



[Visão Geral](#) [Introdução à Reatividade](#) [Gerenciador de Tarefas](#) [Entidade](#) [Persistência](#) [Controlador](#) [Executando](#)

[Verificação de Aprendizado](#)

VISÃO GERAL

Nossa primeira aplicação foi desenvolvida como um bloco monolítico. Dessa forma, a interface gráfica e o restante da aplicação estão integrados em um único arquivo de deploy. Uma aplicação monolítica pode ser útil em várias situações, mas existem casos que você precisa integrar uma aplicação com outra existente. Nesses casos, você precisa de uma API.

Nesta aula vamos construir uma nova aplicação que funciona como uma API, recebendo dados usando a arquitetura REST. Para fazer isso, vamos usar uma nova tecnologia do Spring - o Spring WebFlux.

O [Spring WebFlux](#) é um projeto que permite criar aplicações reativas. Isso significa que a requisição do cliente não fica bloqueada enquanto os processos internos são executados pelo aplicativo no sistema operacional. O resultado disso é uma aplicação com melhor desempenho, retornando rapidamente as requisições do cliente.

Cada seção introduz uma parte da aplicação. Mas, se você tiver pressa, talvez prefira começar pelo vídeo abaixo.

INTRODUÇÃO À REATIVIDADE

O código abaixo mostra um método em uma classe anotada com `@Controller`. Esse método é executado quando a URL `/excluir` é acessada (linha 59). Quando o método recebe a solicitação, o processador escalona uma *thread* em um processo para tratar essa solicitação. Por padrão, as solicitações são *síncronas*. Isso significa que a thread bloqueia a continuidade do método até que a operação solicitada seja completada. A linha 64 exemplifica isso muito bem.

```
59     @GetMapping("/excluir")
60     public String excluir(
61         @RequestParam String nome,
62         @RequestParam String estado) {
63
64         var cidadeEstadoEncontrada = repository.findByNomeAndEstado(nome, estado);
65
66         cidadeEstadoEncontrada.ifPresent(repository::delete);
67
68         return "redirect:/";
69     }
```

Na linha 64, o código verifica a existência de dados em um banco de dados. Isso envolve: (i) recuperar o objeto responsável pela verificação; (ii) acessar o banco de dados; (iii) realizar a consulta; (iv) mapear os dados recebidos em uma entidade; e (v) retornar o resultado para o método. Enquanto essa verificação é feita, o método fica bloqueado.

Na prática, isso significa que o usuário que fez a solicitação deve aguardar o processo ser completado para só então receber o retorno. O processamento nesse exemplo pode ser rápido, mas esse nem sempre é o caso. Além disso, nesse exemplo, o usuário não depende do retorno, pois ele solicitou uma exclusão. Nesse caso, o usuário só precisa que os dados sejam excluídos. Ainda assim, ele precisa ficar esperando o processo terminar.



Esse é um exemplo ideal para o uso de uma operação *assíncrona*. Em uma operação assíncrona, a solicitação de verificação feita na linha 64 seria enviada ao banco de dados e, imediatamente, o restante do processo no método `excluir()` seria liberado, retornando uma resposta imediata ao usuário.

Note que isso não significa que a linha 64 seria executada mais rapidamente, mas sim, que o processador iria cuidar da execução dessa tarefa em algum momento. Isso torna seu aplicativo mais *responsivo*. Isso é similar ao conceito de `Promise` em JavaScript, para aqueles familiarizados com a linguagem JS.

Esse não é um recurso novo em Java, mas gerenciar concorrência costumava ser mais complicado antes da adição de `CompletableFuture`, no Java 8. Em Java, o termo *reatividade* está relacionado com a reação a eventos. Esses eventos acontecem em nível de processamento, em vez de aplicações, como estamos acostumados a ver em aplicações distribuídas.

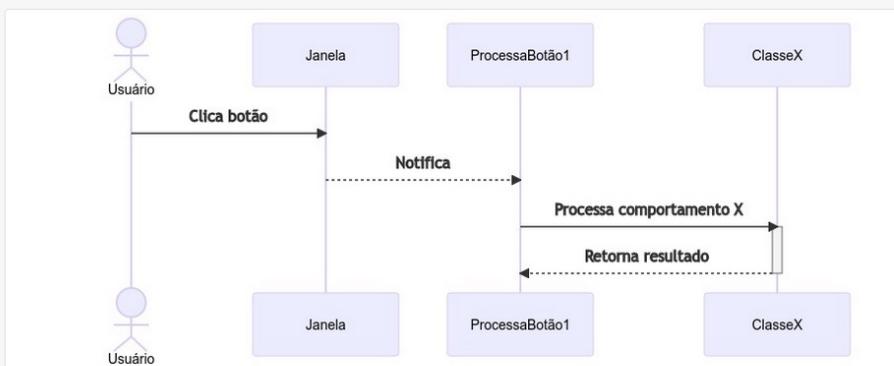
Até agora, temos usado o `Spring MVC` para desenvolver aplicações. O Spring MVC é uma das tecnologias fundamentais do `Spring Framework`. O Spring MVC permite desenvolver aplicações Web baseadas na `API Servlet`. A API Servlet é uma especificação de como tratar solicitações em um container Web. Usar a API Servlet tal como é implementada por containers como `Tomcat` ou servidores de aplicação como `Glassfish` não costumava ser uma tarefa trivial.

O Spring MVC facilita muito a vida do desenvolvedor fornecendo mecanismos para implementar aplicações Web de forma rápida e simples, como temos visto até agora nesse curso. Contudo, o Spring MVC não implementa reatividade. Para isso, precisamos de outro projeto - o `Spring WebFlux`.

O Spring WebFlux usa uma estrutura muito similar ao Spring MVC para o desenvolvimento de aplicações Web. A similaridade facilita a adoção do Spring WebFlux. Contudo, ele introduz diferenças significativas em como o processamento de uma solicitação é realizada. Essas diferenças são necessárias para permitir um processamento *não bloqueante*. A principal diferença está no uso do `padrão publisher-subscriber` (editor-assinante).



O padrão `publisher-subscriber` é muito conhecido em computação, e pode ser melhor introduzido por meio do `projeto observer`. Um exemplo clássico do uso desse padrão é uma janela de uma aplicação qualquer (Figura abaixo). Toda vez que um botão é clicado, ele dispara um evento (*publisher*). Esse evento é publicado para os assinantes (*subscribers*). Outra forma de representar a mesma coisa é dizer que *a janela notifica os responsáveis pelo comportamento do botão*. Esses assinantes são classes responsáveis por processar determinados comportamentos do botão. Observe que a notificação é feita de forma assíncrona. Assim, a janela não fica bloqueada enquanto o comportamento do botão é processado.



GERENCIADOR DE TAREFAS

Um gerenciador de tarefas (TODO) é um aplicativo que permite controlar as tarefas que devem ser executadas para atingir algum propósito. Nosso gerenciador de tarefas deve permitir criar e apagar tarefas, além de marcar se uma tarefa foi concluída. Além disso, esse gerenciador de tarefas deve funcionar como uma API.

API é o acrônimo de *Application Programming Interface*. Uma API funciona como a porta de entrada de uma aplicação. Originalmente, APIs foram criadas para permitir a integração entre interface gráfica e o restante da aplicação [1]. Atualmente, APIs são fundamentais para permitir a integração de aplicações em sistemas orientados a serviços. Frequentemente, APIs são implementadas por frameworks ou bibliotecas. APIs podem ser desenvolvidas usando protocolos, tecnologias e arquiteturas diversas, como RMI, WebSocket, ou REST.

Neste curso, vamos desenvolver o gerenciador de tarefas usando uma API REST. APIs REST fazem requisições usando os métodos do protocolo HTTP para identificar o tipo de operação que será executada. Por exemplo, listar um conjunto de tarefas usa o método **GET**, enquanto a criação de uma tarefa usa o método **POST**.

Uma URL identifica o recurso sobre o qual a operação será realizada. Por exemplo, podemos usar a URL `/todos` para identificar a tarefa, e podemos usar a URL `/user` para identificar o usuário. Se quisermos identificar um recurso específico, podemos adicionar um caminho à URL. Por exemplo, `/todos/a8sd7f687df` pode identificar uma tarefa em particular. Por fim, é possível enviar um corpo junto à requisição. Normalmente, isso é feito especificando um objeto com o formato **JSON**.

Quer saber mais sobre APIs REST? Esse **site** tem uma coleção de links úteis sobre o assunto.

Como uma API, o gerenciador de tarefas não tem interface gráfica. Por isso, para realizar as operações vamos usar um cliente HTTP. Um cliente HTTP é um aplicativo que envia solicitações HTTP para uma determinada URL, assim como seu navegador de Internet. Se você usa Linux, já deve ter o **cURL** instalado - essa é uma ferramenta muito popular no Linux. O **Postman** também é uma ferramenta bastante conhecida e, possivelmente, a mais simples de usar. Se você tem pouca experiência usando clientes HTTP, eu sugiro que use o Postman. Para usuários Mac, o **HTTPie** é um excelente cliente HTTP. Contudo, você tem liberdade para usar o cliente que achar melhor.

O gerenciador de tarefas será desenvolvido como uma API reativa. Para isso, vamos usar o Spring Webflux e o **MongoDB**. O MongoDB é um banco de dados não-SQL (NoSQL) que armazena dados como documentos. No contexto do MongoDB, um documento é como um arquivo em formato **JSON**. Neste projeto, vamos usar o MongoDB porque, no momento em que esse texto está sendo escrito, o Spring Data para JPA ainda não oferece suporte para reatividade. Você não precisa conhecer de MongoDB, mas vai precisar ter ele instalado para executar este projeto. Para isso, basta fazer o [download direto do site oficial](#) e instalar de acordo com seu sistema operacional. Você também pode usar uma [imagem Docker](#) se preferir.

Se você nunca usou MongoDB, não se preocupe. Toda a interação com o MongoDB será feita por dentro da aplicação usando o Spring Data, que você já conhece.

Precisamos criar um novo projeto usando o **Spring Initializr**, assim como foi feito anteriormente. Adicione as dependências *Spring Reactive Web* (Webflux) e *Spring Data Reactive MongoDB*. Faça o download do projeto e abra ele na sua IDE preferida. Não esqueça de iniciar o MongoDB, ou o container Docker antes de iniciar o projeto.

Você também pode clonar o projeto vazio já com as dependências direto do **nossa repositório no Github**, na branch **semana06-10-projeto-vazio**.

[1] ALONSO , G.; CASATI, F.; KUNO, H.; MACHIRAJU, V. Web Services: Concepts, Architectures and Applications. Berlin: Springer, 2004.

ENTIDADE

Vamos usar uma abordagem *bottom-up* para desenvolver esse projeto, começando pela entidade persistente. Para isso, criamos o arquivo `Todo.java` no pacote `demo`. Esse arquivo representa uma *tarefa* no nosso gerenciador de tarefas.

Em vez de uma classe, criamos um `record`. Um java `record` é uma entidade imutável para a transferência de dados entre objetos. `Record` é uma adição da [versão 16 do Java](#). Um detalhe relevante é que um `record` é imutável. Dessa forma, uma vez atribuídos valores, eles não podem ser alterados. Outras características do `record` tornam ele [invíável](#) para uso persistente com JPA. Contudo, ele funciona bem com MongoDB.

Se Java `record` ainda é algo novo pra você, você pode [conhecer mais sobre o assunto nesse link](#).

Diferente de uma classe, um `record` define os atributos logo na sua declaração. Nesse projeto, vamos considerar que uma *tarefa* tem os seguintes atributos:

- `id` que identifica a tarefa como única em uma lista de tarefas;
- `titulo` que define o nome da tarefa;
- `descricao` que descreve detalhes da tarefa; e,
- `feito` que marca a tarefa como concluída ou não.

Para garantir que toda tarefa tem um título minimamente descritivo, vamos adicionar uma validação. Colocamos a validação no construtor do `record`. Desse modo, uma exceção será levantada caso o título seja nulo ou menor que três caracteres. O código na figura abaixo mostra o resultado final do `Todo.java`.

```
1 package com.example.demo;
2
3 public record Todo(String id, String titulo, String descricao, boolean feito) {
4
5     public Todo {
6         if (titulo == null || titulo.length() < 3) {
7             throw new IllegalArgumentException("Um título maior que 3 caracteres é necessário.");
8         }
9     }
10 }
11 }
```

A linha 3 mostra a definição do `record`, seus atributos, e o tipo de cada atributo. As linhas 5 a 9 definem o construtor. A validação entre as linhas 6 e 8 garante um título minimamente válido para a *tarefa*.

Notou a ausência de qualquer anotação identificando essa classe como entidade persistente? Não, não está errado. O Spring Data reconhece o uso da entidade mesmo sem qualquer anotação. Mas atenção: isso não funciona para entidades JPA!

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana06-20-entidade`.

PERSISTÊNCIA

Nossa API do gerenciador de tarefas usa o [MongoDB](#) para salvar as tarefas de forma persistente. O MongoDB é um banco de dados não relacional (NoSQL), que armazena dados como *documentos*. Dessa forma, a estrutura dos dados pode variar a cada *documento*. Para o MongoDB, um documento é como um arquivo em formato [JSON](#).

As *tarefas* no nosso gerenciador de tarefas tem uma estrutura bem definida. Poderíamos usar um banco de dados relacional sem qualquer problema. Porém, estamos usando o Spring WebFlux para desenvolver uma API não bloqueante. No momento em que essa aula está sendo preparada, o Spring Data JPA ainda não oferece suporte para o WebFlux. Porém, você ainda pode usar bancos relacionais com WebFlux adotando o [Spring Data R2DBC](#).

Escolher o tipo mais adequado de banco de dados pode ser bem complicado. Se quiser entender melhor as diferenças entre SQL e as diversas variantes de NoSQL, esses links podem ajudar:

- [Understand data store models](#)
- [From SQL to NoSQL](#)
- [NoSQL Databases: a Survey and Decision Guidance](#)

Felizmente, criar a parte de persistência com o Spring WebFlux não é diferente do que já fizemos com o Spring MVC. Isso porque o Spring Data oferece uma interface para repositórios reativos (`org.springframework.data.repository.reactive.ReactiveCrudRepository`). Tanto a interface para repositórios *reativos* quanto para *não reativos* é baseada na mesma interface (`org.springframework.data.repository.Repository`). Dessa forma, a implementação é exatamente a mesma. A figura abaixo mostra o código da interface `TodoRepository`. Essa interface é responsável pelas operações de gerenciamento dos dados de *tarefas* (representada pelo Java record `Todo`).

```
1 package com.example.demo;
2
3 import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
4 import org.springframework.stereotype.Repository;
5
6 import reactor.core.publisher.Flux;
7
8 @Repository
9 public interface TodoRepository
10    extends ReactiveMongoRepository<Todo, String> {
11
12    Flux<Todo> findByFeito(boolean feito);
13
14 }
```

A linha 08 identifica essa interface como responsável por gerenciar os dados da aplicação. A linha 10 herda comportamentos da interface `org.springframework.data.mongodb.repository.ReactiveMongoRepository`. Os comportamentos herdados correspondem às operações básicas de um CRUD, como ler e salvar. A interface `org.springframework.data.mongodb.repository.ReactiveMongoRepository` é uma especialização da interface `org.springframework.data.repository.reactive.ReactiveCrudRepository` para o MongoDB.

Por que `CidadeRepository` não tem a anotação `org.springframework.stereotype.Repository` e `TodoRepository` tem? Essa anotação é **indicada para identificar** uma interface como responsável por gerenciar dados persistentes. Na prática, ela é indiferente para a execução da aplicação.

Note que a interface é do tipo genérico – assim como no exemplo com JPA. Dessa forma, precisamos identificar a entidade persistente (`Todo`) e o tipo do identificador único (`String`). Enquanto bancos SQL costumam usar identificadores com tipos numéricos (ex: `int`, `long`), bancos NoSQL podem usar diferentes tipos de dados.

Ao herdar os comportamentos da interface `org.springframework.data.mongodb.repository.ReactiveMongoRepository`, nossa interface já **recebe automaticamente** métodos para criar, alterar, excluir e consultar os dados, entre outros. Contudo, nesse projeto criamos um método adicional usando **palavras-chave de consulta** (linha 12). Isso nos permite gerar uma lista de *tarefas* de acordo com seu status – feita ou não.

Observe que o retorno do método `findByFeito()` é do tipo `reactor.core.publisher.Flux`, e não um `java.util.List`. Isso porque `reactor.core.publisher.Flux` é o tipo que **controla uma sequência de dados** variando de zero a muitos.

Pode parecer um pouco confuso no início, mas é importante lembrar que o WebFlux está baseado no padrão *publisher-subscriber*. Dessa forma, o retorno dos dados não é imediato, como acontece em um método não reativo. O objeto é populado à medida que os dados são emitidos pelo *publisher* – nesse caso, o banco de dados. Um `java.util.List` não é apropriado para esse tipo de operação não bloqueante. Por outro lado, o `reactor.core.publisher.Flux` é projetado para isso.

Posso usar um retorno do tipo `reactor.core.publisher.Flux` em um código não reativo? Não. O `reactor.core.publisher.Flux` não é apenas um container para dados, como `java.util.List`. Tem muito mais coisa acontecendo para tornar o código reativo.

Assim como foi feito no projeto anterior, o código que implementa a interface `TodoRepository` é criado automaticamente pelo Spring Data quando o projeto é compilado. Dessa forma, não precisamos nos preocupar com mais nada – nem mesmo com a configuração do MongoDB. **Apenas garanta que o MongoDB estará disponível na porta padrão no seu computador quando executar o projeto.**

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana06-31-persistencia`.

CONTROLADOR

A última parte desse projeto é o controlador. Seguindo o modelo MVC, o controlador tem a responsabilidade de gerenciar a interação entre o usuário e a API. Nesse projeto, nosso controlador age como um *delegador* de solicitações. Dessa forma, os métodos do controlador apenas expoem os endereços da API e repassam as solicitações de e para a interface de persistência.

Usar controladores para delegar solicitações é um padrão muito comum em aplicações CRUD. Mas, você sabia que tem uma forma mais simples de fazer isso? O **Spring Data REST** é um projeto do Spring Data que permite criar APIs diretamente da interface do repositório. Desse modo, não é necessário criar um controlador para delegar solicitações.

O código na figura a seguir mostra a classe do controlador, `TodoRestController`, na sua forma final.

```
14  @RestController
15  public class TodoRestController {
16
17      private final TodoRepository repository;
18
19      public TodoRestController(final TodoRepository repository) {
20          this.repository = repository;
21      }
22
23      @GetMapping("/todos")
24      public Flux<Todo> lerTodos() {
25          return repository.findAll();
26      }
27
28      @GetMapping("/todos/{feito}")
29      public Flux<Todo> lerByFeito(@PathVariable boolean feito) {
30          return repository.findByFeito(feito);
31      }
32
33      @PostMapping("/todos")
34      public Mono<Todo> criar(@RequestBody Todo todo) {
35          return repository.save(todo);
36      }
37
38      @DeleteMapping("/todos/{id}")
39      public Mono<Void> deletar(@PathVariable String id) {
40          return repository.deleteById(id);
41      }
42
43      @PutMapping("/todos/{id}")
44      public Mono<Todo> atualizar(@PathVariable String id) {
45
46          return repository
47              .findById(id)
48              .map(todoAtual -> new Todo(id,
49                  todoAtual.titulo(),
50                  todoAtual.descricao(),
51                  !todoAtual.feito()))
52              .flatMap(repository::save)
53              .onTerminateDetach();
54      }
55  }
```

A classe `TodoRestController` tem cinco métodos. Na linha 24, o método `lerTodos()` é responsável por listar as *tarefas* existentes na base de dados. Observe o tipo de retorno - é o mesmo retorno que usamos no `TodoRepository`. Observe que o corpo do método apenas repassa a solicitação para uma instância de `TodoRepository`, responsável por acessar a base de dados. O resultado de `TodoRepository.findAll()` é retornado diretamente para o solicitante.

O segundo método, na linha 29, é uma variação do primeiro. Enquanto `lerTodos()` retorna todas as *tarefas* na base de dados, `lerByFeito(boolean)` retorna apenas as tarefas que tiverem o atributo `feito` com o valor definido no parâmetro do método - `true` ou `false`. O corpo do método `lerByFeito(boolean)` apenas repassa a solicitação, chamando o método `TodoRepository.findByFeito(boolean)`, que criamos anteriormente.

O terceiro método, na linha 34, é responsável por salvar uma tarefa na base de dados. Note que o método `criar(Todo)` tem um retorno diferente dos dois métodos anteriores. Enquanto os métodos anteriores poderiam retornar mais de uma *tarefa*, o método `criar(Todo)` retorna apenas uma *tarefa*. O tipo `reactor.core.publisher.Mono` funciona da mesma forma que o `reactor.core.publisher.Flux`. Porém, o `reactor.core.publisher.Mono` suporta carregar apenas um objeto. Nesse caso, o retorno corresponde à *tarefa* que foi criada.

Os dois métodos seguintes, `criar(Todo)` (linha 34) e `deletar(String)` (linha 39), funcionam exatamente como os métodos anteriores - repassando a solicitação para o método correspondente no `TodoRepository`. Contudo, o método `criar(Todo)` recebe um objeto do tipo `Todo` como parâmetro - representando a *tarefa* que será criada. Já o método `deletar(String)` recebe uma `String` como parâmetro - representando o identificador da *tarefa* que será removida. Outra diferença é que o método `deletar(String)` não retorna uma *tarefa*, mas sim um retorno vazio - representado pelo objeto `Void`. Isso faz sentido porque a *tarefa* acabou de ser eliminada e, portanto, não existem dados para retornar.

Você pode **personalizar o tipo de retorno** se preferir, por exemplo, adicionando mais informações sobre a solicitação.

O último método, `atualizar(String)` (linha 44), alterna o estado da *tarefa* entre `feito` e `não feito`. O corpo do método recupera a *tarefa* correspondente ao identificador informado (linha 47). Em seguida, uma nova *tarefa* (`Todo`) é criada com os mesmos valores da tarefa existente (linhas 48 a 51). Porém, o valor do atributo `feito` é alternado (linha 51). Depois, a tarefa alterada é salva (linha 52) e os assinantes (*subscribers*) dessa operação são liberados (linha 53).

A classe tem ainda um construtor que recebe `TodoRepository` como argumento. Esse construtor é usado pelo Spring para injetar a dependência no atributo `repository`. Assim, como vimos anteriormente, a injeção de dependência é uma característica marcante do Spring. Note que não precisamos instanciar `TodoRepository` em nenhum momento.

Nesse ponto, você deve estar se perguntando: "E quanto às anotações?". Sem as anotações, nosso código é apenas mais um código Java. São as anotações que fazem o framework entender que esse código deve ser gerenciado pelo Spring. As anotações usadas aqui **não são** diferentes das anotações usadas pelo Spring MVC. Por isso, esse mesmo controlador poderia ser usado tanto para uma API com Spring MVC quanto para uma API com WebFlux. O detalhe, é claro, é o tipo de retorno dos métodos que precisaria ser adequado.

Agora, vamos entender o papel de cada anotação nessa classe:

- `org.springframework.web.bind.annotation.RestController` (linha 14) identifica essa classe como um controlador que atende solicitações do tipo REST.
- `org.springframework.web.bind.annotation.GetMapping` (linhas 23 e 28) associa o método subsequente com uma solicitação HTTP do tipo GET para a URL informada como parâmetro **na anotação**. Observe que na linha 28, a URL tem também um parâmetro variável (`feito`), que deve ser informado como parte da URL na solicitação.
- `org.springframework.web.bind.annotation.PostMapping` (linha 33),
`org.springframework.web.bind.annotation.DeleteMapping` (linha 38) e
`org.springframework.web.bind.annotation.PutMapping` (linha 43) associam o método subsequente com uma solicitação HTTP do tipo POST, DELETE e PUT, respectivamente, para as URLs informadas como parâmetro nas anotações.
- `org.springframework.web.bind.annotation.PathVariable` (linhas 29, 39 e 44) associa o parâmetro da URL com o parâmetro do método atual.
- `org.springframework.web.bind.annotation.RequestBody` (linha 34) associa um objeto JSON enviado no corpo da solicitação REST com o objeto informado no parâmetro do método atual (`Todo`).

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana06-40-controlador`.

EXECUTANDO O PROJETO

Nesse ponto, o código necessário para nossa API de gerenciamento de tarefas está pronto. Esse é um projeto baseado no Spring Boot, assim como nosso projeto anterior. Por isso, para executar o projeto basta abrir o terminal na pasta do projeto e digitar: `mvn clean spring-boot:run`. Contudo, existe alguns detalhes que precisamos nos atentar.

Primeiro, esse projeto usa o MongoDB. Diferente do que fizemos anteriormente, não temo um MongoDB embutido no projeto. Por isso, antes de executar esse projeto, você precisa garantir que você tem o MongoDB rodando localmente no seu computador. Se você nunca teve contato com o MongoDB, minha sugestão é que você use uma imagem Docker, conforme orientado na aba "Gerenciador de Tarefas". Se quiser testar seu ambiente para garantir que está tudo funcionando, você pode rodar os testes unitários disponíveis no [projeto no Github](#). Basta clonar o projeto localmente, abrir a pasta do projeto no terminal, e executar: `mvn test`

Segundo, esse projeto **não tem** interface gráfica. Isso é esperado, uma vez que esse projeto consiste de uma API. Dessa forma, para ver as solicitações funcionando, você precisa de um cliente HTTP. Se precisar de sugestões, veja as informações deixadas na aba "Gerenciador de Tarefas".

Finalmente, execute o projeto usando `mvn clean spring-boot:run` e use seu cliente HTTP para fazer solicitações.

```
sh-3.2$ http get localhost:8080/todos
HTTP/1.1 200 OK
Content-Type: application/json
transfer-encoding: chunked

□

sh-3.2$ http post localhost:8080/todos id=9a87sdf98 titulo=Hello descricao=World feito=false
HTTP/1.1 200 OK
Content-Length: 69
Content-Type: application/json

{
  "descricao": "World",
  "feito": false,
  "id": "9a87sdf98",
  "titulo": "Hello"
}

sh-3.2$ http post localhost:8080/todos id=9a87sdf98 titulo=Hello descricao=World feito=false
```