Criando um projeto Spring Boot e implementando uma API REST com JWT

Autor: Ian Rangel Passos

Disciplina: Sistemas Orientados a Objetos

Requisitos: Spring Boot com API REST
IntelliJ 2022.1
MySQL 8.0
Java 11
Postman

1. Objetivo do tutorial

Este tutorial tem como objetivo mostrar como criar um projeto Java utilizando o framework Spring, como fazer a conexão com o banco de dados relacional MySQL e como fazer a implementação de uma API REST.

2. Tecnologias

2.1. Spring

O Spring é um framework que fornece um modelo de configuração e programação para aplicações modernas do tipo Java. A principal característica desse framework é o suporte da infraestrutura em um nível básico, automatizando e simplificando o desenvolvimento dessas aplicações. Dentre os módulos do Spring, neste projeto será utilizado o Spring Data JPA. Esse é um modelo que permite o acesso a um banco de dados, visando facilitar o acesso a esses dados e ao mesmo tempo, mantendo o padrão de armazenamento dos mesmos.

2.2. MySQL

O MySQL é um software de código aberto usado na criação e gerenciamento de bancos de dados relacionais (RDBMS - Relational Database Management Systems) com um modelo de cliente-servidor. Um banco de dados é um local onde dados são armazenados e gerenciados. No MySQL, cliente e servidor se comunicam usando uma linguagem específica - a SQL (Structured Query Language). Um software RDBMS é escrito em uma outra linguagem de programação, porém sempre utilizará a sintaxe SQL para comunicar-se com o banco de dados.

2.3. API REST

API REST é uma interface de programação de aplicações (Application Programming Interface) que se encontra de acordo com as restrições da arquitetura REST, permitindo a interação com servidores web RESTful. REST é a sigla para Representational State Transfer, ou transferência de estado representacional.

2.3.1. O que é uma API?

Uma API é um conjunto de protocolos e restrições usados no desenvolvimento e na integração de aplicações. Ela pode ser resumida como um um contrato entre um provedor e um usuário, definindo as entradas exigidas pelo consumidor e a resposta exigida pelo produtor.

2.3.2. Arquitetura REST

Para uma arquitetura ser considerada RESTful, ela precisa seguir os seguintes critérios:

- Ter uma arquitetura cliente/servidor com solicitações geradas por HTTP.
- Estabelecer uma comunicação stateless entre cliente e servidor. Ou seja, nenhuma informação do cliente é armazenada em métodos GET e todas as solicitações são separadas.
- Armazenamento de dados em cache, visando a otimização das interações cliente-servidor.
- Interface uniforme entre os componentes para que as informações sejam transmitidas em um formato padrão.
- Sistema em camadas que organiza os tipos de servidores envolvidos na recuperação das informações solicitadas em hierarquias que o cliente não pode ver.
- (Opcional) A capacidade de enviar um código executável do servidor para o cliente para ampliar a funcionalidade disponível ao cliente.

As restrições dessa arquitetura podem ser implementadas conforme necessário, isso torna as APIs REST mais leves e rápidas.

2.4. JWT

JWT (JSON web token) é um padrão que tem como objetivo transmitir ou armazenar de forma compacta objetos JSON entre diferentes aplicações. O JWT é utilizado em dois principais cenários:

- Depois que o usuário já estiver autenticado, cada requisição posterior possuirá o JWT, permitindo que o usuário acesse esse recurso sem precisar colocar a senha novamente.
- Transmissão de informações com segurança entre ambas as partes.

Esses tokens são compostos por 3 partes separadas por pontos:

- Header: Neste componente é colocado o tipo de token (JWT) e o algoritmo de assinatura utilizado.
- Payload: Esta seção contém as claims, ou declarações sobre um objeto, por exemplo o nome e celular de uma pessoa,
- Signature: Para criar a parte da assinatura, você deve pegar o cabeçalho codificado, o payload codificado, a chave secreta, o algoritmo especificado no cabeçalho e assiná-lo.

3. Conceitos para o tutorial

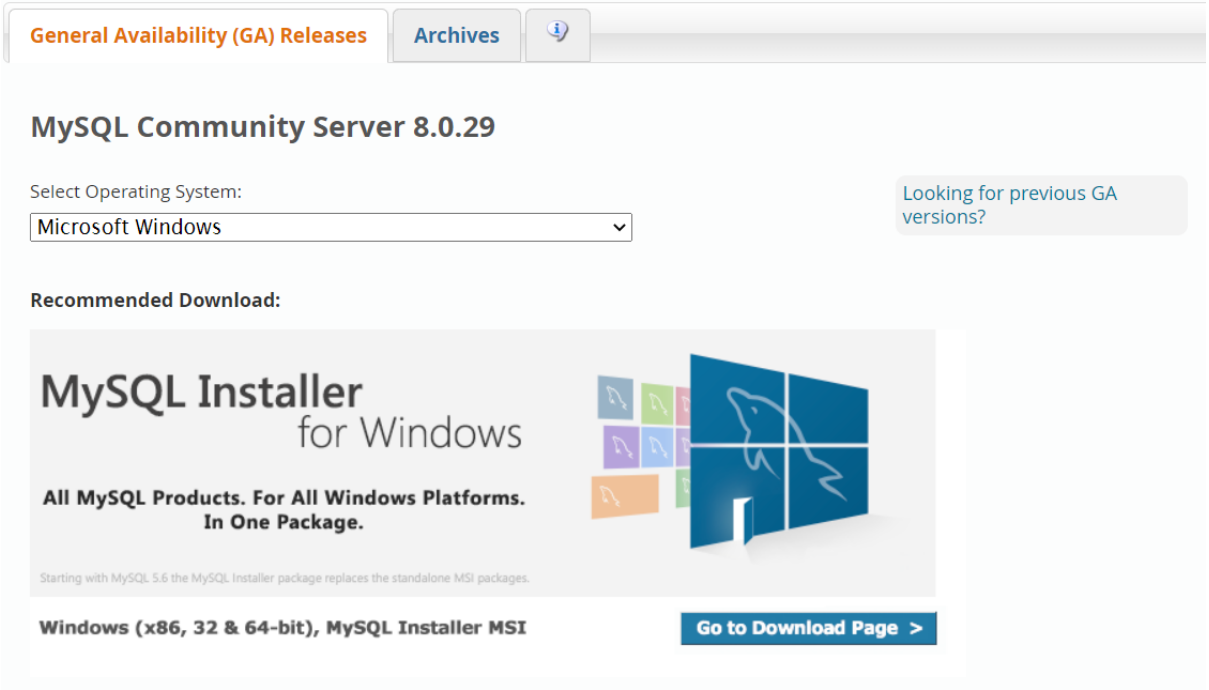
Para a realização deste tutorial, foi criado um projeto utilizando a IDE IntelliJ. A arquitetura utilizada seguirá o seguinte padrão:

- A camada model contém as classes lógicas do sistema.

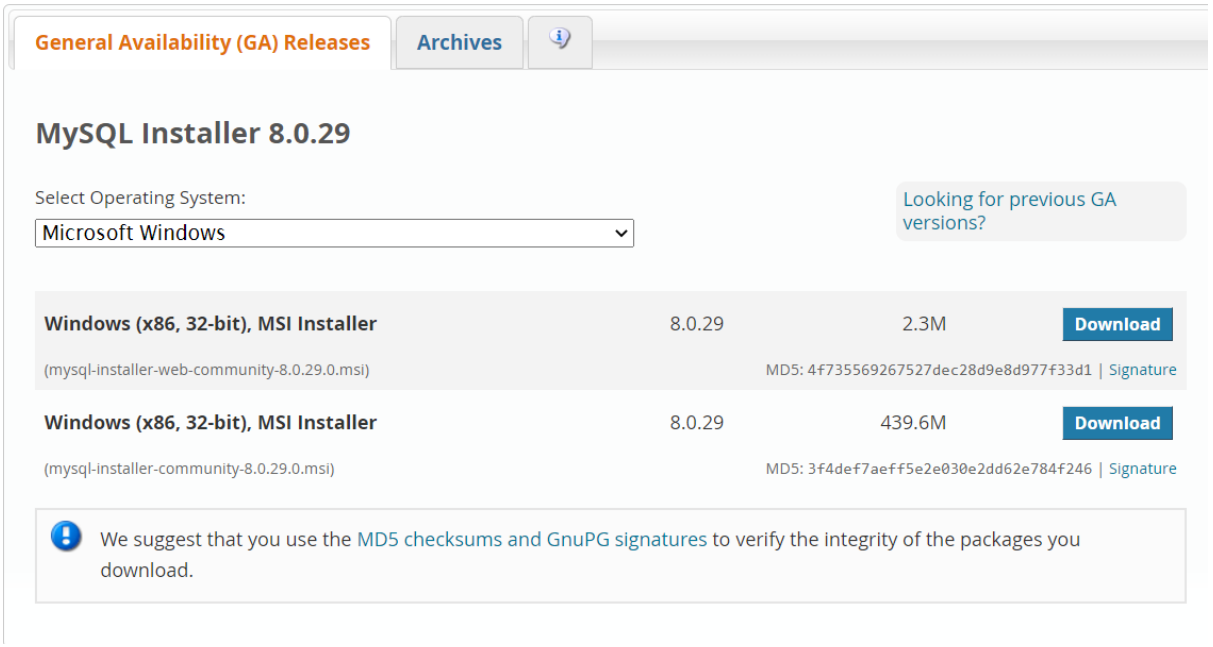
- A camada repository possui as classes de persistência.
- A camada service contém as classes que fazem a ligação entre a camada da aplicação e a camada de repository.

4. Banco de dados MySQL

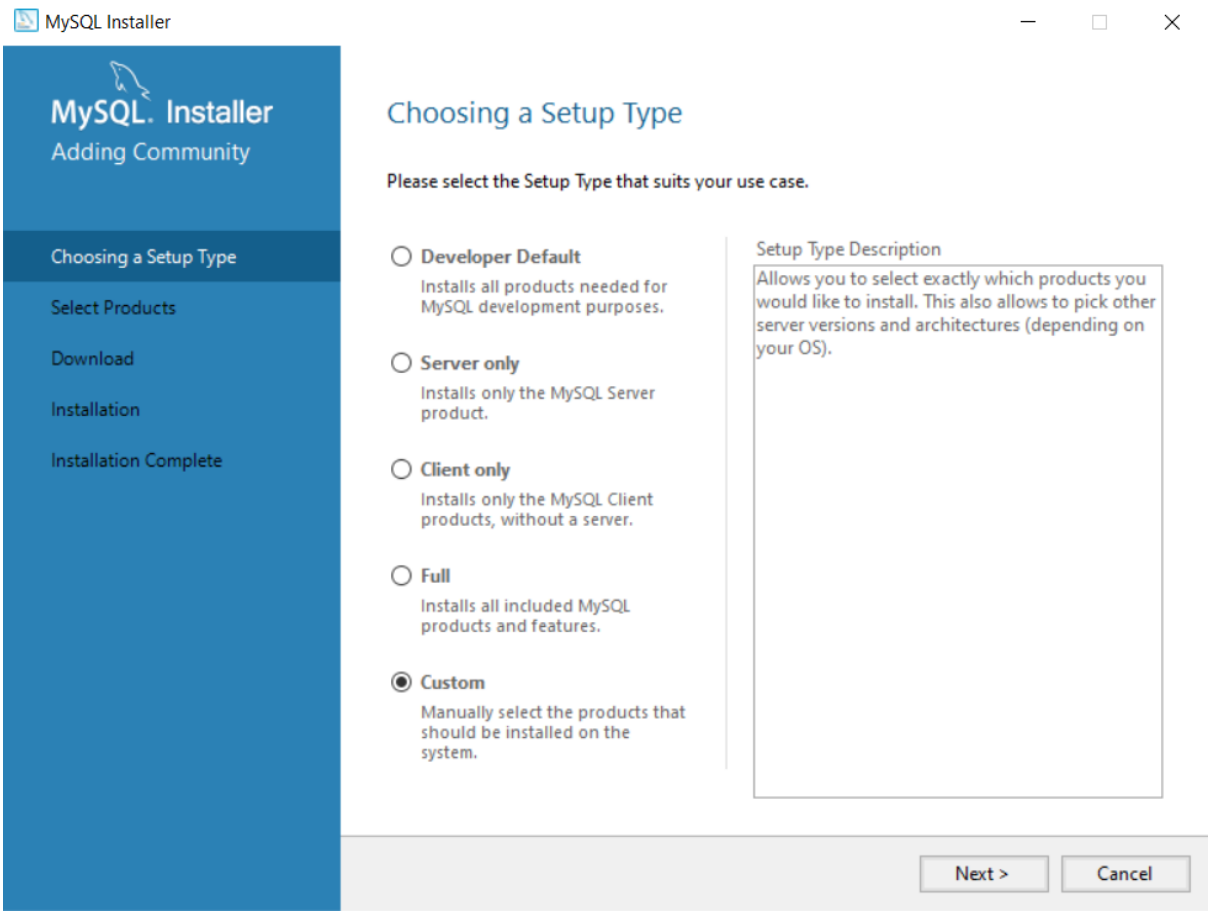
Antes de criar o projeto, é preciso instalar o banco de dados MySQL e gerar uma conexão com o mesmo. Primeiro, acesse o link [MySQL :: Download MySQL Community Server](), e instale o pacote MySQL para windows:

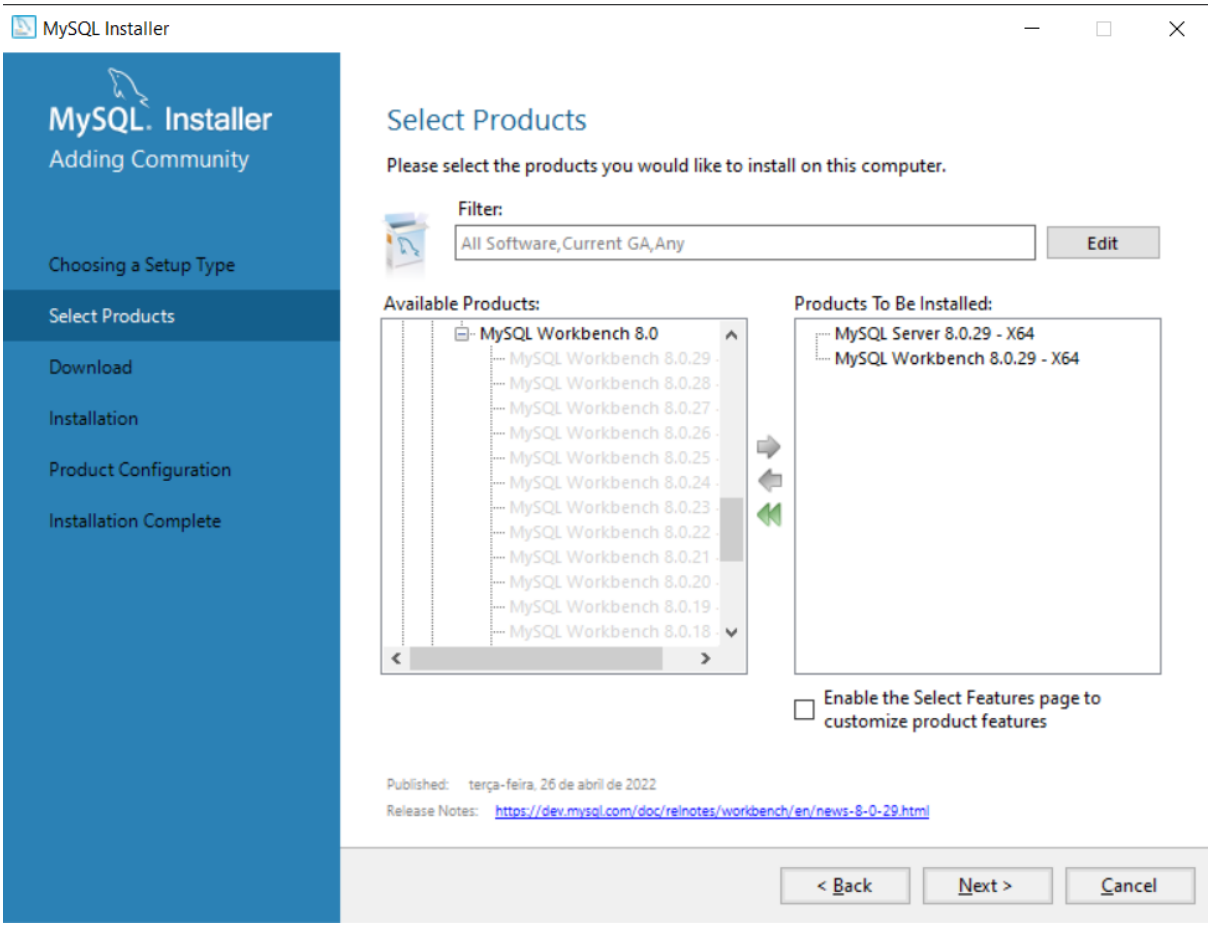


Então, selecione a primeira opção de download, que não inclui todos os pacotes providos pelo MySQL:

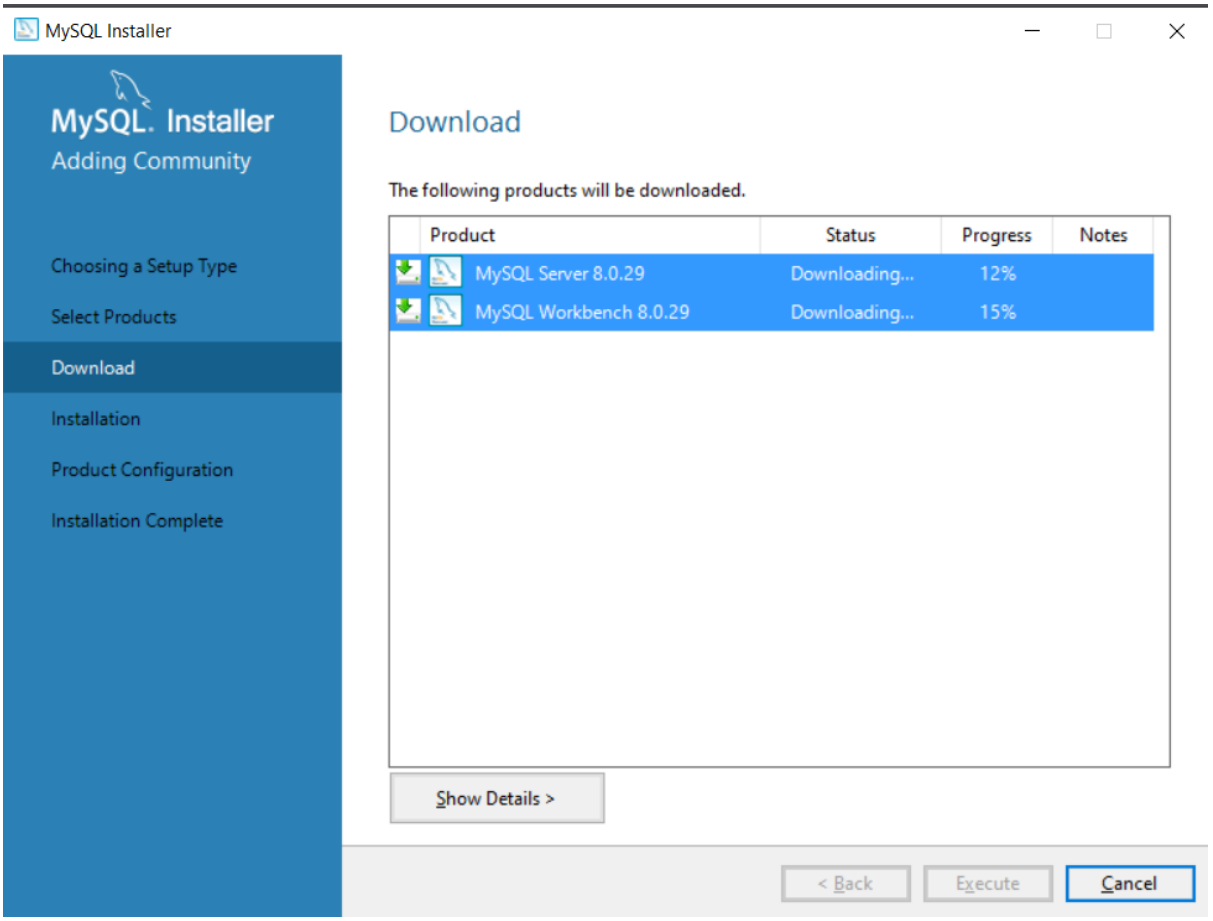


As configurações do instalador serão as seguintes:



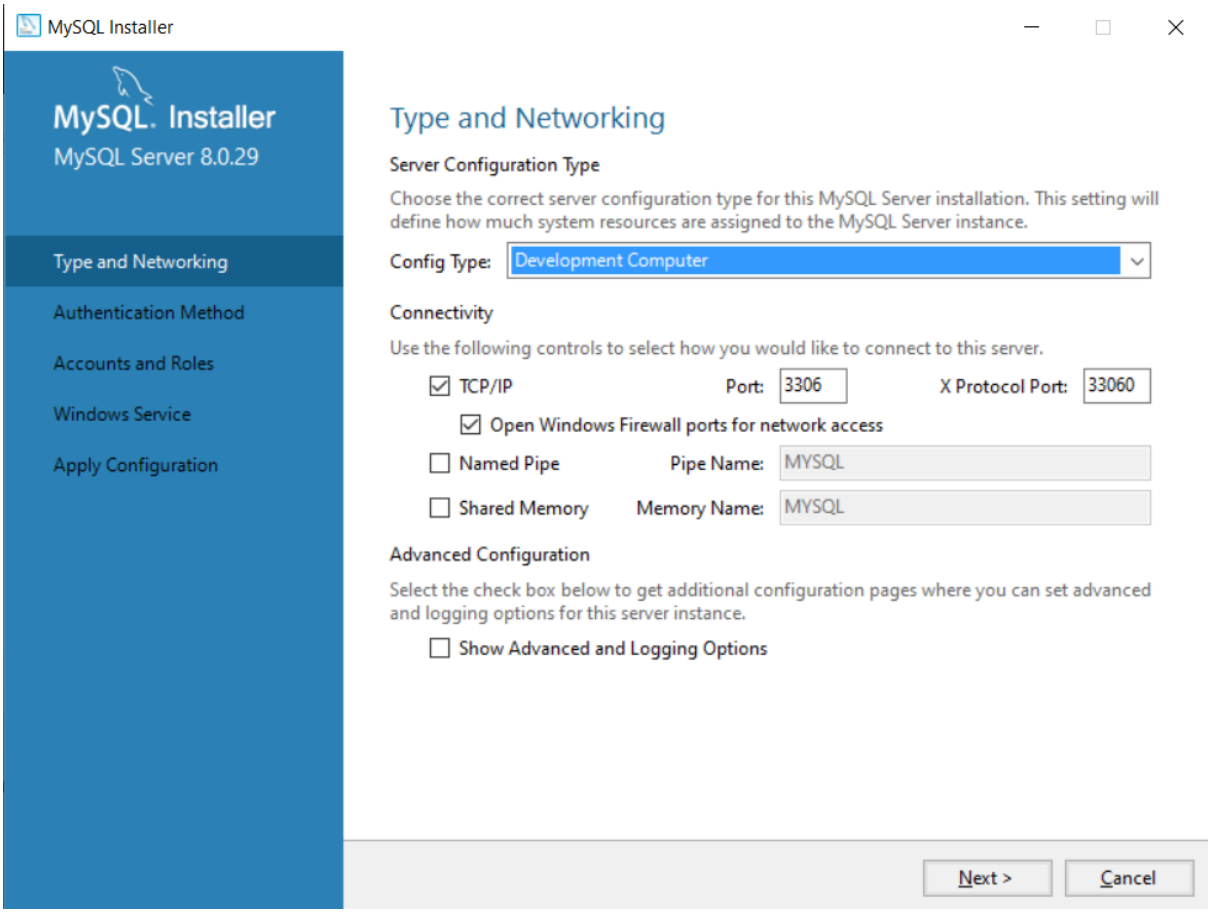


Escolha baixar somente o Server e o Workbench.

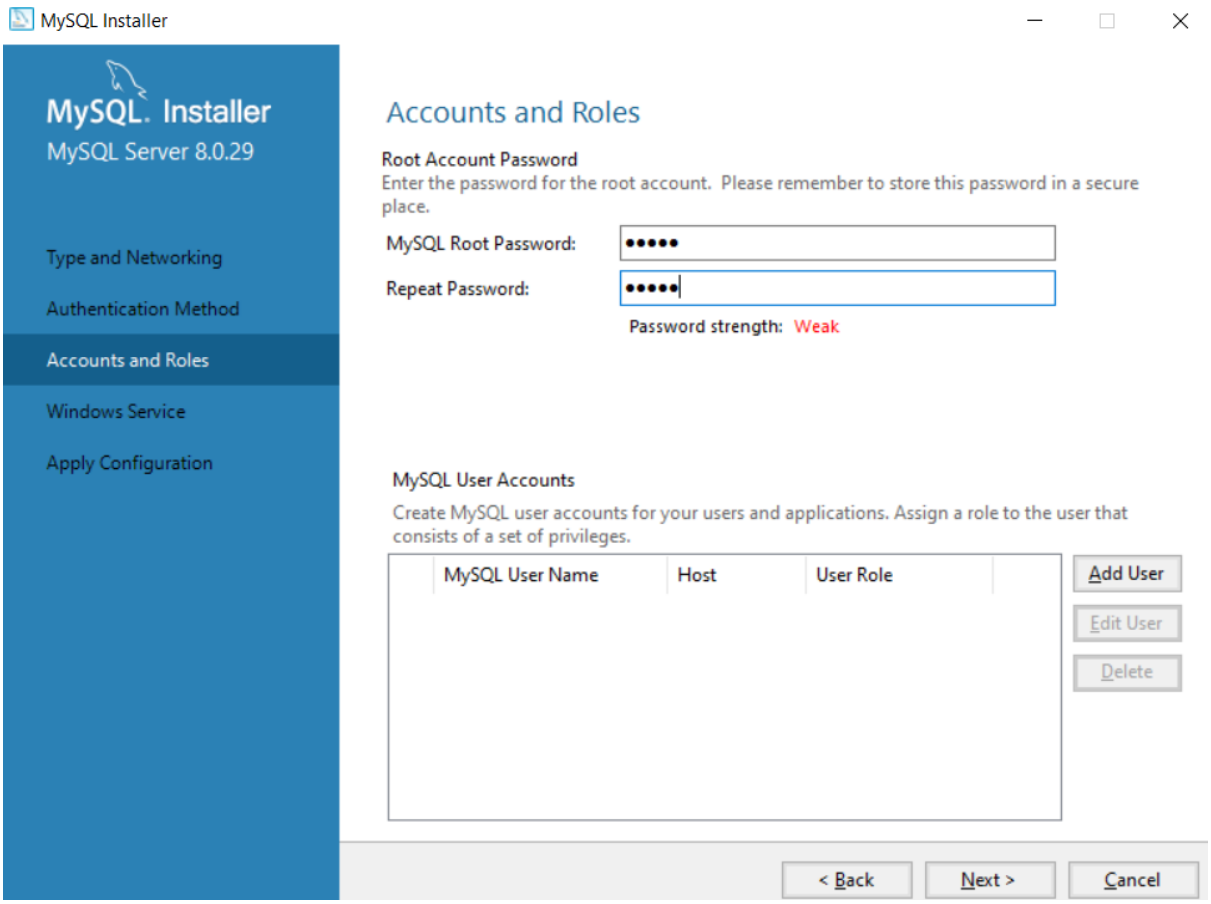


O download e instalação serão executados.

Depois de instalado, é preciso configurar o servidor MySQL:

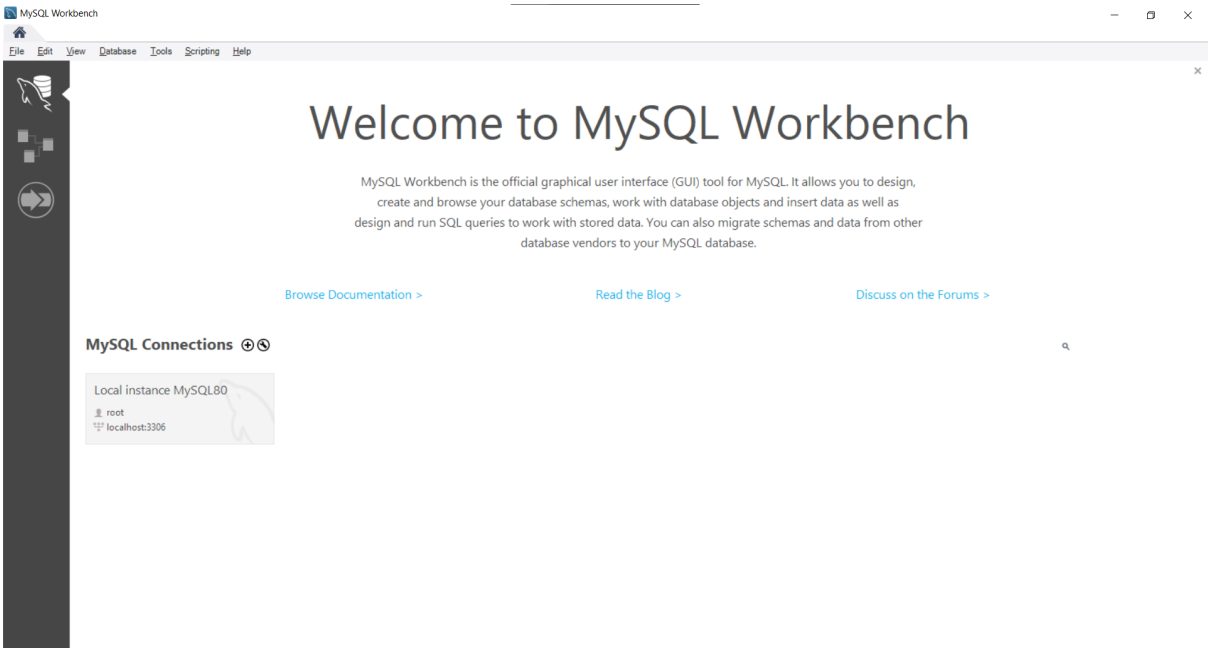


Escolha uma senha:

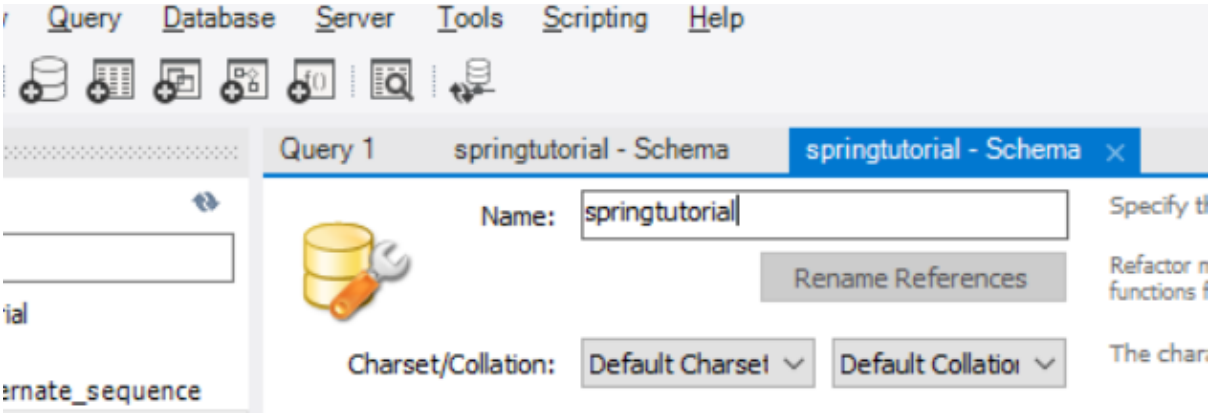


As outras opções podem ser mantidas da mesma forma que são apresentadas.

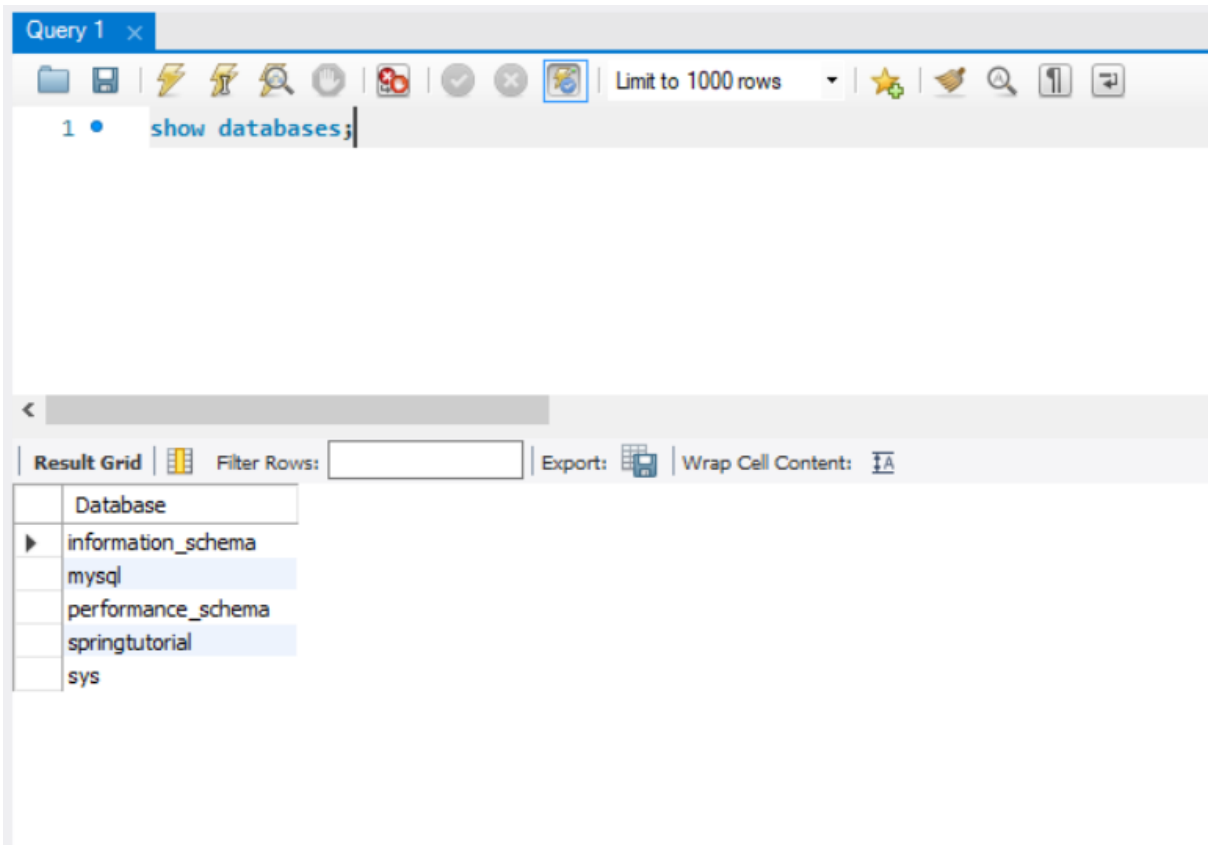
Quando a configuração terminar, o MySQL irá abrir já com uma conexão existente:



Crie um database com o nome springtutorial:



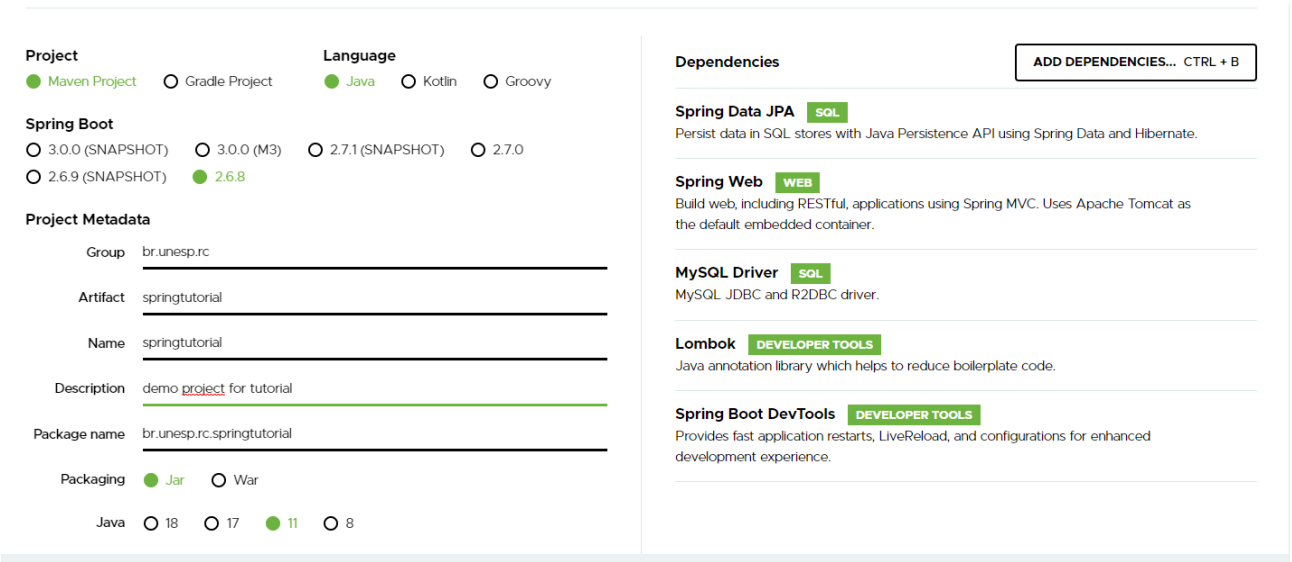
Para testar, acesse essa conexão e execute um comando, como “show databases;” e verifique se a database criada está lá:



Se sim, o banco está configurado e conectado.

5. Criando o projeto

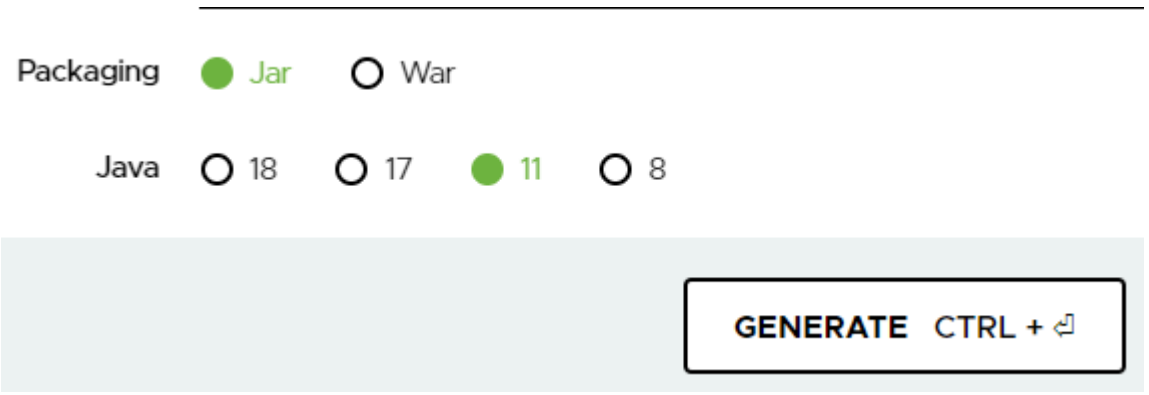
Para criar um projeto com o Spring, basta acessar o site Spring Initializr (https://start.spring.io/) e configurá-lo da seguinte forma:



Quanto às dependências selecionadas:

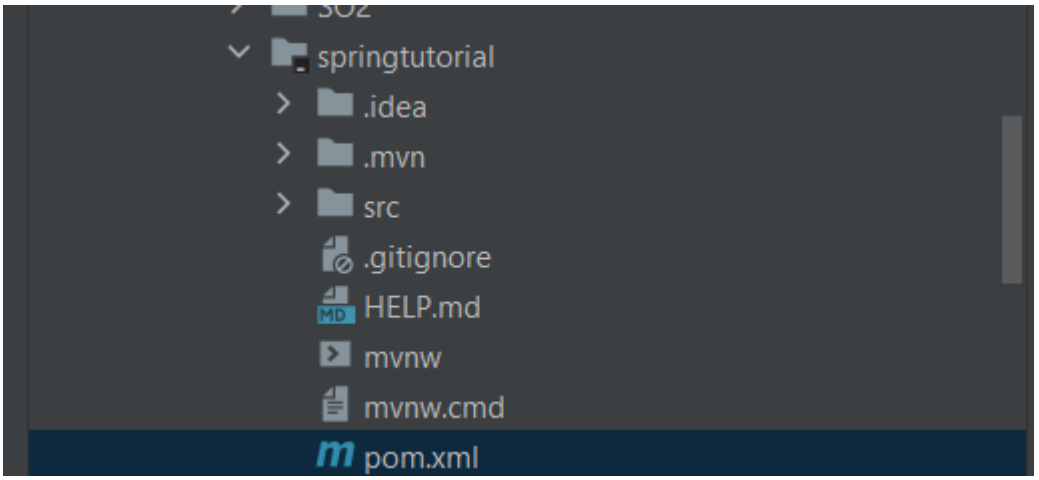
- Spring Data JPA foi selecionado para o trabalho com o banco de dados MySQL
- MySQL driver foi selecionado pelo mesmo motivo
- Spring Boot DevTools garante que as alterações no código sejam atualizadas em tempo real
- Lombok ajuda a reduzir a quantidade de código dentro das classes
- Spring Web foi selecionado para permitir a criação da API REST

Para criar o projeto, clique no botão generate.



6. Projeto Criado

Agora que o projeto foi inicializado, é possível abri-lo no IntelliJ:



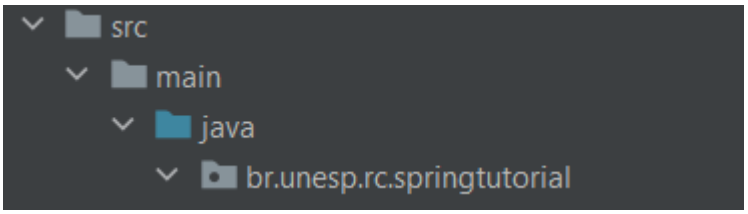
Para isso, selecione o arquivo xml e escolha a opção “open as a project”. Com o projeto criado, explore as seções dele:

5.1. Arquivo XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.8</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>br.unesp.rc</groupId>
    <artifactId>springtutorial</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springtutorial</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
```

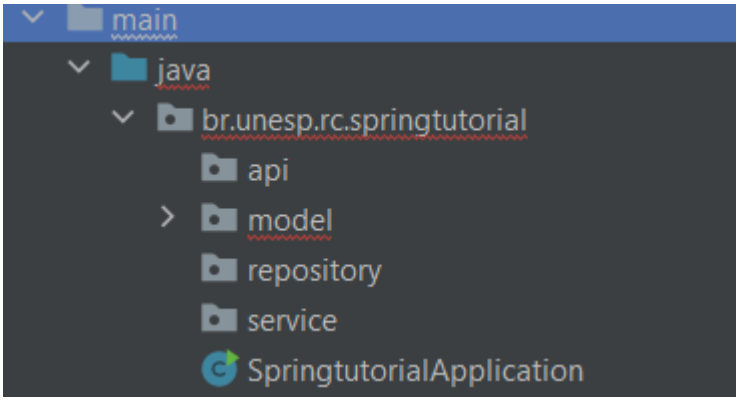
Este é o arquivo que o Spring Boot criou, ele contém todas as dependências que foram escolhidas, a versão do Java, descrição e nome do projeto. Aqui serão acrescentadas novas dependências Maven conforme necessário.

5.2. Pacote principal



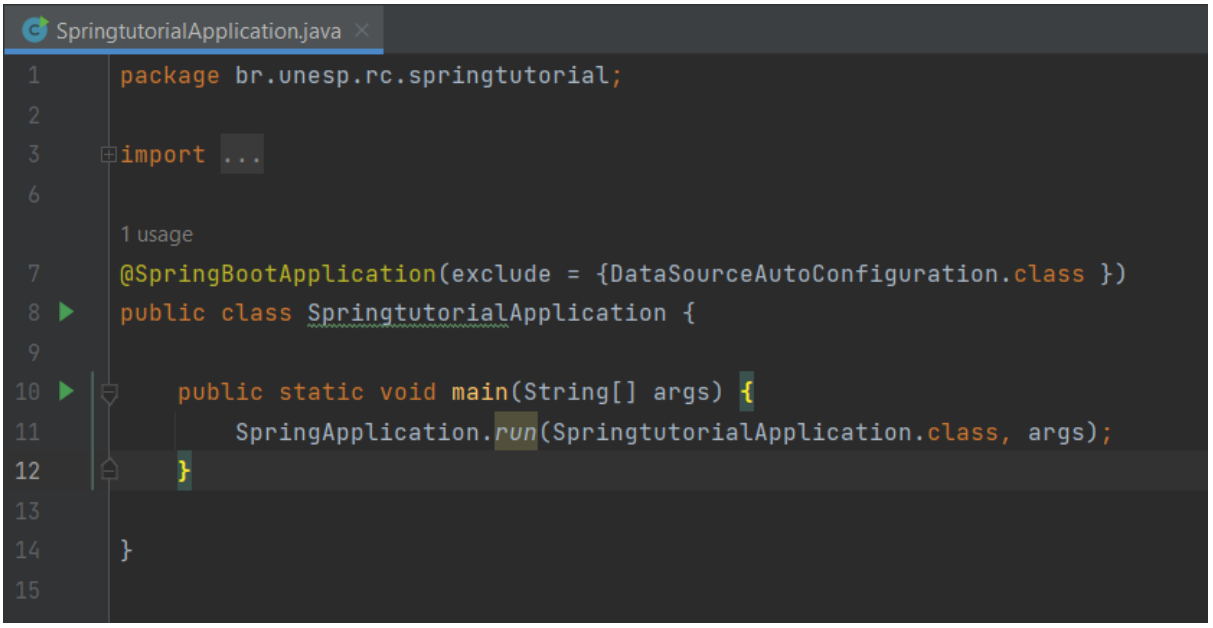
Dentro do pacote br.unesp.rc.springtutorial é onde serão desenvolvidas as camadas do projeto.

Nele, crie os pacotes referentes à cada camada:



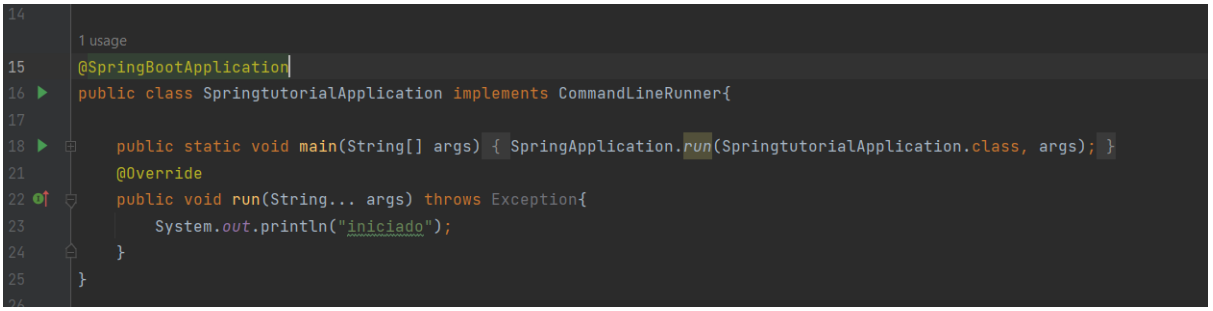
5.4. Application

Dentro do pacote, existe a classe Application, essa classe serve para rodar a aplicação Spring dentro da IDE.



Antes de rodar esta classe (apertando o botão de play verde visto na imagem), configure o projeto, apertando o comando ctrl + alt + shift

E escolha a versão do SDK (Software Development Kit) que será utilizada. Após isso, algumas alterações devem ser feitas nessa classe:

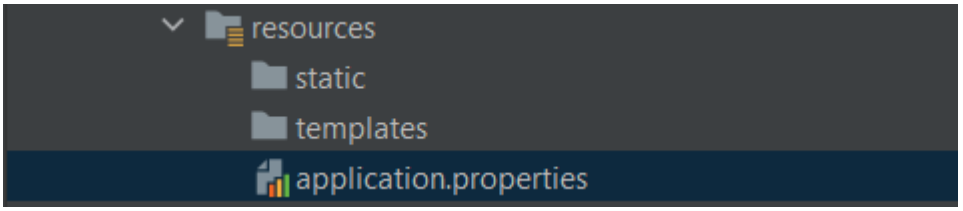


- A exceção exclude DataSourceAutoConfiguration presente na anotação @SpringBootApplicaton foi removida, para permitir que o projeto configure os beans automaticamente de acordo com o contexto de onde eles foram criados.
- O método CommandLineRunner foi implementado, e run foi redefinido. Dentro da função run, é possível fazer operações desejadas para o teste do sistema, neste exemplo, a palavra “iniciado” é escrita no terminal quando o projeto é executado.

```
17:12:56.295 [Thread-0] DEBUG org.springframework.boot.devtools.restart.classloader.RestartClassLoader - Created RestartClassLoader org.springframework.boot.devtools.restart.clas

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.8)

2022-06-21 17:12:56.779  INFO 3700 --- [  restartedMain] b.u.r.s.SpringtutorialApplication        : Starting SpringtutorialApplication using Java 16.0.1 on LAPTOP-VJTKBMUA with P
2022-06-21 17:12:56.780  INFO 3700 --- [  restartedMain] b.u.r.s.SpringtutorialApplication        : No active profile set, falling back to 1 default profile: "default"
2022-06-21 17:12:56.872  INFO 3700 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties' to 'fa
2022-06-21 17:12:56.872  INFO 3700 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' pr
2022-06-21 17:12:58.335  INFO 3700 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2022-06-21 17:12:58.353  INFO 3700 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-06-21 17:12:58.353  INFO 3700 --- [  restartedMain] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.63]
2022-06-21 17:12:58.501  INFO 3700 --- [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2022-06-21 17:12:58.501  INFO 3700 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1628 ms
2022-06-21 17:12:58.892  INFO 3700 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2022-06-21 17:12:58.925  INFO 3700 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-06-21 17:12:58.935  INFO 3700 --- [  restartedMain] b.u.r.s.SpringtutorialApplication        : Started SpringtutorialApplication in 2.622 seconds (JVM running for 3.219)
```

7. Conectando com o banco de dados

Antes de começar a modelagem das classes, é preciso conectar o projeto com o banco de dados configurado anteriormente. Para isso, utilize o arquivo application.properties, localizado no diretório resources:



Escreva os seguinte comandos neste arquivo:

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/springtutorial
spring.datasource.username=root
spring.datasource.password=1245
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
```

Mudando os campos username e password de acordo com o seu banco. Após isso, clique em Build Project e o banco estará conectado ao seu projeto.

8. Criando a camada model

Neste tutorial, serão utilizadas três classes: Pessoa, Física e Acesso. Elas devem ser criadas inicialmente no pacote model, que contém as lógicas das classes do sistema.

8.1. Pessoa

Crie uma nova classe clicando no pacote model e selecionando new->Java class. Depois disso, escreva os atributos dessa classe. A dependência Lombok será utilizada nesse tutorial com o intuito de simplificar a codificação das classes.
O código dessa classe inicialmente será da seguinte forma:

```
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@EqualsAndHashCode
@ToString
@Entity(name = "Pessoa")
@Inheritance(strategy = InheritanceType.JOINED)
public class Pessoa implements Serializable {

    private static final long serialVersionUID = 1L;

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


    @Column(name = "nome", nullable = false)
    private String nome;

    @Column(name = "email",nullable = false)
    private String email;

    @Embedded
    private Acesso acesso;


}
```

- Os comandos Lombok @Getter,@Setter,@EqualsAndHashCode e @ToString criam os métodos getter, setter, equals, hashcode e tostring, respectivamente.

- As anotações @AllArgsConstructor e @NoArgsConstructor criam métodos de construção com e sem argumentos, respectivamente.
- A relação de herança (@Inheritance) será feita com a estratégia JOINED, indicando que existirá uma tabela separada para o filho de Pessoa (classe Física), que possuirá uma coluna contendo os ids da classe pai.
- O atributo id é a chave primária da classe, como definido pela anotação @Id, e @GeneratedValue colocado em AUTO, para indicar que os ids serão incrementados.
- @Column foi colocado como nullable = false nos atributos para indicar que eles são obrigatórios.
- Em e-mail, ele foi definido como obrigatório e único, ou seja, não podem ter dois emails iguais na tabela.
- Pessoa possui um atributo que pertence à classe embeddable Acesso.
- A classe implementa Serializable para permitir que os dados sejam transmitidos para o banco de dados na forma de uma sequência de bytes.

8.2. Acesso

O código da classe acesso está a seguir:

```
6 usages
@AllArgsConstructor
@NoArgsConstructor
@Embeddable
@Getter
@Setter
@EqualsAndHashCode
@ToString
public class Acesso implements Serializable {

    private static final long serialVersionUID = 1L;

    @Column(nullable = false)
    private String usuario;

    @Column(nullable = false)
    private String senha;

}
```

- Esta classe é Embeddable, portanto ao invés de possuir tabela própria, seus atributos serão colocados na tabela de Pessoa.

- O resto das anotações são as mesmas já utilizadas na classe anterior.

8.3. Física

Já no caso de Física, ela possui relação de herança com Pessoa, como dito na seção acima, portanto seu código será da seguinte forma:

```
19 usages
@Entity(name = "PessoaFisica")
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@EqualsAndHashCode(callSuper = false,exclude = {"dataNascimento"})
@PrimaryKeyJoinColumn(name="id")
@ToString(callSuper = true)
public class Fisica extends Pessoa implements Serializable {

    private static final long serialVersionUID = 1L;

    @Column(unique = true, nullable = false)
    private String cpf;

    @Temporal(TemporalType.DATE)
    @Column(name = "data_nascimento", nullable = false)
    private Date dataNascimento;


}
```

- Neste caso, não é necessário implementar Serializable, pois isso já foi feito na classe pai.
- @PrimaryKeyJoinColumn(name=”id”) coloca a chave primária de Pessoa na tabela de Física.
- dataNascimento foi definida com a anotação Temporal(TemporalType.DATE) para indicar que a data será exibida sem horas, minutos e segundos.

9. Repository

Com a camada model pronta, o próximo passo é codificar a camada repository, que possui as classes de persistência.

Somente uma interface será criada dentro desta camada: Física. A classe acesso não é uma entidade, portanto não será incluída; a classe Física já possui todos os atributos de Pessoa, então não é necessário criar uma interface exclusivamente para Pessoa.

O código dessa camada ficará dessa forma:

```
9
                5 usages
10              @Repository
11              public interface FisicaRepository extends JpaRepository<Fisica, Long> {
12
                    1 usage
13                  Fisica findByCpf(String cpf);
14
                    1 usage
15                  Fisica findByEmail(String email);
16
                    2 usages
17                  Fisica findByAcesso_Usuario(String usuario);
18
                    1 usage
19                  boolean existsByCpf(String cpf);
20
21              }
```

- Foram colocadas essas quatro assinaturas para este tutorial, considerando que outras já estão implementadas pelo JpaRepository.
- A anotação @Repository indica que essa classe é um repositório.

10. Service

O próximo passo é codificar a camada Service, que abstrai os métodos implementados no Repository:

```java
@Service
public class FisicaService {

    11 usages
    @Autowired
    private FisicaRepository repository;

    public FisicaService(){

    }

    2 usages
    public Fisica findByCpf(String cpf){
        Fisica pessoaEncontrada = null;

        if (repository!=null){
            pessoaEncontrada = repository.findByCpf(cpf);
        }
        return pessoaEncontrada;
    }

    1 usage
    public Fisica findByEmail(String email){
        Fisica pessoaEncontrada = null;

        if (repository != null){
            pessoaEncontrada = repository.findByEmail(email);
        }

        return pessoaEncontrada;
    }
```

```java
    2 usages
    public Fisica findByUsuario(String usuario){
        Fisica pessoaEncontrada = null;

        if (repository!=null){
            pessoaEncontrada = repository.findByUsuario(usuario);
        }
        return pessoaEncontrada;
    }

    1 usage
    @Transactional
    public Fisica save(Fisica pessoa){
        Fisica pessoaNova = null;

        if (repository != null){
            pessoaNova = repository.save(pessoa);
        }

        return pessoaNova;
    }

    1 usage
    @Transactional
    public void delete(Fisica pessoa){

        if (repository != null){
            repository.delete(pessoa);
        }
    }
```

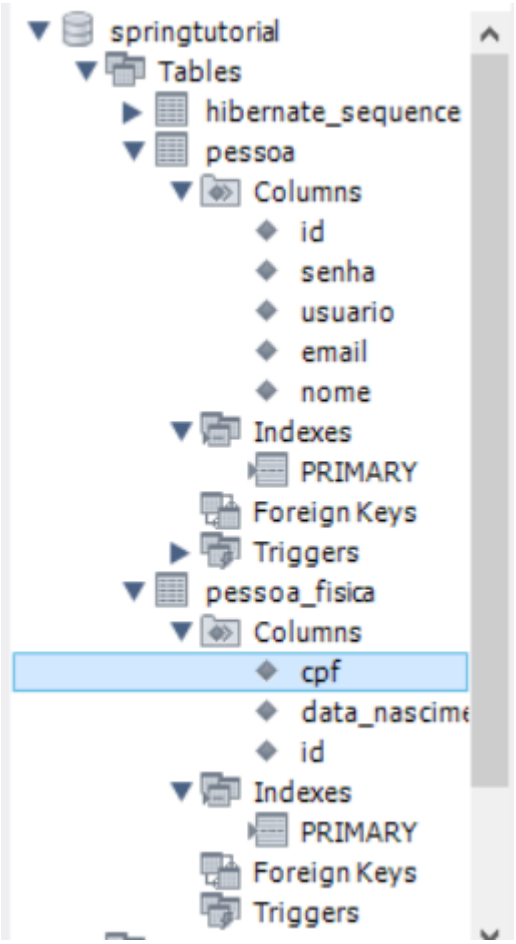
```java
    1 usage
    public List<Fisica> findAll(){
        List<Fisica> list = null;

        if (repository != null){
            list = new ArrayList<>();
            list = repository.findAll();
        }

        return list;
    }

    1 usage
    public boolean existsByCpf(String cpf) { return repository.existsByCpf(cpf); }
}
```

- Esta classe é anotada com @Service, para indicar que ela é um bean encarregado por manter as lógicas de serviço.
- As funções incluídas para a aplicação foram: salvar uma nova pessoa, deletar uma pessoa existente, retornar uma lista com todas as pessoas, encontrar uma pessoa pelo cpf, encontrar uma pessoa pelo e-mail, encontrar uma pessoa pelo usuário e checar se o cpf inserido existe no banco de dados existe.

- A anotação @Transactional foi utilizada nos métodos save e delete, pois se uma dessas operações der errado, será possível dar um rollback sem causar problemas no banco de dados. Como save e delete são operações que interferem diretamente nos dados existentes, é boa prática usar essa anotação.

11. Executando pela primeira vez

Com as três principais camadas do projeto prontas, podemos executá-lo. Para isso basta clicar na opção run na classe SpringTutorialApplication. Para este teste, nenhum dado será inserido no banco de dados, iremos fazer apenas a criação das tabelas dentro dele.



Se tudo for feito de forma correta, este será o resultado no banco de dados.

12. API

Para criar a API, foi criada uma classe chamada FisicaController dentro do pacote de api criado no início do tutorial. O código desta classe é o seguinte:

```java
@RestController
@RequestMapping("/tutorial/api")
public class FisicaController {

    9 usages
    @Autowired
    private FisicaService fisicaService;

    1 usage
    private final PasswordEncoder encoder;

    public FisicaController(PasswordEncoder encoder) { this.encoder = encoder; }


    @PostMapping("/save")
    public ResponseEntity<Object> savePessoa(@RequestBody @Validated FisicaDto fisicaDto){
        if(fisicaService.existsByCpf(fisicaDto.getCpf())){
            return ResponseEntity.status(HttpStatus.CONFLICT).body("Cpf já está em uso.");
        }
        if(fisicaService.findByAcesso_Usuario(fisicaDto.getAcesso().getUsuario()).isPresent()){
            return ResponseEntity.status(HttpStatus.CONFLICT).body("Usuário já está em uso.");
        }
        Fisica fisicaModel = new Fisica();
        BeanUtils.copyProperties(fisicaDto,fisicaModel);
        return ResponseEntity.status(HttpStatus.CREATED).body(fisicaService.save(fisicaModel));
    }
```

```java
    @GetMapping("/todos")
    public ResponseEntity<List<Fisica>> getAllFisica() {
        return ResponseEntity.status(HttpStatus.OK).body(fisicaService.findAll());
    }

    @GetMapping("/cpf/{cpf}")
    public ResponseEntity<Object> getPessoaPorCpf(@PathVariable(value= "cpf")String cpf){
        Optional<Fisica> fisicaModelOptional = Optional.ofNullable(fisicaService.findByCpf(cpf));
        if(!fisicaModelOptional.isPresent()){
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Pessoa não encontrada.");
        }
        return ResponseEntity.status(HttpStatus.OK).body(fisicaModelOptional.get());
    }

    @GetMapping("/email/{email}")
    public ResponseEntity<Object> getPessoaPorEmail(@PathVariable(value= "email")String email){
        Optional<Fisica> fisicaModelOptional = Optional.ofNullable(fisicaService.findByEmail(email));
        if(!fisicaModelOptional.isPresent()){
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Pessoa não encontrada.");
        }
        return ResponseEntity.status(HttpStatus.OK).body(fisicaModelOptional.get());
    }

    @GetMapping("/usuario/{usuario}")
    public ResponseEntity<Object> getPessoaPorUsuario(@PathVariable(value= "usuario")String usuario){
        Optional<Fisica> fisicaModelOptional = fisicaService.findByAcesso_Usuario(usuario);
        if(!fisicaModelOptional.isPresent()){
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Pessoa não encontrada.");
```

```
81
82      @DeleteMapping("/delete/{cpf}")
83      public ResponseEntity<Object> deletePessoaPorCpf(@PathVariable(value= "cpf")String cpf){
84          Optional<Fisica> fisicaModelOptional = Optional.ofNullable(fisicaService.findByCpf(cpf));
85          if(!fisicaModelOptional.isPresent()){
86              return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Pessoa não encontrada.");
87          }
88          fisicaService.delete(fisicaModelOptional.get());
89          return ResponseEntity.status(HttpStatus.OK).body("Pessoa deletada com sucesso.");
90      }
91  }
92
```

- A anotação RestController indica que a classe é a api.
- @RequestMapping(“tutorial/api”) determina a url onde a api será acessada
- Os métodos criados foram: POST, GET ALL, GET by cpf, GET by email e DELETE by cpf.
- Dentro de cada método, são feitos testes para o caso da pessoa selecionada não existir no banco, ou no caso do método POST, caso o CPF selecionado já exista. Esses testes são importantes para evitar problemas nos dados guardados.

13.1. FisicaDto

Para a implementação da API, uma classe chamada FisicaDto foi criada. Esta classe é chamada de Data Transfer Object, responsável pela transferência dos dados inseridos pelo usuário para o sistema. Ou seja, os dados recebidos pelo controller serão inicialmente armazenados no objeto FisicaDto, e posteriormente serão enviados para a camada Model, para depois serem armazenados no banco de dados.

O código dessa classe está a seguir:

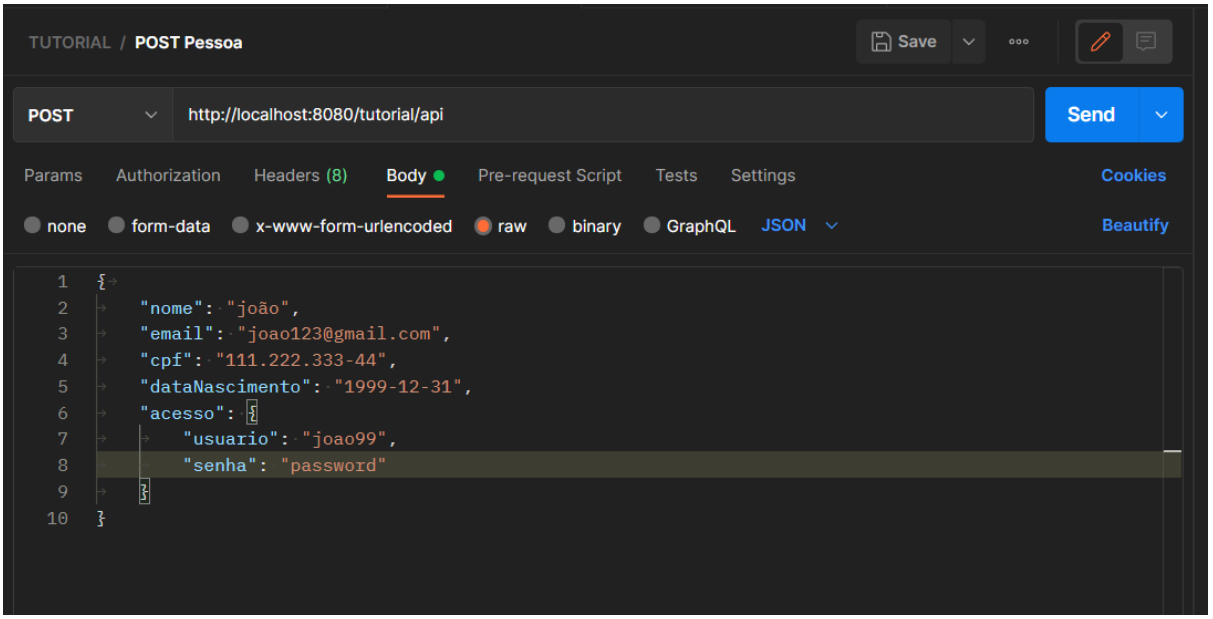
```java
2 usages
@Getter
@Setter
@ToString
@EqualsAndHashCode
@NoArgsConstructor
@AllArgsConstructor
public class FisicaDto {

    @NotBlank
    private String nome;

    @NotBlank
    @Email
    private String email;

    @NotBlank
    @CPF
    private String cpf;

    @NotBlank
    @Past
    @JsonFormat(pattern = "yyyy-MM-dd")
    private Date dataNascimento;

    @NotBlank
    private Acesso acesso;
}
```

13. Testando a API

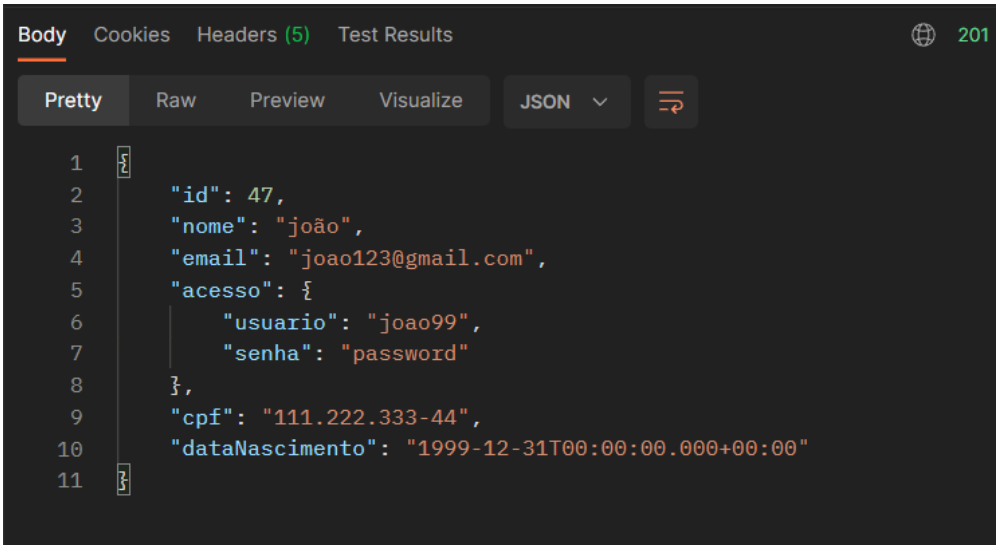
Para testar a API criada, a ferramenta postman foi usada. Os testes feitos foram os seguintes:

13.1. POST

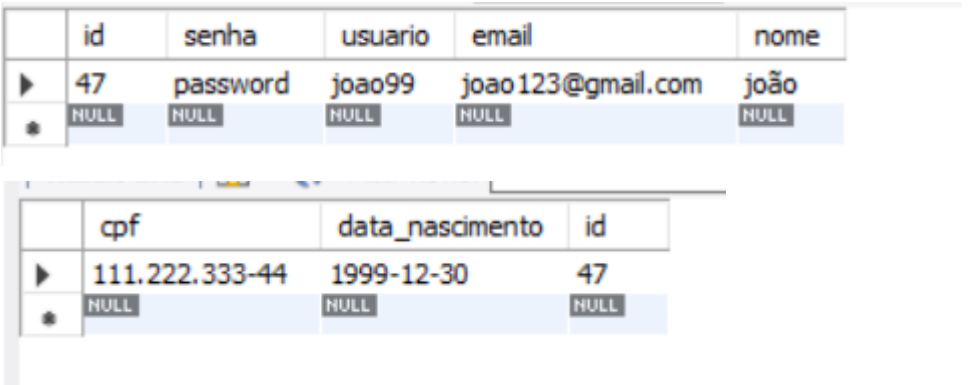
Entrada:



Resultado:



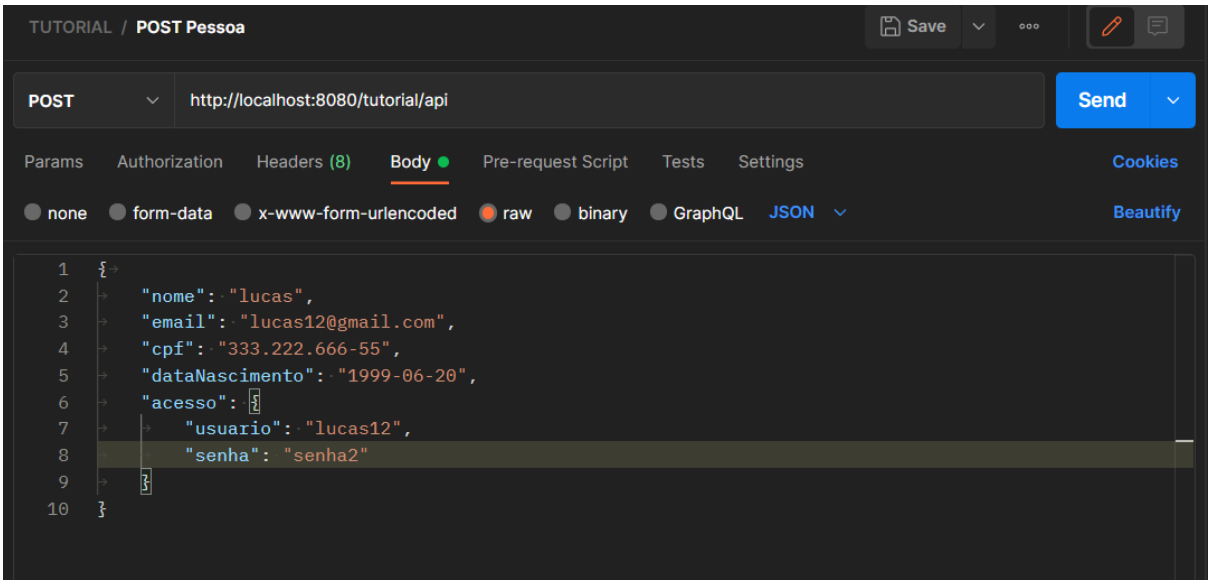
MySQL:



	id	senha	usuario	email	nome
▶	47	password	joao99	joao123@gmail.com	joão
*	NULL	NULL	NULL	NULL	NULL

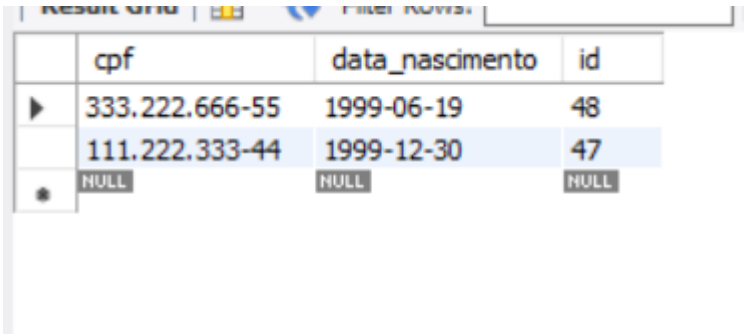
	cpf	data_nascimento	id
▶	111.222.333-44	1999-12-30	47
*	NULL	NULL	NULL

Mais uma entrada:



Resultado:

MySQL:



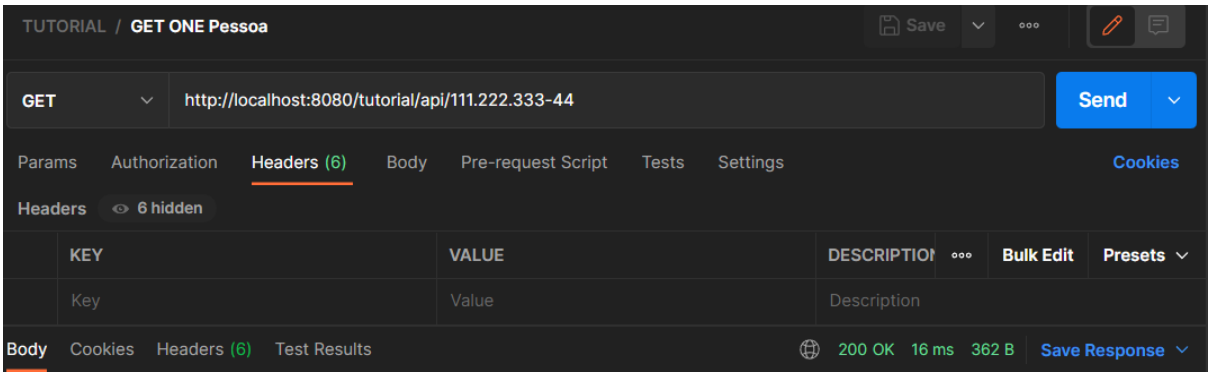
	cpf	data_nascimento	id
▶	333.222.666-55	1999-06-19	48
	111.222.333-44	1999-12-30	47
*	NULL	NULL	NULL

	id	senha	usuario	email	nome
▶	47	password	joao99	joao123@gmail.com	joão
	48	senha2	lucas12	lucas12@gmail.com	lucas
*	NULL	NULL	NULL	NULL	NULL

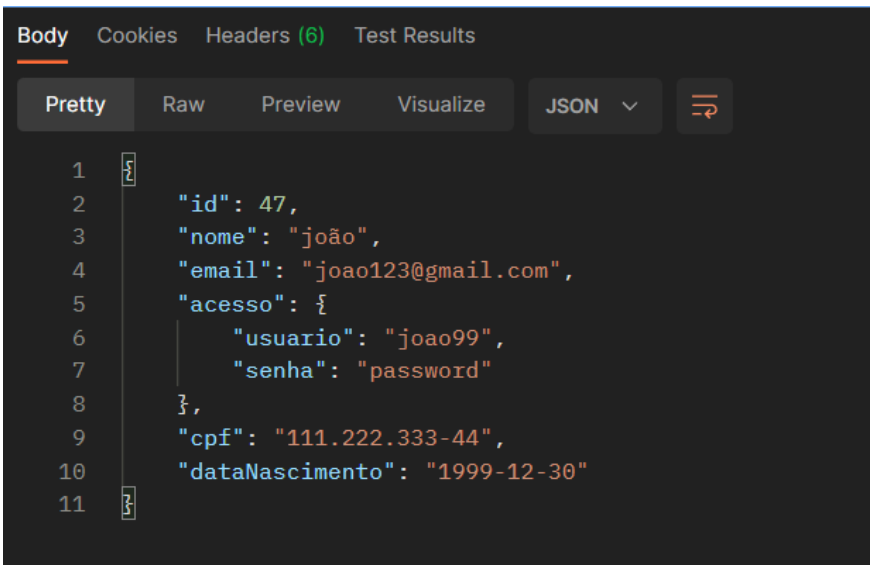
13.2. GET by cpf

Nessa operação, um cpf é inserido na url, e o retorno será os dados da pessoa com o cpf escolhido, caso o cpf não exista no banco de dados, uma mensagem de erro é mostrada:

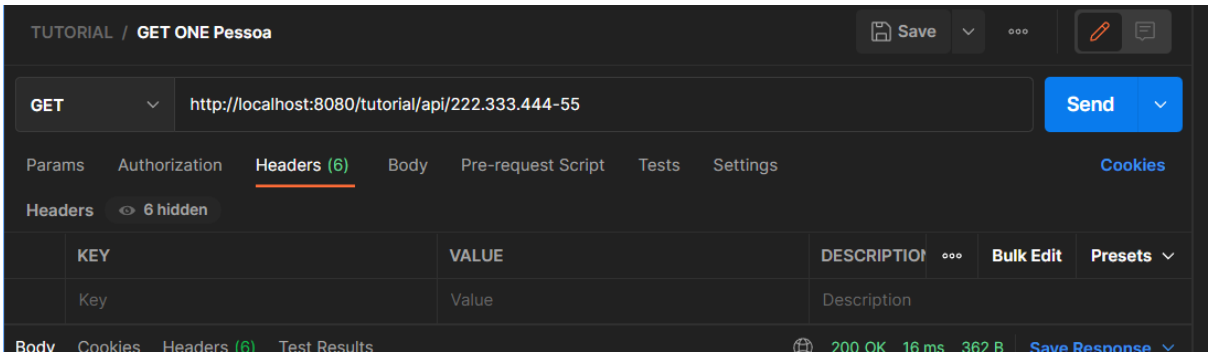
Entrada:



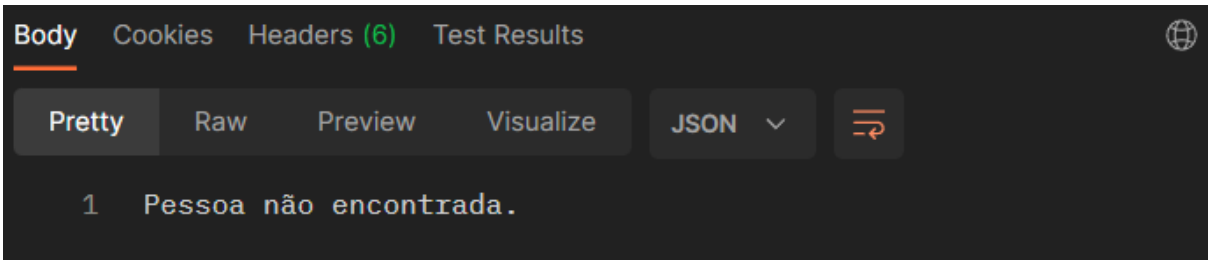
Saída:



Entrada com cpf inválido:



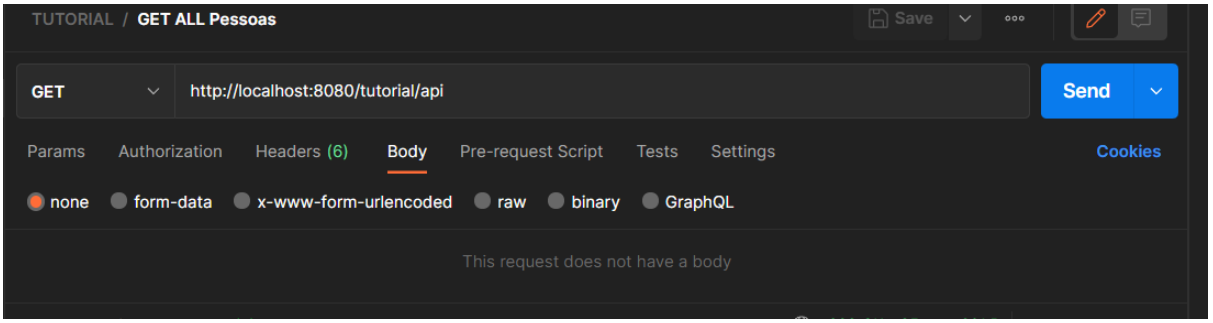
Saída:



13.3. GET ALL

Esse método retorna todos os dados salvos no banco:

Entrada:

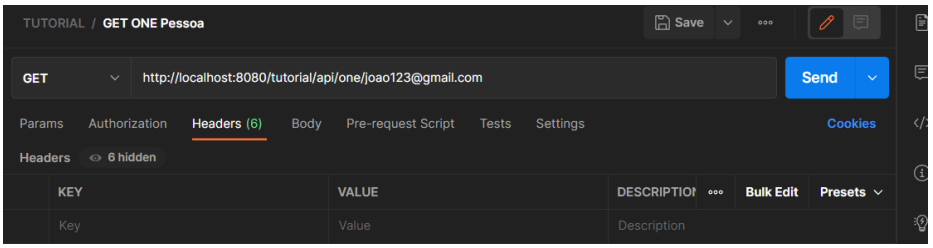


Retorno:

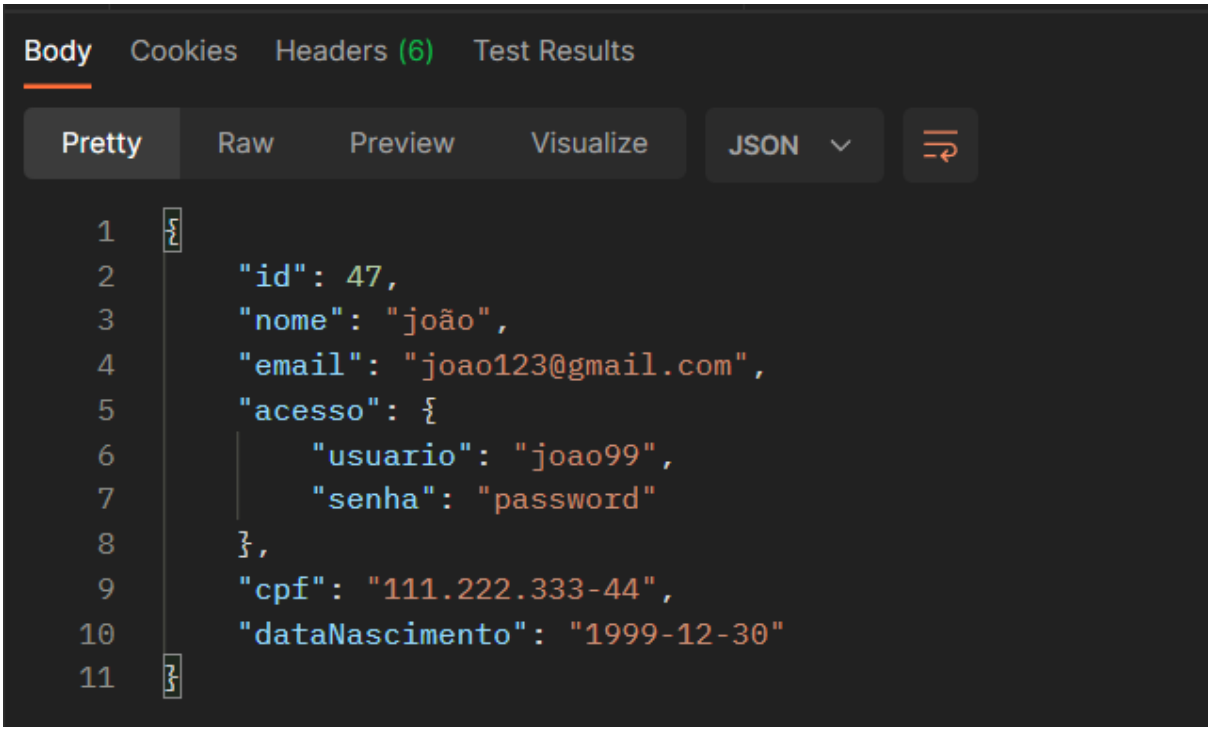
```
    {
        "id": 48,
        "nome": "lucas",
        "email": "lucas12@gmail.com",
        "acesso": {
            "usuario": "lucas12",
            "senha": "senha2"
        },
        "cpf": "333.222.666-55",
        "dataNascimento": "1999-06-19"
    },
    {
        "id": 47,
        "nome": "joão",
        "email": "joao123@gmail.com",
        "acesso": {
            "usuario": "joao99",
            "senha": "password"
        },
        "cpf": "111.222.333-44",
        "dataNascimento": "1999-12-30"
    }
```


13.4. GET by email

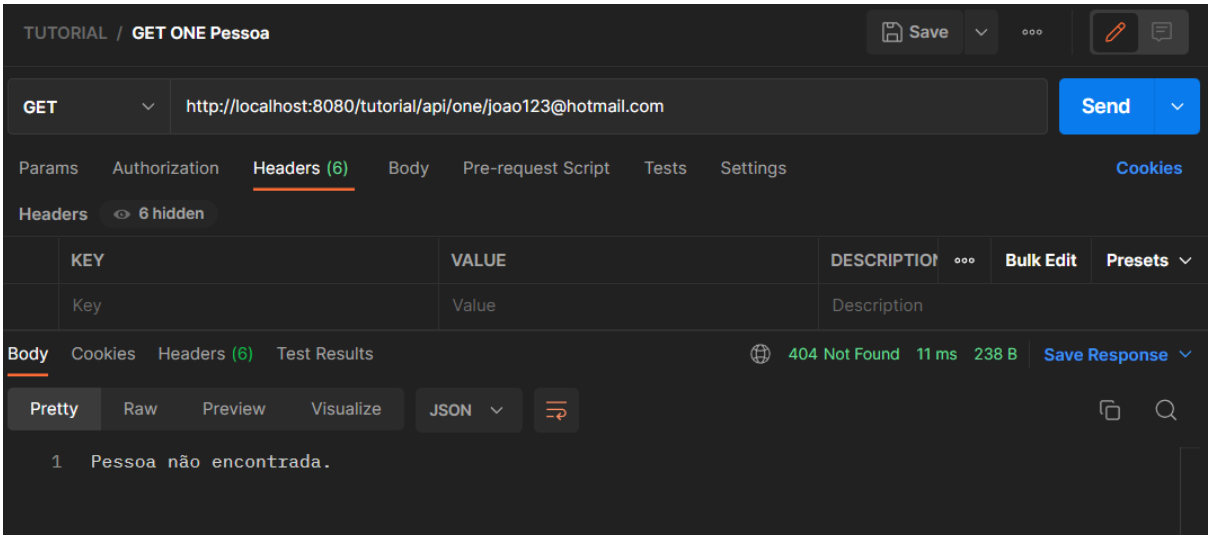
Entrada:



Saída:



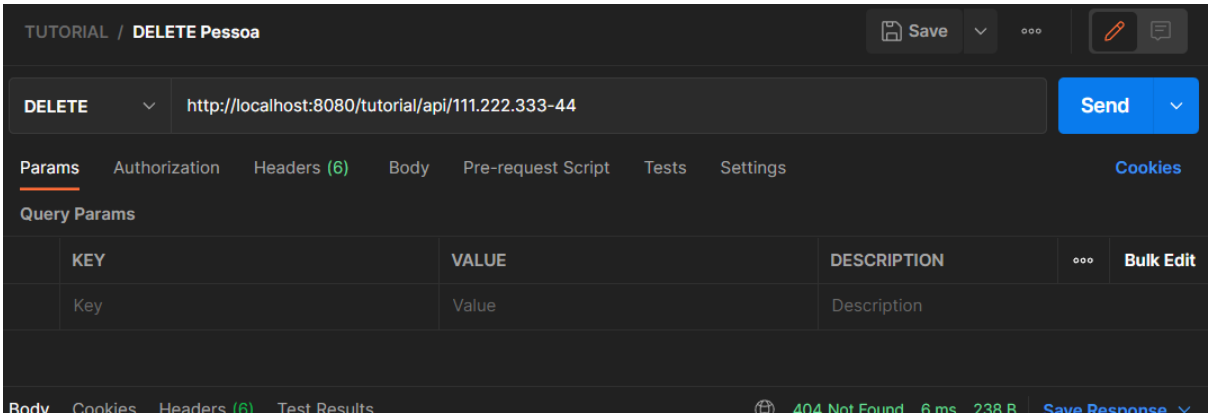
Caso inválido:



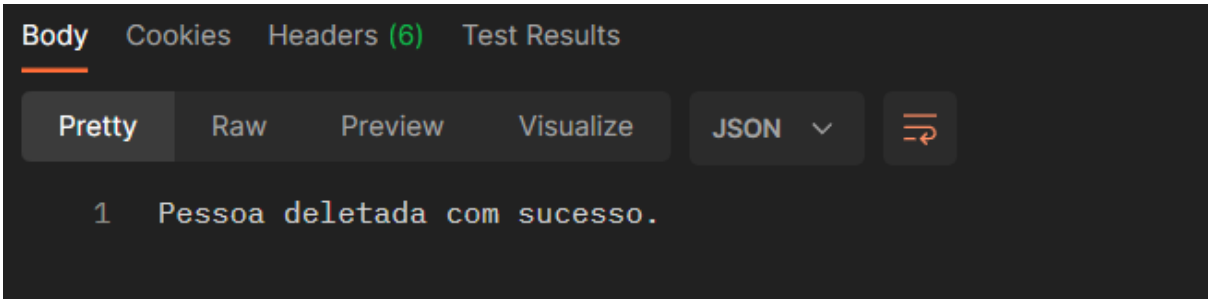
13.5. DELETE by CPF

Esse método deleta os dados de uma pessoa que possui o cpf inserido.

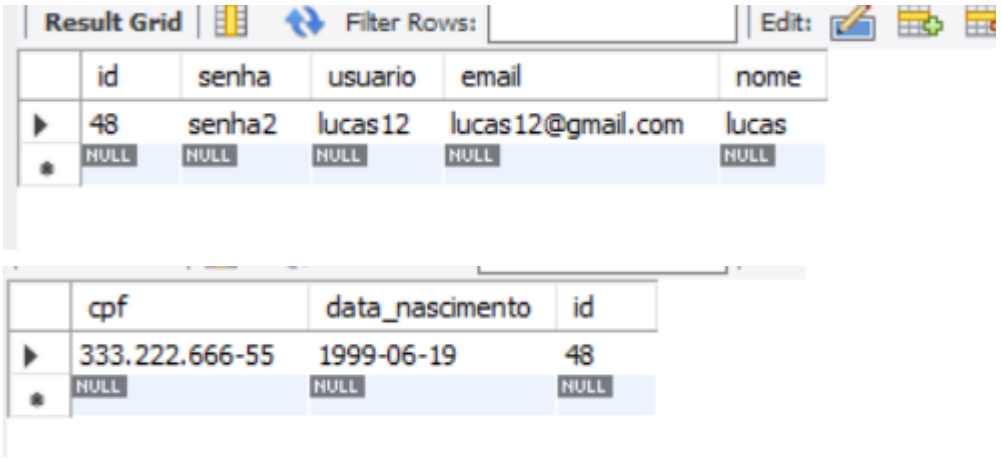
Entrada:



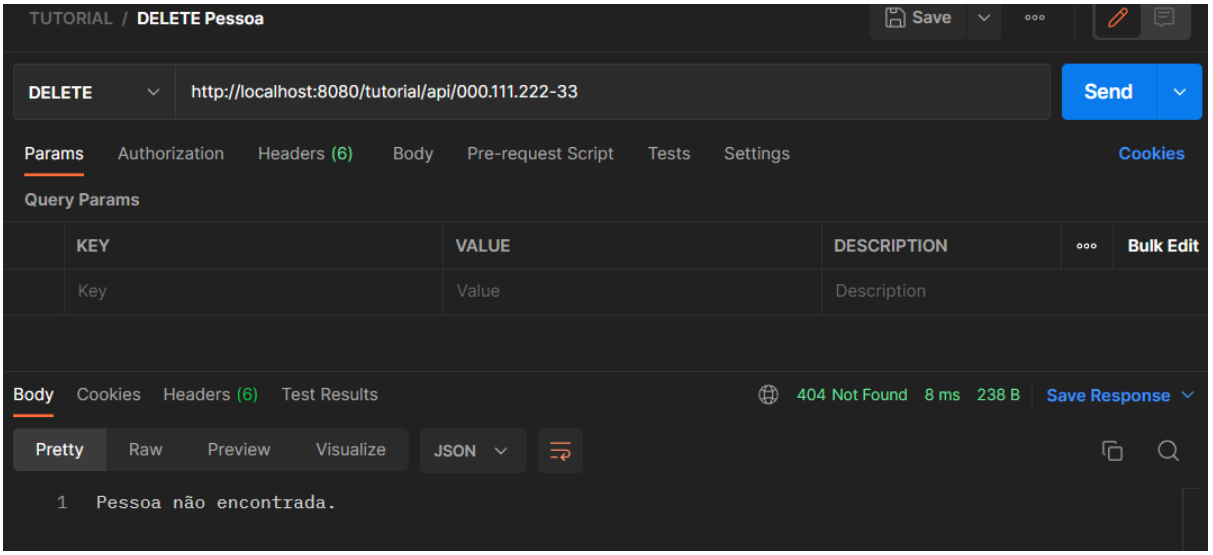
Saída:



MySQL:



Caso inválido:



14. Criptografando a senha

Antes de implementar a autenticação com JWT, iremos criar um método que criptografa a senha do usuário antes de armazená-la no banco, pois não é seguro guardar uma senha no banco de dados de forma desprotegida.

Para isso, vamos inserir o seguinte método na classe SpringTutorialApplication:

```
@Bean
public PasswordEncoder getPasswordEncoder(){
    BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
    return encoder;
}
```

- Esse método é um Bean do sistema.
- Ele chama o algoritmo de criptografia BCrypt para a senha.

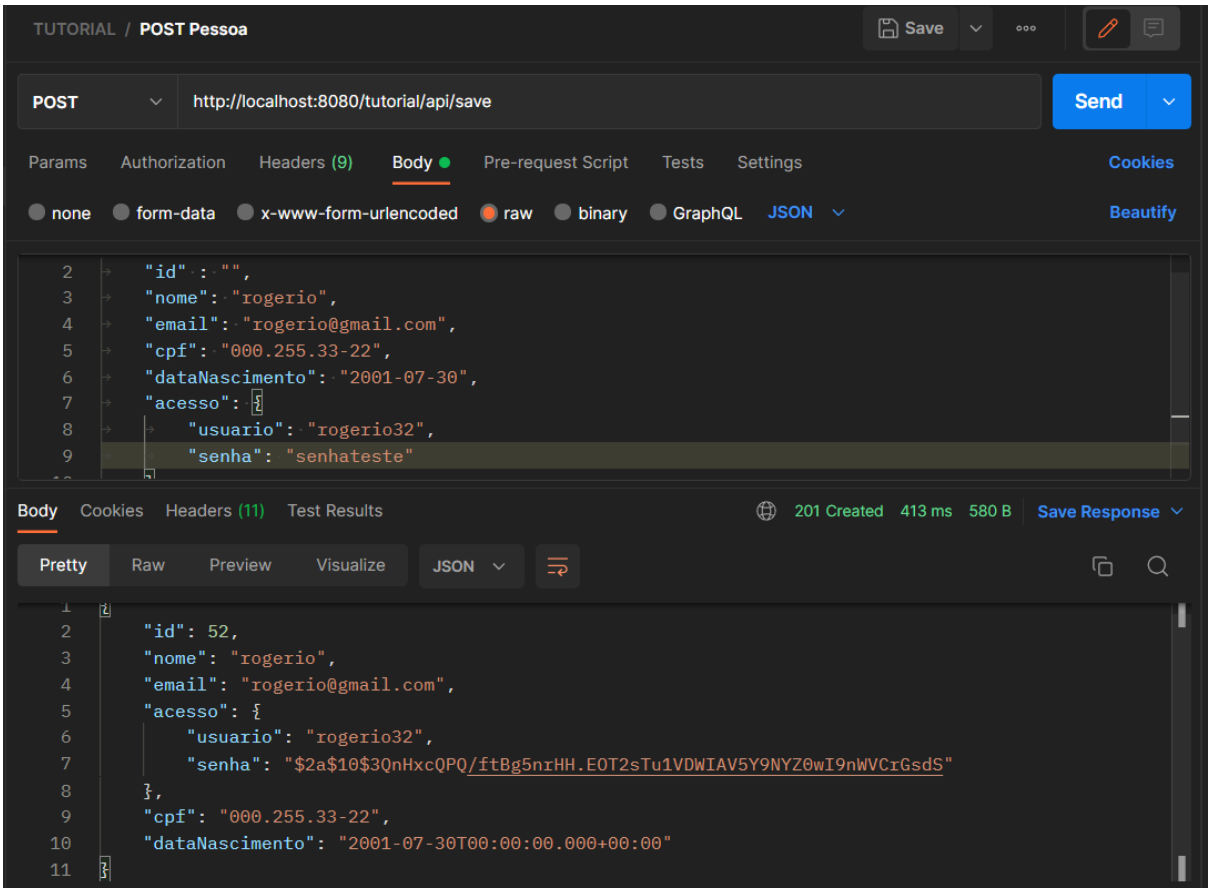
Depois disso, coloque a seguinte linha de comando no método Post de salvar uma nova pessoa na API:

```
41        Fisica fisicaModel = new Fisica();
42        BeanUtils.copyProperties(fisicaDto,fisicaModel);
43        fisicaModel.getAcesso().setSenha(encoder.encode(fisicaModel.getAcesso().getSenha()));
44        return ResponseEntity.status(HttpStatus.CREATED).body(fisicaService.save(fisicaModel));
45    }
```

(linha 43)

- Com isso, a senha inserida pelo usuário será criptografada antes de ser enviada para o banco.

Exemplo:



- Veja a senha inserida, e a senha retornada após o método Post.

15. Implementação da autenticação com JWT

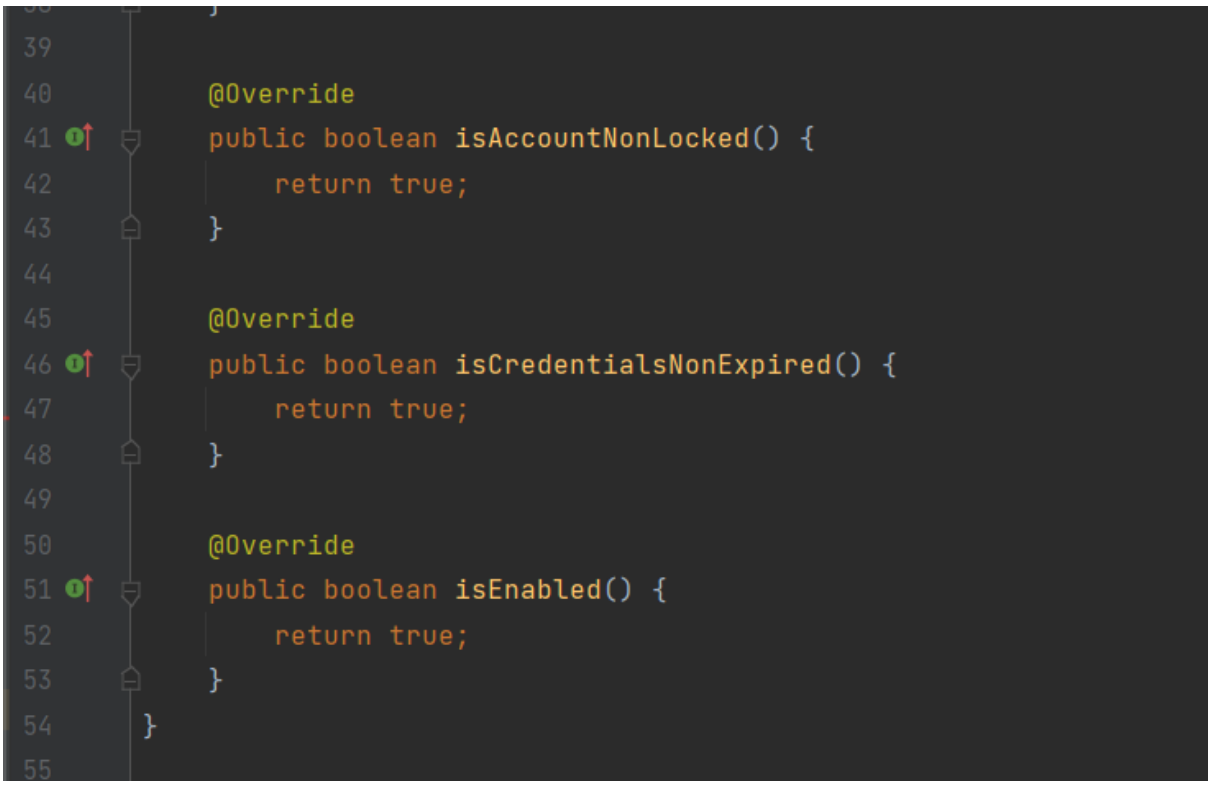
É de extrema importância incluir uma forma de autenticação dentro da API criada, para garantir a segurança da aplicação.

O primeiro passo para a implementação do JWT é incluir a dependência auth0 no arquivo pom.xml, ela pode ser encontrada no link Maven Repository: com.auth0 » java-jwt (mvnrepository.com), e a dependência do spring security no link Maven Repository: org.springframework.security » spring-security-web » 5.7.1 (mvnrepository.com).

15.1. Detalhes do usuário

Agora, crie uma classe num pacote novo Data, que irá conter os detalhes do usuário, e outros métodos que serão utilizados para verificação:

```java
public class DetalheUsuarioData implements UserDetails {

    3 usages
    private final Optional<Fisica> entity;

    1 usage
    public DetalheUsuarioData(Optional<Fisica> entity) {
        this.entity = entity;
    }


    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return new ArrayList<>();
    }

    @Override
    public String getPassword() {
        return entity.orElse(new Fisica()).getAcesso().getSenha();
    }

    @Override
    public String getUsername() {
        return entity.orElse(new Fisica()).getAcesso().getUsuario();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

```



```
39
40          @Override
41          public boolean isAccountNonLocked() {
42              return true;
43          }
44
45          @Override
46          public boolean isCredentialsNonExpired() {
47              return true;
48          }
49
50          @Override
51          public boolean isEnabled() {
52              return true;
53          }
54      }
55
```

- Veja que a classe implementa a interface UserDetails, do Spring Security.
- A classe possui métodos para verificação dos dados inseridos, assim como métodos Getters para as informações de login.

15.2. DetalhesUsuarioService

Depois disso, crie uma nova classe da camada service, com o nome DetalhesUsuarioServiceImpl, responsável pela abstração e implementação dos métodos que têm relação com a classe anterior:

```java
import java.util.Optional;

@Component
public class DetalheUsuarioServiceImpl implements UserDetailsService {

    2 usages
    private final FisicaRepository repository;

    public DetalheUsuarioServiceImpl(FisicaRepository repository) {
        this.repository = repository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Optional<Fisica> usuario = Optional.ofNullable(repository.findByUsuario(username));
        if(usuario.isEmpty()){
            throw new UsernameNotFoundException("Usuário [" + usuario + "] não encontrado.");
        }

        return new DetalheUsuarioData(usuario);
    }
}
```

- Observe que este método é um bean, por isso foi anotado com @Component.
- Essa classe implementa o método loadUserByUsername, que checa se o usuário inserido existe na base de dados, se sim, retorna o usuário.

15.3. Security

Agora, um novo pacote será criado com o nome Security, onde as classes responsáveis por configurar, autenticar e validar os dados de login serão feitas.

15.3.1. JwtAutenticarFilter

Responsável por fazer a autenticação do login:

```java
public class JwtAutenticarFilter extends UsernamePasswordAuthenticationFilter {

    1 usage
    public static final int  TOKEN_EXPIRACAO = 600_000;
    2 usages
    public static final String TOKEN_SENHA = "4d56b40e-a059-4189-a575-01297350d821"; //Esta senha é colocada no código
                                                                                     //fonte apenas para o
                                                                                     //desenvolvimento do tutorial
    2 usages
    private final AuthenticationManager authenticationManager;

    1 usage
    public JwtAutenticarFilter(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request,
                                                HttpServletResponse response) throws AuthenticationException {

        try {
            Fisica usuario = new ObjectMapper().readValue(request.getInputStream(),Fisica.class);

            return authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(
                    usuario.getAcesso().getUsuario(),
                    usuario.getAcesso().getSenha(),
                    new ArrayList<>()
            ));
        } catch (IOException e) {
            throw new RuntimeException("Falha ao autenticar usuário", e);
        }
    }
```

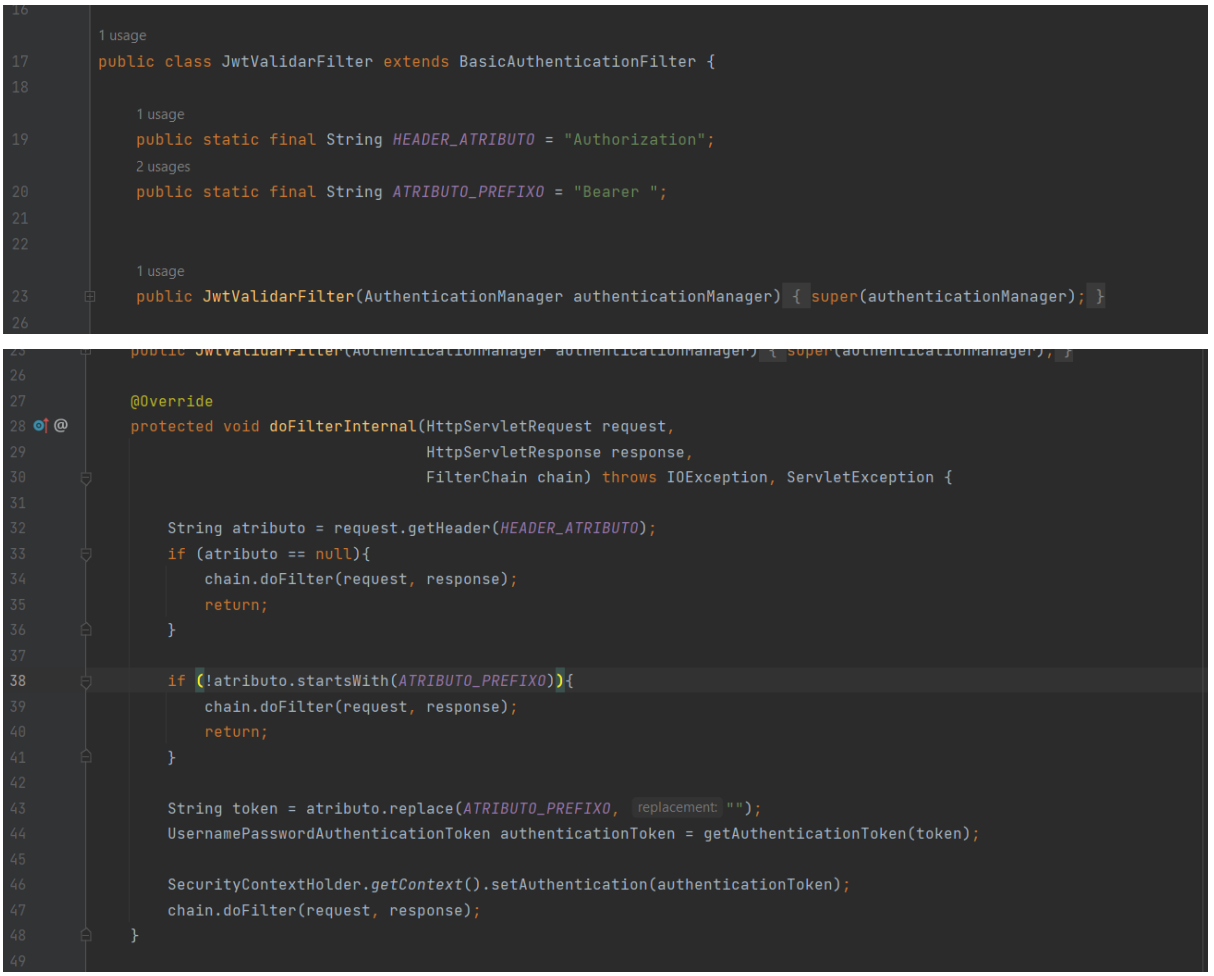
```java
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest request,
                                            HttpServletResponse response,
                                            FilterChain chain,
                                            Authentication authResult) throws IOException, ServletException {

        DetalheUsuarioData usuarioData = (DetalheUsuarioData)authResult.getPrincipal();
        String token = JWT.create()
                .withSubject(usuarioData.getUsername())
                .withExpiresAt(new Date(System.currentTimeMillis() + TOKEN_EXPIRACAO))
                .sign(Algorithm.HMAC512(TOKEN_SENHA));

        response.getWriter().write(token);
        response.getWriter().flush();
    }
}
```

- Essa classe possui o método que tenta autenticar o login, e caso seja bem sucedido,
- No início da classe, a constante TOKEN_EXPIRACAO é setada para 600_000, indicando que o token gerado tem a duração de 600.000 milissegundos (10 minutos), esse valor pode ser alterado conforme necessário.

- A constante TOKEN_SENHA também foi gerada, com um valor GUID gerado aleatoriamente. Esse valor pode ser gerado utilizando outros métodos mais seguros, mas isso não será abordado neste tutorial.
- Vale notar que as duas práticas citadas não devem ser reproduzidas numa aplicação real, valores como esses devem ser armazenados em um arquivo separado do código fonte.
- Quando a função successfulAuthentication é executada, ela gera um código JWT referente ao usuário que chamou o método.

15.3.2. JwtValidarFilter

Responsável por validar os dados:



```
1 usage
public class JwtValidarFilter extends BasicAuthenticationFilter {

    1 usage
    public static final String HEADER_ATRIBUTO = "Authorization";
    2 usages
    public static final String ATRIBUTO_PREFIXO = "Bearer ";


    1 usage
    public JwtValidarFilter(AuthenticationManager authenticationManager) { super(authenticationManager); }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain) throws IOException, ServletException {

        String atributo = request.getHeader(HEADER_ATRIBUTO);
        if (atributo == null){
            chain.doFilter(request, response);
            return;
        }

        if (!atributo.startsWith(ATRIBUTO_PREFIXO)){
            chain.doFilter(request, response);
            return;
        }

        String token = atributo.replace(ATRIBUTO_PREFIXO,  replacement: "");
        UsernamePasswordAuthenticationToken authenticationToken = getAuthenticationToken(token);

        SecurityContextHolder.getContext().setAuthentication(authenticationToken);
        chain.doFilter(request, response);
    }
```

```
49
                1 usage
50  @       private UsernamePasswordAuthenticationToken getAuthenticationToken(String token){
51              String usuario = JWT.require(Algorithm.HMAC512(JwtAutenticarFilter.TOKEN_SENHA)) Verification
52                      .build() JWTVerifier
53                      .verify(token) DecodedJWT
54                      .getSubject();
55
56              if(usuario == null){
57                  return null;
58              }
59
60              return new UsernamePasswordAuthenticationToken(usuario,  credentials: null, new ArrayList<>());
61          }
62      }
63
```

- Essa classe será executada depois que o token já foi criado. Ela faz o tratamento desse token, e executa a validação dele. O Spring Security é responsável pela chamada desses métodos.

15.3.3. JwtConfiguração

A última classe da camada security é a classe de configuração:

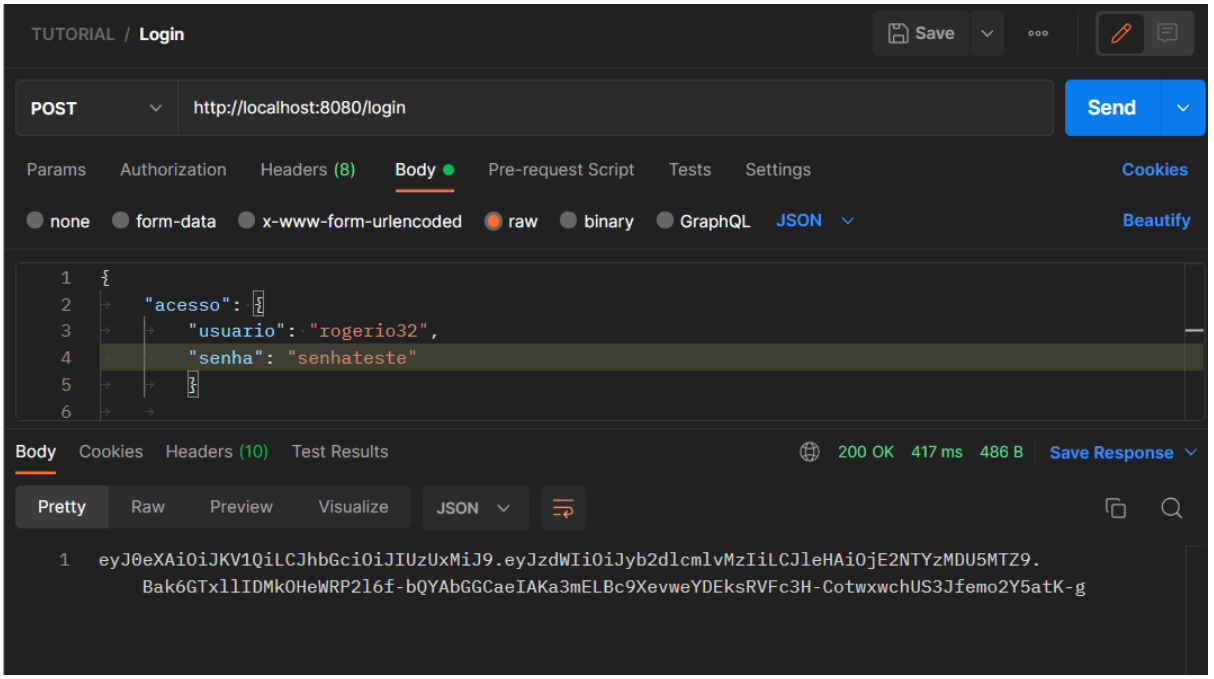
```
18  @EnableWebSecurity
19  public class JwtConfiguracao extends WebSecurityConfigurerAdapter {
20
        2 usages
21      private final DetalheUsuarioServiceImpl usuarioService;
        2 usages
22      private final PasswordEncoder passwordEncoder;
23
24      public JwtConfiguracao(DetalheUsuarioServiceImpl usuarioService, PasswordEncoder passwordEncoder) {
25          this.usuarioService = usuarioService;
26          this.passwordEncoder = passwordEncoder;
27      }
28
29      @Override
30      protected void configure(AuthenticationManagerBuilder auth) throws Exception {
31          auth.userDetailsService(usuarioService).passwordEncoder(passwordEncoder);
32      }
33
34      @Override
35      protected void configure(HttpSecurity http) throws Exception {
36          http.csrf().disable().authorizeRequests().antMatchers(HttpMethod.POST, ...antPatterns: "/login").permitAll()
37                  .anyRequest().authenticated()
38                  .and()
39                  .addFilter(new JwtAutenticarFilter(authenticationManager()))
40                  .addFilter(new JwtValidarFilter(authenticationManager()))
41                  .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
42      }
43
```

```
    @Bean
    CorsConfigurationSource corsConfigurationSource() {
        final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

        CorsConfiguration corsConfiguration = new CorsConfiguration().applyPermitDefaultValues();
        source.registerCorsConfiguration( pattern: "/**", corsConfiguration);

        return source;
    }

}
```

- A primeira função configure é responsável por pegar o nome do usuário e da senha inseridos.
- A segunda define o método POST na url /login como um método que não precisa de autorização para ser executado, e certifica-se de gerar um token (caso a autenticação seja bem sucedida) e de validar esse token.
- CorsConfigurationSource é um bean que permite que a aplicação receba requisições de domínios além da web, caso isso não seja desejado, essa configuração pode ser removida.

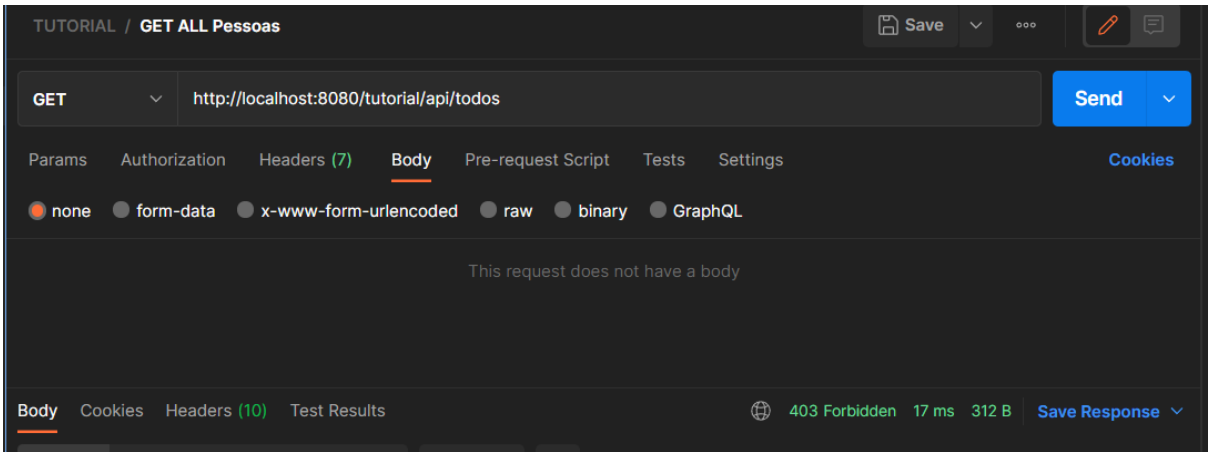
16. Testando a autenticação

Para testar essa implementação, abra o postman e faça uma requisição POST com a url http://localhost:8080/login, que foi selecionada na classe de configuração. Então, coloque no corpo desta requisição os valores de usuário de senha de uma pessoa que já está no banco de dados. Utilizaremos o usuário inserido na seção de criptografia da senha:



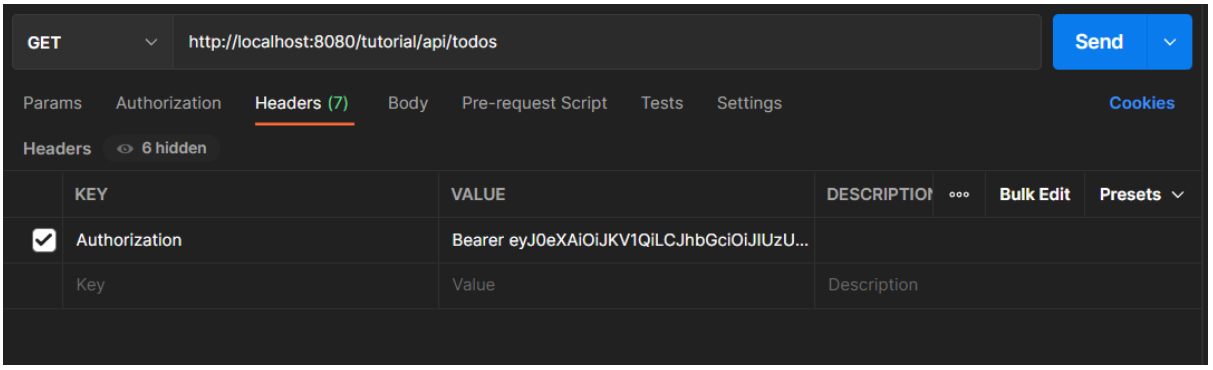
- O método retorna um token, que será utilizado para acessar as outras requisições da aplicação.

Agora, vamos primeiro executar uma requisição sem o token:



- Temos um status http 403, indicando que não temos autorização para realizar a requisição GET ALL pessoas.

Para inserir o token, vá para a seção Headers, e insira um parâmetro com o nome: Authorization e o Value: Bearer “token”:



- Esses nomes foram os definidos na classe de validação, eles podem ser mudados caso desejado.

Apertando em Send, o método GET ALL pessoas deve ser executado como esperado:

17. Conclusão

Com isso, concluímos o tutorial de como criar um projeto Spring Boot e implementar uma API RESTful com autenticação JWT. Ainda existem diversas alterações que podem ser feitas para tornar a aplicação mais eficiente e segura, além de outras funções que não foram tratadas neste tutorial. Este documento serve apenas como um guia de como começar o seu sistema de forma prática e didática.