

Criando uma simples Spring Rest API

criado por Erick Tomaz Oliveira

O começo

Java é uma linguagem razoavelmente antiga e que é amplamente usada para o desenvolvimento de aplicações desktop e para backend de todos os tipos de serviço e para entrar nesse mundo de possibilidades é necessário fazer uma decisão muito importante e que gera muitas discussões entre programadores: “editor de código ou IDE?”. Por ser de forte tipagem, juntamente com o “tempo de vida” e uma comunidade grande, ferramentas muito interessantes e úteis foram desenvolvidas para facilitar a vida do programador e devido às especificidades da linguagem, eu recomendo fortemente utilizar uma IDE especializada como é o caso do Eclipse (experiência própria tentando aprender Java utilizando o Visual Studio Code).

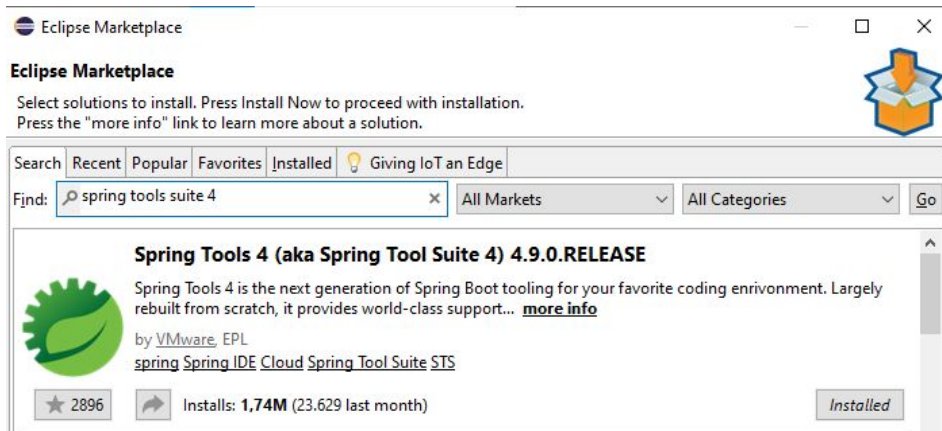
Além disso, como toda linguagem, essa possui um sistema de pacotes, que em outras linguagens podem ser chamados de módulos mas é o mesmo princípio de um compilado de códigos que possuem funções específicas para facilitar o desenvolvimento de aplicações. Com o passar do tempo, um conjunto desses pacotes passam a ser necessários para simplesmente inicializar um projeto e fazer esse manejo de conteúdos manualmente é uma tarefa desnecessariamente trabalhosa. Para isso existe um framework denominado Spring que cria um ecossistema com a base para a criação de projetos para a web e possui uma ferramenta para iniciar esse tipo de aplicação.

Inicializando um projeto

O Eclipse possui um método simples para a inicialização de projetos Spring mas antes de tudo é importante instalar o Spring Tools Suite 4, um conjunto de ferramentas para complementar a IDE.

Na área de pesquisa, digite “Marketplace” e clique na primeira opção que abrirá uma nova janela. Nela, digite “spring tools suite 4” e clique no botão de instalar.





Existem duas formas de iniciar um projeto spring: a primeira é utilizar o site <https://start.spring.io/> para selecionar as opções para a criação de projeto e depois de configurar tudo, baixar o projeto em zip e abri-lo no eclipse; o segundo parte utiliza das ferramentas embutidas na IDE para criar esse pacote de base:

Na barra de pesquisa digite e selecione “Spring Starter Project” e irá surgir a seguinte janela:

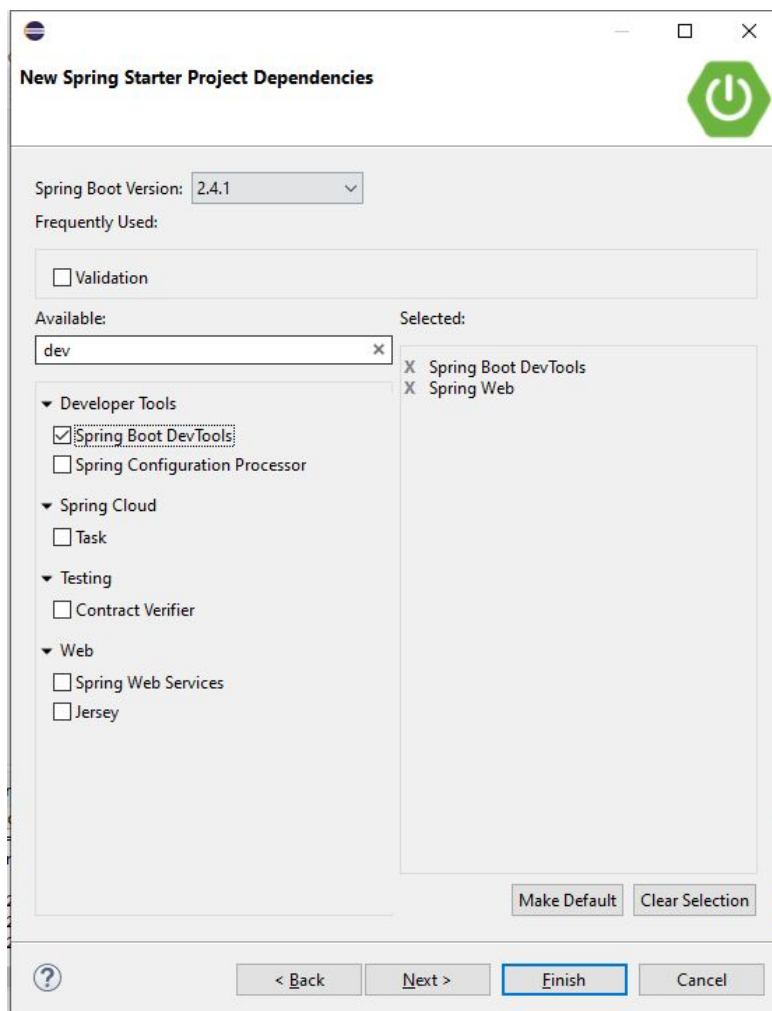
The screenshot shows the 'New Spring Starter Project' dialog box. It has a title bar with the Eclipse logo and a green power button icon. The main area contains several fields and checkboxes. The 'Service URL' field is set to 'https://start.spring.io'. The 'Name' field is 'demo'. There is a checked checkbox for 'Use default location' and a 'Location' field set to 'C:\Users\User\eclipse-workspace\demo'. Below this are fields for 'Type' (Maven), 'Packaging' (Jar), 'Java Version' (11), and 'Language' (Java). There are also fields for 'Group' (com.example), 'Artifact' (demo), 'Version' (0.0.1-SNAPSHOT), 'Description' (Demo project for Spring Boot), and 'Package' (com.example.demo). At the bottom, there is a 'Working sets' section with an unchecked checkbox for 'Add project to working sets' and a 'New...' button. Below that is a 'Working sets:' dropdown menu and a 'Select...' button. At the very bottom, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'.

Essa janela é utilizada para configurar as características do projeto:

- Service URL: local de onde o projeto base será gerado e instalado.
- Name: Nome do projeto.
- Group: O nome do pacote raiz do projeto.
- Artifact: Nome do projeto.
- Package: também tem relação ao nome do pacote raiz. Se não for especificado, o valor do group será utilizado.
- Description: descrição do projeto.
- Version: versão do projeto.

Mudar os valores que são padrão da ferramenta é a critério, mas se for a primeira vez que estiver desenvolvendo um projeto, não é necessário; é melhor focar no desenvolvimento e ter noção do que essas configurações afetam a estrutura de um projeto Spring.

Selecione as opções de “Spring Web” e “Spring Boot DevTools”



O Spring Web vai instalar os pacotes necessários para desenvolver uma aplicação web e o Spring Boot DevTools irá auxiliar em alguns pontos no desenvolvimento de código em Java.

A maior parte do código é desenvolvida na mesma pasta que está localizada o arquivo utilizado para rodar o código. Nesse exemplo, esse arquivo é o `com.example.demo`.

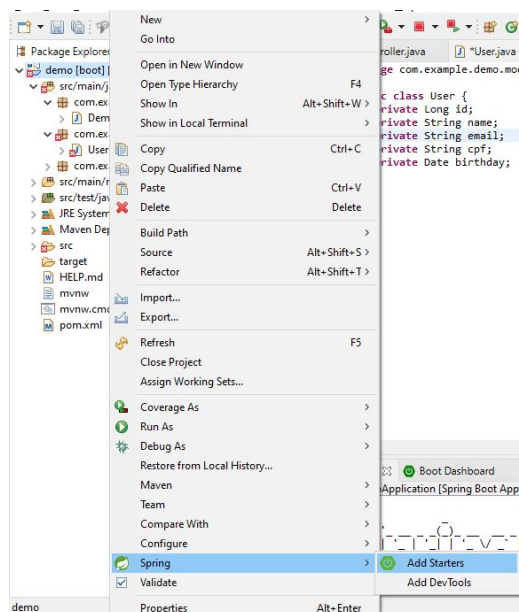
Características do projeto

Para esse projeto, iremos criar uma API Rest que irá receber “nome”, “email”, “CPF” e “data de nascimento”, fazer a validação dos dados, inserir no banco de dados e, como toda API REST, retornar o status correto e utilizar o método adequado para essa transação. Também iremos criar uma rota que retorne uma listagem de todos os cadastrados para facilitar a depuração do código.

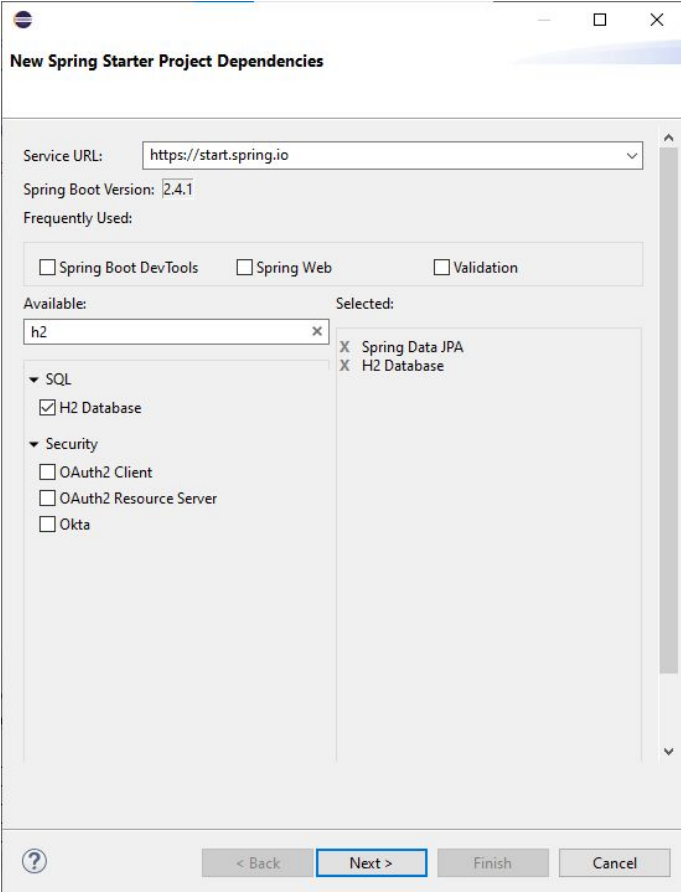
Pacotes extras para o projeto

Alguns pacotes serão necessários mais a frente.

Clique com o botão direito na pasta raiz e selecione spring > add stater



Selezione H2 (Hibernate) e Spring Data JPA



New Spring Starter Project Dependencies

Service URL:

Spring Boot Version:

Frequently Used:

☐ Spring Boot DevTools ☐ Spring Web ☐ Validation

Available:

Selected:

- X Spring Data JPA
- X H2 Database

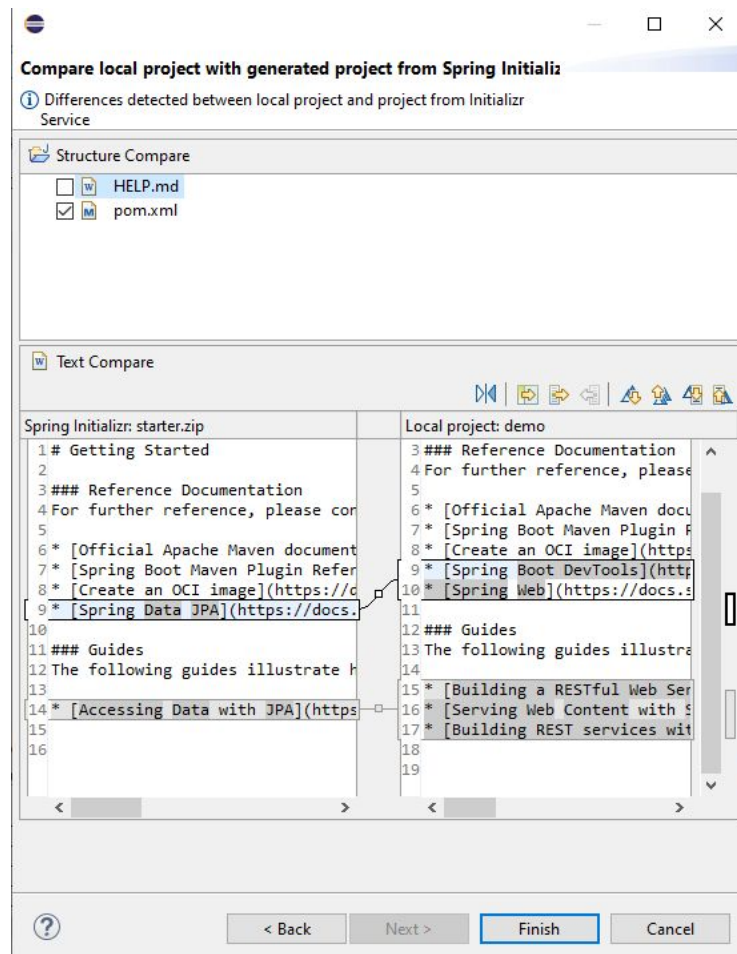
▼ SQL

- ☒ H2 Database

▼ Security

- ☐ OAuth2 Client
- ☐ OAuth2 Resource Server
- ☐ Okta

Marque a opção de “pom.xml” para poder efetuar mudanças nesse arquivo.

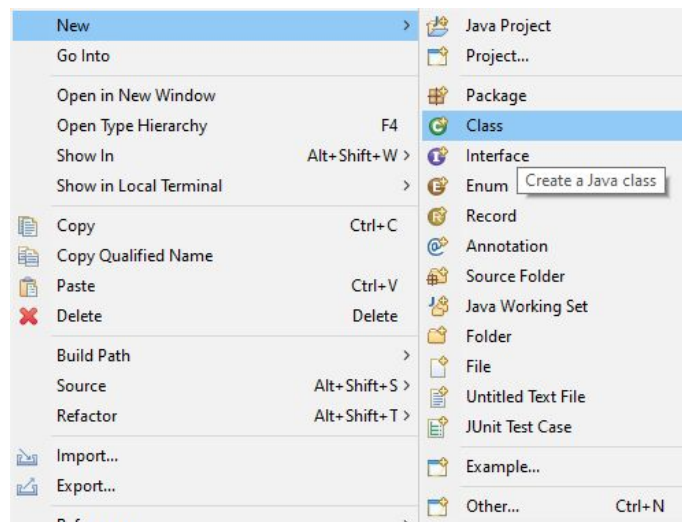


Criando um model

O model é responsável pela estruturação, lógica e a validação dos dados.

Tenho por preferência criar um model primeiro, principalmente quando sei quais serão os dados que serão armazenados pois assim fica tudo organizado para a criação das outras etapas da API.

O código abaixo é o model da aplicação para os dados do usuário. Esse model verifica se todos os dados estão presentes e se o email, CPF e data de aniversário estão estruturados corretamente.



```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Name is required")
    @Column(nullable = false)
    private String name;

    @NotBlank(message = "Email is required")
    @Email(message = "Email format invalid")
    @Column(nullable = false, unique = true)
    private String email;

    @NotBlank(message = "CPF is required")
    @CPF(message = "CPF invalid")
    @Column(nullable = false, unique = true)
    private String cpf;

    @NotNull(message = "Birthday is required")
    @Column(nullable = false)
    @Past
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy")
    private LocalDate birthday;

    //Getters and setters

}
```

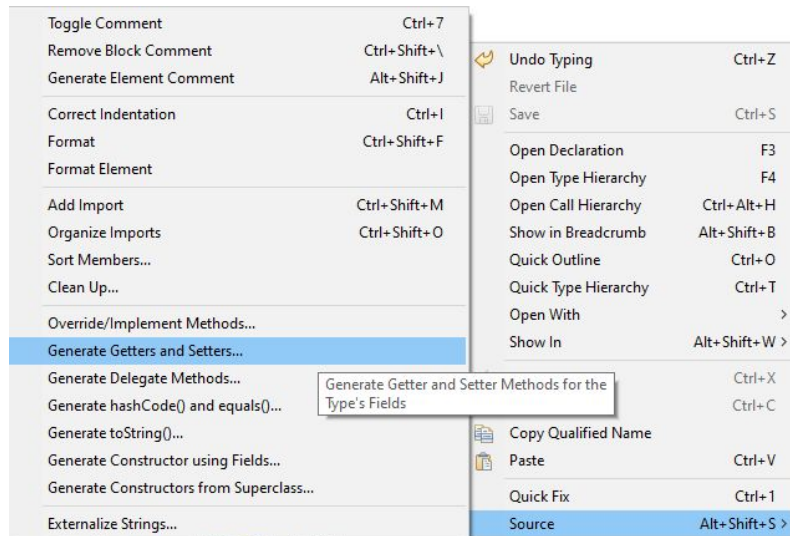
Cada bean tem uma funcionalidade em específica:

- **@Entity**: Ele define a classe como um model, a estrutura de dados.
- **@Id**: Define a variável como o ID da tabela.
- **@GeneratedValue**: define a forma que os ids serão gerados; a forma IDENTITY diz que a escolha de geração fica a cargo do banco de dados.
- **@NotBlank**: define a validação onde é obrigatório passar um valor para o banco de dados e ele não pode ser null.

- **@Column**: naturalmente, o spring consegue dizer que cada nova variável criada dentro da classe com **@Entity** é uma nova coluna da tabela. Entretanto, caso seja necessário informar alguma característica da tabela, é necessário utilizar **@Column** para definir essa característica da coluna em si, como é o caso de nullable que quando definido como false diz que a coluna não pode estar vazia (a diferença entre caracterizar assim e usar **@NotBlank** é que o segundo é uma validação antes de inserir ou tentar inserir os valores no banco de dados enquanto o primeiro é uma característica da coluna, ou seja, caso não seja feita uma validação, a tabela ainda sim não irá aceitar e irá lançar um erro). Outro parâmetro que pode ser passado é o unique que define que a coluna não pode possuir dois campos com o mesmo valor.
- **@CPF**: valida a formatação do CPF. Este possui algumas características intrínsecas aos valores e à organização dos mesmos.
- **@Email**: utilizado para validar a string como uma estrutura de email.
- **@NotNull**: só identifica se o campo não é nulo. O tipo LocalDate não aceita **@NotBlank**.
- **@Past**: valida se a string, já no formato correto, possui uma data do passado.
- **@JsonFormat**: utilizado para desestruturar e verificar a estrutura de uma string em um json de acordo com o formato passado como parâmetro.

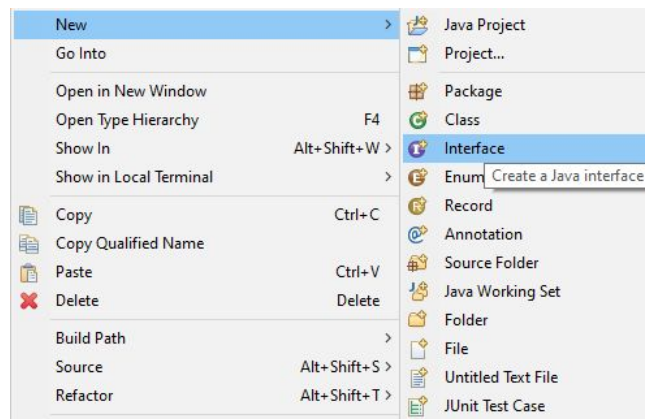
Alguns desses beans validations possuem um parâmetro “message” que facilita a passagem de um mensagem customizada para caso o dado não preencha os requisitos para entrar no banco de dados. A maioria das validações são feitas pelo javax e somente o CPF é feito pelo Hibernate.

Por fim, é necessário criar os getters e setters. No eclipse, é só clicar com o botão direito e ir em source > generate getters and setters.



Criando um repository

O repository é a interface entre o banco de dados e a aplicação. Com ele é possível utilizar as regras e estruturas criadas no model para efetuar ações concretas no banco de dados. Para criar um repository simples utilizando JPA é necessário criar uma interface e usar um extends para um JpaRepository e passar o model/entity criado anteriormente e o tipo de dado que é o ID do model/entity como parâmetros:



```
package com.example.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.example.demo.model.User;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

}
```

Criando um controller


O controller é a parte do código que gerencia todos os processos, recebe informações e processa da forma desejada. Nele é onde se colocam as rotas.

Iremos adicionar essa classe em uma pasta separada de controllers e para isso é necessário adicionar “.controller” no campo “package”.

New Java Class

Java Class

Create a new Java class.



Source folder:

demo/src/main/java

Browse...

Package:

com.example.demo.controller

Browse...

☐ Enclosing type:

Browse...

Name:

UserController

Modifiers:

☒ public

☐ package

☐ private

☐ protected

☐ abstract

☐ final

☐ static

Superclass:

java.lang.Object

Browse...

Interfaces:

Add...

Remove

Which method stubs would you like to create?


☐ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments



Finish

Cancel

```

package com.example.demo.controller;

//importações

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

    @Autowired
    private UserRepository userRepository;

    @GetMapping
    public List<User> get() {
        return userRepository.findAll();
    }

    @RequestMapping(method=RequestMethod.POST)
    public ResponseEntity<String> save(@Valid @RequestBody User user, BindingResult errors) {

        if(errors.hasErrors()) {
            List<String> errosArray = new ArrayList<>();

            for(ObjectError error: errors.getAllErrors()){
                errosArray.add(error.getDefaultMessage().toString());
            };

            return ResponseEntity.badRequest().body(errosArray.toString());
        }

        try {
            userRepository.save(user);
            return new ResponseEntity<String>(HttpStatus.CREATED);
        } catch(DataIntegrityViolationException e) {
            return ResponseEntity.badRequest().body("CPF or email already registered");
        }
    }
}

```

- **Para definir um controller**, é necessário utilizar a notação **@RestController** e para definir a rota principal desse controller é adiciona-se uma string contendo o nome da rota como parâmetro da notação **@RequestMapping**.
- **Para podermos realizar ações no banco de dados**, criaremos um objeto do repository criado anteriormente e marcamos com um ponto de injeção com a notação **@Autowired**.
- **Para criar uma rota**, utiliza-se da notação de **@(nome do método http)Mapping**, como **@GetMapping** que é utilizado para rotas do tipo get. É possível também utilizar a anotação **@RequestMapping** e passar o método escolhido como parâmetro. Nesse caso, foi utilizada a segunda opção para a rota do tipo post, pois a notação **@Valid**, que veremos adiante, não funcionou na primeira notação, ou seja, **@PostMapping**.
- **ResponseEntity** é uma classe para poder responder a uma solicitação.
- **@Valid** faz com que o model User seja validado quando os dados passados pelo body forem capturados por essa variável utilizando a notação **@RequestBody**.
- A classe **BindingResult** captura os resultados da validação caso haja algum dado inválido.
- O **if** é utilizado para caso haja dados inválidos, eles sejam mapeados para poder ser retornado um array das mensagens que foram personalizadas no modelo.

- O **try/catch** fiscaliza o método `save` do `userRepository` e caso haja um erro, especialmente `DataViolationExceptionHandler` (caso aconteça alguma violação das regras colocadas no banco de dados, no caso deste código seria tentar colocar valores iguais na coluna CPF e email), retorna uma mensagem no corpo da resposta com o status HTTP apropriado. Caso não haja erros, é retornado somente o status `CREATED` como resposta à requisição.

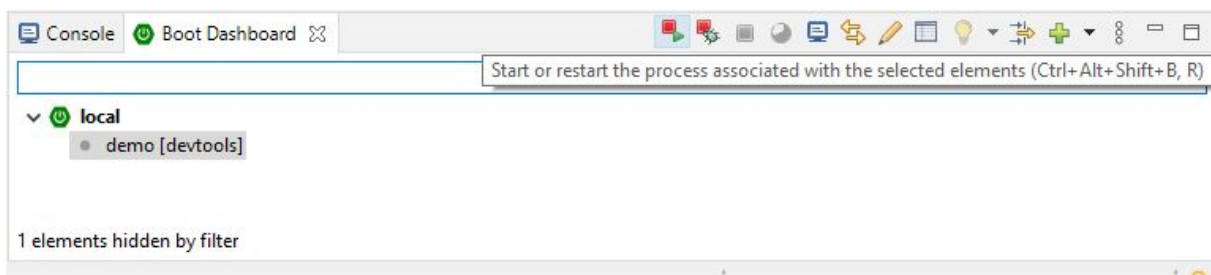
Testando o código

Se tudo foi feito corretamente, é possível rodar o código seguindo os seguintes passos:

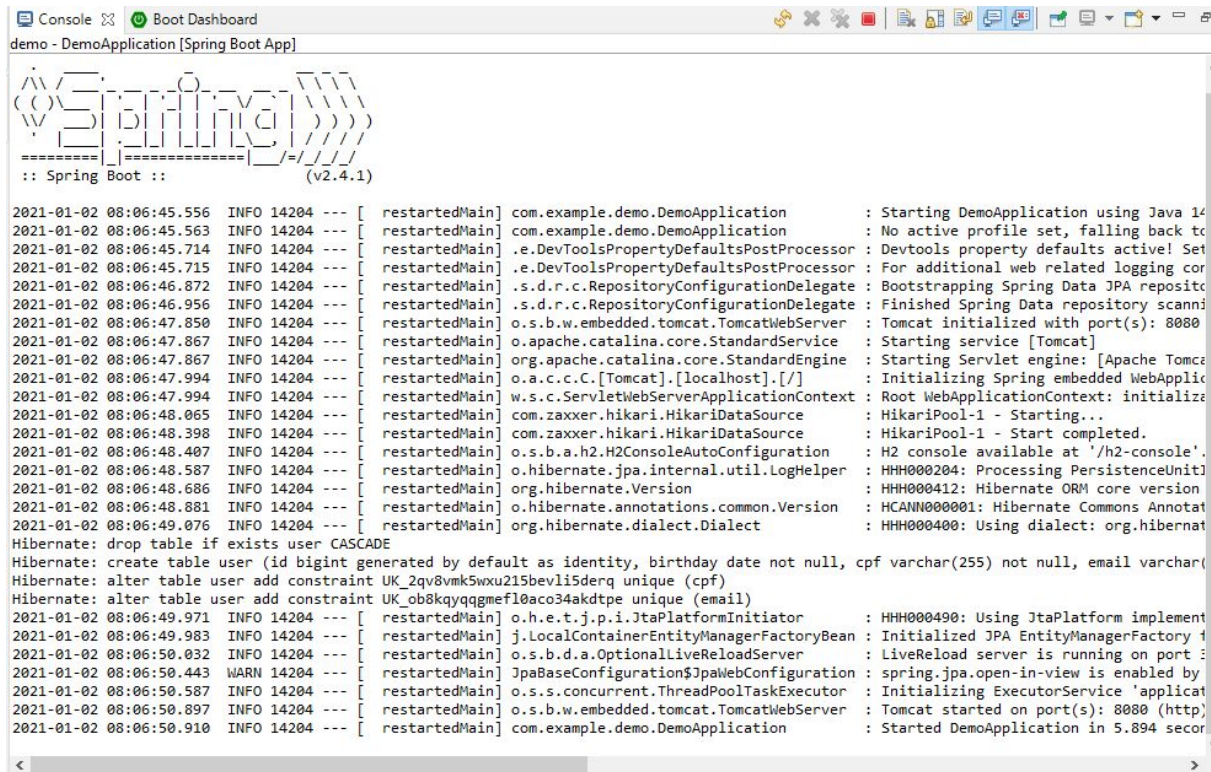
Selecione o projeto a ser rodado na aba de Boot Dashboard



Selecione o botão para iniciar a aplicação



Se estiver tudo correto, o console apresentará uma mensagem parecida com a seguinte,



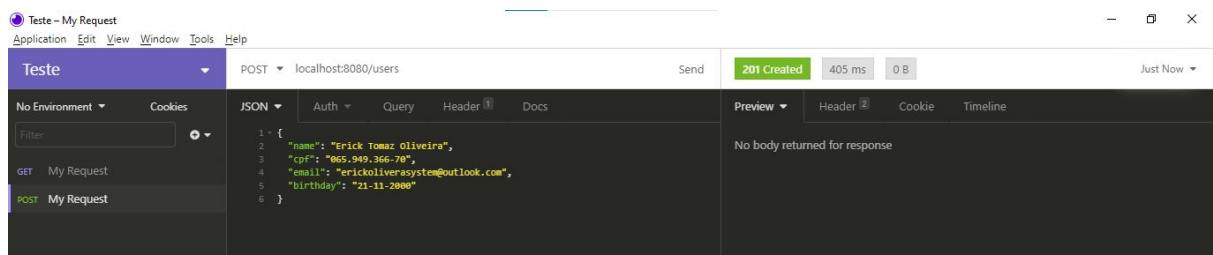
```
demo - DemoApplication [Spring Boot App]
:: Spring Boot ::
(v2.4.1)

2021-01-02 08:06:45.556 INFO 14204 --- [ restartedMain] com.example.demo.DemoApplication : Starting DemoApplication using Java 14
2021-01-02 08:06:45.563 INFO 14204 --- [ restartedMain] com.example.demo.DemoApplication : No active profile set, falling back to
2021-01-02 08:06:45.714 INFO 14204 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set
2021-01-02 08:06:45.715 INFO 14204 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging cor
2021-01-02 08:06:46.872 INFO 14204 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA reposi
2021-01-02 08:06:46.956 INFO 14204 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scan
2021-01-02 08:06:47.850 INFO 14204 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080
2021-01-02 08:06:47.867 INFO 14204 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-01-02 08:06:47.867 INFO 14204 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomc
2021-01-02 08:06:47.994 INFO 14204 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplic
2021-01-02 08:06:47.994 INFO 14204 --- [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initializ
2021-01-02 08:06:48.065 INFO 14204 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2021-01-02 08:06:48.398 INFO 14204 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2021-01-02 08:06:48.407 INFO 14204 --- [ restartedMain] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'.
2021-01-02 08:06:48.587 INFO 14204 --- [ restartedMain] org.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnit
2021-01-02 08:06:48.686 INFO 14204 --- [ restartedMain] org.hibernate.Version : HHH000412: Hibernate ORM core version
2021-01-02 08:06:48.881 INFO 14204 --- [ restartedMain] org.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotat
2021-01-02 08:06:49.076 INFO 14204 --- [ restartedMain] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate
Hibernate: drop table if exists user CASCADE
Hibernate: create table user (id bigint generated by default as identity, birthday date not null, cpf varchar(255) not null, email varchar(
Hibernate: alter table user add constraint UK_2qv8vmk5wxu215bevli5derq unique (cpf)
Hibernate: alter table user add constraint UK_ob8kqyqqgmefl0aco34akdtpe unique (email)
2021-01-02 08:06:49.971 INFO 14204 --- [ restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implement
2021-01-02 08:06:49.983 INFO 14204 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory f
2021-01-02 08:06:50.032 INFO 14204 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 3
2021-01-02 08:06:50.443 WARN 14204 --- [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by
2021-01-02 08:06:50.587 INFO 14204 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicat
2021-01-02 08:06:50.897 INFO 14204 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http
2021-01-02 08:06:50.910 INFO 14204 --- [ restartedMain] com.example.demo.DemoApplication : Started DemoApplication in 5.894 secon
```

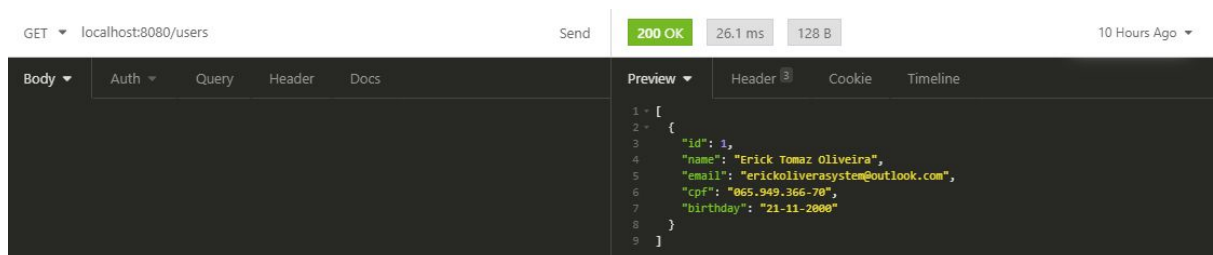
Para testar as rotas, utiliza-se de alguma ferramenta para fazer solicitações. Nesse projeto, utilizaremos Insomnia, ferramenta leve e muito intuitiva.

No Insomnia, definimos um nome para o workspace, o nome, tipo e URL da rota e o corpo da requisição.

Para a rota de criação de usuário,



Para a rota de listagem de usuário,



E assim, finalizamos o projeto. Obrigado pela atenção.