}

Precisamos ainda acrescentar a seguinte anotação em **Application** e atualizar os imports **:**

@EntityScan(basePackages = "br.edu.unoesc.springboot.progII.model")



Agora, podemos startar o projeto novamente, ir até o pgAdmin e atualizar com F5 e a tabela Usuario e as três colunas devem ter sido criadas.



O passo seguinte é criar o Repository para armazenar os dados no banco de dados.

Vamos construir um package

br.edu.unoesc.springboot.progII.repository

e nele vamos começar a construir **a interface** **UsuarioRepository** que extende JpaRepository assim:

@Repository

public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

}

Dessa forma, ao declarar um objeto de UsuarioRepository, teremos diversos métodos para persistência associados a ele, tais como save, delete, find, count, etc

Podemos declarar um objeto de UsuarioRepository e outro endpoint em **GreetingsController** assim:

@Autowired // injeção de dependência

private UsuarioRepository usuarioRepository;

@RequestMapping(value = "testeGravar/{nome}", method = RequestMethod.GET)

@ResponseStatus(HttpStatus.OK)

public String testeGravar(@PathVariable String nome){

Usuario usuario = new Usuario();

usuario.setNome(nome);

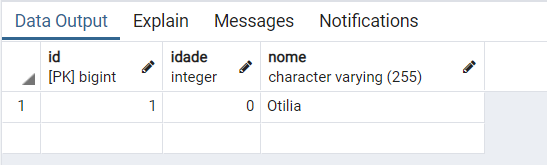
usuarioRepository.save(usuario);

return "Gravado";

}

E no browser podemos testar a gravação e ver o resultado no pgAdmin com:

http://localhost:8000/testeGravar/Otilia



Criação de API

Vamos começar construindo um novo end-point em **GreetingsController** que vai retornar uma lista de usuários existentes (consultando no banco de dados) em formato json para o corpo de uma requisição e precisamos também importar java.util.list para trabalhar com listas:

@GetMapping(value="listatodos")

@ResponseBody

public ResponseEntity<List<Usuario>>listaUsuario(){

List<Usuario> usuarios = usuarioRepository.findAll();

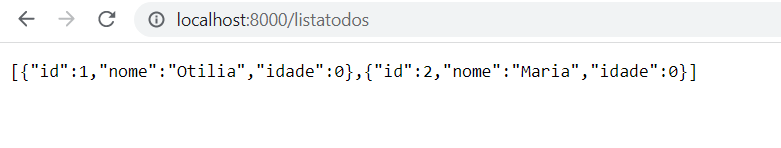
return new ResponseEntity<List<Usuario>>(usuarios, HttpStatus.OK) ;

}



Estou usando a extensão JsonFormatter 6.3 no Chrome e por isso obtemos essa visualização.

Sem nenhuma extensão, a visualização será assim:



Lembrando que temos um array de objetos json sendo mostrados.

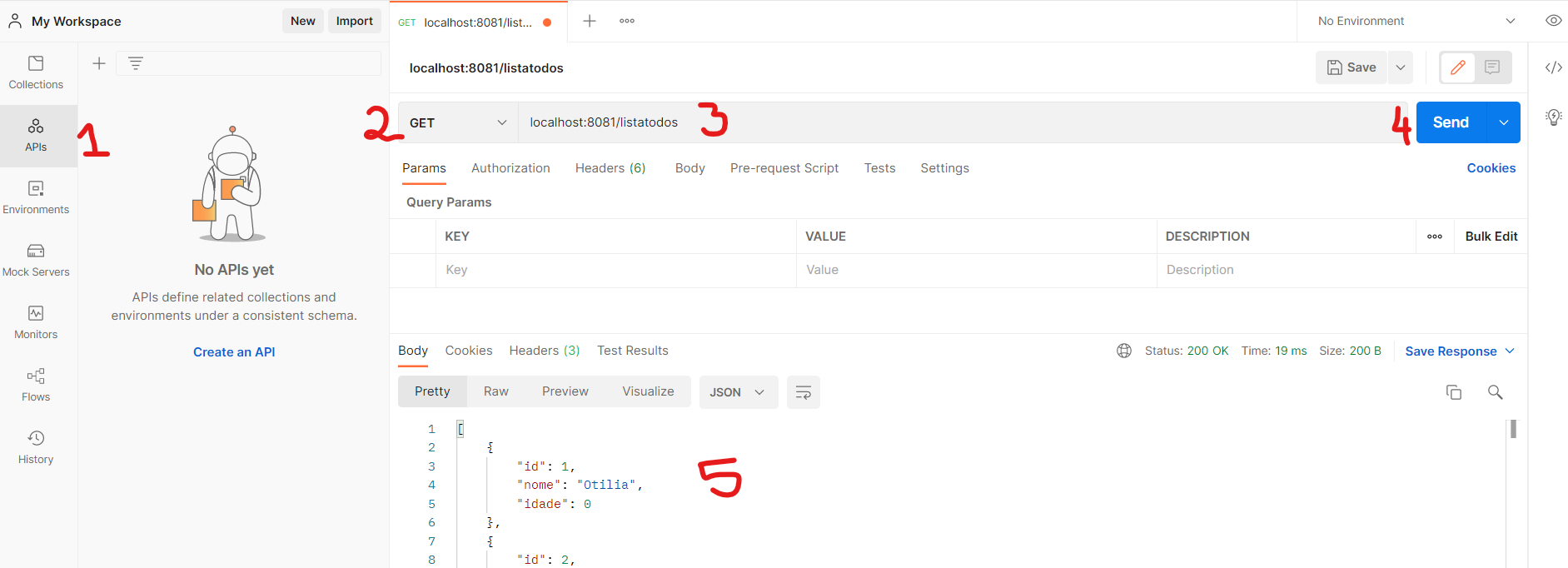
Para testar as requisições, podemos também utilizar o **Postman** (veja o material para a instalação do Postman). Será ele que utilizaremos para ver as respostas das requisições a partir de agora:

Caso ao levantar novamente o projeto no Tomcat sejamos informados que a porta 8000 não está disponível isso pode ser devido ao Postman estar rodando nessa porta.

Para mudar, devemos modificar no projeto em application.properties e passar da porta 8000 para a porta 8081, por exemplo:

server.port=8081

Podemos testar a requisição para listatodos no Postman seguindo esses passos:



1 - escolhemos API

2 - adicionamos uma request (+) e escolhemos o verbo (GET)

3 - informamos a requisição (com o servidor rodando): http://localhost:8001/listatodos

4 - clicamos em Send

5 - visualizamos o retorno da requisição

Para salvar dados no banco de dados (enviando um json) vamos utilizar a seguinte construção em **GreetingsController**.

@PostMapping(value = "salvar")

@ResponseBody

public ResponseEntity<Usuario> salvar(@RequestBody Usuario usuario){

Usuario user = usuarioRepository.save(usuario);

return new ResponseEntity<Usuario>(user, HttpStatus.CREATED);

}

Para testar no Postman usamos a seguinte sequência, com o seguinte json:

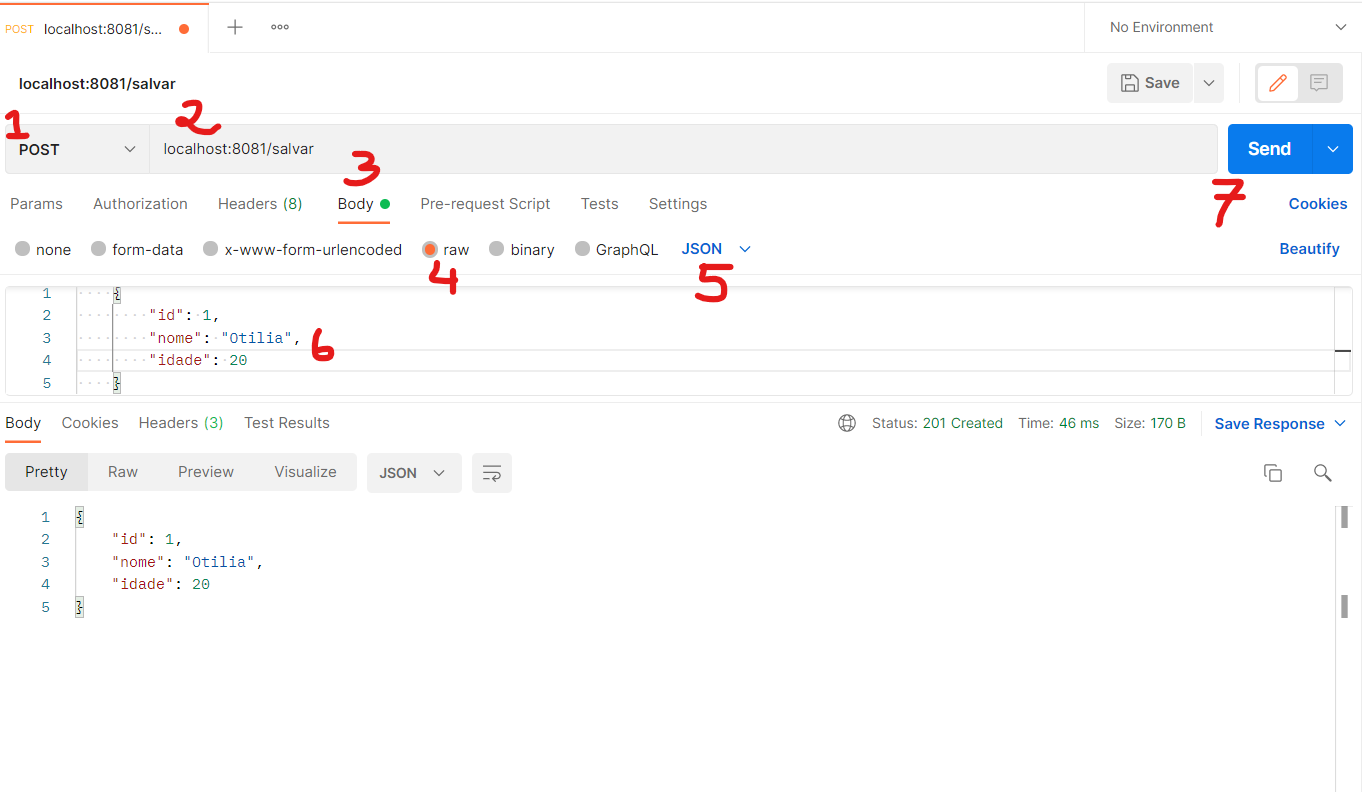
{

"id": 1,

"nome": "Otilia",

"idade": 20

}



1 - escolhemos POST (já que queremos enviar os dados para o banco de dados)

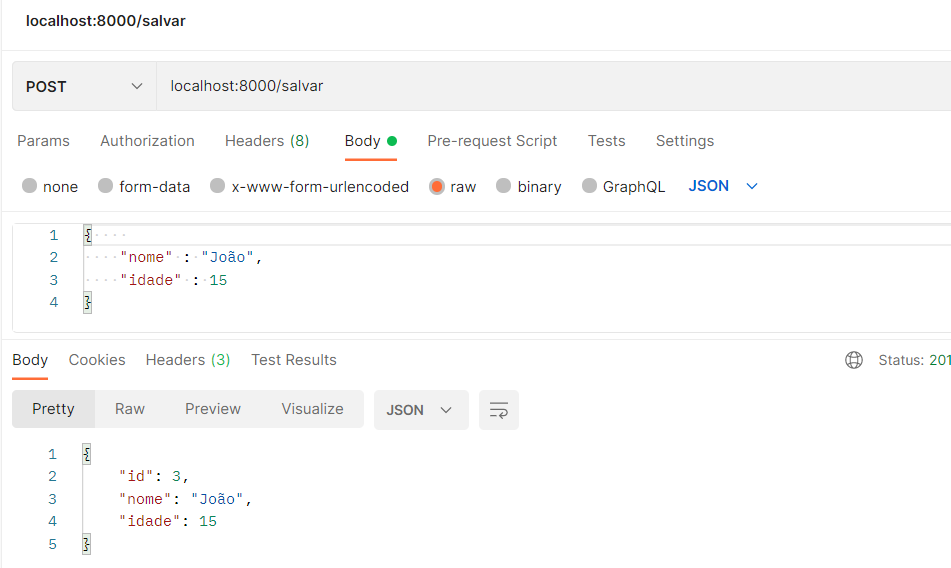
2 - informamos a requisição (com o servidor rodando a aplicação) localhost:8081//salvar

3/4/5 - escolhemos Body -> raw -> JSON -> informamos o json com os dados a serem salvos

7 - Send para enviar a requisição e veremos em seguida o resultado dos dados salvos.

Se informar um **id existente**, os dados do registro existente serão atualizados e salvos.

Para informar **um novo registro**, usar apenas o nome e a idade (sem o id que será gerado).



Para excluir registros vamos ter a seguinte construção:

@DeleteMapping(value = "delete")

@ResponseBody

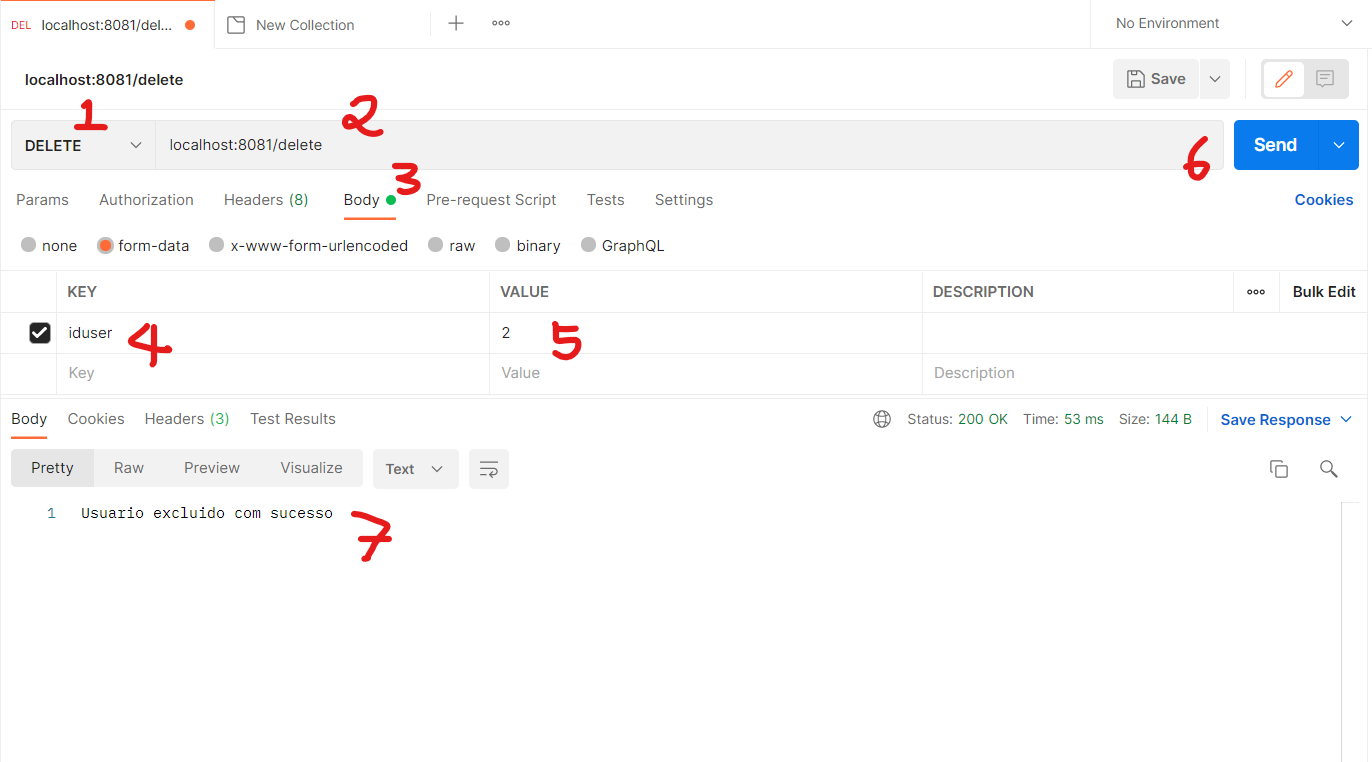
public ResponseEntity<String> delete(@RequestParam Long iduser) {

usuarioRepository.deleteById(iduser);

return new ResponseEntity<String>("Usuario excluido com sucesso", HttpStatus.OK);

}

E para testar com o Postman usamos a seguinte sequência:



1 - escolher o verbo DELETE

2 - informar a requisição: localhost:8081//delete

3 - escolher Body

4/5 - informar o parâmetro que será enviado (iduser) e o valor (2)

6 - enviar a requisição

7 - conferir a resposta

O end-point para buscar um determinado registro no banco de dados será assim:

@GetMapping(value = "buscaruserid")

@ResponseBody

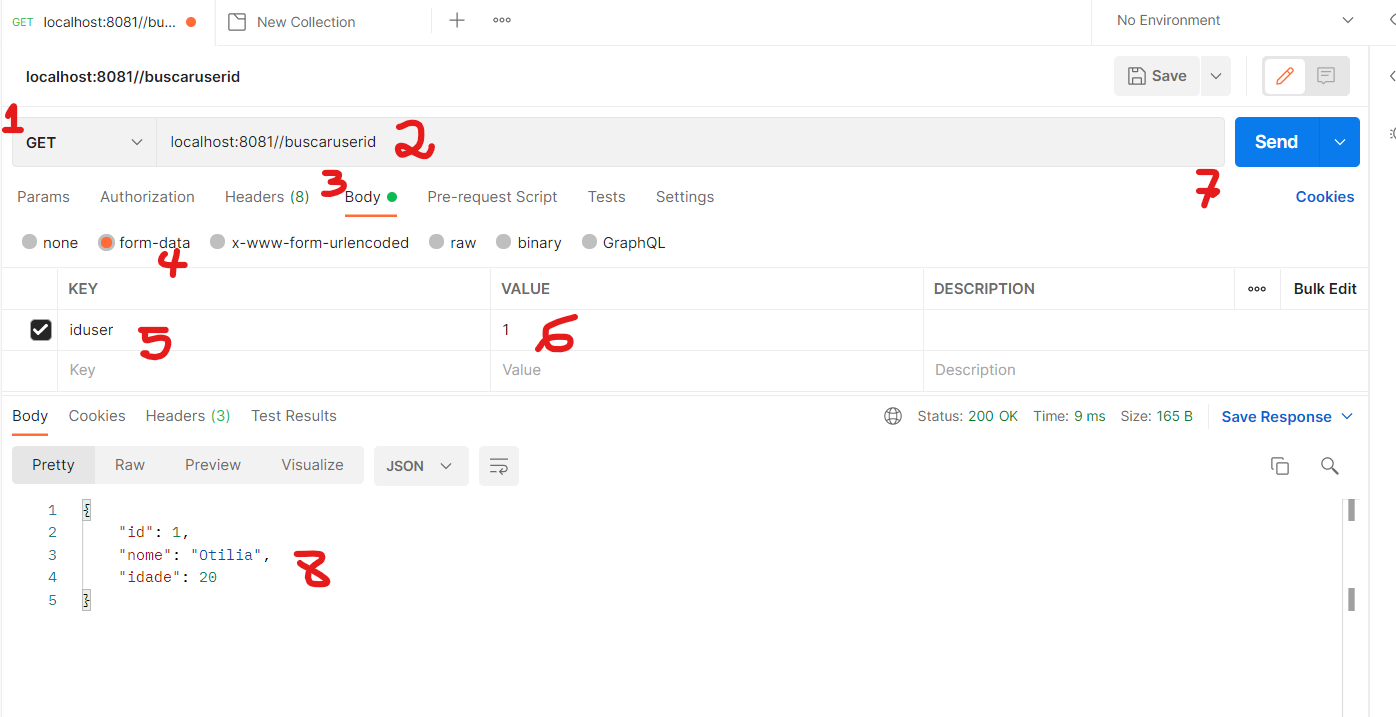
public ResponseEntity<Usuario> buscaruserid(@RequestParam(name = "iduser") Long iduser){

Usuario usuario = usuarioRepository.findById(iduser).get();

return new ResponseEntity<Usuario>(usuario, HttpStatus.OK);

}

Para testar no Postman fazemos assim:



Para atualizar podemos ter a seguinte construção:

@PutMapping(value = "atualizar")

@ResponseBody

public ResponseEntity<?> atualizar(@RequestBody Usuario usuario){

if(usuario.getId()==null) {

return new ResponseEntity<String>("Id não informado para atualizar", HttpStatus.OK);

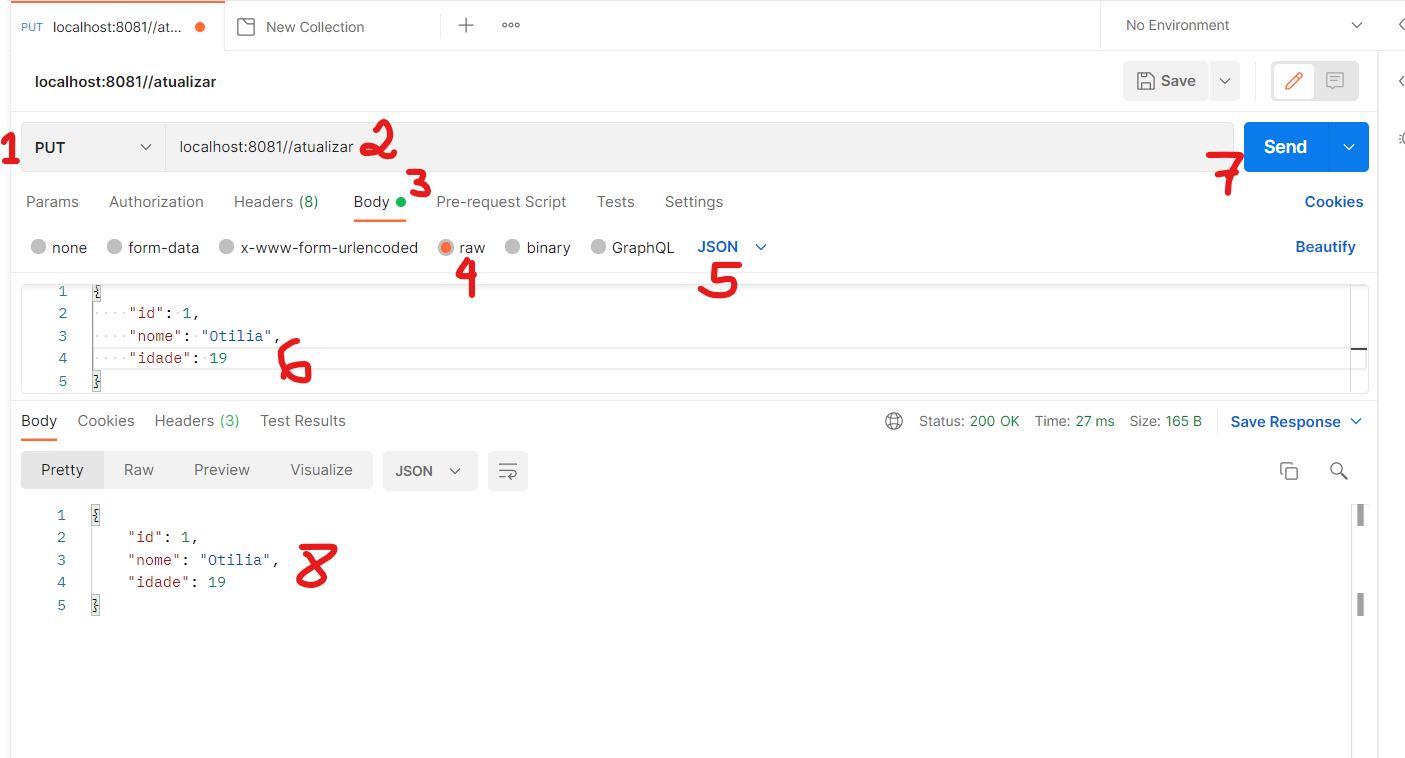
}

Usuario user = usuarioRepository.saveAndFlush(usuario);

return new ResponseEntity<Usuario>(user, HttpStatus.OK);

}

E para testar seguimos os seguintes passos:

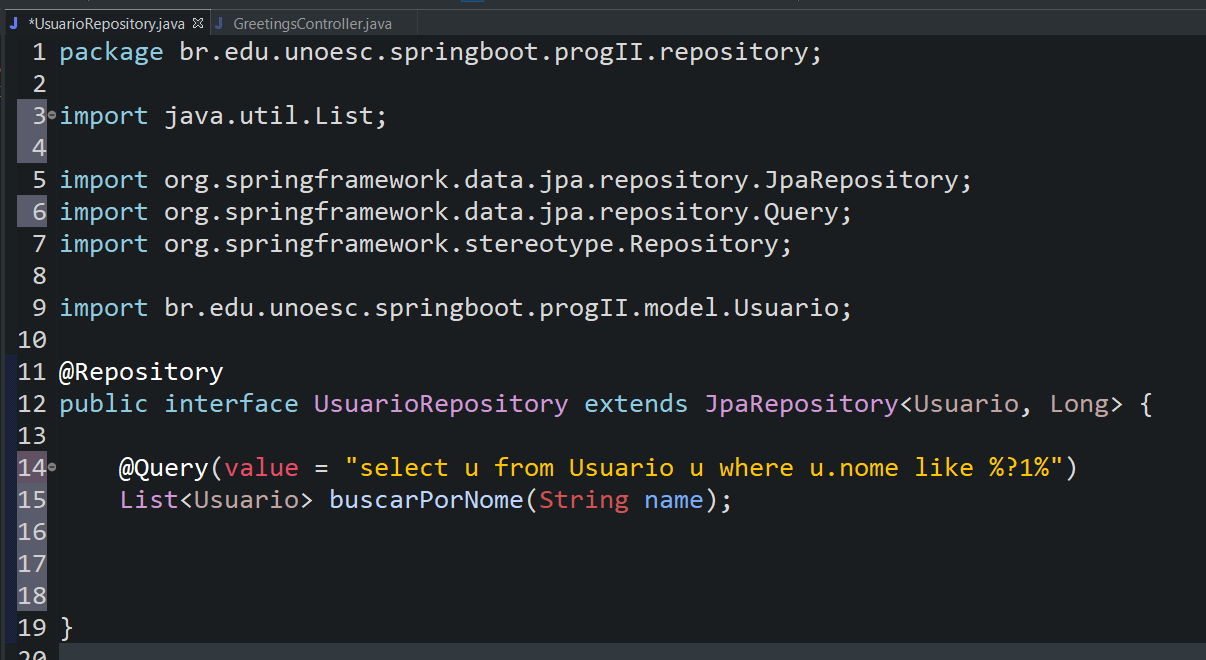


Para realizar uma consulta por parte do nome do usuário precisamos modificar a interface **UsuarioRepository**, já que esse método não existe em JpaRepository.

Assim, vamos acrescentar esse método na interface UsuarioRepository e teremos assim:

@Query(value = "select u from Usuario u where u.nome like %?1%")

List<Usuario> buscarPorNome(String name);



Ou podemos melhorar assim:

@Query(value = "select u from Usuario u where upper(trim(u.nome)) like %?1%")

Em GreetingsController faremos a seguinte construção:

@GetMapping(value = "buscarpornome")

@ResponseBody

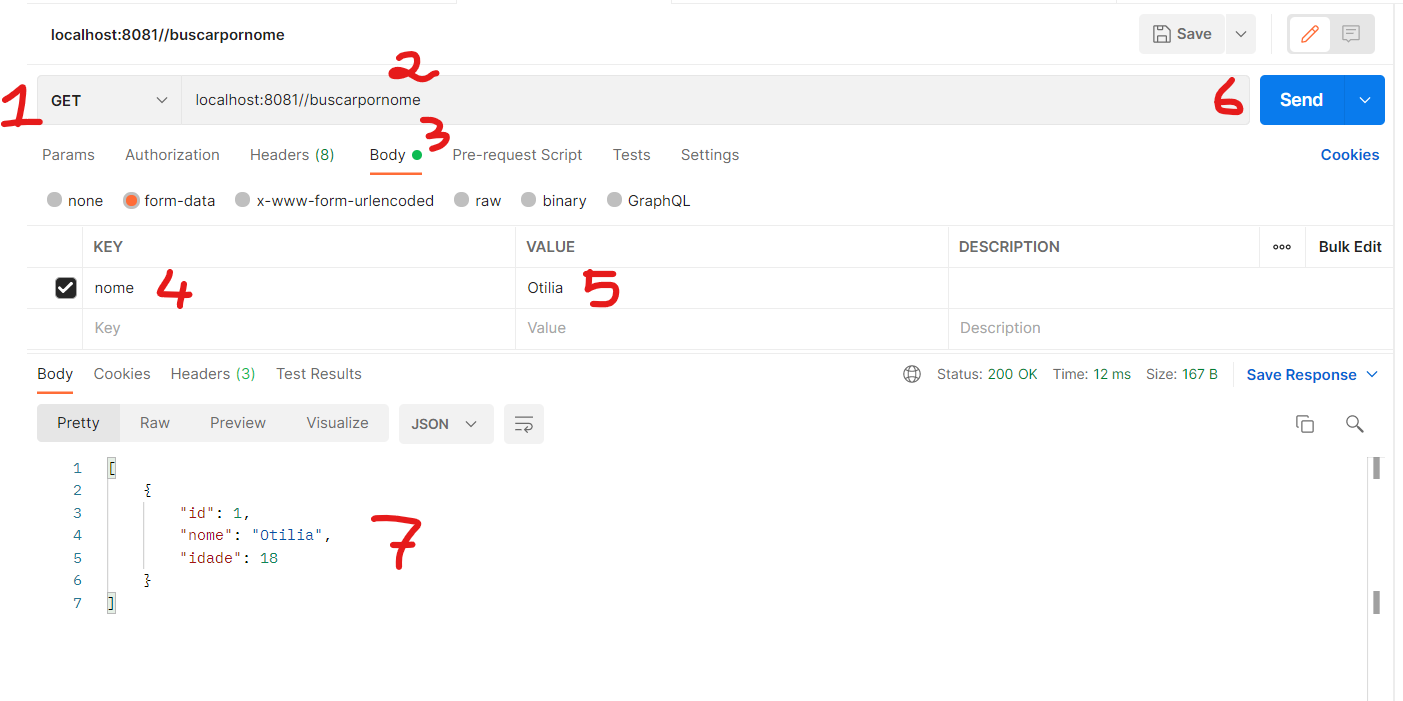
public ResponseEntity<List<Usuario>>bucarpornome(@RequestParam(name = "nome") String nome){

List<Usuario> usuario = usuarioRepository.buscarPorNome(nome.trim().toUpperCase());

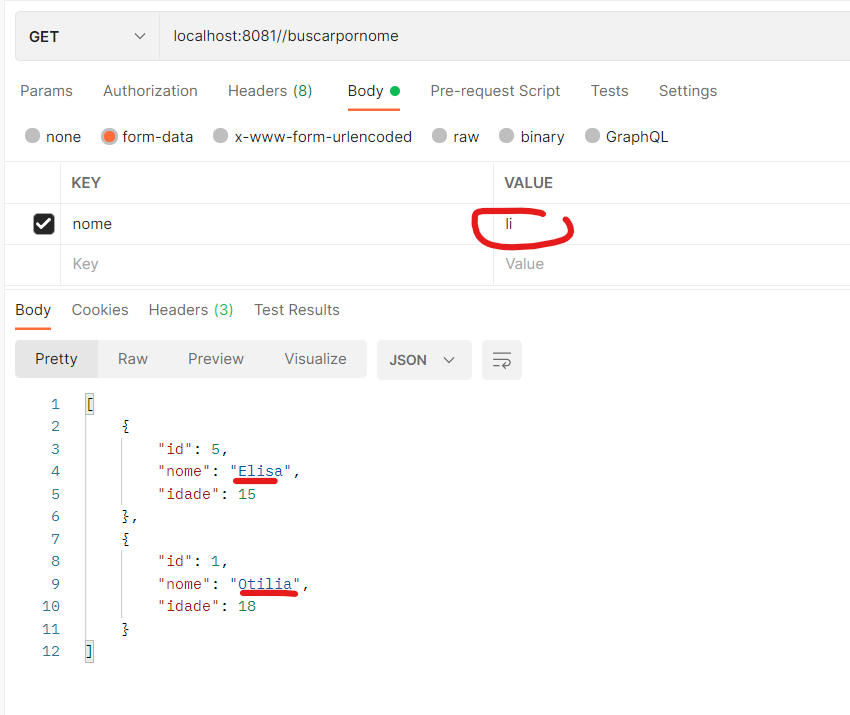
return new ResponseEntity<List<Usuario>>(usuario, HttpStatus.OK);

}

E para testar seguiremos os seguintes passos:



Podemos obter mais de um registro, como nesse exemplo:



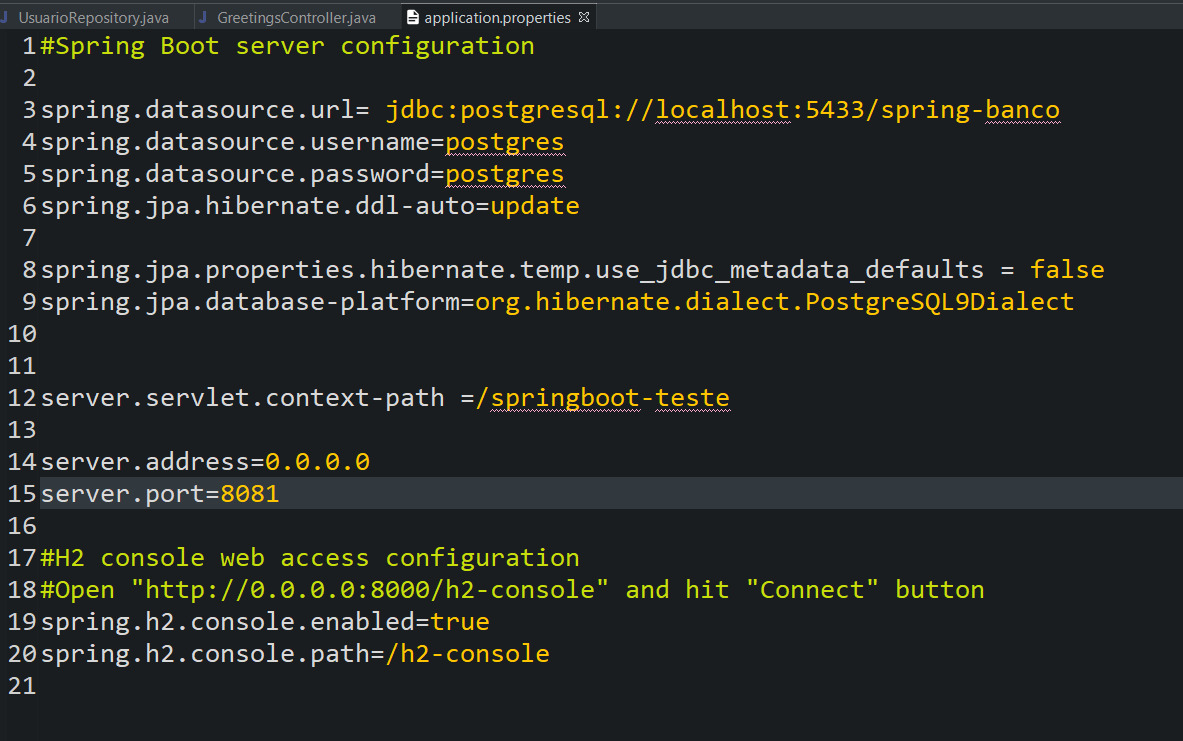
O projeto estava sem um contexto até esse momento e os end-points eram chamados diretamente a partir de localhost:8081//xxxxx.

Para mudar o contexto, vamos modificar application-properties assim:

server.servlet.context-path =/springboot-teste

server.adress=0.0.0.0

server.port=8081



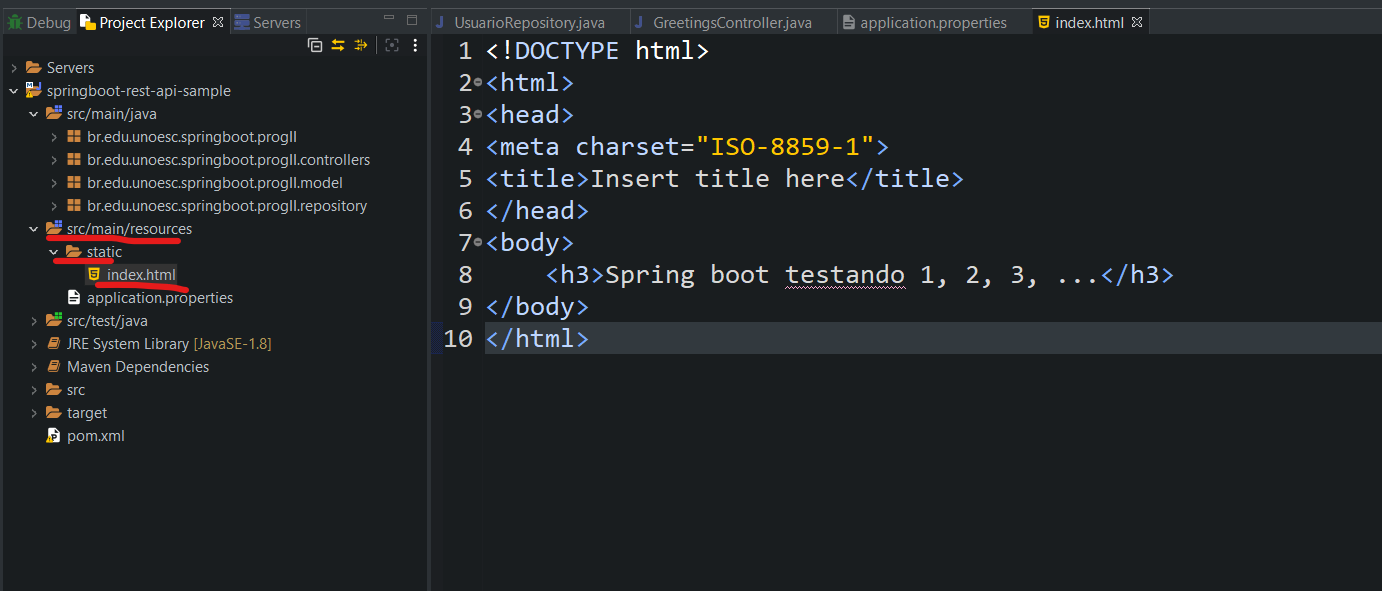
A partir de agora, para chamar as requisições precisamos fazer assim:

localhost:8081//springboot-teste/xxxxx

**Isso deve ser feito tanto no browser quanto no Postman.**

Precisamos agora construir os formulários para consumir as requisições que foram construídas.

Começamos com o **file** chamado **index.html** que deve ficar dentro do **package** src/main/resources (já existente) e dentro de uma nova **pasta** (folder) chamada **static.**



Ao utilizar o contexto do projeto já veremos o arquivo index.html.



Podemos acrescentar recursos do Bootstrap para que fique com uma melhor aparência.

De forma simples podemos buscar modelos em:

<https://getbootstrap.com/docs/5.1/getting-started/introduction/>

E o arquivo index deve ficar assim nesse primeiro momento.

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<meta name="viewport" content="width=device-width, initial-scale=1">

<!-- Bootstrap CSS -->

<link

href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"

rel="stylesheet"

integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"

crossorigin="anonymous">

<title>Spring boot - teste</title>

</head>

<body>

<h3>Spring boot testando 1, 2, 3, ...</h3>

<!-- Option 1: Bootstrap Bundle with Popper -->

<script

src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"

integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p"

crossorigin="anonymous"></script>

</body>

</html>

Assim, já teremos uma aparência um pouco diferente:



Para a construção do formulário vamos precisar construir um formulário com os campos, labels e botões. Vamos começar acrescentando um form assim dentro de body:

<form action="" id="formCadastroUser">

<div class="mb-3">

<label for="id" class="form-label">ID</label>

<input type="text" class="form-control" id="id" readonly="readonly">

</div>

<div class="mb-3">

<label for="nome" class="form-label">Nome do usuário</label>

<input type="text" class="form-control" id="nome" placeholder="Informe o nome" required = "required">

</div>

<div class="mb-3">

<label for="idade" class="form-label">Idade</label>

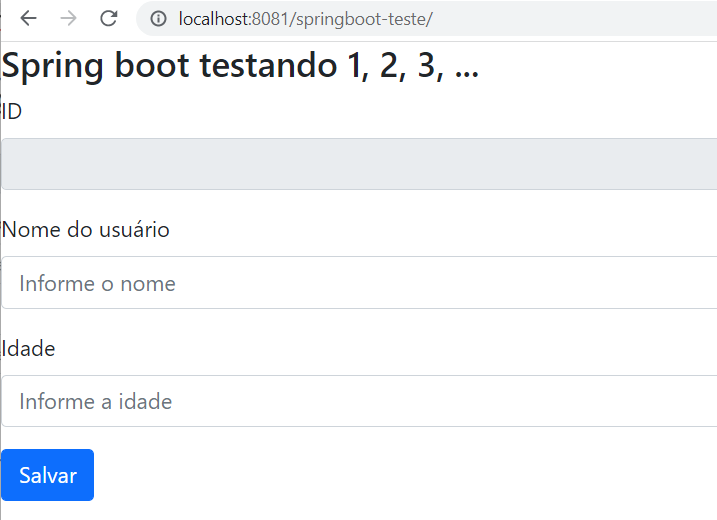
<input type="number" class="form-control" id="idade" >

</div>

</form>

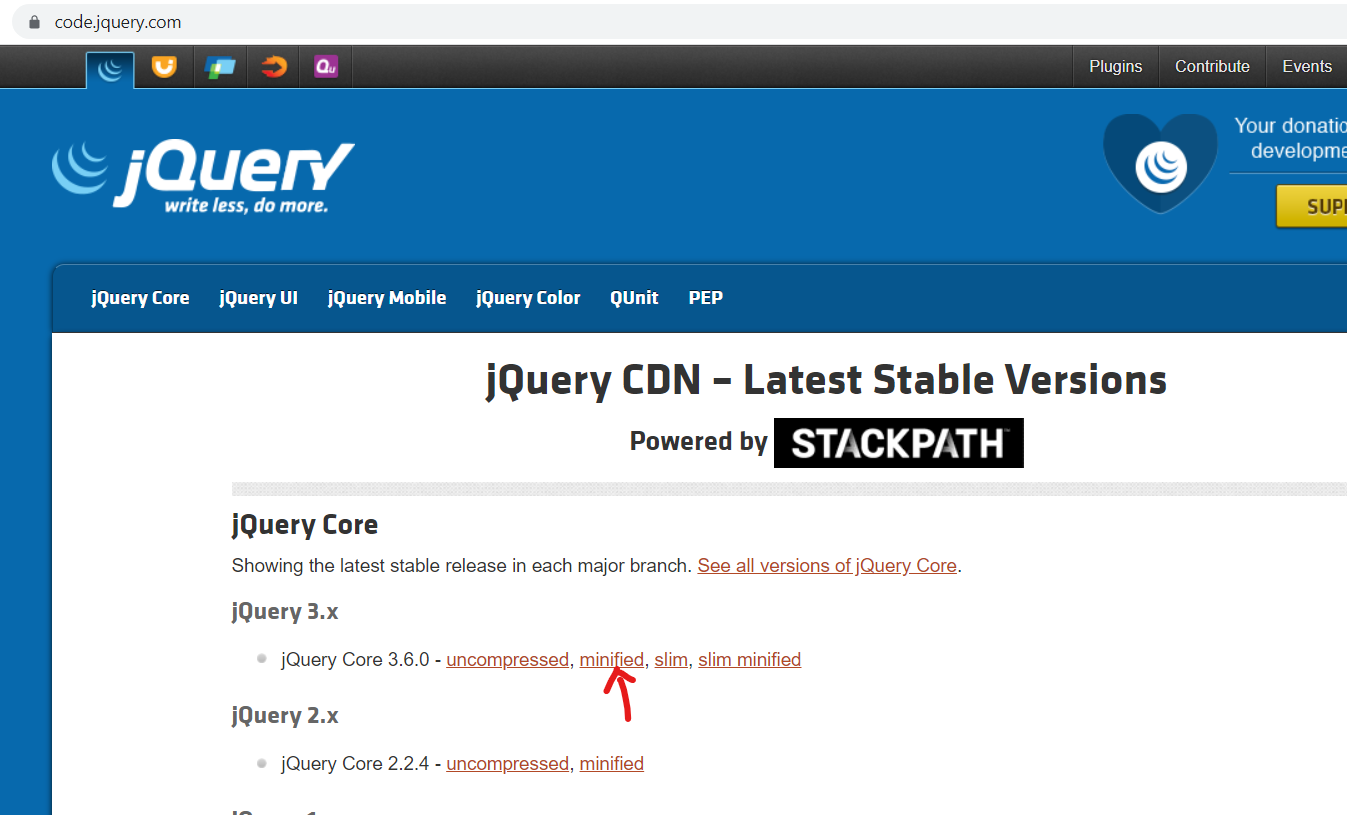
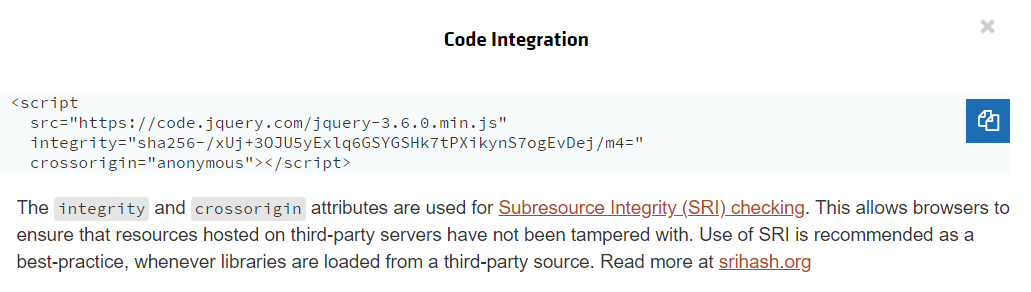
<button type="button" class="btn btn-primary">Salvar</button>

Para acrescentar botões com formatos já definidos podemos entrar em https://getbootstrap.com/docs/5.1/components/buttons/ e escolher o código.

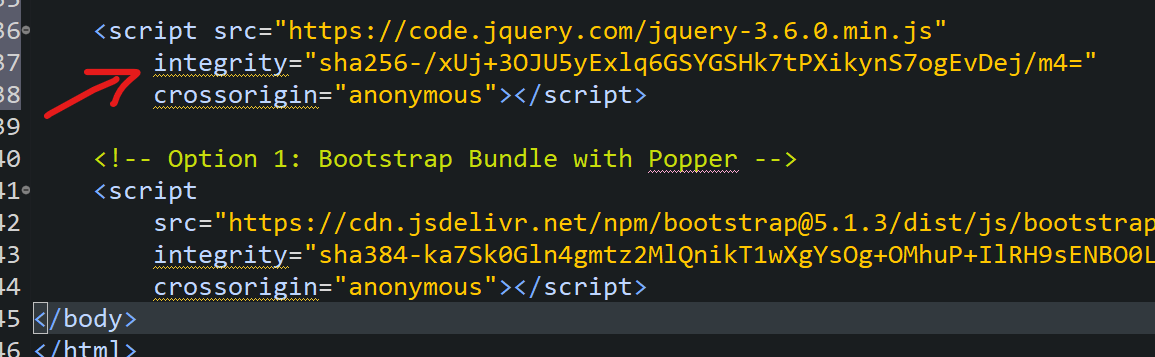


Para a interação podemos utilizar várias ferramentas. Aqui vamos usar JQuery, JavaScript e AJAX.

Para instalar vamos entrar em: <https://code.jquery.com/>, vamos escolher minified e copiar o script.



Vamos copiar o script para inserir no arquivo index.html, no body, **depois dos formulários mas antes do script do Bootstrap.**



Agora podemos utilizar o **botão Salvar**, criando um script para ele. Vamos mudar o botão para acrescentar a chamada ao script assim:

<button type="button" class="btn btn-primary" onclick="salvarUsuario()">Salvar</button>

E vamos construir a função salvarUsuario(), **acrescentar depois dos scripts** já inseridos e vamos testar, mostrando uma mensagem com os dados recuperados:

<script type"text/javascript">

function salvarUsuario(){

var id = $("#id").val();

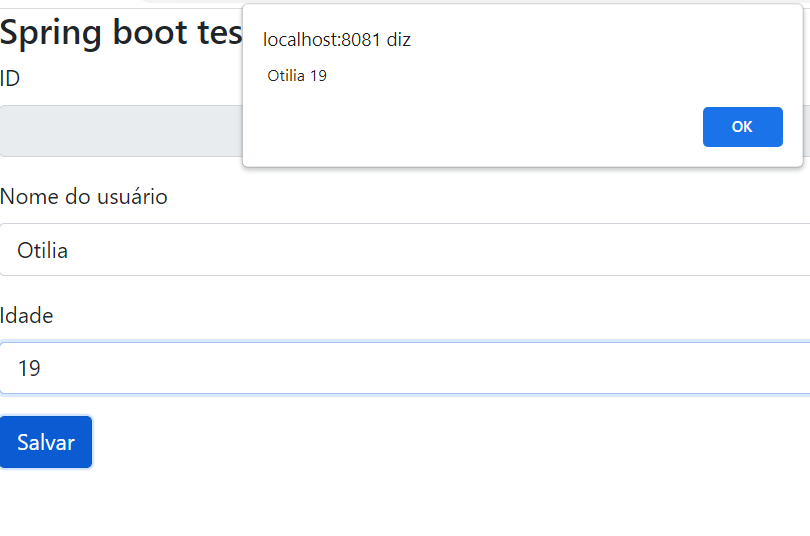
var nome = $("#nome").val();

var idade = $("#idade").val();

alert(id + " " + nome + " " + idade)

}

</script>



Agora, em lugar de alert, vamos usar AJAX para conectar o formulário com as requisições assim:

<script type="text/javascript">

function salvarUsuario(){

var id = $("#id").val();

var nome = $("#nome").val();

var idade = $("#idade").val();

if(nome == ""){

alert("Informe o nome! ");

$("#nome").focus();

return;

}else{

$.ajax({

method : "POST",

url : "salvar",

data : JSON.stringify({id : id, nome : nome, idade: idade}),

contentType : "application/json; charset=utf-8",

success : function(response){

$("#id").val(response.id);

alert("Salvo com sucesso!");

}

}).fail(function (xhr, status, errorThrown){

alert("Erro ao salvar: " + xhr.responseText);

});

}

}

</script>

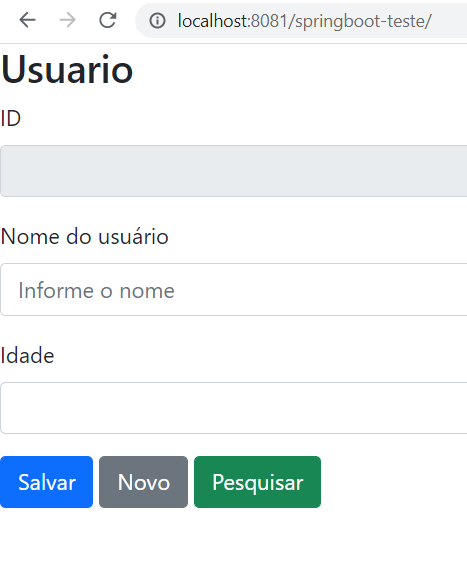
Assim, os dados informados no formulário serão enviados por post para o end-point mapeado como salvar e os dados enviados serão salvos no banco de dados.

Vamos acrescentar um botão **Novo** para limpar o formulário assim:

<button type="button" class="btn btn-secondary" onclick ="document.getElementById('formCadastroUser').reset();">Novo</button>

E também um botão **Pesquisar** assim:

<button type="button" class="btn btn-success">Pesquisar</button>



Vamos construir uma tela modal para pesquisar por dados já cadastrados. Podemos ver alguns modelos em de modal em: <https://getbootstrap.com/docs/5.1/components/modal/#varying-modal-content> e vamos utilizar o seguinte código para criar uma janela modal. Esse código deve ser acrescentado no documento depois dos botões e antes dos scripts:

<div class="modal fade" id="modalPesquisarUser" tabindex="-1"

aria-labelledby="Pesquisar Usuários" aria-hidden="true">

<div class="modal-dialog">

<div class="modal-content">

<div class="modal-header">

<h5 class="modal-title" id="exampleModalLabel">Pesquisa de

usuários:</h5>

<button type="button" class="btn-close" data-bs-dismiss="modal"

aria-label="Fechar"></button>

</div>

<div class="modal-body">

<form>

<div class="mb-3">

<label for="nomeBusca" class="col-form-label">Informe o

nome:</label> <input type="text" class="form-control" id="nomeBusca">

</div>

<button type="button" class="btn btn-success">Buscar</button>

</form>

</div>

<div class="modal-footer">

<button type="button" class="btn btn-secondary"

data-bs-dismiss="modal">Fechar</button>

</div>

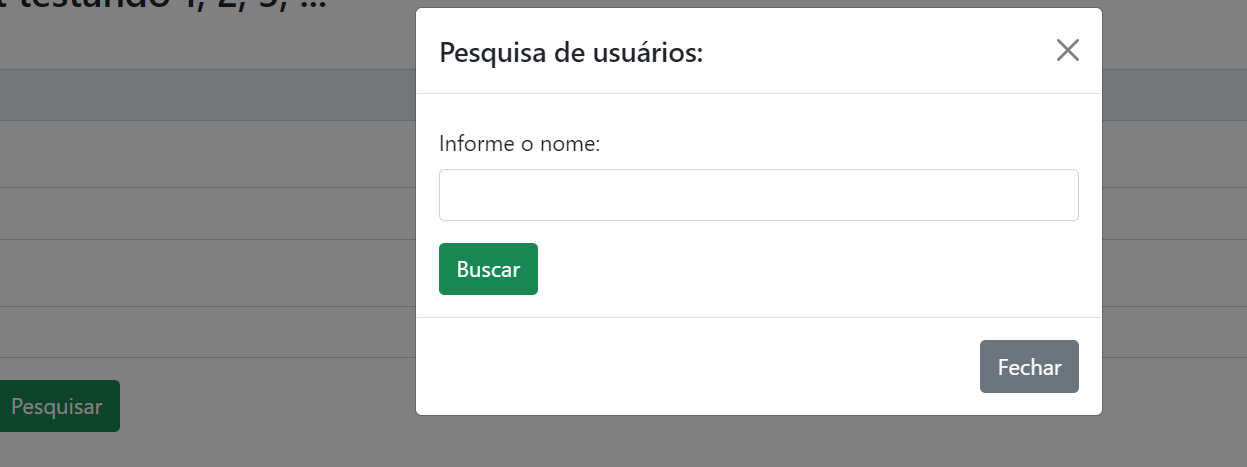
</div>

</div>

</div>

Precisamos mudar o botão Pesquisar para que ele chame o modal assim:

<button type="button" class="btn btn-success" data-bs-toggle="modal" data-bs-target="#modalPesquisarUser">Pesquisar</button>



Vamos acrescentar uma tabela dentro do modal, depois do *form* e antes do *footer* do modal, para mostrar os dados pesquisados assim:

<div style="height:300px; overflow:auto;">

<table class="table table-striped" id="tabelaresultados">

<thead>

<tr>

<th scope="col">ID</th>

<th scope="col">Nome</th>

<th scope="col">Editar</th>

<th scope="col">Excluir</th>

</tr>

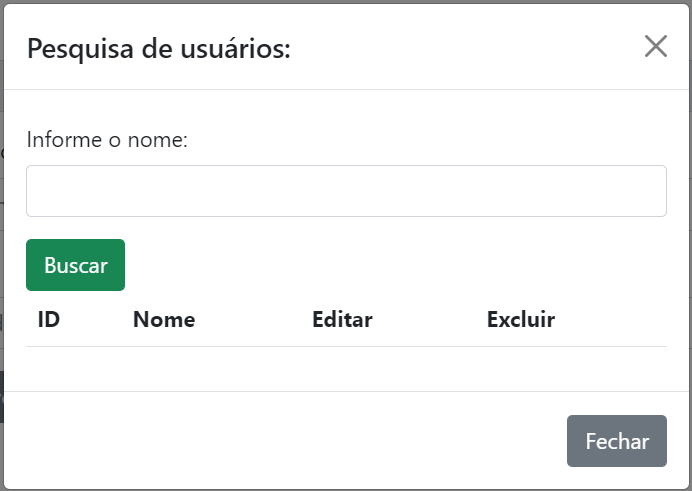
</thead>

<tbody>

</tbody>

</table>

</div>



Vamos construir o script para o botão Buscar que vai selecionar o nome informado pelo usuário, efetuar a pesquisa através do end-point, e mostrar o resultado na tabela de forma dinâmica. Essa função será colocada depois da function salvarUsuario() já existente, dentro da mesma tag <script>

<button type="button" class="btn btn-success" onclick="pesquisarUsuario()">Buscar</button>

...

function pesquisarUsuario(){

var nomeB = $("#nomeBusca").val();

if (nomeB !=null && nomeB.trim()!=''){

$.ajax({

method : "GET",

url : "buscarpornome",

data : "nome=" + nomeB,

success :

function(response){

$("#tabelaresultados > tbody > tr").remove();

for (var i = 0; i < response.length; i++){

$('#tabelaresultados > tbody').append('<tr><td>' +

response[i].id + '</td><td>' +

response[i].nome +'</td><td> <button type="button" onclick="colocarEmEdicao('+response[i].id+')"class="btn btn-primary">+</button></td></tr>')

}

}

}).fail(function (xhr, status, errorThrown){

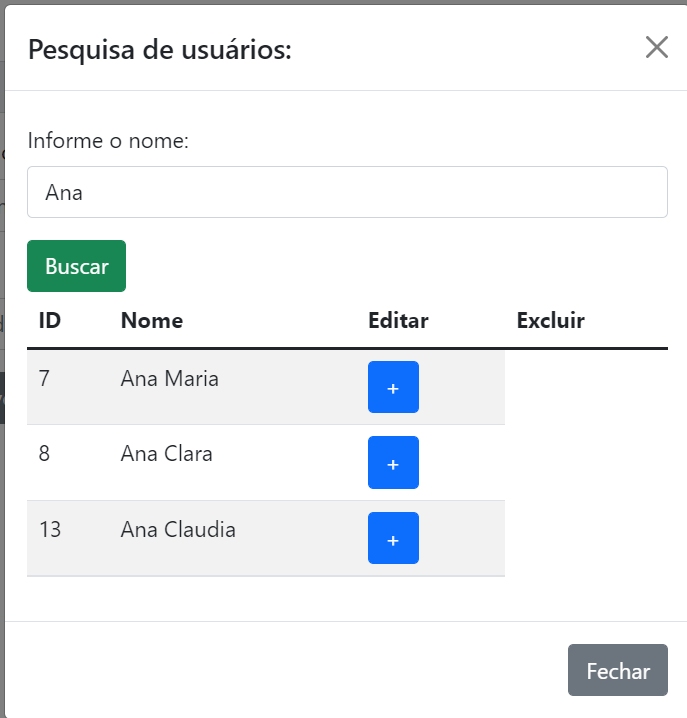
alert("Erro ao buscar usuário: " + xhr.responseText);

});

}

}

Ao realizar a pesquisa teremos resultados assim:



Agora precisamos implementar a função para executar o botão +, que coloca em edição o registro escolhido, colocarEmEdicao, assim:

function colocarEmEdicao(id) {

$.ajax({

method : "GET",

url : "buscaruserid",

data : "iduser=" + id ,

success : function(response) {

$("#id").val(response.id);

$("#nome").val(response.nome);

$("#idade").val(response.idade);

$('#modalPesquisarUser').modal('hide');

}

}).fail(function(xhr, status, errorThrown) {

alert("Erro ao buscar usuario por id: " + xhr.responseText);

});

}

O botão + já teve sua codificação definida de forma dinâmica na função pesquisarUsuario().

Com essa função, ao clicar no botão “+”, o registro escolhido é copiado para o formulário principal e pode ser editado.

A última rotina que precisa ser implementada é a exclusão dos registros. Vamos modificar a função pesquisarUsuario para mostrar o botão de exclusão, também de forma dinâmica:

...

for (var i = 0; i < response.length; i++){

$('#tabelaresultados > tbody').append('<tr id="'+response[i].id+'"><td>' +

response[i].id + '</td><td>' +

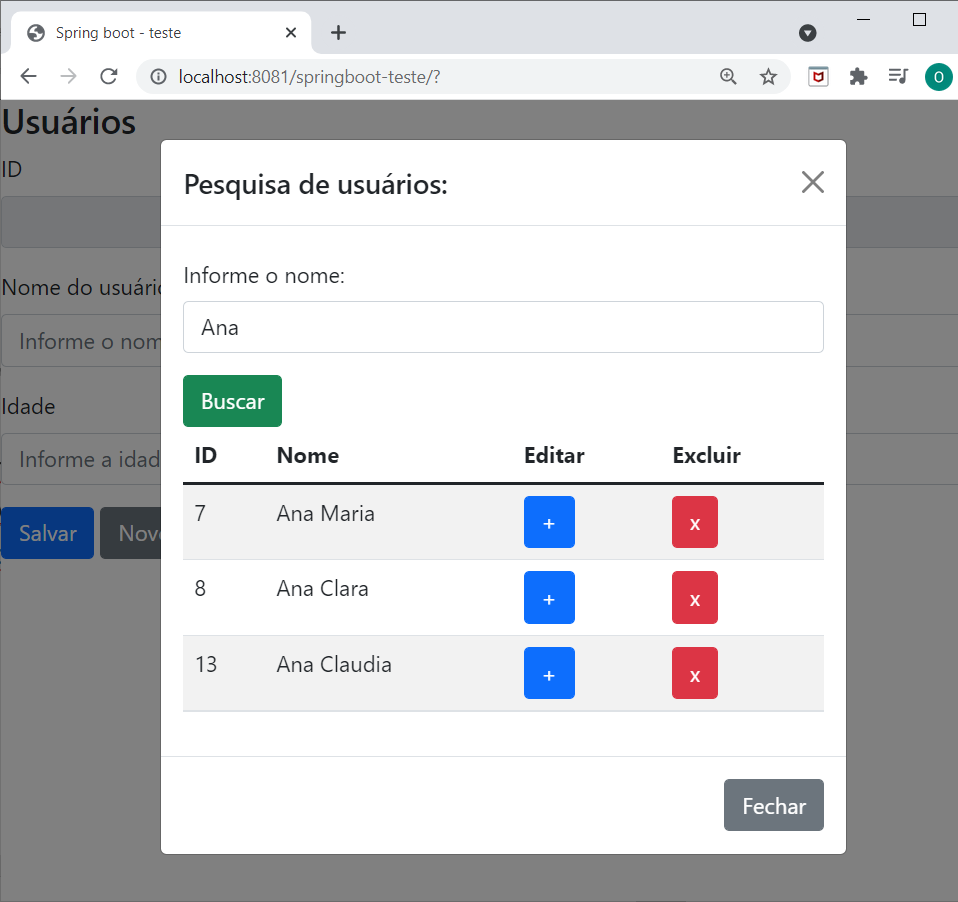
response[i].nome +'</td><td> <button type="button" onclick="colocarEmEdicao('+

response[i].id+')"class="btn btn-primary">+</button></td> <td><button type="button" class="btn btn-danger" onclick="excluirUsuario('+response[i].id+')">x</button></td></tr>')

}

...

Assim, teremos os botões para exclusão sendo apresentados e o script precisará ser construído:



function excluirUsuario(id){

if(confirm('Deseja realmente excluir?')) {

$.ajax({

method : "DELETE",

url : "delete",

data : "iduser=" + id ,

success : function(response) {

$('#'+ id).remove();

alert(response);

}

}).fail(function(xhr, status, errorThrown) {

alert("Erro ao deletar usuario por id: " + xhr.responseText);

});

}

}

Podemos usar a mesma rotina para excluir sendo chamada a partir de um botão Excluir no formulário principal. Vamos acrescentar um botão Excluir assim:

<button type="button" class="btn btn-danger" onclick="excluirUsuarioBotao()">Excluir</button>

E vamos implementar o script excluirUsuarioBotao() dessa forma:

function excluirUsuarioBotao(){

var id = $('#id').val();

if(id != null && id.trim() != ''){

excluirUsuario(id);

document.getElementById('formCadastroUser').reset();

}

}

Para melhorar as margens, podemos usar CSS (acrescentar no head):

<style type = "text/css">

html{

margin: 5%;

}

</style>

Para fazer o deploy e testar a aplicação podemos usar o Heroku (veja o tutorial).