



Desenvolvimento de Sistemas

Linguagem de programação para *back-end* (APIs REST)

Neste conteúdo, você terá uma visão mais abrangente sobre o desenvolvimento do *back-end* especificamente para APIs REST, usando o Spring Framework. Durante a leitura, você verá muitos processos familiares, mas com detalhes minuciosos que diferem um “website” de um “web service”.

Além de construir um projeto de API completo, com explicações durante todo o processo, será introduzida uma nova ferramenta muito útil para testar serviços *web*: o Postman. Não se preocupe em instalar agora, pois isso será feito assim que chegar a hora. E após finalizar os testes, será feita a integração de seu serviço *web* a um *website* totalmente isolado do projeto para você entender como funciona na prática a integração de sistemas.

Prepare seu NetBeans e seu Spring Initializr que você começará suas programações.

Criação do projeto

O primeiro passo é acessar a página do Spring Initializr.



Project
☒ Gradle - Groovy
☐ Gradle - Kotlin ☐ Maven

Language
☒ Java ☐ Kotlin
☐ Groovy

Spring Boot
☐ 3.1.0 (SNAPSHOT) ☐ 3.1.0 (M2) ☐ 3.0.6 (SNAPSHOT)
☒ 3.0.5 ☐ 2.7.11 (SNAPSHOT) ☐ 2.7.10

Project Metadata
Group
Artifact
Name
Description
Package name
Packaging ☒ Jar ☐ War
Java ☐ 20 ☒ 17 ☐ 11 ☐ 8

Dependencies ADD DEPENDENCIES... CTRL + B
No dependency selected

GENERATE CTRL + G EXPLORE CTRL + SPACE SHARE...

Figura 1 – Spring Initializr

Fonte: Spring Initializr (s.d.)

Defina nessa página as seguintes configurações:

- ◆ Project: Maven
- ◆ Language: Java
- ◆ Spring Boot: 3.0.6 (ou maior versão estável)
- ◆ Group: com.api
- ◆ Artifact: tarefas
- ◆ Description: API Rest de Projeto de Tarefas
- ◆ Packaging: Jar
- ◆ Java: 17 (ou versão instalada em seu computador)

Por fim, você precisa definir as dependências de seu projeto. Para projetos de APIs REST, deve ser usado apenas o **Spring Web**.

Dependencies

[ADD DEPENDENCIES...](#) CTRL + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Figura 2 – Spring Initializr

Fonte: Spring Initializr (s.d.)

Após concluir todas as configurações, clique em **Generate** e será feito o *download* de uma pasta compactada (.zip) contendo o projeto completo.

Finalizado o *download*, você pode extrair o conteúdo em seu computador e abrir a pasta do projeto com o Apache NetBeans IDE.

Se é a sua primeira vez abrindo um projeto Maven com o Spring Boot no NetBeans, é possível que você se depare com a mensagem “unloadable”.

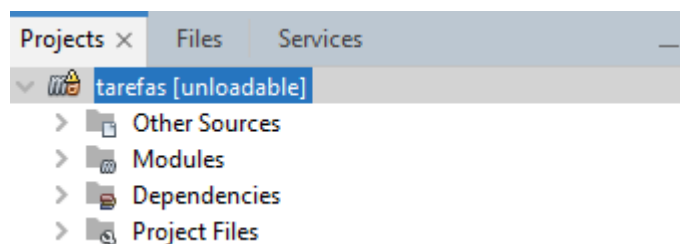


Figura 3 – Estrutura do projeto no NetBeans

Fonte: Apache NetBeans IDE (2023)

Para corrigir isso, clique com o botão direito do *mouse* sobre o projeto e selecione a opção “Resolve Project Problems”. Uma nova janela será aberta.

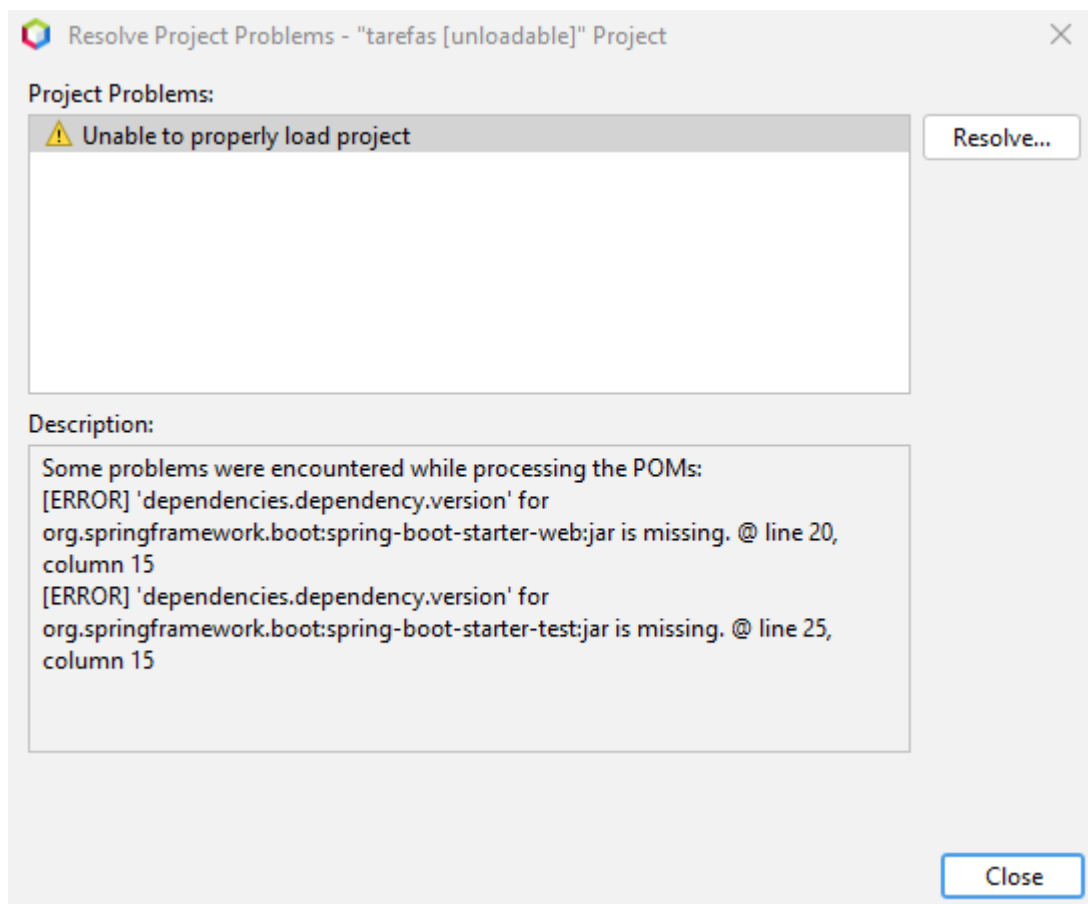


Figura 4 – Janela de “Resolve Project Problems” do NetBeans

Fonte: Apache NetBeans IDE (2023)

Nessa nova tela, há mais detalhes sobre o erro que se refere às dependências `org.springframework.boot:spring-boot-starter-web:jar` e `org.springframework.boot:spring-boot-starter-test:jar`. Para corrigir esse problema, é muito simples! Clique no botão **Resolve...** e o NetBeans se encarregará de fazer todas as correções.

Assim que as correções forem concluídas, a mensagem “This problem was resolved” aparecerá. Clique no botão “Close” para fechar a janela.

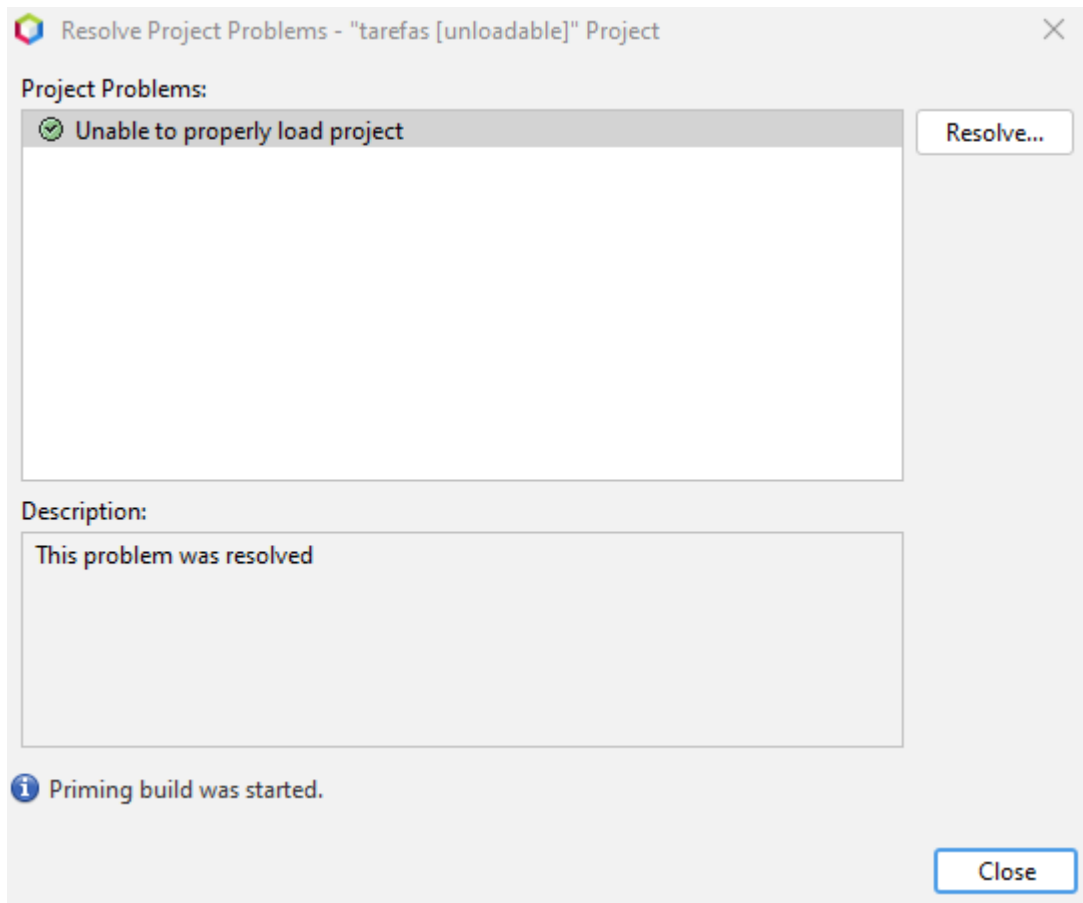


Figura 5 – Janela de “Resolve Project Problems” do NetBeans

Fonte: Apache NetBeans IDE (2023)

Esse processo só precisa ser feito na primeira vez que você abrir um projeto Maven com o Spring. Após isso, o NetBeans salvará as correções para que esse erro não ocorra novamente durante a abertura de novos projetos.

Agora que as configurações foram finalizadas, execute o arquivo principal do projeto para verificar se está tudo funcionando.

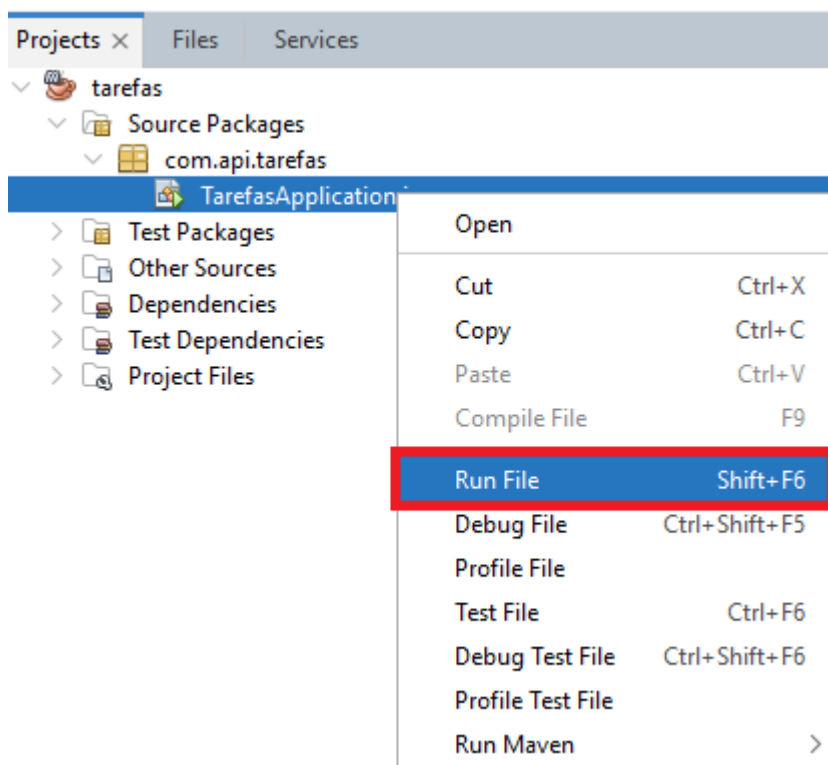


Figura 6 – Estrutura de arquivos do projeto no NetBeans

Fonte: Apache NetBeans IDE (2023)

Quando você executa um projeto Spring Web, o Spring Boot iniciará um servidor *web* embutido (por padrão, o Tomcat) e implantará o seu aplicativo nesse servidor.

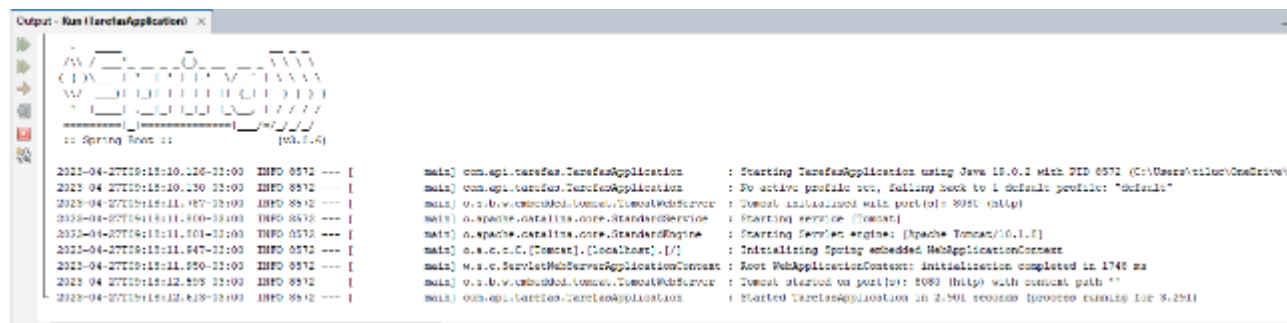


Figura 7 – Terminal do NetBeans executando o projeto

Fonte: Apache NetBeans IDE (2023)

Ao executar o aplicativo Spring Web, você pode acessá-lo por meio de um navegador da *web* ou por meio de outras ferramentas de teste de API, como o Postman, que será visto mais para a frente ainda neste conteúdo. Por padrão, a URL desse acesso é <http://localhost:8080>.

A seguir, serão feitas modificações no projeto que incluem a criação de novas classes, pacotes etc. Essas mudanças não são reconhecidas pelo servidor *web* já em execução. Então lembre-se de finalizar o processo de execução do projeto no NetBeans e iniciá-lo

novamente para que as alterações sejam reconhecidas.



Estrutura do projeto

Nesse projeto, será usada a arquitetura MVC para organizar o conteúdo. Para começar, crie um pacote chamado **model** e outro pacote chamado **controller** dentro do pacote **com.api.tarefas**.

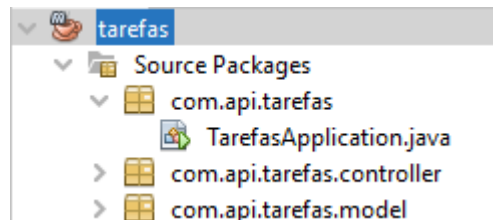


Figura 8 – Estrutura do projeto no NetBeans

Fonte: Apache NetBeans IDE (2023)

- ◆ **Model:** esse pacote será responsável por representar os dados do sistema e as regras de negócios que se aplicam aos dados.
- ◆ **Controller:** esse pacote conterá todas as classes que processarão as requisições HTTP feitas pelos endpoints da API.

Classe Principal

Esse é o arquivo executado para rodar seu projeto. E como já visto, a classe principal da aplicação Spring é responsável por definir as configurações básicas do Spring, inicializar o servidor *web* e iniciar a execução da aplicação.

Mantenha esse arquivo onde ele está e não faça nenhuma alteração nele. Os pacotes criados são destinados apenas para as novas classes que serão criadas a seguir.

Criação do Model

Para começar, será criada uma classe Java chamada “Tarefa”, que representará uma tarefa dentro do pacote **com.api.tarefas.model**. Antes de escrever os atributos e os métodos dessa classe, adicione a anotação `@Component` para indicar que essa classe é um componente Spring. Essa anotação é feita em cima da declaração da classe.

```
package com.api.tarefas.model;
import org.springframework.stereotype.Component;
@Component
public class Tarefa {
}
```

Atenção: não se esqueça de adicionar a declaração de importação de `org.springframework.stereotype.Component`. Essa é uma importação necessária para utilizar a anotação `@Component` em uma classe Java e torná-la gerenciada pelo Spring. Toda anotação do Spring Framework precisa de sua devida declaração de importação. Você pode adicioná-la facilmente clicando sobre o ícone de lâmpada que aparecer ao lado e selecionando a opção “Add import for <caminho>”.

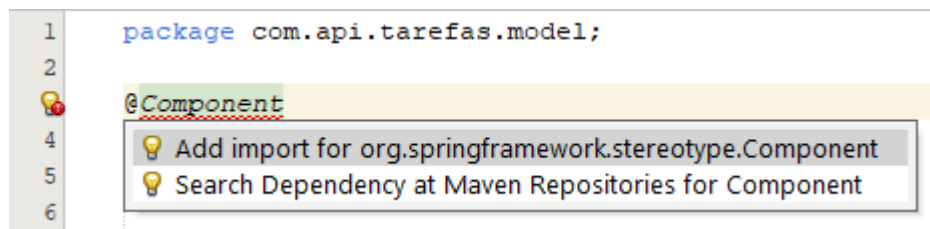


Figura 9 – Editor de texto do NetBeans

Fonte: Apache NetBeans IDE (2023)

Agora é possível seguir com a codificação da classe “Tarefa”. Essa classe terá os seguintes atributos:

- ◆ **id**: um identificador numérico para identificar cada tarefa
- ◆ **descricao**: texto descrevendo a tarefa que será realizada
- ◆ **completa**: um booleano para informar se a tarefa foi concluída (valor *true*) ou não (valor *false*)

Também será implementado o método construtor da classe e os métodos *getters* e *setters* de cada atributo. O código ficará da seguinte maneira:

```
package com.api.tarefas.model;
import org.springframework.stereotype.Component;
@Component
public class Tarefa {
    // Atributos
    private int id;
    private String descricao;
```



```
private boolean completa;

// Construtores
public Tarefa() {}

public Tarefa(int id, String descricao, boolean completa) {
    this.id = id;
    this.descricao = descricao;
    this.completa = completa;
}

// Métodos Getters e Setters
public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getDescricao() { return descricao; }
public void setDescricao(String descricao) { this.descricao = descricao; }
public boolean isCompleta() { return completa; }
public void setCompleta(boolean completa) { this.completa = completa; }
}
```

Com isso, está concluída a criação da classe Model.

Criação do Controller

Crie uma classe chamada **TarefaController** em **com.api.tarefas.controller** que será responsável por lidar com as requisições HTTP feitas na API.

```
package com.api.tarefas.controller;
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/tarefas")
public class TarefaController {

}
```

Essa classe deve ter a anotação **@RestController** para indicar que ela é uma controladora REST e deve ser registrada com o Spring. É uma anotação com propósito semelhante ao **@Controller** de Spring MVC, mas específico para serviços RESTful. Também é incluída a anotação **@RequestMapping** com valor **"/tarefas"**, indicando que os serviços disponibilizados por essa classe de controle serão acessados pela URL **http://localhost:8080/tarefas**.

Variáveis auxiliares



Como seu projeto não utiliza banco de dados, serão armazenadas as tarefas criadas em uma lista dentro da classe `TarefaController`. Isso permitirá que as tarefas sejam mantidas na memória durante a execução do programa. Além disso, também será criada uma variável auxiliar para garantir que o id das tarefas não se repita quando for feito o cadastro – caso estivesse sendo utilizado um banco de dados, isso seria feito automaticamente pela propriedade `AUTO_INCREMENT`.

```
package com.api.tarefas.controller;
import java.util.ArrayList;
import java.util.List;
import org.springframework.web.bind.annotation.*;
import com.api.tarefas.model.Tarefa;
@RestController
@RequestMapping("/tarefas")
public class TarefaController {
    // Variáveis auxiliares
    private List tarefa = new ArrayList<>();
    private int proximoId = 1;
}
```

Agora serão vistos os métodos.

Métodos GET, POST, PUT e DELETE

Para lidar com as diferentes operações da API, serão criados os seguintes métodos dentro da classe:

- ◆ `buscarTodasTarefas`: para buscar todas as tarefas cadastradas na lista
- ◆ `buscarTarefaPorId`: para buscar uma tarefa específica pelo id
- ◆ `criarTarefa`: para adicionar uma nova tarefa à lista
- ◆ `atualizarTarefa`: para atualizar os dados de uma tarefa cadastrada
- ◆ `deletarTarefa`: para remover uma tarefa cadastrada

Cada método estará associado a um verbo HTTP específico, que pode ser GET, POST, PUT ou DELETE. Logo, é necessário utilizar as anotações do Spring para indicar qual método HTTP deve ser usado por cada método, sendo elas:

- ◆ @GetMapping: para associar ao método GET, usar nos métodos buscarTodasTarefas e buscarTarefaPorId. Assim, esses métodos serão acionados apenas ao receber requisições com método GET para a URL do projeto.
- ◆ @PostMapping: para associar ao método POST, usar no método criarTarefa. O método será executado apenas quando chegar requisição do tipo POST ao sistema.
- ◆ @PutMapping: para associar ao método PUT, usar no método atualizarTarefa. O método responderá apenas às requisições com método PUT que chegarem ao sistema.
- ◆ @DeleteMapping: para associar ao método DELETE, usar no método deletarTarefa. O método responderá às requisições de método DELETE.

Isso significa que, diferentemente do projeto Spring MVC, não se depende necessariamente de mapeamentos na URL, como localhost:8080/tarefas/gravar ou localhost:8080/tarefas/excluir. Apenas será usado http://localhost:8080/tarefas especificando o método de requisição e, a partir disso, a classe Controller selecionará o método correto a ser executado.

Clique ou toque para visualizar o conteúdo.

GET

```
@GetMapping
public List buscarTodasTarefas() {
    return tarefas;
}
```

Esse método é um *endpoint* do tipo GET que retorna todas as tarefas cadastradas na API. Ele simplesmente retorna a lista de tarefas que está armazenada na variável `tarefas`.

```
@GetMapping("/{id}")
public Tarefa buscarTarefaPorId(@PathVariable int id) {
    for (Tarefa tarefa : tarefas) {
        if (tarefa.getId() == id) {
            return tarefa;
        }
    }
    return null;
}
```

Esse método também é um endpoint do tipo GET, porém ele recebe um parâmetro na URL que indica o ID da tarefa que se deseja buscar. O parâmetro “id” é capturado pela anotação `@PathVariable` – essa anotação é responsável por extrair valores de variáveis de caminho em uma URL. O método faz uma iteração sobre a lista de tarefas até encontrar a tarefa com o ID informado, e então retorna essa tarefa. Caso nenhuma tarefa seja encontrada com o ID informado, ele retorna “null”.

Em suma, quando uma requisição não trazer um parâmetro extra, `buscarTodasTarefas()` será executado. Caso haja um parâmetro, o método `buscarTarefaPorId()` é que executará.

POST

```
@PostMapping
public Tarefa criarTarefa(@RequestBody Tarefa tarefa) {
    int id = tarefas.size() + 1;
    tarefa.setId(id);
}
```

```
    tarefas.add(tarefa);  
    return tarefa;  
}
```

Esse método é um *endpoint* do tipo POST que recebe um JSON no corpo da requisição contendo as informações de uma nova tarefa. Ele gera um novo ID para a tarefa (o ID é igual ao número de tarefas cadastradas mais um), adiciona a tarefa na lista de tarefas e então retorna a tarefa que acabou de ser criada.


Aqui, é usada a anotação `@RequestBody`, que é uma anotação usada para indicar que o parâmetro de um método deve ser extraído do corpo da requisição HTTP – nesse caso, refere-se ao parâmetro “tarefa”.

Ao usar essa anotação, o Spring mapeia o corpo da requisição HTTP para o objeto Java correspondente ao tipo do parâmetro anotado com `@RequestBody`. Em outras palavras, o Spring converte o conteúdo do corpo da requisição em um objeto Java.

DELETE

```
@PutMapping("/{id}")  
public Tarefa atualizarTarefa(@PathVariable int id, @RequestBody Tarefa tarefaAtualizada) {  
    for (int i = 0; i < tarefas.size(); i++) {  
        Tarefa tarefa = tarefas.get(i);  
        if (tarefa.getId() == id) {  
            tarefa.setDescricao(tarefaAtualizada.getDescricao());  
            tarefa.setCompleta(tarefaAtualizada.isCompleta());  
            return tarefa;  
        }  
    }  
    return null;  
}
```

Perceba que aqui há o uso em conjunto de duas anotações vistas na criação dos métodos anteriores: `@PathVariable` e `@RequestBody`. Isso porque esse método receberá dois parâmetros: primeiro é um inteiro que representa o ID da tarefa e é obtido a partir da anotação `@PathVariable`. O segundo parâmetro é um objeto da classe `Tarefa` que representa a nova versão da tarefa e é obtido a partir da anotação `@RequestBody`.



Esse método é um *endpoint* do tipo PUT que recebe um parâmetro na URL indicando o ID da tarefa que se deseja atualizar, e um JSON no corpo da requisição contendo as informações atualizadas da tarefa. Ele faz uma iteração sobre a lista de tarefas até encontrar a tarefa com o ID informado, atualiza as informações da tarefa com as informações que foram passadas no JSON e então retorna a tarefa atualizada. Caso nenhuma tarefa seja encontrada com o ID informado, ele retorna “null”.

PUT

```
@DeleteMapping("/{id}")
public boolean deletarTarefa(@PathVariable int id) {
    for (int i = 0; i < tarefas.size(); i++) {
        Tarefa tarefa = tarefas.get(i);
        if (tarefa.getId() == id) {
            tarefas.remove(i);
            return true;
        }
    }
    return false;
}
```

Esse método é um *endpoint* do tipo DELETE que recebe um parâmetro na URL indicando o ID da tarefa que se deseja deletar. Ele faz uma iteração sobre a lista de tarefas até encontrar a tarefa com o ID informado, remove a tarefa da lista e retorna *true*. Caso nenhuma tarefa seja encontrada com o ID informado, ele retorna *false*. Note que como o método retorna um booleano, é possível utilizá-lo para saber se a tarefa foi deletada com sucesso ou não.

Para finalizar, é necessário adicionar mais duas anotações à classe Controller.

A primeira é a anotação *@RequestMapping* para mapear uma URL específica para que a classe “controller” seja acessada. Para esse exemplo, será usada a URL **/tarefas**. Assim, se um cliente fizer uma requisição GET para a URL “/tarefas”, o Spring vai automaticamente chamar o método *listarTarefas* da classe *TarefaController*. Da mesma forma, se o cliente fizer uma requisição POST para a URL “/tarefas”, o Spring vai chamar o método *adicionarTarefa*.

A segunda anotação é a *@CrossOrigin*, que permitirá o acesso à API a partir de um domínio diferente. Quando rodar o arquivo *TarefasApplication.java*, a API será iniciada em um servidor *web* Tomcat com o endereço <http://localhost:8080> – esse endereço eletrônico é o domínio da API. Se você criar um arquivo **index.html**, por exemplo, na área de trabalho do Windows, e abrir esse arquivo, o domínio dessa página web será o caminho do arquivo, algo como **C:/Users/MeuUsuario/Desktop/index.html**. Perceba que os domínios são bem diferentes um do outro.

GET

```
@GetMapping
public List buscarTodasTarefas() {
    return tarefas;
}
```

Esse método é um *endpoint* do tipo GET que retorna todas as tarefas cadastradas na API. Ele simplesmente retorna a lista de tarefas que está armazenada na variável `tarefas`.

```
@GetMapping("/{id}")
public Tarefa buscarTarefaPorId(@PathVariable int id) {
    for (Tarefa tarefa : tarefas) {
        if (tarefa.getId() == id) {
            return tarefa;
        }
    }
    return null;
}
```

Esse método também é um endpoint do tipo GET, porém ele recebe um parâmetro na URL que indica o ID da tarefa que se deseja buscar. O parâmetro “id” é capturado pela anotação `@PathVariable` – essa anotação é responsável por extrair valores de variáveis de caminho em uma URL. O método faz uma iteração sobre a lista de tarefas até encontrar a tarefa com o ID informado, e então retorna essa tarefa. Caso nenhuma tarefa seja encontrada com o ID informado, ele retorna “null”.

Em suma, quando uma requisição não trazer um parâmetro extra, `buscarTodasTarefas()` será executado. Caso haja um parâmetro, o método `buscarTarefaPorId()` é que executará.

POST

```
@PostMapping
public Tarefa criarTarefa(@RequestBody Tarefa tarefa) {
    int id = tarefas.size() + 1;
    tarefa.setId(id);
    tarefas.add(tarefa);
    return tarefa;
}
```



```
}
```

Esse método é um *endpoint* do tipo POST que recebe um JSON no corpo da requisição contendo as informações de uma nova tarefa. Ele gera um novo ID para a tarefa (o ID é igual ao número de tarefas cadastradas mais um), adiciona a tarefa na lista de tarefas e então retorna a tarefa que acabou de ser criada.

Aqui, é usada a anotação `@RequestBody`, que é uma anotação usada para indicar que o parâmetro de um método deve ser extraído do corpo da requisição HTTP – nesse caso, refere-se ao parâmetro “tarefa”.

Ao usar essa anotação, o Spring mapeia o corpo da requisição HTTP para o objeto Java correspondente ao tipo do parâmetro anotado com `@RequestBody`. Em outras palavras, o Spring converte o conteúdo do corpo da requisição em um objeto Java.

DELETE

```
@PutMapping("/{id}")
public Tarefa atualizarTarefa(@PathVariable int id, @RequestBody Tarefa tarefaAtualizada) {
    for (int i = 0; i < tarefas.size(); i++) {
        Tarefa tarefa = tarefas.get(i);
        if (tarefa.getId() == id) {
            tarefa.setDescricao(tarefaAtualizada.getDescricao());
            tarefa.setCompleta(tarefaAtualizada.isCompleta());
            return tarefa;
        }
    }
    return null;
}
```

Perceba que aqui há o uso em conjunto de duas anotações vistas na criação dos métodos anteriores: `@PathVariable` e `@RequestBody`. Isso porque esse método receberá dois parâmetros: primeiro é um inteiro que representa o ID da tarefa e é obtido a partir da anotação `@PathVariable`. O segundo parâmetro é um objeto da classe Tarefa que representa a nova versão da tarefa e é obtido a partir da anotação `@RequestBody`.

Esse método é um *endpoint* do tipo PUT que recebe um parâmetro na URL indicando o ID da tarefa que se deseja atualizar, e um JSON no corpo da requisição contendo as informações atualizadas da tarefa. Ele faz uma iteração sobre a lista de tarefas até

encontrar a tarefa com o ID informado, atualiza as informações da tarefa com as informações que foram passadas no JSON e então retorna a tarefa atualizada. Caso nenhuma tarefa seja encontrada com o ID informado, ele retorna “null”.

PUT

```
@DeleteMapping("/{id}")
public boolean deletarTarefa(@PathVariable int id) {
    for (int i = 0; i < tarefas.size(); i++) {
        Tarefa tarefa = tarefas.get(i);
        if (tarefa.getId() == id) {
            tarefas.remove(i);
            return true;
        }
    }
    return false;
}
```

Esse método é um *endpoint* do tipo DELETE que recebe um parâmetro na URL indicando o ID da tarefa que se deseja deletar. Ele faz uma iteração sobre a lista de tarefas até encontrar a tarefa com o ID informado, remove a tarefa da lista e retorna *true*. Caso nenhuma tarefa seja encontrada com o ID informado, ele retorna *false*. Note que como o método retorna um booleano, é possível utilizá-lo para saber se a tarefa foi deletada com sucesso ou não.

Para finalizar, é necessário adicionar mais duas anotações à classe Controller.

A primeira é a anotação *@RequestMapping* para mapear uma URL específica para que a classe “controller” seja acessada. Para esse exemplo, será usada a URL **/tarefas**. Assim, se um cliente fizer uma requisição GET para a URL “/tarefas”, o Spring vai automaticamente chamar o método *listarTarefas* da classe *TarefaController*. Da mesma forma, se o cliente fizer uma requisição POST para a URL “/tarefas”, o Spring vai chamar o método *adicionarTarefa*.

A segunda anotação é a *@CrossOrigin*, que permitirá o acesso à API a partir de um domínio diferente. Quando rodar o arquivo *TarefasApplication.java*, a API será iniciada em um servidor *web* Tomcat com o endereço `http://localhost:8080` – esse endereço eletrônico é

o domínio da API. Se você criar um arquivo **index.html**, por exemplo, na área de trabalho do Windows, e abrir esse arquivo, o domínio dessa página web será o caminho do arquivo, algo como **C:/Users/MeuUsuario/Desktop/index.html**. Perceba que os domínios são bem diferentes um do outro.

Por padrão, as solicitações HTTP entre diferentes domínios são bloqueadas pelos navegadores como medida de segurança, isso é conhecido como política de segurança de mesma origem (Same-origin policy). Essa política impede que *scripts* de um *site* mal-intencionado tenham acesso aos dados de outro *site*.

No entanto, às vezes, é necessário permitir que um *site* externo faça solicitações para uma API, e a anotação `@CrossOrigin` pode ajudar a permitir essas solicitações. Nesse exemplo, será inserida essa anotação com o valor **origins = ""**, que significa que estão sendo permitidas solicitações de qualquer domínio. Você também pode especificar um conjunto de domínios que são permitidos a fazer solicitações para sua API caso queira.

O código completo da classe Controller ficará assim:

```
package com.api.tarefas.controller;
import java.util.ArrayList;
import java.util.List;
import org.springframework.web.bind.annotation.*;
import com.api.tarefas.model.Tarefa;
@RestController
@CrossOrigin(origins = "")
@RequestMapping("/tarefas")
public class TarefaController {
    private List tarefas = new ArrayList<>();
    private int proximoId = 1;
    @PostMapping("")
    public Tarefa criarTarefa(@RequestBody Tarefa tarefa) {
        tarefa.setId(proximoId++);
        tarefas.add(tarefa);
        return tarefa;
    }
    @GetMapping("")
    public List buscarTarefas() {
        return tarefas;
    }
    @GetMapping("/{id}")
    public Tarefa buscarTarefa(@PathVariable int id) {
        for (Tarefa tarefa : tarefas) {
            if (tarefa.getId() == id) {
                return tarefa;
            }
        }
    }
}
```

```
    }  
    return null;  
}  
@PutMapping("/{id}")  
public Tarefa atualizarTarefa(@PathVariable int id, @RequestBody Tarefa tarefa) {  
    for (int i = 0; i < tarefas.size(); i++) {  
        Tarefa t = tarefas.get(i);  
        if (t.getId() == id) {  
            t.setDescricao(tarefa.getDescricao());  
            t.setCompleta(tarefa.isCompleta());  
            return t;  
        }  
    }  
    return null;  
}  
@DeleteMapping("/{id}")  
public boolean deletarTarefa(@PathVariable int id) {  
    for (int i = 0; i < tarefas.size(); i++) {  
        Tarefa tarefa = tarefas.get(i);  
        if (tarefa.getId() == id) {  
            tarefas.remove(i);  
            return true;  
        }  
    }  
    return false;  
}  
}
```

Com isso, está concluída a criação da classe **Controller** e da API.

POSTMAN

O Postman é uma plataforma utilizada no desenvolvimento de APIs que permite aos desenvolvedores testar, documentar e compartilhar APIs de maneira rápida e fácil. O Postman oferece uma interface de usuário amigável que permite que os desenvolvedores enviem solicitações HTTP para APIs e examinem as respostas.

Ele tem diversas funcionalidades, como a capacidade de criar e salvar solicitações HTTP personalizadas, criar testes automatizados para garantir que a API esteja funcionando corretamente, gerenciar variáveis e ambientes, colaborar com outros desenvolvedores, documentar APIs, entre outras.

Instalação e configuração

O Postman está disponível para uso na *web* e também como um aplicativo *desktop* para os sistemas operacionais Windows, Linux e Mac.

O aplicativo *web* do Postman está em desenvolvimento ativo e, por isso, existem recursos que você só conseguirá acessar no aplicativo de *desktop*. Por esse motivo, será usada a versão *desktop* em vez da versão *web* e recomenda-se que você use também.

Para obter a versão mais recente do aplicativo *desktop*, visite a página de *download* oficial e baixe a versão para a sua plataforma.

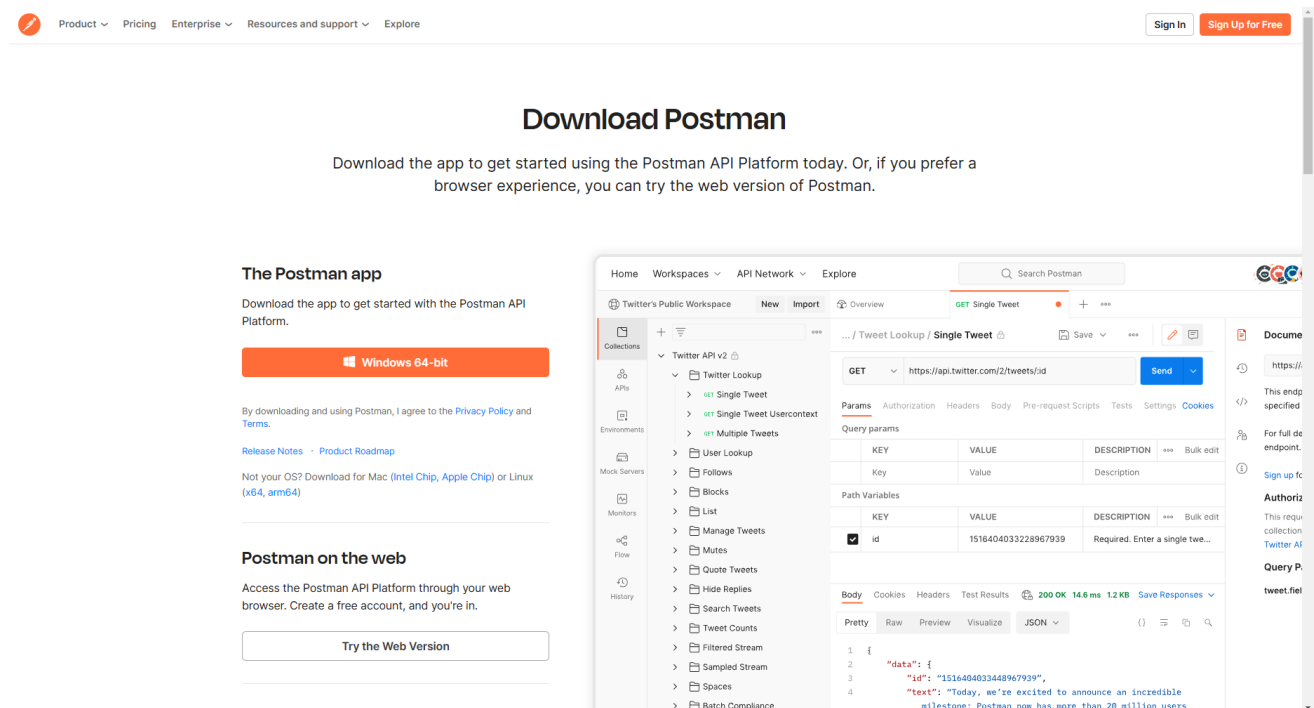


Figura 10 – Página oficial de *download* do Postman

Fonte: Postman (c2023)

Após concluir o *download*, execute o instalador e o Postman será instalado automaticamente no seu sistema. Após terminar a instalação, será aberta uma tela de apresentação da ferramenta em que você poderá escolher entre criar uma nova conta (opção *Create Free Account*) ou usar uma já existente (opção *Sign in*). Você pode realizar esse processo de *login* ou escolher a opção *Skip and go to the app* para pular essa etapa e ir direto ao aplicativo.

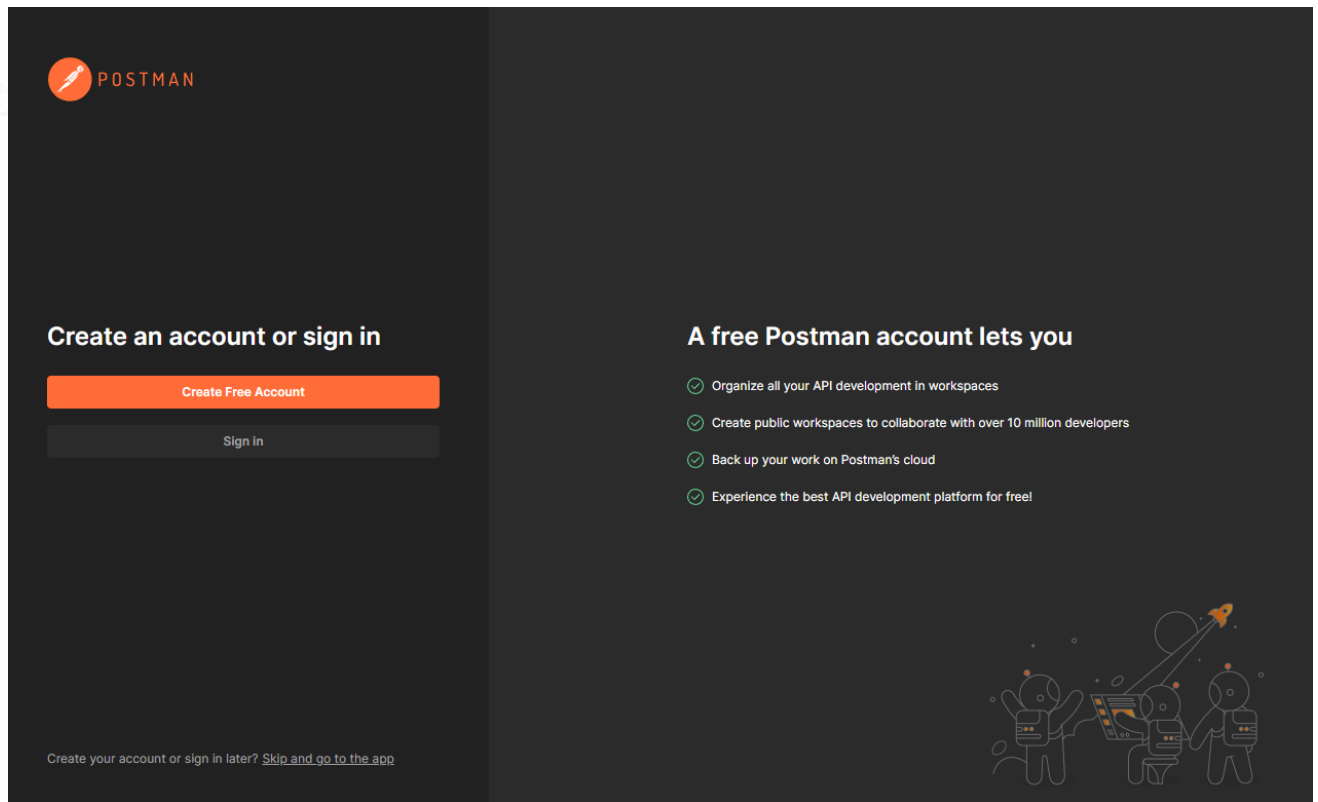


Figura 11 – Tela inicial do Postman

Fonte: Postman (2023)

Teste da API

Para testar a API desenvolvida, basta iniciar a aplicação (execute o arquivo principal no NetBeans – nesse exemplo, trata-se do arquivo *TarefasApplication.java*) e fazer requisições HTTP para o endereço `http://localhost:8080/tarefas`. A seguir, você verá como fazer as requisições POST, GET, PUT e DELETE através do Postman.

POST

Para criar uma nova tarefa, é enviada uma requisição POST com um JSON no corpo da requisição.

1. Selecione o método POST no menu à esquerda da barra de endereço.
2. Insira a URL da API na barra de endereço: `http://localhost:8080/tarefas`.
3. Clique na guia “Body” abaixo da barra de endereço.
4. Selecione “raw” na parte inferior da guia e escolha o formato JSON.

5. Em seguida, digite o corpo da solicitação. Nesse exemplo, será cadastrada a descrição da tarefa como “Fazer compras” e o seu *status* de completa como *false* para sinalizar que a tarefa não está completa ainda.

```
{
  "descricao": "Fazer compras",
  "completa": false
}
```

6. Clique no botão “Send” para enviar a solicitação.
7. O resultado da solicitação será exibido na seção de resposta abaixo da área de envio.

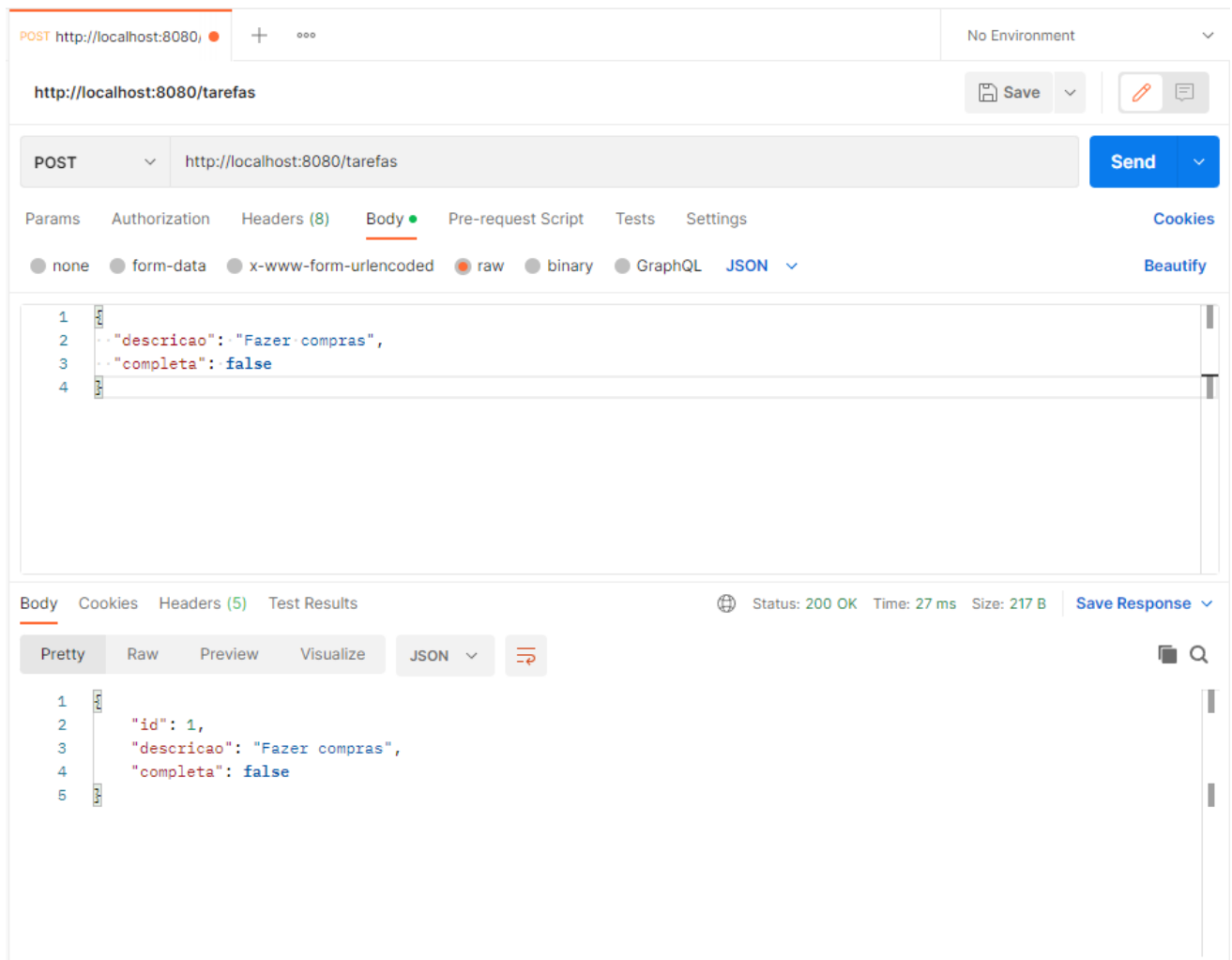


Figura 12 – Requisição POST no Postman

Fonte: Postman (2023)

GET

Para buscar todas as tarefas, é usada a requisição GET na URL.



O processo aqui é mais simples!

1. Selecione o método GET no menu à esquerda da barra de endereço.
2. Insira a URL da API na barra de endereço: `http://localhost:8080/tarefas`.
3. Clique no botão “Send” para enviar a solicitação.

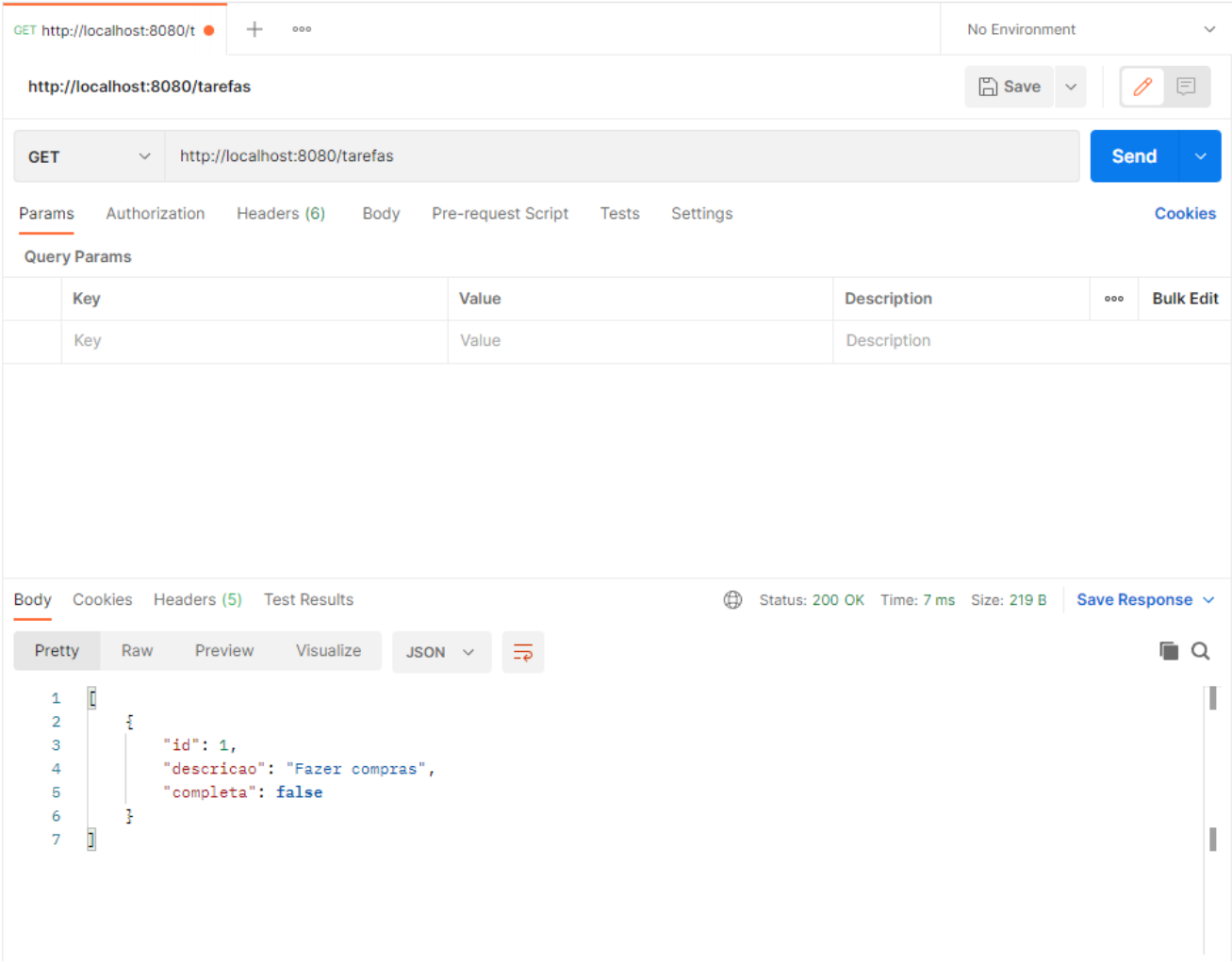


Figura 13 – Requisição GET no Postman

Fonte: Postman (2023)

Para buscar uma tarefa específica, envie uma requisição GET com o ID da tarefa na URL. No exemplo a seguir, foi buscada a tarefa com ID 1 (pois foi cadastrada apenas uma tarefa) usando a URL `http://localhost:8080/tarefas/1`.

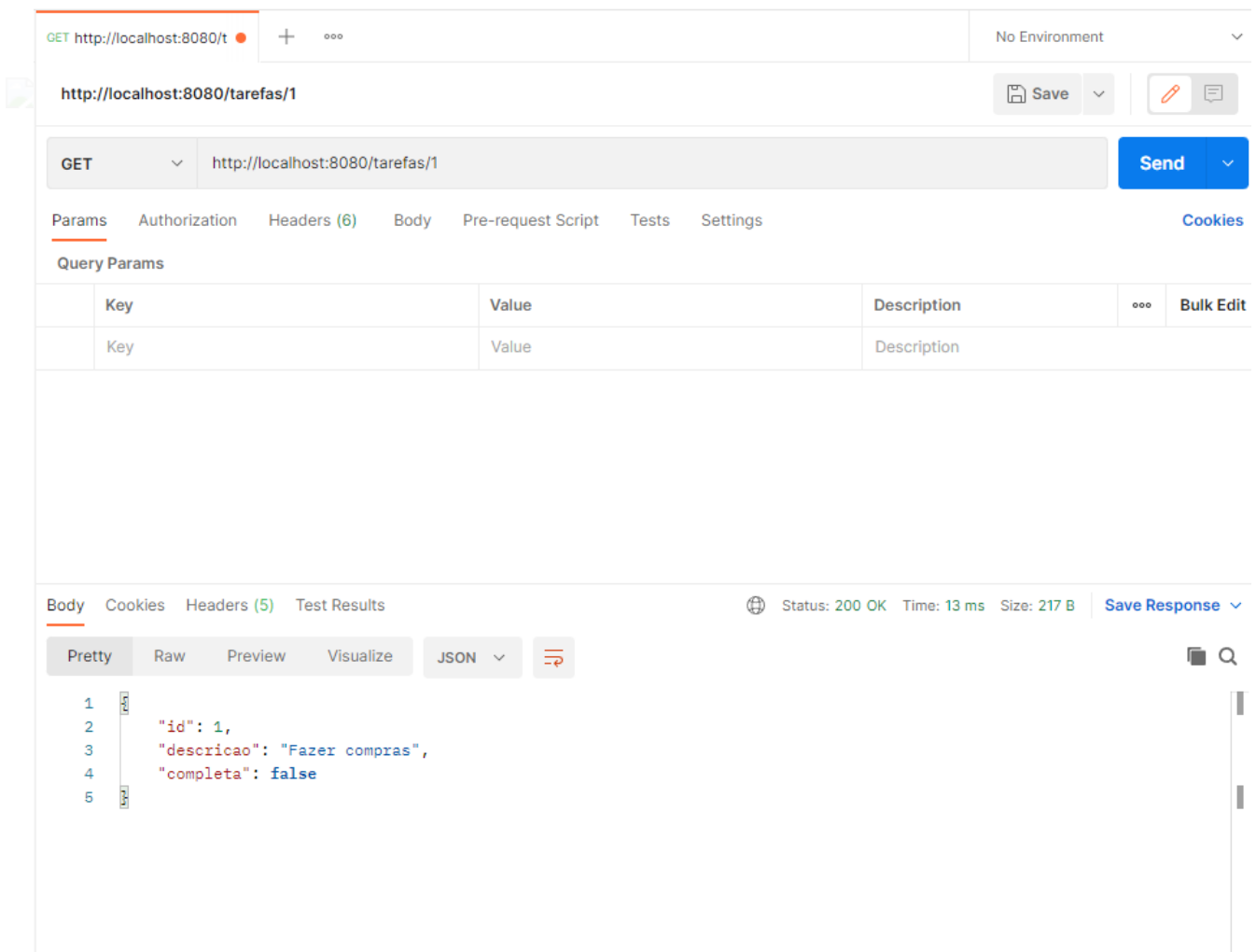


Figura 14 – Requisição GET no Postman

Fonte: Postman (2023)

PUT

Para atualizar uma tarefa, envie uma requisição PUT com o ID da tarefa na URL e um JSON no corpo da requisição.

1. Selecione o método PUT no menu à esquerda da barra de endereço.
2. Insira a URL da API na barra de endereço: `http://localhost:8080/tarefas/1`.
3. Clique na guia “Body” abaixo da barra de endereço.
4. Selecione “raw” na parte inferior da guia e escolha o formato JSON.
5. Em seguida, digite o corpo da solicitação. Nesse exemplo, será alterada a descrição da tarefa e também o seu *status* de completa para *true*.

```
{
  "descricao": "Fazer compras no mercado",
  "completa": true
}
```

```
}
```

6. Clique no botão “Send” para enviar a solicitação.
7. O resultado da solicitação será exibido na seção de resposta abaixo da área de envio.

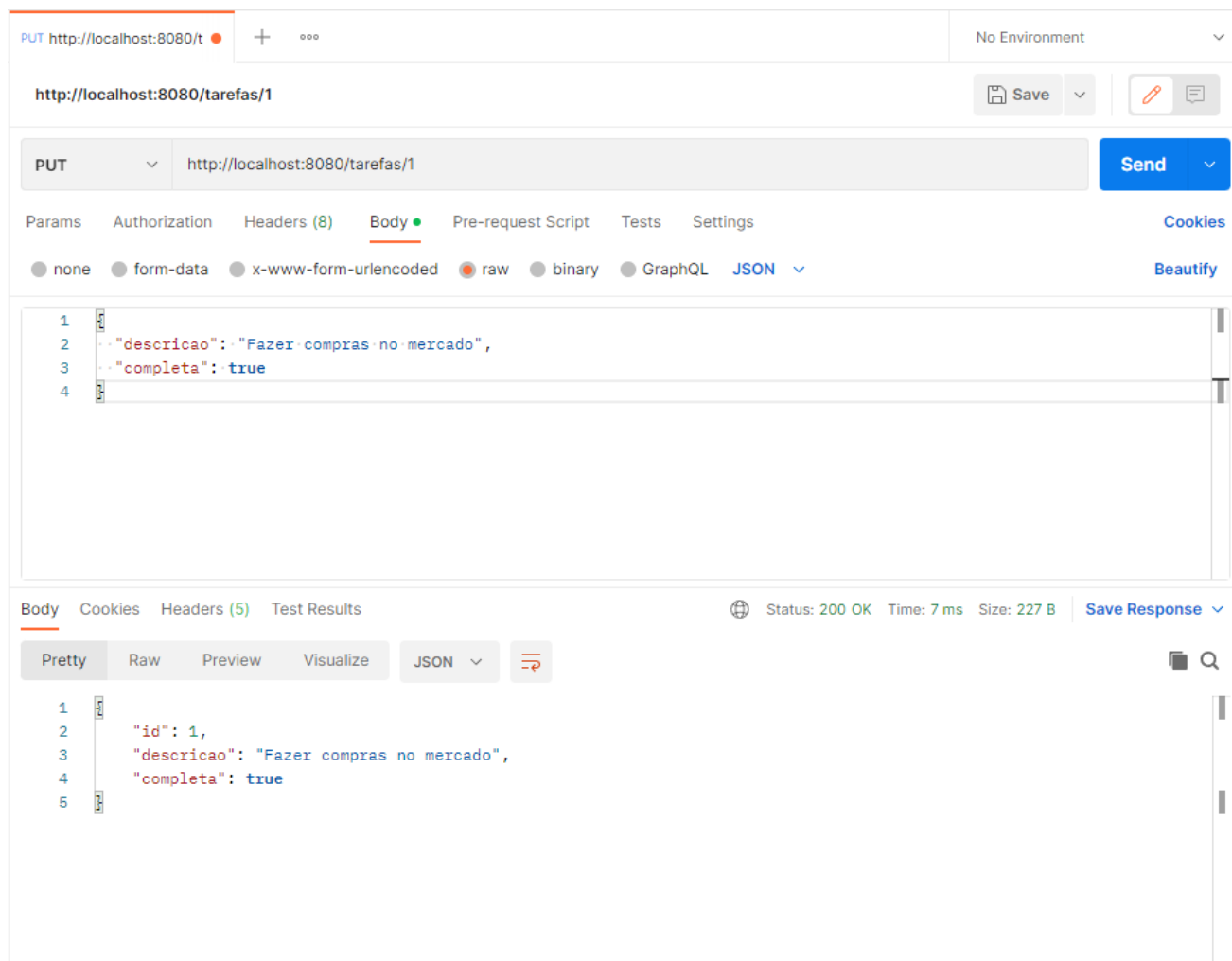


Figura 15 – Requisição PUT no Postman

Fonte: Postman (2023)

DELETE

Para deletar uma tarefa, é enviada uma requisição DELETE com o ID da tarefa na URL.

1. Selecione o método DELETE no menu à esquerda da barra de endereço.
2. Insira a URL da API na barra de endereço com o ID da tarefa:
`http://localhost:8080/tarefas/1`.
3. Clique no botão “Send” para enviar a solicitação.

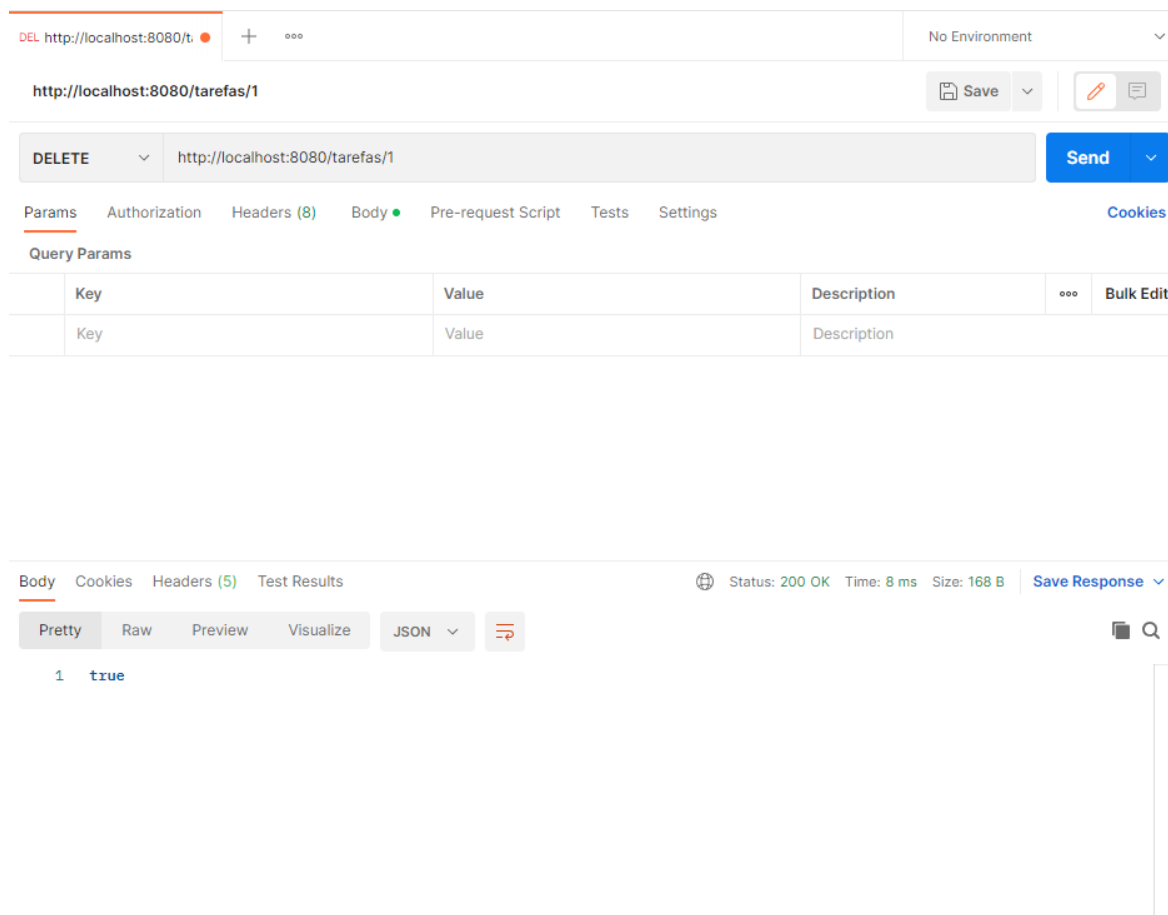


Figura 16 – Requisição DELETE no Postman

Fonte: Postman (2023)

É importante lembrar que esse código não utiliza um banco de dados, portanto, todas as tarefas serão armazenadas na memória durante a execução do programa. Isso significa que, ao reiniciar a aplicação, todas as tarefas serão perdidas.

Crie um projeto Spring RESTful para livros, permitindo cadastro, consulta de livro por id ou todos os livros e exclusão. Para cada livro, considere pelo menos título e autor. Faça testes com Postman.

Integração com sistema web

Agora que você já construiu sua API, já realizou todos os testes com o Postman e tem tudo funcionando, chegou a hora de construir um projeto *web* separado para consumir essa API.

Esse projeto será bem simples, contendo apenas uma página HTML para o usuário interagir e os *scripts* em Javascript para fazer as requisições à API – para facilitar, será usada a biblioteca *jQuery*.



Para continuar, crie uma pasta vazia com o nome de sua escolha e, dentro dela, crie um arquivo **index.html**. O código desse arquivo HTML ficará da seguinte maneira:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Minha Lista de Tarefas</title>
</head>
<body>
  <h1>Minha Lista de Tarefas</h1>
  <h2>Cadastro de tarefas</h2>
  <form id="formCriarTarefa">
    <label for="descricao">Descrição:</label>
    <input type="text" id="descricao" name="descricao">
    <button type="submit">Adicionar</button>
  </form>
  <h2>Tarefas cadastradas</h2>
  <table id="tabelaTarefas">
    <thead>
      <tr>
        <th>ID</th>
        <th>Descrição</th>
        <th>Concluída</th>
        <th>Excluir</th>
      </tr>
    </thead>
    <tbody></tbody>
  </table>

  <script src="https://code.jquery.com/jquery-3.6.4.min.js"></script>
  <script src="script.js"></script>
</body>
</html>
```



Esse é um código HTML básico para criar uma página *web* simples. Ele tem um título "Minha Lista de Tarefas", um formulário para criar novas tarefas e uma área para a lista de tarefas que será preenchida com as tarefas da API. O código HTML também inclui dois *scripts* JavaScript - um para usar a biblioteca jQuery e outro para o *script* personalizado que será criado a seguir.

Agora, crie um arquivo chamado **script.js** e insira o seguinte código:

```
$(document).ready(function () {  
  // Função para carregar as tarefas da API e exibi-las na tel  
  a  
  function carregarTarefas() {  
    $.ajax({  
      url: 'http://localhost:8080/tarefas',  
      method: 'GET',  
      success: function (data) {  
        // Limpa a tabela de tarefas  
        $('#tabelaTarefas tbody').empty();  
        // Adiciona cada tarefa à tabela  
        for (let i = 0; i < data.length; i++) {  
          let tarefa = data[i];  
          let checkbox = $('<input>')  
            .attr('type', 'checkbox')  
            .prop('checked', tarefa.completa)  
            .change(function () {  
              atualizarTarefa($(this).parent().parent  
                ().attr('data-id'), {  
                  descricao: tarefa.descricao,  
                  completa: $(this).prop('checked')  
                });  
            });  
          let id = $('<td>')  
            .text(tarefa.id);  
          let descricao = $('<input>')  
            .attr('type', 'text')  
            .val(tarefa.descricao)  
            .blur(function () {  
              atualizarTarefa($(this).parent().attr('da  
                ta-id'), {  
                  descricao: $(this).val(),  
                  completa: tarefa.completa
```



```

        });
    });
    let concluida = $('<td>')
        .append(checkbox);
    let botaoDeletar = $('<button>')
        .text('Excluir')
        .click(function () {
            deletarTarefa($(this).parent().parent().a
ttr('data-id'));
        });
    let excluir = $('<td>')
        .append(botaoDeletar);
    let tr = $('<tr>')
        .attr('data-id', tarefa.id)
        .append(id)
        .append(descricao)
        .append(concluida)
        .append(excluir);
    $('#tabelaTarefas tbody').append(tr);
}
},
error: function () {
    alert('Não foi possível carregar as tarefas da AP
I. ');
}
});
}

//Função para criar uma nova tarefa na API
function criarTarefa(tarefa) {
    $.ajax({
        url: 'http://localhost:8080/tarefas',
        method: 'POST',
        contentType: 'application/json',
        data: JSON.stringify(tarefa),
        success: function (data) {
            // Limpa o formulário e recarrega a lista de ta
refas

            $('#descricao').val('');
            carregarTarefas();
        },
        error: function () {
            alert('Não foi possível criar a tarefa na AP

```

```
I. ');  
    }  
    });  
}  
// Adiciona uma tarefa ao enviar o formulário  
$('#formCriarTarefa').submit(function (event) {  
    event.preventDefault();  
    let descricao = $('#descricao').val();  
    if (!descricao) {  
        alert('Por favor, preencha a descrição da tarefa');  
        return;  
    }  
    let tarefa = {  
        descricao: descricao,  
        completa: false  
    };  
    criarTarefa(tarefa);  
});  
// Atualiza uma tarefa na API  
function atualizarTarefa(id, tarefa) {  
    $.ajax({  
        url: 'http://localhost:8080/tarefas/' + id,  
        method: 'PUT',  
        contentType: 'application/json',  
        data: JSON.stringify({  
            descricao: tarefa.descricao,  
            completa: tarefa.completa  
        }),  
        success: function (data) {  
            // Recarrega a lista de tarefas  
            alert('Tarefa atualizada na API com sucesso!');  
            carregarTarefas();  
        },  
        error: function () {  
            alert('Não foi possível atualizar a tarefa na API');  
        }  
    });  
}  
// Deleta uma tarefa na API  
function deletarTarefa(id) {  
    $.ajax({
```

```
url: 'http://localhost:8080/tarefas/' + id,
method: 'DELETE',
success: function (data) {
    alert('Tarefa removida na API com sucesso!');
    // Recarrega a lista de tarefas
    carregarTarefas();
},
error: function () {
    alert('Não foi possível deletar a tarefa na AP
I. ');
}
});
}
// Carrega as tarefas ao abrir a página
carregarTarefas();
});
```

Esse código já está comentado explicando tudo que foi implementado, mas será feito um resumo geral. O arquivo JavaScript contém várias funções que são executadas quando a página é carregada ou quando o usuário interage com a página.

- `carregarTarefas()`: é responsável por fazer uma requisição GET para a API e exibir as tarefas na lista. Essa função é chamada tanto ao carregar a página quanto após a criação ou exclusão de uma tarefa.
- `criarTarefa()`: é responsável por criar uma nova tarefa na API e limpar o formulário após a submissão. Essa função é chamada quando o usuário clica no botão de adicionar tarefa.
- `deletarTarefa()`: é responsável por excluir uma tarefa existente na API e atualizar a lista de tarefas após a exclusão. Essa função é chamada quando o usuário clica no botão de excluir uma tarefa.
- `atualizarTarefa()`: é responsável por atualizar uma tarefa existente na API e atualizar a lista de tarefas após a atualização. Essa função é chamada quando o usuário marca ou desmarca a caixa de seleção de uma tarefa.

Ao final do arquivo JavaScript, há um trecho de código que associa a função "criarTarefa" ao evento "submit" do formulário e chama a função "carregarTarefas" para exibir as tarefas na lista ao carregar a página.

Agora, com a API rodando no NetBeans, abra o arquivo **index.html** e tente cadastrar uma tarefa.

Minha Lista de Tarefas

Cadastro de tarefas

Descrição:

Tarefas cadastradas

ID	Descrição	Concluída	Excluir
----	-----------	-----------	---------

Figura 17 – Figura Minha lista de Tarefas

Fonte: Senac EAD (2023)

Nesse exemplo, foi feito o cadastro da tarefa “Teste”. Ao clicar no botão de adicionar, a requisição POST é feita para a API. Após concluir o cadastro, é realizada a requisição GET para solicitar a lista de tarefas completa e exibi-la na tabela. Perceba que a descrição da tarefa é apresentada dentro de um elemento *input* do HTML. Isso é proposital para ter uma forma simples de editar o texto caso seja cometido algum erro de digitação.

Foram testadas as requisições POST e GET da API. Agora será testada a requisição PUT alterando a descrição da tarefa “Teste” para “Teste de API”.

Minha Lista de Tarefas

Cadastro de tarefas

Descrição: Adicionar

Tarefas cadastradas

ID	Descrição	Concluída	Excluir
1	<input type="text" value="Teste"/>	<input type="checkbox"/>	<button>Excluir</button>

Figura 18 – Figura Minha lista de tarefas

Fonte: Senac EAD (2023)

Como é possível ver, o texto da tarefa será atualizado e a requisição PUT realizada sempre que se desfocar a *input* da descrição.

Falando em requisição PUT, outra atualização para testar é o *status* da tarefa que é representado por uma *checkbox*. Ao marcar essa *checkbox*, a tarefa fica marcada como concluída. Para isso, basta clicar no elemento *checkbox* da tarefa.

Minha Lista de Tarefas

Cadastro de tarefas

Descrição: Adicionar

Tarefas cadastradas

ID	Descrição	Concluída	Excluir
1	<input type="text" value="Teste de API"/>	<input type="checkbox"/>	<button>Excluir</button>

Figura 19 – Figura Minha Lista de Tarefas

Fonte: Senac EAD (2023)

Perceba que tarefas marcadas como concluídas podem ser desmarcadas, basta clicar sobre o elemento *checkbox* novamente. Por fim, será testada a exclusão de tarefas. Para remover uma tarefa cadastrada, basta clicar no botão “Excluir”.

Minha Lista de Tarefas



Cadastro de tarefas

Descrição:

Tarefas cadastradas

ID	Descrição	Concluída	Excluir
1	<input type="text" value="Teste de API"/>	<input type="checkbox"/>	<input type="button" value="Excluir"/>

Figura 20 – Minha Lista de Tarefas

Fonte: Senac EAD (2023)

Ao excluir uma tarefa, a requisição DELETE é enviada para a API e o item é removido da lista de tarefas. Após concluir a remoção, é realizada a requisição GET novamente para solicitar a lista de tarefas atualizada.

Implemente uma página para o projeto RESTful de livros do desafio anterior. Use AJAX para solicitar as operações de inclusão, consulta e exclusão. Aplique outros elementos que considerar necessários, como campos de entrada e tabelas.

Encerramento

Neste conteúdo, você aprendeu como instalar e configurar as ferramentas do ambiente de desenvolvimento para construir e testar APIs em Java e sua integração com um sistema *web*. Como desenvolvedor, você deve estar comprometido em continuar aprimorando os *softwares* para atender às necessidades dos usuários. A busca pela melhoria contínua deve ser constante, pois sempre há espaço para otimizar e evoluir os *softwares*. Portanto, continue seus estudos para alcançar esse objetivo.