



## 1-Configurando o Visual Studio Code

Clica no ícone de extensões  e instala as seguintes extensões:

- Extension Pack for Java
- Spring Boot Extension Pack

## 2-Criando o Projeto Spring Boot

Menu view > comand Pallet

Digite:Spring inicializr Maven Project

### Configurar o Projeto:

Project: Maven Project

Language: Java

Spring Boot: 3.3.1

Group: br.com.exemplo

Artifact: api

Packaging: Jar

Java: 17 (ou superior)

### Dependências (funções):

Spring Web

Spring Boot DevTools

## 2-Executando Projeto

Clica na classe ApiApplication, clica em Run.



```

package br.com.exemplo.api;

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

@SpringBootApplication
public class ApiApplication {

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(ApiApplication.class, args);
    }
}

```

No seu navegador digite <http://localhost:8080/>

### Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Jun 25 15:06:48 BRT 2024

There was an unexpected error (type=Not Found, status=404).

No static resource:

```

org.springframework.web.servlet.resource.NoResourceFoundException: No static resource
at org.springframework.web.servlet.mvc.ResourceHttpRequestHandler.handleRequest(ResourceHttpRequestHandler.java:585)
at org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter.handle(HttpRequestHandlerAdapter.java:52)
at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1089)
at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:979)
at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1014)
at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:903)
at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:564)
at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:885)
at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:658)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:140)
at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:51)

```

Isso quer dizer que está configurado corretamente, faltando definir o controle de rotas

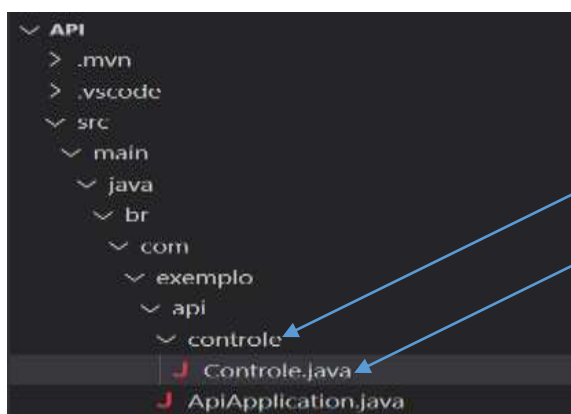
## 3-Implementando o Controle

O controle tem como objetivo criar rotas em sua aplicação.

Clica em cima de api -> clica com o botão direito -> nova pasta

Pasta chamada controle

Dentro dessa pasta cria um arquivo chamado Controle.java



Pasta controle

Classe Controle

Na classe Controle crie um método mensagem

```

package br.com.exemplo.api.controle;

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

@RestController
public class Controle {

    @GetMapping("/")
    public String mensagem(){
        return "Olá, Mundo";
    }
}

```

```

@GetMapping("/boasVindas")
public String boasVindas(){
    return "Seja bem vindo(a)";
}

```

## Utilizando @PathVariable

Essa annotation tem como função, obter dados de uma url. Vamos supor que temos a seguinte rota: `http://localhost:8080/curso/200`, o termo `curso` é a nossa rota, já o valor `200` é o valor que desejamos obter utilizando a annotation `@PathVariable`.

```

@GetMapping("/boasVindas/{nome}")
public String boasVindas(@PathVariable String nome){
    return "Seja bem vindo(a) " + nome;
}

```

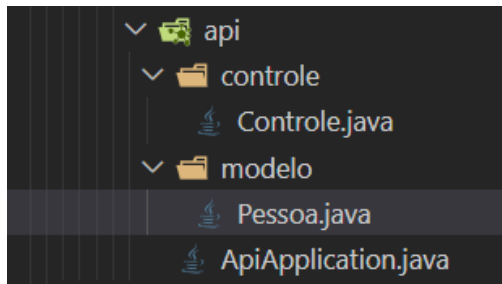
## 4-Trabalhando com Modelos

O modelo é uma representação de dados, ele vai ter duas funcionalidades importantes que são: a manipulação de dados e a criação de tabelas.

Quando trabalhamos com uma API, geralmente temos muitos dados para manipular, e como podemos tornar fácil essa transição de dados? Através de um objeto, que será criado a partir de um modelo. Todo o dado que você queira receber ou enviar de uma API que não seja por url, deverá ter um modelo para o Spring saber como trabalhar com determinadas informações.

Em api crie uma nova pasta chamada modelo

Em modelo crie uma classe Pessoa.java



Crie os métodos get e set

```
package br.com.exemplo.api.modelo;

public class Pessoa {
    private String nome;
    private int idade;

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public int getIdade() {
        return idade;
    }
    public void setIdade(int idade) {
        this.idade = idade;
    }
}
```

## 5-Vinculando modelos e controles

Na classe Controle cria uma rota

```
@PostMapping("/pessoa")
public Pessoa pessoa(@RequestBody Pessoa p){
    return p;
}
```

## FASE 2

### Testando projeto com Thunder Client

Baixa a extensão thunder client

Clica no símbolo do thunder cliente

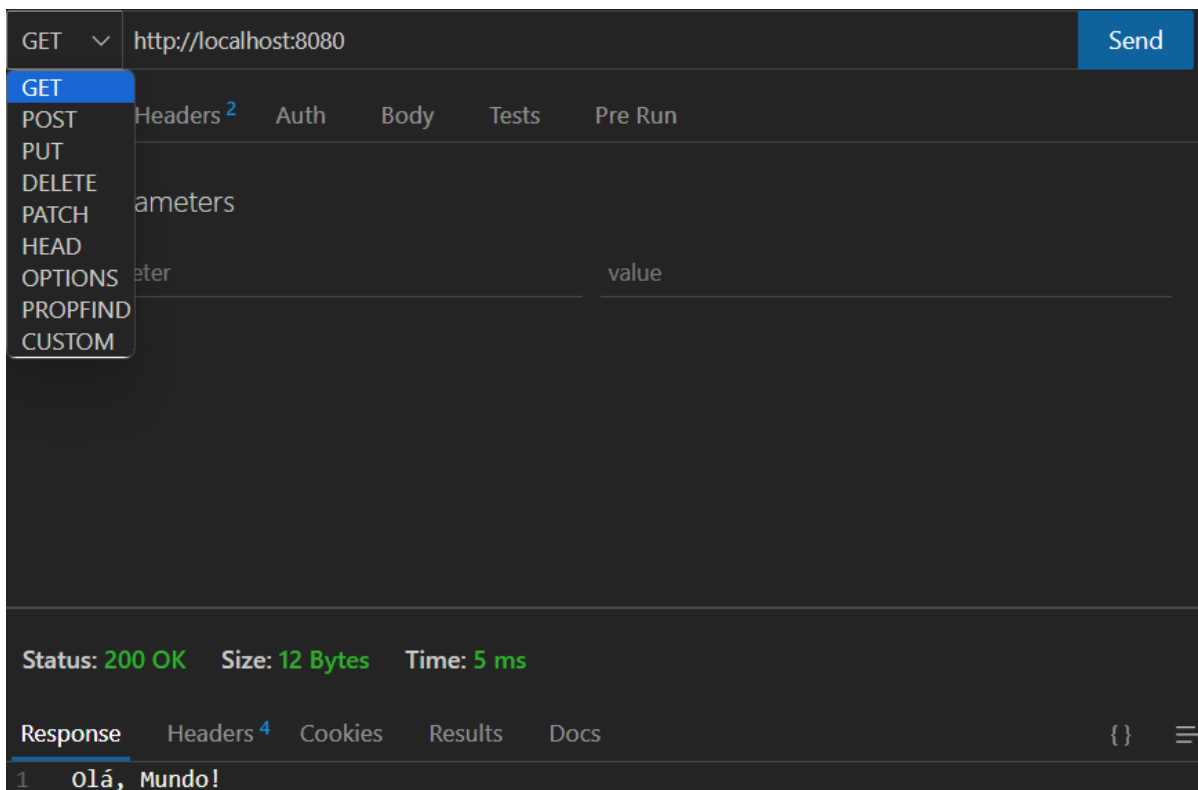


Clica em

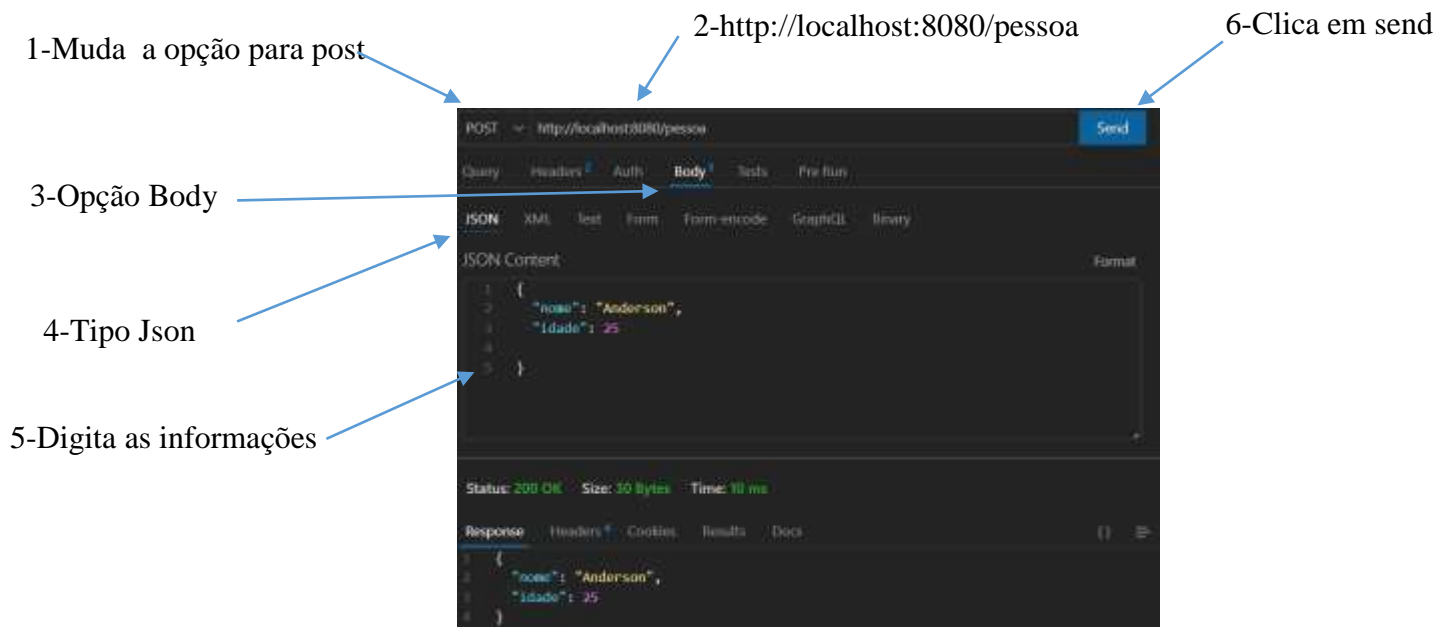
New Request



Escolhe a opção GET, digita a url inicial e clica em send



Deverá aparecer as informações da url

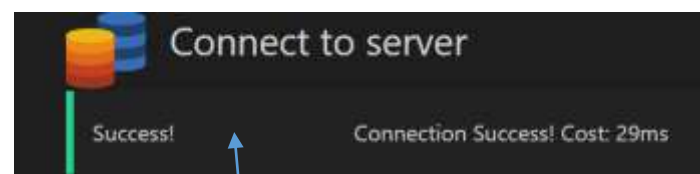
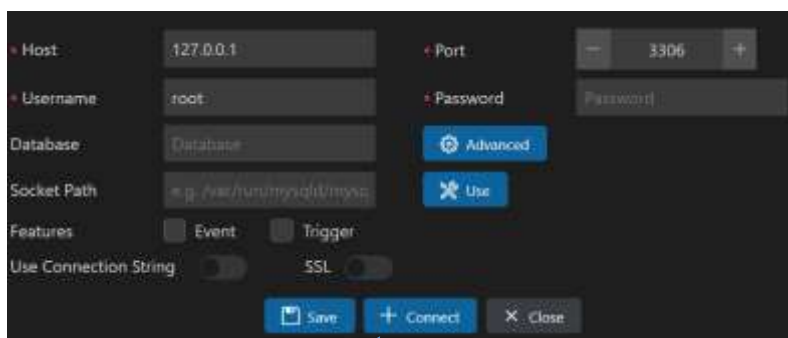
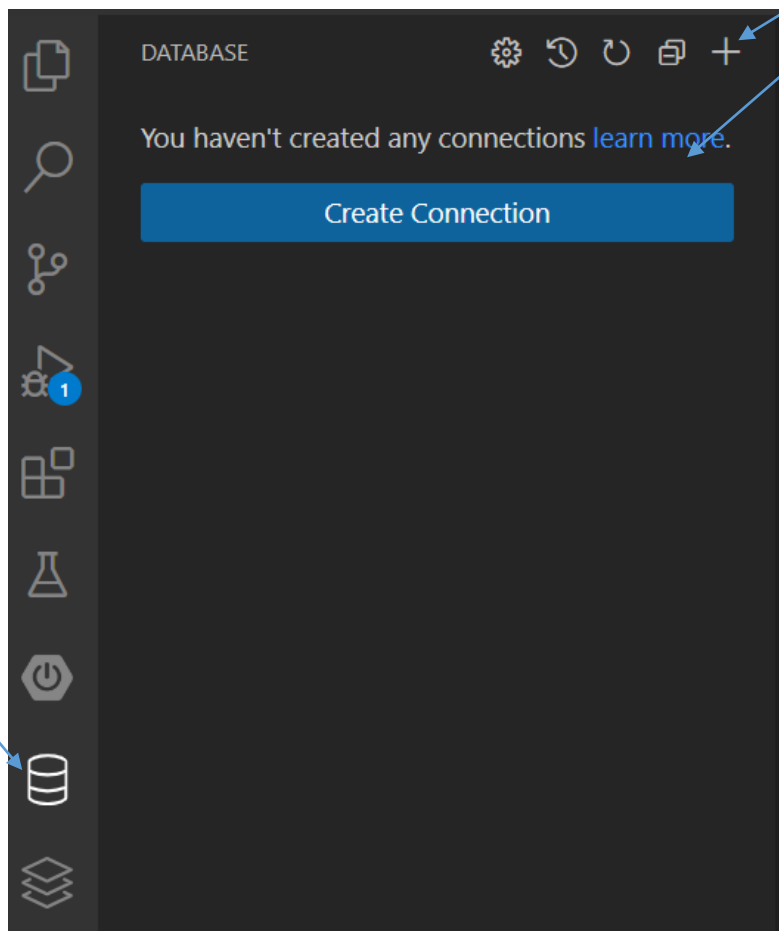


## Configurando banco de dados MySQL

Instala a extensão do mysql

2-Clica em create connection

1-Clica em database



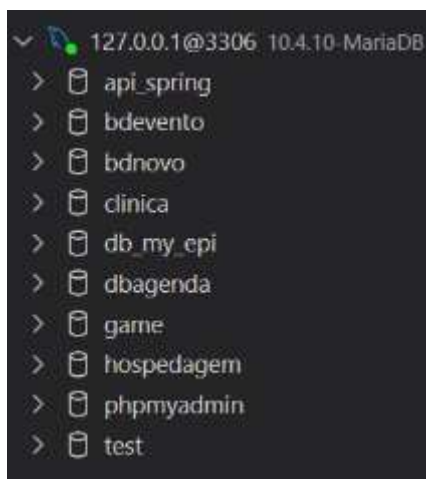
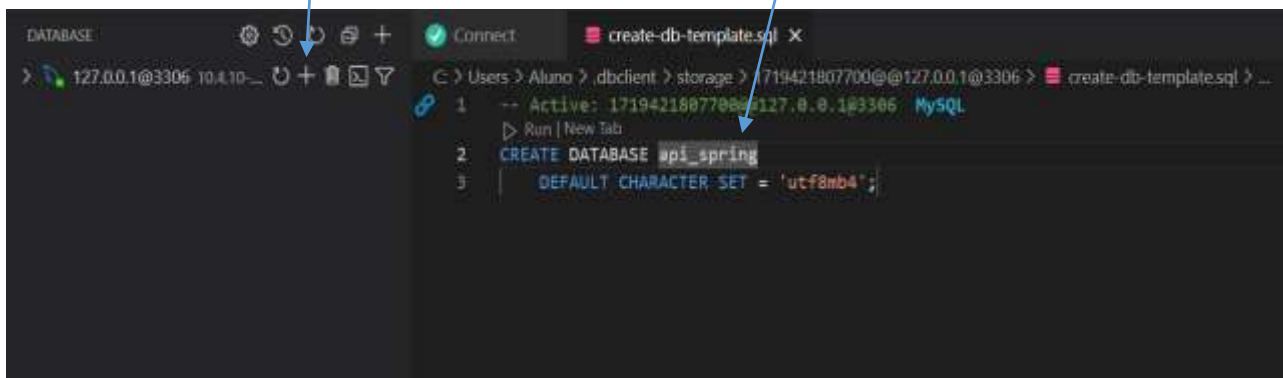
3-Clica em connect

4-Conexão estabelcida

## Criando Base

Clica em new database

Digite o nome da base de dados

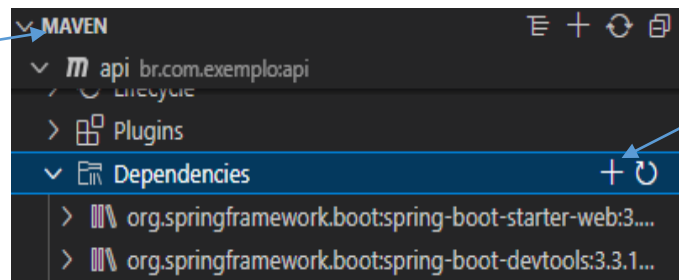


## Implementando as dependências MySQL e JPA

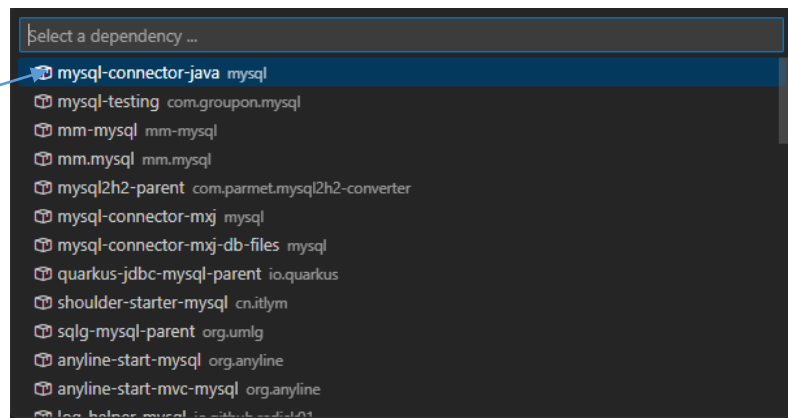
Para o servidor



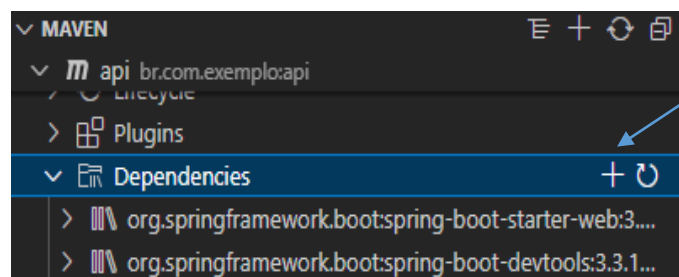
Clica em Maven



Digita mysql -> enter



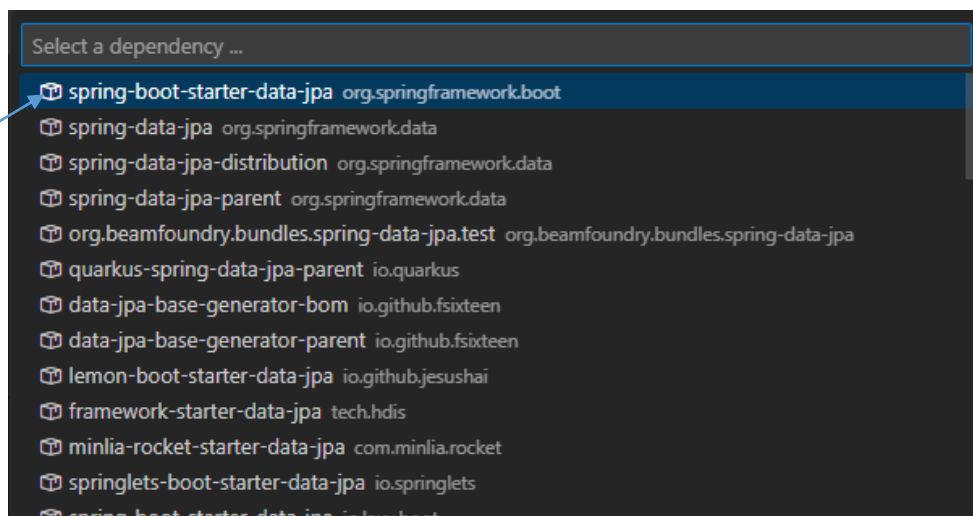
Na lista escolha



add

Digita: data-jpa -> enter

Escolha



## Configurando modelo para gerar tabela

Na classe pessoa crie um novo atributo chamado código

Especifica a criação da  
tabela

Responsável pela  
criação da chave  
primária

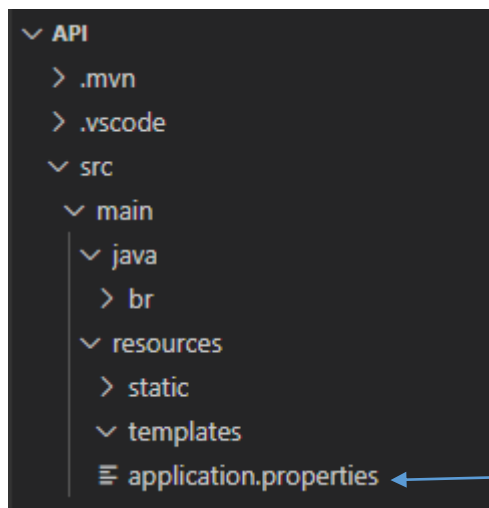
Autoincrement

```
@Entity
public class Pessoa {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int codigo;
    private String nome;
    private int idade;
}
```

## FASE 3

### Conexão com MySQL



Digita as informações

Clica em:

Muda para o nome  
da base criada:  
api\_spring

```
# Altera a estrutura da tabela caso a entidade tenha mudanças.  
spring.jpa.hibernate.ddl-auto=update  
  
# Acesso ao banco de dados  
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/nome_base_de_dados  
  
# Usuário do banco de dados  
spring.datasource.username=usuário  
  
# Senha do banco de dados  
spring.datasource.password=senha
```

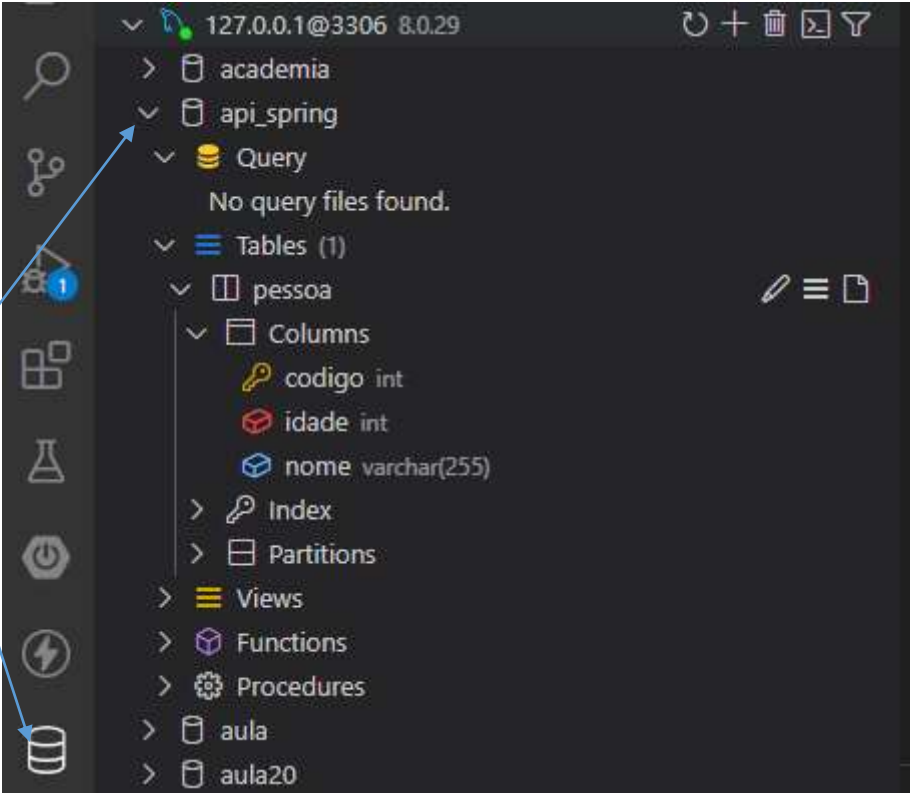
Executa o projeto

Após executar

1-Clica em database

2-Expande a base  
api\_spring

Observe que a  
tabela foi criada  
automaticamente

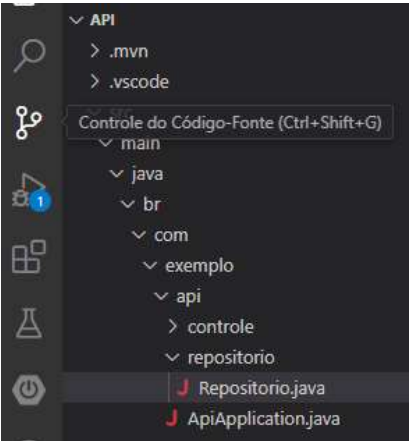


## Criando repositório

Repositório em uma aplicação Spring, tem como objetivo, dispor funcionalidades de manipulação de registros com algum banco de dados, também é conhecido como camada de persistência. Quando efetuamos uma implementação do CrudRepostory, teremos acesso a funções básicas o banco de dados como: cadastrar, selecionar, alterar, excluir, filtrar...

Clica com o botão direito em api -> nova pasta -> repositorio

Dentro da pasta cria o arquivo Repositorio.java



O arquivo será uma interface que herda de CrudRepository

CrudRepository: Dispõe funcionalidades de CRUD (Create, Read, Update e Delete)

Agora podemos codificar nosso arquivo de repositório:

```
@Repository
public interface Repositorio extends CrudRepository<Pessoa, Integer>{
    //
}
```

## Annotation @Autowired

Responsável pela injeção de dependência.

## O que é injeção de dependência?

É um padrão de desenvolvimento adotado por vários frameworks, quando é necessário manter o baixo nível de acoplamento e a alta coesão em um projeto.

**Baixo acoplamento:** Uma classe não deve depender exclusivamente de outra para o seu funcionamento

**Alta coesão:** Quando uma classe é designada para realizar ações específicas. Vamos supor que precisamos manipular data e hora, se formos trabalhar todas as ações em uma só classe, teremos uma baixa coesão. Caso separarmos em duas classes, sendo elas Data e Hora por exemplo, estaremos implementando o conceito de alta coesão

No arquivo Controle digite o código abaixo

```
@RestController
public class Controle {
    @Autowired
    private Repositorio acao;
```

## Efetuando cadastros com save()

O comando save() é responsável por cadastrar ou alterar registros em uma tabela no banco de dados. Não será necessário criar nenhum método para efetuar uma inserção na tabela, quando

temos um repositório, automaticamente o método save() estará pronto para executarmos um cadastramento ou alteração.

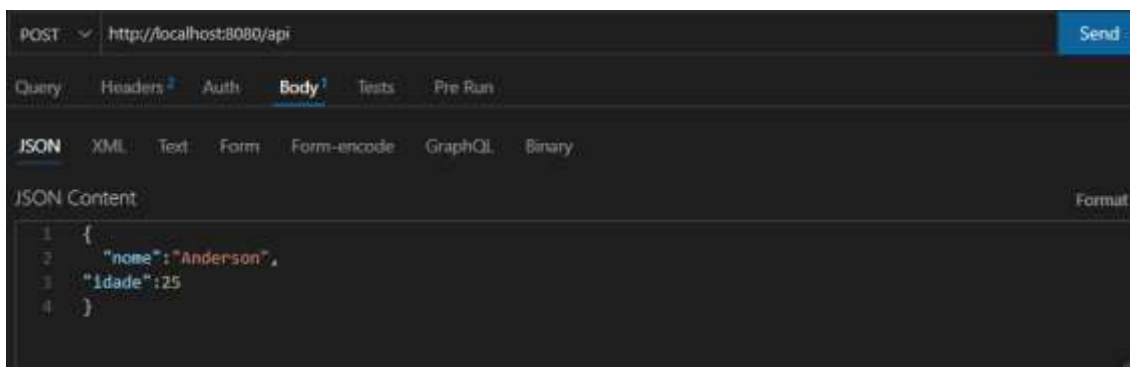
```
@RestController
public class Controle {

    @Autowired
    private Repositorio acao;

    @PostMapping("/api")
    public Pessoa cadastrar(@RequestBody Pessoa obj){
        return acao.save(obj);
    }
}
```

Vamos testar:

Clica no Thunder cliente > New Request > muda para post

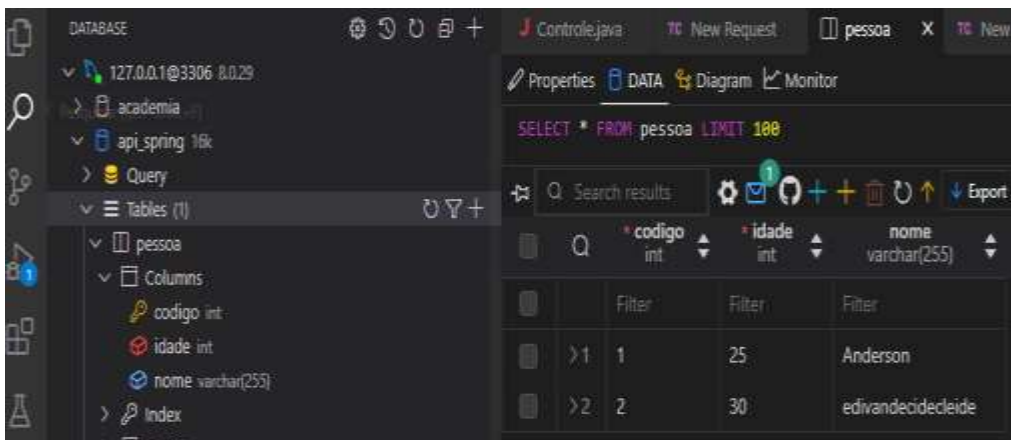


Observe que cria o código automático

	Response	Headers <sup>4</sup>	Cookies	Results	Docs
1	{				
2	"codigo": 1,				
3	"nome": "Anderson",				
4	"idade": 25				
5	}				

Clica em database->duplo clique na tabela pessoa

Irá verificar as informações inseridas no banco



## FASE 4

### Listando dados com o comando findAll()

Abra seu arquivo de repositório e faça a seguinte implementação:

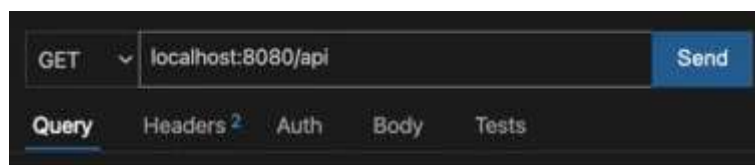
```
@Repository
public interface Repositorio extends CrudRepository<Pessoa, Integer>{

    List<Pessoa> findAll();
}
```

Agora podemos ir ao arquivo de controle e criar uma rota para efetuar a seleção:

```
@GetMapping("/api")
public List<Pessoa> selecionar(){
    return acao.findAll();
}
```

Podemos testar de duas maneiras, uma sendo via Thunder Client



```

{
  "codigo": 1,
  "nome": "Anderson",
  "idade": 25
},
{
  "codigo": 2,
  "nome": "edivandecidecleide",
  "idade": 30
}

```

E outra através o navegador (localhost:8080/api):

```
[{"codigo":1,"nome":"Anderson","idade":25},{"codigo":2,"nome":"edivandecidecleide","idade":30}]
```

### Filtrando dados com o findBy()

O comando **findBy()** serve como se fosse o comando **WHERE** do SQL.

Vamos abrir o nosso repositório e criar o método findByCodigo

```
Pessoa findByCodigo(int codigo);
```

Vamos criar nossa rota no arquivo de controle:

```

@GetMapping("/api/{codigo}")
public Pessoa selecionarPeloCodigo(@PathVariable int codigo){
    return acao.findByCodigo(codigo);
}

```

### Alterando dados com save()

funciona de maneira parecida com o cadastro, porém para que funcione, será necessário passar um objeto completo, contendo todas as características de uma pessoa. Nosso modelo é composto por: código, nome e idade, então para que funcione, será necessário passar todos esses dados.

Utilizamos também uma nova annotation de requisição, chamada **@PutMapping**, que faz com que nossa API saiba que uma rota com esse tipo de requisição será responsável por atualizar um registro.

abra o arquivo de controle e implemente o seguinte código:

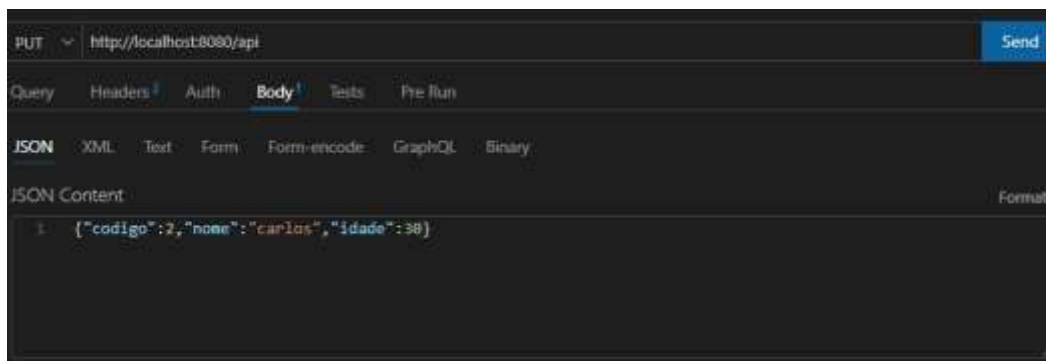
```

@PutMapping("/api")
public Pessoa editar(@RequestBody Pessoa obj){
    return acao.save(obj);
}

```



Utilizando o Thunder Client, podemos testar:



## Removendo dados com delete()

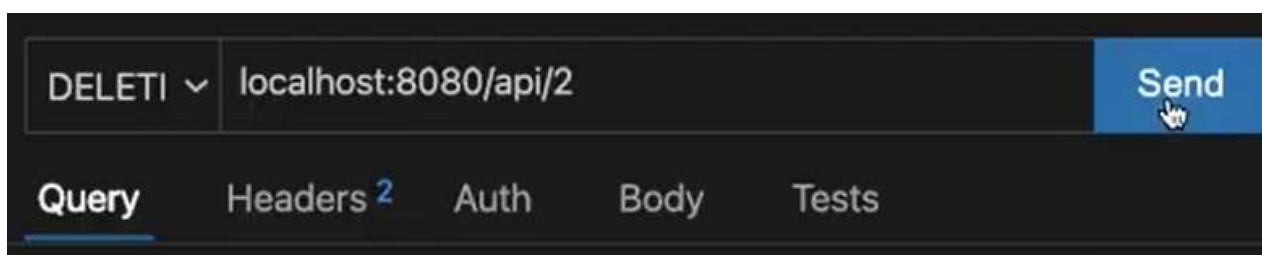
Para o método delete() funcionar, basta enviarmos um objeto completo. Em nossa rota para efetuarmos a exclusão.

Haverá o uso de mais uma annotation para especificar o tipo de requisição, para informar uma rota para exclusão de registros, utilize a annotation **@DeleteMapping**.

No arquivo de controle, crie a seguinte rota:

```
@DeleteMapping("/api/{codigo}")
public void remover(@PathVariable int codigo){
    Pessoa obj = selecionarPeloCodigo(codigo);
    acao.delete(obj);
}
```

Podemos testar utilizando o Thunder Client:



Acesse o a url pelo navegador: localhost:8080/api