

# SPRING BOOT

## Microserviços na prática

---

**1ª EDIÇÃO**

**CAIO COSTA**

# Spring Boot

Microserviços na prática

CAIO COSTA

Copyright © 2021 Caio Costa  
Todos os direitos reservados.  
ISBN: 9798710953402  
DEDICATÓRIA

Gostaria de dedicar esse livro aos meus pais, minha esposa Lorraine e a todas as pessoas que acreditaram e acreditam em mim a cada nova ideia, por mais maluca que possa parecer.

## SUMÁRIO

### [SUMÁRIO](#)

### [PREFÁCIO](#)

### [Capítulo 1](#)

#### [1.1 Pré-Requisitos](#)

#### [1.2 Explicando a Stack](#)

### [Capítulo 2](#)

#### [2.1 Criação do projeto](#)

### [Capítulo 3](#)

#### [3.1 Eclipse IDE](#)

#### [3.2 H2 Database](#)

### [Capítulo 4](#)

#### [4.1 Arquitetura do Projeto](#)

### [Capítulo 5](#)

#### [5.1 Implementação da Entidade](#)

#### [5.2 Implementação do Repositório](#)

#### [5.3 Implementação da Service](#)

#### [5.4 Implementação da Controller](#)

[5.5 Swagger](#)

[Capítulo 6](#)

[6.1 Testes](#)

[6.2 Testes unitários com JUnit e Mockito](#)

[6.3 Testes de Integração com Rest Assured](#)

[Capítulo 7](#)

[7.1 Docker - Preparando a Aplicação](#)

[7.2 Docker - Dockerfile](#)

[Capítulo 8](#)

[8.1 Conclusão e considerações finais](#)

## PREFÁCIO

Este livro tem o intuito de ser um guia prático e objetivo de como construir um Microserviço **RESTful** de forma rápida e eficiente, tentando sempre focar na prática, de uma forma que não seja apenas teoria e processos complexos, para que seja possível ser aplicada no dia-a-dia de trabalho, em qualquer empresa de desenvolvimento de software que queira adotar essa forma de trabalhar.

Como experiência própria, já tendo passado por diversos projetos e diferentes realidades no mundo de desenvolvimento, tento aqui demonstrar o que seria para mim, uma das formas mais ágeis e produtivas de se produzir um Microserviço utilizando Java, tentando seguir princípios **S.O.L.I.D**, que pode ser utilizado em qualquer tipo de projeto, mas, que é mais comumente utilizado no mundo corporativo para sistemas de pequeno, médio e grande porte. Isso tudo, sem esquecer da qualidade e tentando sempre promover a diminuição da complexidade em arquiteturas que não se fazem necessárias e focar os esforços do desenvolvedor no

que realmente importa para o resultado final da aplicação, que é cumprir seus requisitos de negócio de forma assertiva.

Esse é um guia prático, por isso, recomendo fortemente que se faça a leitura como forma de guia, utilizando um computador e exercitando o que é mostrado a cada passo. Lembrando que tudo que for mostrado aqui é apenas uma sugestão de arquitetura e não uma regra, por isso, tome a liberdade de fazer os ajustes necessários para melhor aplicar isso a sua própria realidade.



# CAPÍTULO 1

## 1.1 PRÉ-REQUISITOS

Antes de se começar a desenvolver o assunto, precisamos garantir que temos nossos ambientes devidamente configurados. A seguir, vou mostrar a lista de tudo que utilizaremos e em seguida, realizar uma breve explicação sobre o motivo de cada uma dos componentes dessa Stack. Apenas ressaltando, tentei tornar esse projeto o mais enxuto possível, trazendo apenas componentes que tenham um real contributo para tornar o trabalho do desenvolvedor mais ágil.

Lista dos principais componentes:

- Java 11 (JDK)
- Eclipse IDE
- Spring Boot
- Spring Actuator
- H2 Database
- Querydsl
- JUnit 5
- Maven
- Lombok
- Swagger
- Docker
- Git

Outras ferramentas que serão utilizadas no decorrer do projeto:

- Postman



Como este é um livro voltado para desenvolvedores, estou tendo como premissa que todos sabem como configurar o ambiente, mas caso tenha alguma dúvida, sempre pode utilizar o bom e velho Google, existem centenas de tutoriais sobre como instalar e configurar todas essas ferramentas.

## 1.2 EXPLICANDO A STACK

Essa vai ser uma breve explicação sobre a Stack escolhida, apenas para garantir o nivelamento de conhecimento de todos os leitores no caso de alguma componente ser novidade.

**Java 11 (JDK):** Se você está lendo esse livro e tem experiência, eu não creio que seja necessário justificar o motivo de ter a JDK instalada e configurada, porém, como disse antes, preciso garantir um nivelamento. Como o nome já diz, a JDK é o Java Development Kit, ou seja, precisamos dela sempre que queremos criar, compilar e executar um código Java. Esse é o componente core de qualquer aplicação escrita em Java.

**Eclipse IDE:** Como já disse antes, você leitor, tem total liberdade para fazer escolhas e adaptações necessárias para que se sinta mais confortável no andamento do projeto, no meu caso, depois de já trabalhar há anos com o Eclipse, pra mim é a escolha óbvia, e nesse projeto é a IDE que vou utilizar para realizar as demonstrações, porém, sinta-se livre para optar por qualquer outra IDE.

**Spring Boot:** Desde que o Spring surgiu, acredito que o intuito dos criadores sempre foi facilitar nossa vida e com o Spring Boot não é diferente, ele nos entrega uma forma simples e rápida de se criar uma aplicação Standalone, visto que traz consigo “embutido” um Apache TomCat, um WebServer muito bem difundido no mercado há muitos anos.

**Spring Actuator:** Uma framework da família Spring que veio para nos auxiliar a monitorar nossos Microserviços. Com essa biblioteca é possível coletar uma série de métricas, entender o tráfego e a saúde do nosso Microserviço.

**H2 Database:** Apenas utilizada no projeto a nível de desenvolvimento, mas não é recomendada para produção, por ser uma base de dados do tipo in-memory.

**Querydsl:** Basicamente é um framework que nos permite a criação de queries type-safe, por meio de sua API fluente que suporta JPA, Hibernate Search, MongoDB, Lucene, e outras mais. Tudo isso com uma sintaxe muito similar ao JPQL, SQL e etc.

**JUnit 5:** Parte muito importante do projeto e que na maioria das vezes é negligenciada nos projetos, mas que considero fundamental para garantir a qualidade dos sistemas desenvolvidos, neste livro, iremos apenas mostrar na prática como utilizar o JUnit para fazer uma cobertura básica de testes, pois nosso projeto não terá grande complexidade. Porém recomendo fortemente que caso ainda não o utilize nos seus projetos, que comece a utilizar. E para quem não conhece ou nunca utilizou, JUnit é uma framework de testes para aplicações Java.

**Maven:** Framework responsável por gerenciar dependências, automatizar builds e etc. Novamente, pela minha experiência profissional, eu optei pelo Maven, mas se você tem mais familiaridade com o Gradle ou qualquer outra ferramenta similar, não há nenhum problema em optar por elas.

**Lombok:** Muitos ainda não conhecem, mas o Lombok é uma framework focado no aumento de produtividade e redução de boilerplate. Ele utiliza anotações adicionadas ao nosso código e em tempo de compilação, cria código Java e adiciona a sua aplicação.

O Lombok consegue te ajudar a não ter mais que escrever todos aqueles Getters e Setters do nosso projeto, além dos hashCode e Equals (que a maioria das pessoas não implementa e que são de

grande importância para uma boa construção de qualquer sistema), além de várias outras coisas mais.

**Swagger:** Se você trabalhou com REST APIs, provavelmente já ouviu falar ou utilizou o Swagger. Atualmente o Swagger é uma das ferramentas mais utilizadas no desenvolvimento de OpenAPI Specification (OAS). O Swagger nos permite uma forma muito simples de documentar uma API.

**Docker:** Talvez uma das maiores novidades para a maioria das pessoas, porque apesar de ter sido lançado em 2013, e ser uma ferramenta incrível, ainda não é utilizada de uma forma muito difundida. Docker é uma ferramenta que permite a criação de containers, facilitando a manutenção e administração dos ambientes. É uma ferramenta extremamente útil e importante quando se quer construir um Microserviço.

**Postman:** Ferramenta para auxiliar nos testes manuais do nosso Microserviço. Também poderia ser feito com SoapUI ou qualquer outra ferramenta do gênero.

**Git:** Como todo bom projeto, precisamos de uma ferramenta para versionamento do nosso código. No decorrer do nosso guia, também vou deixar o link do nosso repositório para que possam consultar o projeto de uma forma mais fácil.

# CAPÍTULO 2

## 2.1 CRIAÇÃO DO PROJETO

Depois de toda essa explicação que deveria ser breve sobre as componentes que aqui serão utilizadas, enfim chegou o momento em que vamos começar a realmente pôr as mãos na massa, ou melhor, no código. Primeiramente precisamos criar o nosso projeto, para isso, iremos utilizar o Spring Initializer, que torna esse processo de criação do projeto muito rápido e fácil.

Para começar, devemos acessar o site <https://start.spring.io>. Se nunca utilizou o Spring Initializer, a ideia é bastante simples, nós precisamos selecionar qual o nosso gerenciador de dependências, linguagem, versão do Spring Boot, informamos alguns dados relativos ao Project Metadata, como Group, Artifact, Name, Description, Package name, forma de empacotamento, se utilizaremos Jar ou War e por fim versão do Java. Caso seja necessário, também é possível adicionar as dependências que serão utilizadas no projeto. Ao terminar de informar tudo que é necessário, clicamos em Generate e a mágica acontece. Não se preocupe, nós vamos fazer tudo isso juntos a partir de agora.

Em primeiro lugar, precisamos definir o que nosso Microserviço irá fazer, para isso, separei um tema bastante simples, que será um Microserviço capaz de fazer o gerenciamento de tarefas, a princípio, sem nenhuma regra de negócio complicada nem nada do gênero, nesse momento, o importante é entender a estrutura do projeto.

Então já tendo nosso “problema” definido, vamos tratar de iniciar o projeto. Na primeira seção relativa às informações básicas do projeto, linguagem e versão do Spring Boot, vamos selecionar as seguintes opções: Maven, Java, 2.4.3 (A versão do Spring Boot

pode variar em virtude do espaço de tempo entre a escrita deste livro e o momento em que se está fazendo a leitura).

**Project**

☒ Maven Project ☐ Gradle Project

**Language**

☒ Java ☐ Kotlin ☐ Groovy

**Spring Boot**

☐ 2.5.0 (SNAPSHOT) ☐ 2.5.0 (M2) ☐ 2.4.4 (SNAPSHOT) ☒ 2.4.3

☐ 2.3.10 (SNAPSHOT) ☐ 2.3.9

O próximo passo é preencher as informações relativas ao Project Metadata, no campo Group inserimos a informação do group que queremos dar ao nosso projeto, no nosso caso, com.book. Preenchemos também em seguida o campo Artifact, ao preencher esse campo, nomeei nosso artifact como task, automaticamente já são preenchidos os campos Name e Package Name. Por último, insiro uma breve descrição sobre o projeto no campo Description, “Microserviço para gestão de tarefas”.

No campo Packaging, deixamos marcado a opção Jar, que vai representar o tipo de empacotamento que pretendemos e em seguida selecionamos o Java 11.

**Project Metadata**

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 15 ☒ 11 ☐ 8

Em seguida, na seção Dependencies, iremos adicionar as seguintes dependências: Lombok, Spring Boot DevTools, Spring HATEOAS, Spring Data JPA, Spring Web, Rest Repositories, Rest Repositories HAL Explorer, Spring Boot Actuator, H2 Database.

**Dependencies**

ADD DEPENDENCIES... CTRL + B

**Lombok** DEVELOPER TOOLS  
Java annotation library which helps to reduce boilerplate code.

**Spring Boot DevTools** DEVELOPER TOOLS  
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

**Spring HATEOAS** WEB  
Eases the creation of RESTful APIs that follow the HATEOAS principle when working with Spring / Spring MVC.

**Spring Data JPA** SQL  
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

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

**Rest Repositories** WEB  
Exposing Spring Data repositories over REST via Spring Data REST.

**Rest Repositories HAL Explorer** WEB  
Browsing Spring Data REST repositories in your browser.

**Spring Boot Actuator** OPS  
Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.

**H2 Database** SQL  
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Um ponto interessante, é que nesse momento o Spring Initializer já nos permite analisar a estrutura do projeto que vai ser criado, assim como o pom.xml e etc. Para isso, basta clicar no botão EXPLORE

Deverá surgir uma tela parecida com essa:

task.zip

.gitignore

.mvn

HELP.md

mvnw

mvnw.cmd

pom.xml

src

main

java

com

book

task

TaskApplication.java

resources

application.properties

static

templates

test

java

com

book

task

TaskApplicationTests.java

DOWNLOAD

COPY

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.4.3</version>
9     <relativePath/> <!-- Lookup parent from repository -->
10  </parent>
11  <groupId>com.book</groupId>
12  <artifactId>task</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>task</name>
15  <description>Micro Serviço para gestão de tarefas</description>
16  <properties>
17    <java.version>11</java.version>
18  </properties>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework.boot</groupId>
22      <artifactId>spring-boot-starter-actuator</artifactId>
23    </dependency>
24    <dependency>
25      <groupId>org.springframework.boot</groupId>
26      <artifactId>spring-boot-starter-data-jpa</artifactId>
27    </dependency>
28    <dependency>
29      <groupId>org.springframework.boot</groupId>
30      <artifactId>spring-boot-starter-data-rest</artifactId>
31    </dependency>
32    <dependency>
33      <groupId>org.springframework.boot</groupId>
34      <artifactId>spring-boot-starter-hateoas</artifactId>
```

DOWNLOAD

CTRL + ↵

CLOSE

ESC

Nesse momento, vale revisar para ver se não faltou preencher nenhum campo, nem incluir alguma dependência e caso esteja tudo correto, clicar no botão Generate, ou se ainda estiver com na tela do EXPLORE, botão DOWNLOAD.

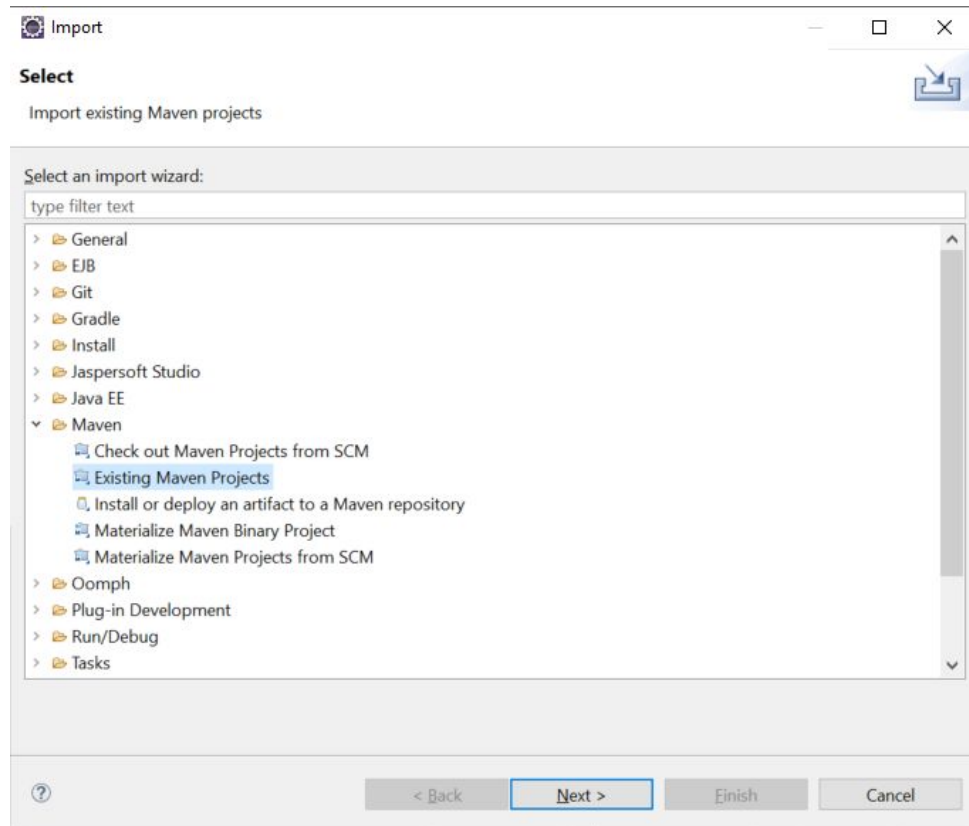
Pronto, você já tem um projeto inicial criado e o será iniciado o download do projeto.

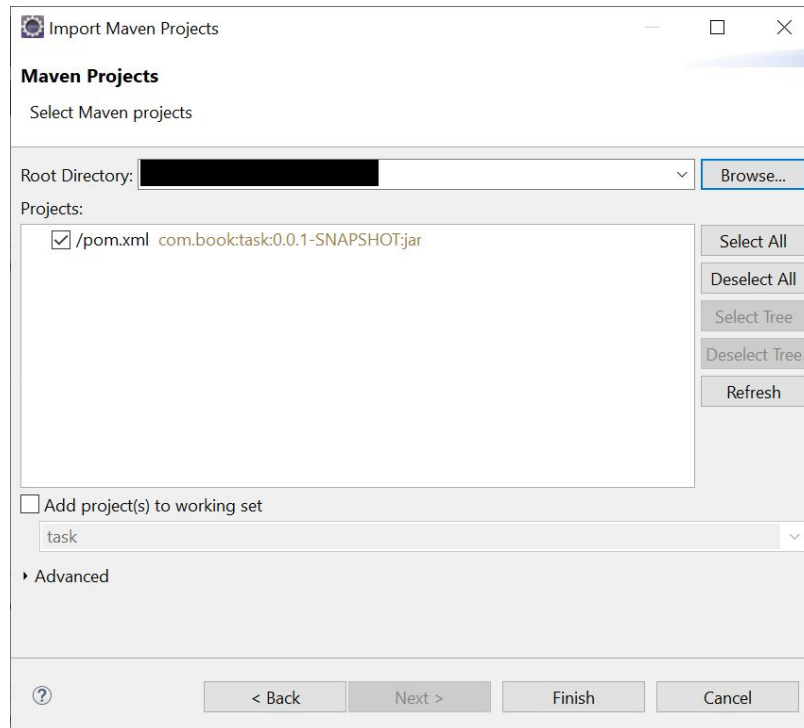


# CAPÍTULO 3

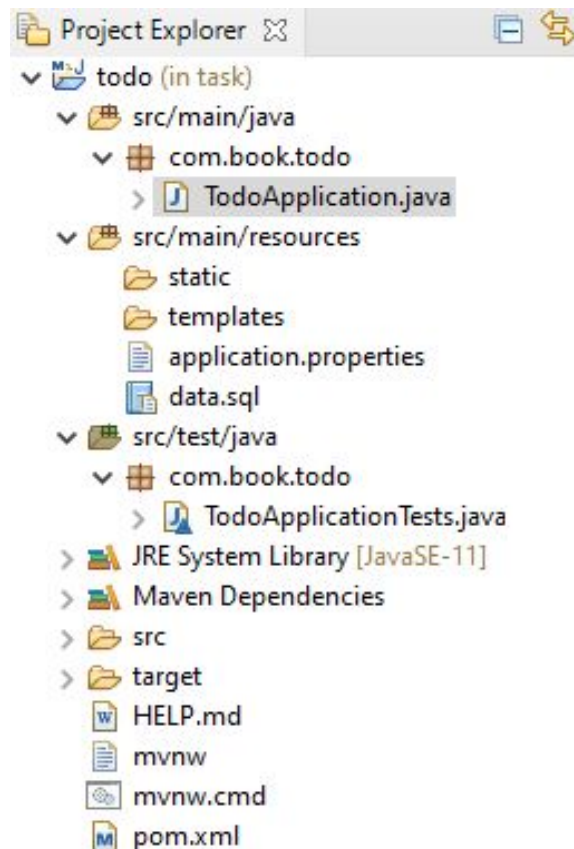
## 3.1 ECLIPSE IDE

Caso ainda não tenha feito download do Eclipse ou de qualquer outra IDE, esse é o momento, faça download e em seguida importe o projeto gerado pelo Spring Initializer, lembrando apenas que é um projeto Maven e deve ser importado como tal.





Após a importação do projeto, olhando para o Project Explorer, devemos ter uma estrutura de projeto muito parecida com essa:



Nesse momento, se tentarmos iniciar a aplicação, fazendo “Run As > Java Application” na classe `TaskApplication.java`, pelo Eclipse, iremos ter um erro de conexão com a base de dados, visto que ainda não fizemos nenhuma configuração para a mesma.



```
=====
:: Spring Boot ::
(v2.4.2)

2021-02-20 11:13:49.176 INFO 2800 --- [ restartedMain] com.beak.task.TaskApplication : Starting TaskApplication using Java 11.0.9 on GAD-DESKTOP with PID 2800 (D:\dev\task\target\classes started by C:\o in D:\dev\task)
2021-02-20 11:13:49.182 INFO 2800 --- [ restartedMain] com.beak.task.TaskApplication : No active profile set, falling back to default profiles: default
2021-02-20 11:13:49.255 INFO 2800 --- [ restartedMain] o.s.DevToolsPropertyDefaultsPostProcessor : DevTools property defaults active! Set 'spring.devtools.add-properties' to 'false' to disable
2021-02-20 11:13:49.355 INFO 2800 --- [ restartedMain] o.s.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' property to 'DEBUG'
2021-02-20 11:13:50.279 INFO 2800 --- [ restartedMain] s.s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2021-02-20 11:13:50.291 INFO 2800 --- [ restartedMain] s.s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 5 ms. Found 0 JPA repository interfaces.
2021-02-20 11:13:50.405 INFO 2800 --- [ restartedMain] o.a.s.b.w.embedded.tomcat.TomcatStarter : Tomcat initialized with port(s): 8080 (http)
2021-02-20 11:13:51.005 INFO 2800 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-02-20 11:13:51.006 INFO 2800 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.45]
2021-02-20 11:13:51.006 INFO 2800 --- [ restartedMain] o.a.c.c.c.[tomcat].[localhost].[/] : Initializing Spring embedded webApplicationContext
2021-02-20 11:13:51.006 INFO 2800 --- [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1731 ms
2021-02-20 11:13:51.022 WARN 2800 --- [ restartedMain] ConfigServletWebServerApplicationContext : Exception encountered during context initialization - cancelling refresh attempt: org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'InMemoryDatabaseShutdownExecutor' defined in class path resource [org/springframework/boot/devtools/autoconfigure/devtools/datasource/autoconfiguration.class]: Unsatisfied dependency expressed through method 'InMemoryDatabaseShutdownExecutor' parameter 0; nested exception is org.springframework.beans.factory.BeanCreationException: error creating bean with name 'datasource' defined in class path resource [org/springframework/boot/autoconfigure/dbc/datasource/configuration/ Hikari.class]: Bean instantiation via factory method failed; nested exception is org.springframework.beans.BeanInstantiationException: Failed to instantiate [com.zaxxer.hikari.HikariDataSource]: Factory method 'datasource' threw exception; nested exception is org.springframework.boot.autoconfigure.jdbc.datasourceproperties.HikariDataSourceProperties.HikariDataSourceBeanCreationException: Failed to determine a suitable driver class
2021-02-20 11:13:51.020 INFO 2800 --- [ restartedMain] o.apache.catalina.core.StandardService : Stopping service [Tomcat]
2021-02-20 11:13:51.053 INFO 2800 --- [ restartedMain] ConditionEvaluationReportLoggingListener :

Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2021-02-20 11:13:51.082 ERROR 2800 --- [ restartedMain] o.s.b.f.LoggingFailureAnalysisReporter :

=====
APPLICATION FAILED TO START
=====

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class


Action:

Consider the following:
  If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
  If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).
```

## 3.2 H2 DATABASE

Como já havia dito antes, mas sempre vale a pena reforçar, essa não é uma Database para ser utilizada em produção, por isso, seu uso é recomendado apenas para ambiente de desenvolvimento quando se quer ter um ambiente mais ágil.

Para começar, precisamos fazer as seguintes configurações no arquivo `application.properties`, que pode ser localizado em `/src/main/resources`.

Habilitar console H2;

Inserir configurações do datasource (URL de conexão, driver, user, pass e platform);

Definir um path de acesso para o WebServer

Então o conteúdo do arquivo será semelhante a esse:

`## H2`

`spring.h2.console.enabled=true`

`## Datasource`

`spring.datasource.url=jdbc:h2:mem:testdb`

`spring.datasource.driverClassName=org.h2.Driver`

`spring.datasource.username=root`

`spring.datasource.password=root`

`spring.jpa.database-platform=org.hibernate.dialect.H2Dialect`

`## WebServer`

`server.servlet.context-path=/api`

Lembrando que todas as configurações acima feitas e várias outras que possam ser úteis, sempre podem ser encontradas acessando a documentação oficial do Spring Boot.

Link para acesso da documentação:

<https://docs.spring.io/spring-boot/docs/current/reference/html/index.html>

\*O endereço de acesso pode variar devido a atualizações por parte dos criadores, portanto, caso o link não esteja funcionando, sempre é possível encontrar o endereço fazendo uma rápida busca no Google.

Após isso, precisamos definir a estrutura de tabelas que serão criadas pelo H2 ao iniciarmos a aplicação, para isso, crie um arquivo chamado data.sql no diretório /src/main/resources. Para nosso exemplo, o arquivo deverá conter o seguinte conteúdo:

```
DROP TABLE IF EXISTS task;
```

```
CREATE TABLE task (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  name VARCHAR(250) NOT NULL,  
  description VARCHAR(250) NOT NULL  
);
```

```
INSERT INTO task (name, description) VALUES  
  ('name1', 'Primeira tarefa'),  
  ('name2', 'Segunda tarefa'),  
  ('name3', 'Terceira tarefa');
```

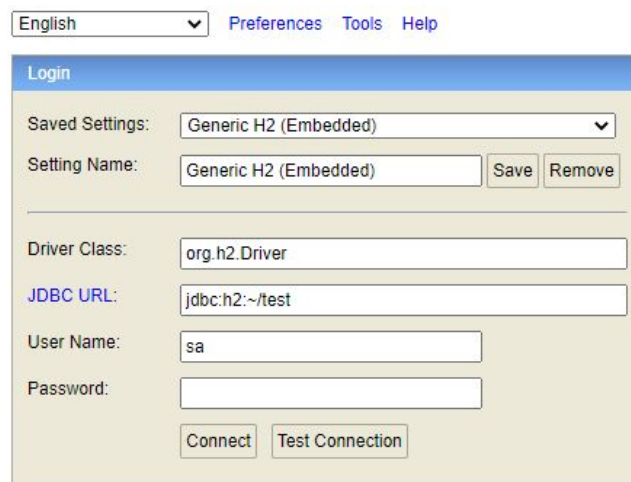
Nesse momento, já temos tudo que precisamos para iniciar a aplicação pela primeira vez sem erros. Então faça a Run As > Java Application na classe TaskApplication.java, pelo Eclipse. Poderemos então ver a seguinte mensagem no Console:

```
: Initializing ExecutorService 'applicationTaskExecutor'  
: Exposing 2 endpoint(s) beneath base path '/actuator'  
: Tomcat started on port(s): 8080 (http) with context path '/api'  
: Started TaskApplication in 4.221 seconds (JVM running for 4.691)
```

O que nos indica que a aplicação conseguiu se conectar com o H2 e subir sem problemas. No momento em que a aplicação é iniciada, ela se conecta ao H2 e faz a criação da tabela que foi definida anteriormente no script data.sql. Ou seja, todas as informações que forem persistidas nessa Database durante a utilização da sua aplicação, se não forem incluídas no script SQL para que a cada início da aplicação sejam inseridas novamente, serão perdidas. Por isso, se tiver informações de que vai precisar utilizar todas as vezes em que a aplicação estiver rodando, inclua no seu script.

Para garantirmos que a tabela e os dados foram criados com sucesso, podemos acessar um console web que o H2 disponibiliza que fizemos a ativação no application.properties.

Para isso, acesse no seu browser: <http://localhost:8080/api/h2-console>.



The screenshot shows the H2 database web console interface. At the top, there is a language dropdown set to 'English' and navigation links for 'Preferences', 'Tools', and 'Help'. The main section is titled 'Login'. It contains a 'Saved Settings' dropdown menu currently showing 'Generic H2 (Embedded)'. Below this is a 'Setting Name' field, also containing 'Generic H2 (Embedded)', with 'Save' and 'Remove' buttons to its right. A horizontal line separates this from the connection details. The 'Driver Class' field is set to 'org.h2.Driver'. The 'JDBC URL' field is set to 'jdbc:h2:~/test'. The 'User Name' field is set to 'sa', and the 'Password' field is empty. At the bottom, there are 'Connect' and 'Test Connection' buttons.

Para conseguir se conectar, precisamos que as informações estejam conforme definimos no application.properties, então, no

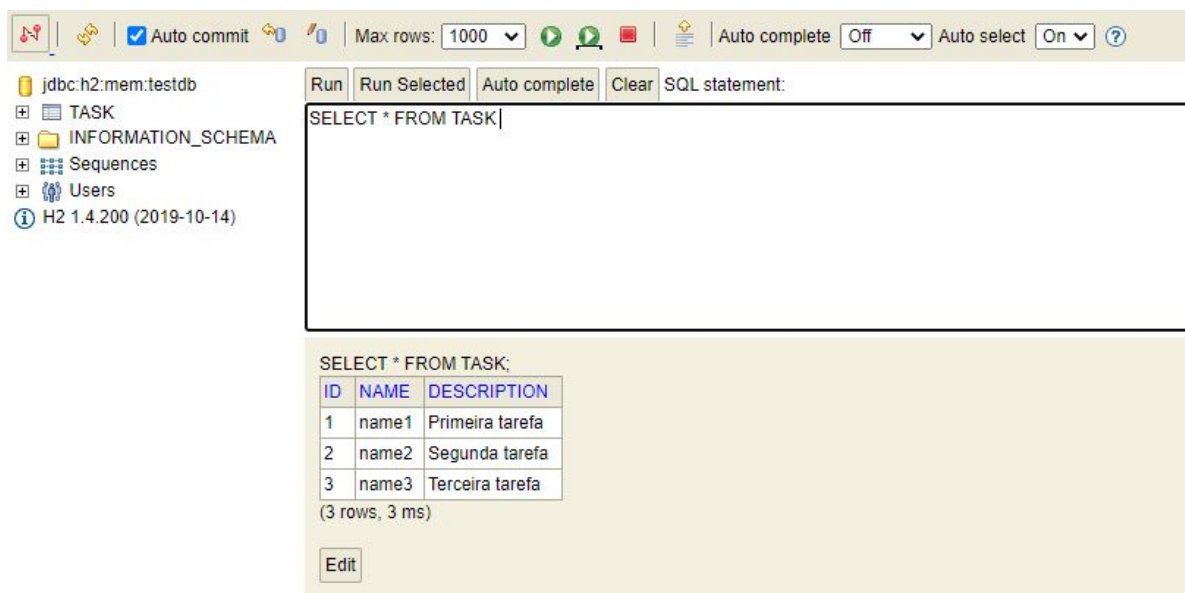
nosso caso deverão ser:

JDBC URL: jdbc:h2:mem:testdb

User name: root

Password: root

Podemos então agora realizar uma consulta para verificar se os dados foram corretamente inseridos.



## CAPÍTULO 4

### 4.1 ARQUITETURA DO PROJETO

Neste capítulo, iremos entender a arquitetura do projeto, em que camadas está dividida e suas responsabilidades. Lembrando que essa não é uma arquitetura criada por mim, apenas gosto de trabalhar dessa forma, porque na minha visão, é uma arquitetura mais que suficiente para a esmagadora maioria dos casos.

Então, vamos lá. O projeto está dividido em 4 camadas, são elas: Controller, Service, Repository e Database. Cada camada tem



sua função e responsabilidade definida e é importante que seja assim para que não haja dúvidas na hora da implementação. Vamos então agora detalhar um pouco o que cada camada faz e quais suas responsabilidades.

## **Controller**

Talvez a coisa mais importante que eu posso dizer aqui seja: “Nessa camada não deve ter nenhuma regra de negócio e nem acesso a Database”. Para quem já está acostumado a trabalhar assim, pode parecer simples, mas por incrível que pareça, é muito comum ver uma tonelada de regras de negócio implementadas nas classes de controle. Então você pode me perguntar: “Mas então, o que devemos ter nas nossas classes de controle?” e eu te respondo: “Única e exclusivamente a declaração dos seus endpoints, assim como o tipo correto de verbo HTTP que deve ser utilizado para aquela operação. A Controller deve apenas ser utilizada para invocar quem de fato realiza alguma lógica de negócio, que nessa arquitetura é a camada de Service”.

Alguns outros pontos que podemos abordar sobre a Controller, são as Annotations que devem ser utilizadas. É muito comum ver desenvolvedores com dúvidas sobre quais Annotations utilizar, por exemplo: “Devo utilizar `@RestController` ou `@Controller`?”.

A explicação é bastante simples, na realidade, a Annotation `@Controller` foi introduzida pelo Spring MVC para mapearmos uma classe como uma Controladora Spring. Já a Annotation `@RestController`, foi introduzida no Spring 4.0 para que possamos anotar especialmente Controladoras que sejam RESTful, logo, como nosso intuito aqui é produzir um Microserviço RESTful, ou seja, um Microserviço que tenha capacidade de utilizar REST, então iremos utilizar a Annotation `@RestController`.

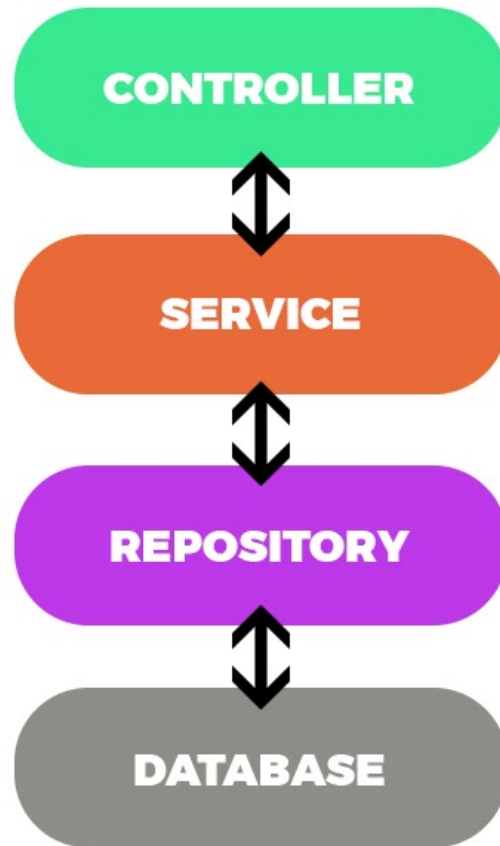
## **Service**

Como havia dito antes, essas são as classes responsáveis por todas as regras de negócio e elas devem ser implementadas única e exclusivamente aqui. Diferentemente das Controllers, nas classes Service, nós podemos injetar nossa camada de Repository para que possamos manipular os dados.

## **Repository**

São essas as classes responsáveis pela abstração da comunicação com a Database, à grosso modo, apenas nessas classes é que fazemos de fato o acesso a Database.

Então para ilustrar, nossas camadas seguem esse fluxo:



# CAPÍTULO 5

## 5.1 IMPLEMENTAÇÃO DA ENTIDADE

Agora que já conhecemos a arquitetura que vamos seguir e que já fizemos as configurações básicas e já é até possível subir nossa aplicação, finalmente vamos começar a implementação do nosso Microsserviço. Primeiramente, precisamos criar nossa classe Task, já que criamos uma tabela na nossa Database, agora precisamos de uma classe Java para representar essa entidade, ou seja, fazer nosso ORM. E ORM (Object Relational Mapper) nada mais é do que a representação de uma entidade de base de dados no nosso código, assim facilitando os relacionamentos e operações que possam envolver essa entidade.

Para isso, criamos uma classe Task, no pacote, com.book.todo.entity. Agora o que precisamos fazer é mapear os atributos da classe de acordo com os campos que definimos anteriormente na criação da tabela.

Nossa classe vai ficar assim:

```
package com.book.todo.entity;

import lombok.Data;

import javax.persistence.*;
import javax.validation.constraints.NotNull;

@Entity
@Data
public class Task {
```

```

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

@Column
@NotNull(message="{NotNull.Task.name}")
private String name;

@Column
@NotNull(message="{NotNull.Task.description}")
private String description;

}

```

Caso haja algum problema com o import relacionado à classe NotNull, pode ser necessário adicionar a dependência do JavaEE API ao pom.xml do projeto.

Para facilitar, deixo aqui a dependência:

```

<!--
https://mvnrepository.com/artifact/javax.annotation/javax.annotation-api -->
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
  <version>1.3.2</version>
</dependency>

```

Agora uma breve explicação sobre as Annotations utilizada na nossa entidade.

## **@Entity**

Annotation para informar que essa classe também é uma entidade, então, a partir disso, vai ser estabelecida uma ligação entre essa entidade e a tabela com o mesmo nome na Database.

## **@Data**

Annotation do Lombok, é basicamente a junção de outras Annotations que muito comumente são utilizadas juntas. São elas: **@Getter**, **@Setter**, **@ToString**, **@EqualsAndHashCode** e **@RequiredArgsConstructor**.

**@Getter** - Injeta em tempo de compilação os métodos Get de todos os atributos;

**@Setter** - Injeta em tempo de compilação os métodos Set de todos os atributos;

**@ToString** - Injeta em tempo de compilação a implementação do método toString;

**@EqualsAndHashCode** - Injeta em tempo de compilação a implementação dos métodos equals e hashCode;

**@RequiredArgsConstructor** - Injeta em tempo de compilação a implementação de um construtor com 1 parâmetro para cada atributo de sua classe. Você pode gerar um construtor para 1 ou mais parâmetros da sua classe de acordo com a sua necessidade.

## **@Id**

Annotation para informar qual é o atributo da nossa classe é responsável pela Primary Key da tabela.

## **@GeneratedValue**

Annotation que é utilizada para informar que o valor da Primary Key da nossa entidade será gerado e gerenciado pelo provedor de persistência. Esta Annotation é utilizada em conjunto a Annotation @Id e deve ser adicionada logo após a mesma. Caso não façamos isso, significa que a responsabilidade de gerar e gerenciar as Primary Keys será da nossa aplicação.

### **@Column**

Annotation que é utilizada para especificar mais detalhes sobre uma coluna/campo na Database.

### **@NotNull**

Annotation muito útil que nos auxilia a fazer validações diretamente na nossa entidade. Ela apenas realiza a validação para verificar se o valor do atributo onde está anotado não é nulo.

## 5.2 IMPLEMENTAÇÃO DO REPOSITÓRIO

Nesse momento já temos nossa entidade corretamente mapeada e implementada, podemos começar a implementação da nossa classe Repository. Vamos criar uma classe com o nome TaskRepository, dentro do pacote com.book.todo.repository. De forma similar como fizemos anteriormente, primeiro vemos como a classe foi escrita e em seguida, fazemos os comentários do que for relevante.

Mas antes, precisamos novamente adicionar uma dependência ao pom.xml, dessa vez, precisamos adicionar a dependência do Querydsl.

Segue a dependência que deverá ser adicionada:

```
<!-- https://mvnrepository.com/artifact/com.querydsl/querydsl-jpa -->
<dependency>
  <groupId>com.querydsl</groupId>
  <artifactId>querydsl-jpa</artifactId>
</dependency>
```

Além disso, também é necessário adicionar um plugin que faz o controle da geração das classes de Query do Querydsl.

```
<plugin>
  <groupId>com.mysema.maven</groupId>
  <artifactId>apt-maven-plugin</artifactId>
  <version>1.1.3</version>
  <executions>
    <execution>
```



```

        <goals>
            <goal>process</goal>
        </goals>
        <configuration>
            <outputDirectory>target/generated-
sources/java</outputDirectory>
            <processor>com.querydsl.appt.jpa.JPAAnnotationProcessor</processor>
        </configuration>
    </execution>
</executions>
<dependencies>
    <dependency>
        <groupId>com.querydsl</groupId>
        <artifactId>querydsl-apt</artifactId>
        <version>${querydsl.version}</version>
    </dependency>
    <dependency>
        <groupId>com.querydsl</groupId>
        <artifactId>querydsl-jpa</artifactId>
        <version>${querydsl.version}</version>
    </dependency>
</dependencies>
</plugin>

```

Após a inclusão desse plugin, é necessário compilar o projeto usando o Maven, para que ele gere corretamente as classes necessárias para a utilização do Querydsl.

Se seu projeto estiver sendo executado, pare a execução, faça um mvn clean install no projeto e em seguida inicie novamente.

Lembrando que todas essas dependências podem ser achadas facilmente no repositório oficial do Maven, e após a adição das dependências no pom.xml, pode ser necessário em alguns casos, realizar Maven > Update Projects no projeto.

Segue também o endereço para acesso do Maven Repo:

<https://mvnrepository.com/>

Um outro adendo que é importante dizer é que no início do projeto, incluímos uma dependência chamada Spring Boot Dev Tools, essa dependência fica monitorando o classpath do nosso projeto e a cada vez que fazemos uma alteração, ela faz restart a nossa aplicação.

É possível também controlar o comportamento dessa dependência, incluindo uma propriedade no arquivo application.properties.

**spring.devtools.restart.enabled=true**

Para o nosso projeto, não vamos desabilitar, pois a mesma ajuda a ter uma melhora considerável na produtividade e além disso, por padrão o Spring já desabilita essa propriedade quando nossa aplicação é empacotada, visando não ter esse comportamento em produção.

Implementação da classe **TaskRepository**:

```
package com.book.todo.repository;
```

```
import java.util.Collection;
```

```
import java.util.List;
```

```
import java.util.Optional;
```

```
import org.springframework.data.domain.Page;
```

```
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import
org.springframework.data.jpa.repository.JpaSpecificationExecutor;
import org.springframework.data.jpa.repository.Query;
import
org.springframework.data.querydsl.QuerydslPredicateExecutor;
import org.springframework.data.repository.query.Param;
import
org.springframework.data.rest.core.annotation.RepositoryRestRes
ource;
import
org.springframework.data.rest.core.annotation.RestResource;
import com.book.todo.entity.Task;
```

```
@RepositoryRestResource(path = "task", collectionResourceRel =
"tasks")
```

```
public interface TaskRepository extends JpaRepository<Task,
Integer>, JpaSpecificationExecutor<Task>,
QuerydslPredicateExecutor<Task> {
```

```
    Page<Task> findByIdIn(@Param(value = "id") List<Integer>
eventid, Pageable pageable);
```

```
    Page<Task> findByNameIn(@Param("name")
Collection<String> names, Pageable pageable);
```

```
    @Query(name="Task.findByName", nativeQuery = true)
List<Task> findByName(@Param("name") String name);
```

```
    Page<Task> findAll(Pageable pageable);
```

```
@Query(name="Task.findById", nativeQuery = true)
@RestResource(exported = false)
Optional<Task> findById(@Param("id") long id);
}
```

## **@RepositoryRestResource**

Se não conhece essa Annotation, ela pode te causar alguma surpresa. Normalmente, quando utilizamos Spring Data, o mais comum seria utilizar a Annotation @Repository e estender a classe CrudRepository, com isso já teríamos todo o acesso da Database preparado. A Annotation @RepositoryRestResource também faz isso, e até aí, não é novidade nenhuma, mas o pulo do gato vem agora, repare que juntamente com a Annotation, incluímos duas propriedades: path e collectionResourceRel.

Basicamente uma delas é responsável pelo path para acessar esse RestResource e a outra, collectionResourceRel, por definir por qual nome chamaremos uma coleção dos nossos registros. Mas você deve estar se perguntando, “o que? pra que eu quero um path e um nome específico pra coleção na minha classe de repositório?”, pois bem, eu disse que era aqui o pulo do gato. Além de nos criar todo aquele CRUD da camada de repositório, ele automaticamente irá expor o nosso CRUD através de uma API REST, sim, você não leu errado, utilizando o @RepositoryRestResource você não vai precisar criar nem Controller e nem Service.

E se ainda sim, você está achando bom demais pra ser verdade, é simples, basta fazer uma requisição GET, usando o Postman ou o Browser, para o endereço: <http://localhost:8080/api/task>, executando assim uma requisição para esse endpoint e ver a magia acontecer.

```

    "_embedded": {
      "tasks": [
        {
          "name": "name1",
          "description": "Primeira tarefa",
          "_links": {
            "self": {
              "href": "http://localhost:8080/api/task/1"
            },
            "task": {
              "href": "http://localhost:8080/api/task/1"
            }
          }
        }
      ],
    },
  ],
}

```

Ah, e tem mais, lembra que eu disse que ele expõe todo o CRUD? Pois é, vou deixar aqui a lista das operações que ele disponibiliza, assim como, seus correspondentes verbos HTTP.

- GET <http://localhost:8080/api/task>
- GET <http://localhost:8080/api/task/{id}>
- GET <http://localhost:8080/api/task/search>
- POST <http://localhost:8080/api/task>
- PUT <http://localhost:8080/api/task/{id}>
- PATCH <http://localhost:8080/api/task/{id}>
- DELETE <http://localhost:8080/api/task/{id}>

Bem simples né? Ah, se você reparar bem, a API de consulta já vem com a paginação pronta, então é realmente um grande auxílio.

```

"page": {
  "size": 20,
  "totalElements": 3,
  "totalPages": 1,
  "number": 0
}

```

Como nem tudo são flores, infelizmente essa implementação nos deixa um pouco amarrados no caso de precisarmos implementar algo que não seja apenas pequenas validações a nível de modelo. Porém, se seu objetivo é apenas ter uma API REST para fazer operações CRUD, sem ter muita complexidade, isso é perfeito e é uma mão na roda.

### **JpaRepository, JpaSpecificationExecutor e QuerydslPredicateExecutor**

No caso dos JpaRepository, JpaSpecificationExecutor, basicamente, estamos fazendo Extend de algumas especificações para que possamos utilizar a API Criteria JPA. Já o QuerydslPredicateExecutor, atua como uma ponte entre o Spring Data e o Querydsl, nos permitindo utilizar os Predicates construídos com Querydsl, para fazer consultas via Spring Data.

## 5.3 IMPLEMENTAÇÃO DA SERVICE

Como foi dito anteriormente, nem tudo são flores, então nós queremos poder implementar alguma regra de negócio no nosso Microserviço, ou apenas, customizar livremente a resposta que nossa APIs terá. Para isso então, vamos construir uma classe Service para nossa API de Tasks.

Antes de começar a implementação da nossa classe Service, vamos construir uma classe *DTO* (Data transfer object), que utilizaremos para representar nossa entidade Task. Então, dentro do pacote: `com.book.todo.dto`, vamos criar a classe `TaskDto.java`.

```
package com.book.todo.dto;
```

```
import lombok.Data;
```

```
@Data
```

```
public class TaskDto {
```

```
    private long id;
```

```
    private String name;
```

```
    private String description;
```

```
    private String createdAt;
```

```
}
```

É bastante interessante ver o quanto o Lombok consegue reduzir no boilerplate do nosso projeto e essa classe é um excelente exemplo disso.

Depois de criar nosso *DTO*, precisamos adicionar uma nova dependência ao nosso arquivo `pom.xml`, para nos auxiliar a fazer Map da nossa entidade para *DTO* e vice-versa.

A dependência que vamos utilizar é o Modelmapper, e sua utilização é bastante simples, mas já iremos ver a seguir.

Mais uma vez, para facilitar, aqui fica a dependência do ModelMapper no Maven Repository.

```
<!--  
https://mvnrepository.com/artifact/org.modelmapper/modelmapper  
-->  
<dependency>  
  <groupId>org.modelmapper</groupId>  
  <artifactId>modelmapper</artifactId>  
  <version>2.3.9</version>  
</dependency>
```

Agora que já temos nossa dependência importada e nosso *DTO* construído, vamos ver o código da nossa classe Service. A classe vai ser criada no pacote com.book.todo.service, e vai se chamar TaskService.java.

Segue a implementação da nossa classe:

```
package com.book.todo.service;  
  
import lombok.extern.slf4j.Slf4j;  
  
import java.util.Optional;  
  
import org.modelmapper.ModelMapper;  
import org.springframework.data.domain.Page;  
import org.springframework.data.domain.Pageable;  
import org.springframework.stereotype.Service;
```



```
import com.book.todo.dto.TaskDto;
import com.book.todo.entity.Task;
import com.book.todo.repository.TaskRepository;

@Slf4j
@Service
public class TaskService {

    private TaskRepository tasksRepository;

    public TaskService(TaskRepository tasksRepository) {
        this.tasksRepository = tasksRepository;
    }

    public Page<Task> getTasks(Pageable pageable) {
        return tasksRepository.findAll(pageable);
    }

    public Task getTask(long taskId) {
        Optional<Task> task = tasksRepository.findById(taskId);
        return task.get();
    }

    public Task saveTask(TaskDto taskDto) {
        ModelMapper modelMapper = new ModelMapper();
        Task task = modelMapper.map(taskDto, Task.class);

        return tasksRepository.save(task);
    }
}
```

No método `saveTask`, podemos observar a utilização do `ModelMapper`, como já foi dito anteriormente, sua implementação é bastante simples, bastando utilizar o método `map` e informar o objeto que gostaria de mapear e a classe de destino.

Para mais informações a respeito do `ModelMapper`, pode se consultar a sua documentação oficial no endereço: <http://modelmapper.org/>.

## 5.4 IMPLEMENTAÇÃO DA CONTROLLER

Continuando na lógica de ter controle total do nosso Microserviço, vamos construir uma classe Controller, que será a porta de entrada para nossa API de Tasks.

Antes de vermos o código da nossa classe Controller, vamos implementar algumas classes auxiliares.

A primeira delas vai ser a classe TaskUri.java, que vai ser criada no pacote: com.book.todo.uri, essa classe tem a finalidade apenas de deixar nosso código mais organizado, centralizando assim as informações relativas às URIs da nossa aplicação.

```
package com.book.todo.uri;
```

```
import org.springframework.hateoas.server.EntityLinks;  
import org.springframework.stereotype.Component;
```

```
@Component
```

```
public class TaskUri {
```

```
    private final EntityLinks entityLinks;
```

```
    public static final String TASKS = "/tasks";
```

```
    public static final String TASK = "/task/{id}";
```

```
    public static final String CREATE_TASK = "/task";
```

```
    public TaskUri(EntityLinks entityLinks) {
```

```
        this.entityLinks = entityLinks;
```

```
    }
```

```
}
```

Agora então, podemos implementar nossa Controller, a classe vai ser criada no seguinte pacote: `com.book.todo.controller` e vai se chamar `TaskController.java`.

Implementação da **TaskController.java**:

```
package com.book.todo.controller;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import
org.springframework.data.rest.webmvc.PersistentEntityResourceA
ssembler;
import
org.springframework.data.rest.webmvc.RepositoryRestController;
import org.springframework.data.web.PagedResourcesAssembler;
import org.springframework.hateoas.EntityModel;
import org.springframework.hateoas.Link;
import org.springframework.hateoas.PagedModel;
import
org.springframework.hateoas.server.mvc.WebMvcLinkBuilder;
import static
org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
```

```
import
org.springframework.web.bind.annotation.RequestMapping;
import
org.springframework.web.server.ResponseStatusException;
```

```
import com.book.todo.dto.TaskDto;
import com.book.todo.entity.Task;
import com.book.todo.service.TaskService;
import com.book.todo.uri.TaskUri;
```

```
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
```

```
@Slf4j
@RepositoryRestController
@RequestMapping("/todo/")
@RequiredArgsConstructor
public class TaskController {
```

```
    private final PagedResourcesAssembler
pagedResourcesAssembler;
    private final TaskService taskService;
```

```
    @GetMapping(path = TaskUri.TASKS)
    public ResponseEntity<?> getTasks(TaskDto taskDto, Pageable
pageable, PersistentEntityResourceAssembler
resourceAssembler) {
        log.info("TasksController: " + taskDto);
        Page<Task> events = taskService.getTasks(pageable);
        PagedModel<?> resource =
pagedResourcesAssembler.toModel(events, resourceAssembler);
        return ResponseEntity.ok(resource);
    }
```

```
}
```

```
@GetMapping(path = TaskUri.TASK)
public ResponseEntity<?> getTask(@PathVariable("id") int
taskId, Pageable pageable, PersistentEntityResourceAssembler
resourceAssembler) {
```

```
    try {
```

```
        log.info("TasksController::: " + taskId);
        Task task = taskService.getTask(taskId);
        Link selfLink = WebMvcLinkBuilder.linkTo( methodOn(
this.getClass() ).getTask(taskId, pageable, resourceAssembler)
).withSelfRel();
        Link allTasksLink = WebMvcLinkBuilder.linkTo(
this.getClass() ).slash("/tasks").withRel("all tasks");
```

```
        EntityModel<Task> entityModel = EntityModel.of( task );
        entityModel.add(selfLink, allTasksLink);
```

```
        return ResponseEntity.ok(entityModel);
    } catch (RuntimeException exc) {
        throw new
ResponseStatusException(HttpStatus.NOT_FOUND, "Task Not
Found", exc);
    }
}
```

```
@PostMapping(path = TaskUri.CREATE_TASK)
public ResponseEntity<?> createTask(@RequestBody TaskDto
taskDto, Pageable pageable, PersistentEntityResourceAssembler
resourceAssembler) {
```

```
log.info( "TasksController: " + taskDto );  
Task events = taskService.saveTask( taskDto );
```

```
Link selfLink = WebMvcLinkBuilder.linkTo( methodOn(  
this.getClass() ).createTask(taskDto, pageable,  
resourceAssembler)).withSelfRel();
```

```
Link allTasksLink = WebMvcLinkBuilder.linkTo(  
this.getClass() ).slash("/tasks").withRel("all tasks");
```

```
EntityModel<Task> taskResource = EntityModel.of( events );  
taskResource.add(selfLink, allTasksLink);
```

```
HttpHeaders responseHeaders = new HttpHeaders();  
responseHeaders.set("CustomResponseHeader",  
"CustomValue");
```

```
return new ResponseEntity<EntityModel<Task>>  
(taskResource, responseHeaders, HttpStatus.CREATED);  
}
```

```
}
```

Até agora essa foi a nossa maior classe e vou tentar detalhar um pouco mais os pontos que considero serem mais importantes. Vamos começar com as Annotations.

## **@Slf4j**

Anotação do Lombok que nos auxilia a injetar o log do Slf4j em nossa classe, basicamente evita que façamos aquele trabalho mecânico que estamos acostumados a fazer na criação de todas as classes, por exemplo, com a implementação padrão do Slf4j, faríamos:

```
Logger log = LoggerFactory.getLogger(TaskController.class);
```

Basicamente o que o Lombok faz, é injetar esse código pra gente, então não precisamos de o escrever em todas as classes que queremos mandar alguma informação para os logs da aplicação.

### **@RepositoryRestController**

Apenas informa ao Spring Boot que esse é um componente responsável por ser uma Controller REST.

### **@RequestMapping("/todo/")**

Annotation responsável pelo mapeamento do path, tanto da Controller inteira, quanto dos métodos individualmente.

### **@GetMapping & @PostMapping**

Annotations responsáveis por fazer o mapeamento dos verbos HTTP dos nossos métodos. Poderiam ser utilizadas ainda, @DeleteMapping, @PatchMapping e @PutMapping. Lembrando que existe uma especificação para cada tipo de verbo HTTP e é importante segui-la corretamente.

Além dessas Annotations, podemos observar algumas implementações onde estamos fazendo uso do Spring HATEOAS. O Spring HATEOAS nos fornece algumas APIs para facilitar a criação de representações REST que seguem o princípio HATEOAS. O principal problema que ele tenta resolver é a criação de links e montagem de representação dos objetos. Com o Spring HATEOAS, conseguimos enviar na resposta do nosso serviço, os links construídos para navegar entre os conteúdos relacionados.

Agora que já implementamos todas as nossas camadas, Repository, Service e Controller, podemos iniciar a nossa aplicação e fazer alguns testes utilizando o Postman, para verificar o comportamento dos métodos que implementamos.



Primeiramente, vou realizar uma requisição GET, para o método `getTasks`, definido na nossa Controller. Para acessar esse método, vamos utilizar a URL base da nossa aplicação + RequestMapping da nossa Controller + RequestMapping do nosso método.

O endereço então, ficaria assim:  
<http://localhost:8080/api/todo/tasks>.

Ao executar uma requisição GET no Postman, obtive a seguinte resposta:

```
"_embedded": {  
  "tasks": [  
    {  
      "name": "name1",  
      "description": "Primeira tarefa",  
      "_links": {  
        "self": {  
          "href": "http://localhost:8080/api/task/1"  
        },  
        "task": {  
          "href": "http://localhost:8080/api/task/1"  
        }  
      }  
    }  
  ],  
}
```

Sinta-se livre para explorar o funcionamento dos outros métodos implementados.

## 5.5 SWAGGER

Antes de mais nada, precisamos incluir a dependência do Swagger em nosso pom.xml, segue as dependência:

```
<!-- https://mvnrepository.com/artifact/io.springfox/springfox-boot-
starter -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

Para a implementação do Swagger, iremos criar dentro do pacote com.book.todo.config, a classe Swagger2Configuration.java. Essa classe será responsável pelas configurações relativas ao Swagger e terá o seguinte conteúdo:

```
package com.book.todo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.PathSelectors;
import
springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import
springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
```

@EnableSwagger2

```
public class Swagger2Configuration {
```

```
    @Bean
```

```
    public Docket api() {
```

```
        return new Docket(DocumentationType.SWAGGER_2)
```

```
            .select()
```

```
            .apis(RequestHandlerSelectors.basePackage("com.book.todo.controller"))
```

```
            .paths(PathSelectors.ant("/**"))
```

```
            .build();
```

```
    }
```


```
}
```

Após isso, todas as vezes que compilarmos nosso projeto, o Swagger automaticamente vai gerar toda a documentação de acesso para todas as controllers que estiverem sob o pacote “com.book.todo.controller”.

Então, ao acessar o endereço:

<http://localhost:8080/api/swagger-ui/>.

Devemos ter uma página como essa:

 **Swagger**  
Supported by SMARTBEAR

# Api Documentation 1.0

[ Base URL: localhost:8080 ]  
<http://localhost:8080/api/v2/api-docs>

Api Documentation

[Terms of service](#)

[Apache 2.0](#)

## task-controller Task Controller

POST

/api/todo/task createTask

GET

/api/todo/task/{id} getTask

GET

/api/todo/tasks getTasks

Models

Toda essa documentação é gerada automaticamente pelo Swagger e pode ser customizada a sua necessidade. Para mais informações a respeito do Swagger, é possível encontrar na sua documentação oficial: <https://swagger.io/docs/>.

# CAPÍTULO 6

## 6.1 TESTES

Nenhum sistema está pronto (ou pelo menos não deveria) se não foram implementados os testes, por mais básicos que sejam, é um hábito que deveria ser trabalhado dentro de cada um de nós para que possamos cada vez mais produzir sistemas com mais qualidade.

Por mais que esse livro não seja voltado para qualidade de código, essa é uma característica que todo bom desenvolvedor deve trazer consigo, então acho sempre importante ressaltar a necessidade da implementação de testes unitários e de integração a cada vez que formos implementar algo novo, seja um Microserviço, uma API, um método, ou ainda, alterar código legado e garantir que nada se quebrou e que as coisas continuam funcionando como deveriam.

Sei que na teoria é tudo muito fácil de se fazer, e que quando o “calo aperta” e o prazo de entregar aquela nova funcionalidade ou correção está acabando, a primeira coisa que fazemos é sacrificar os testes, então, aí está nosso erro. Eu digo nosso, porque seria hipocrisia dizer que nunca fiz e nem faço isso, mas o que eu posso dizer é: “Faça!”, por mais básico e menor que seja a cobertura dos testes, se não começarmos, nunca vamos tê-los.

Se algum dia não conseguir implementar os testes, converse com o time de desenvolvimento e crie (se ainda não existir) um lugar para registrar as dívidas técnicas do projeto.

Criar cobertura de testes em projetos de desenvolvimento, mais do que uma prática saudável para o projeto, é um trabalho de

evangelização que ainda precisamos fazer sempre que tivermos a oportunidade.

## 6.2 TESTES UNITÁRIOS COM JUNIT E MOCKITO

Como já havia mencionado antes, o JUnit vai ser responsável pelos testes unitários do nosso projeto. Nossos testes unitários vão fazer testes aos componentes do nosso sistema de forma isolada. Se o seu componente depender de informações de qualquer outro lugar, vamos fazer um Mock dessas informações. Nos testes unitários, devemos nos preocupar única e exclusivamente com o componente do sistema que estamos testando.

Um outro ponto importante é que os testes unitários devem ser extremamente rápidos, tanto para implementar, quanto para executar. Isso porque, como todas as informações que precisamos estarão disponíveis via Mock de dados, não vamos ter qualquer tráfego de rede ou qualquer outra que possa gerar alguma lentidão, tendo isso em mente, vamos começar.

Primeiramente, iremos adicionar todas as dependências necessárias para o JUnit e o Mockito.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
```

```

        <artifactId>junit-jupiter-engine</artifactId>
        <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <scope>test</scope>
</dependency>

```

Na estrutura do nosso projeto, em src/test/java, vamos criar uma classe chamada TaskMock, nesta classe apenas vamos atribuir alguns valores aos atributos, para quando precisarmos de dados nos nossos testes.

```
package com.book.todo.mock;
```

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import com.book.todo.entity.Task;
```



```

import java.util.ArrayList;
import java.util.List;

public class TaskMock {

    public static Page<Task> createTasks() {

        List<Task> taskList = new ArrayList<>();
        Task task1 = new Task();
        task1.setId(1);
        task1.setName("Tarefa 1");
        task1.setDescription("Descrição da tarefa 1");

        Task task2 = new Task();
        task2.setId(2);
        task2.setName("Tarefa 2");
        task2.setDescription("Descrição da tarefa 2");

        taskList.add(task1);
        taskList.add(task2);
        Page<Task> pagedResponse = new PageImpl(taskList);
        return pagedResponse;
    }

}

```

É sempre interessante criar classes de Mock, para os objetos que temos, ao invés de instanciar os objetos dentro de cada testes unitário e atribuir valor, isso porque, torna muito mais ágil o processo de criação e manutenção dos testes unitários.

Agora, na mesma estrutura de testes, iremos criar uma classe chamada `TaskServiceTest.java`, dentro do pacote `com.book.todo.service`. Perceba aqui, que a estrutura que utilizaremos para nossos testes, vai na maioria das vezes refletir a estrutura que utilizamos ao desenvolver nossa aplicação.

```
package com.book.todo.service;
```

```
import com.book.todo.entity.Task;
import com.book.todo.mock.TaskMock;
import com.book.todo.repository.TaskRepository;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.junit.platform.runner.JUnitPlatform;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.Mockito;
```

```
import org.mockito.junit.jupiter.MockitoExtension;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
```

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
```

```
@ExtendWith(MockitoExtension.class)
@RunWith(JUnitPlatform.class)
```

```

public class TaskServiceTest {

    @Mock
    TaskRepository tasksRepository;
    private TaskService taskService;

    @BeforeEach
    public void setup() {

        taskService = new TaskService(tasksRepository);
        Pageable pageable = PageRequest.of(0, 5, Sort.by(
            Sort.Order.asc("name"),
            Sort.Order.desc("id")));
        Mockito.lenient().when(tasksRepository.findAll(pageable)).th
enReturn(TaskMock.createTasks());
    }

    @Test
    @DisplayName("Should return all tasks")
    public void getTasks_happypath() {

        Pageable pageable = PageRequest.of(0, 5, Sort.by(
            Sort.Order.asc("name"),
            Sort.Order.desc("id")));

        Page<Task> tasks = taskService.getTasks(pageable);

        assertEquals(tasks.getTotalPages(), 1);
        assertEquals(tasks.getNumberOfElements(), 2);
        assertNotNull(tasks);
    }
}

```

```
@Nested
@DisplayName("Happy Tests")
class happycases {

    @Test
    void justtest() {
        String name = "just testing";
        assertEquals(name, "just testing");
    }

    @Test
    void justtest1() {
        String name = "just testin";
        assertEquals(name, "just testin");
    }
}
```

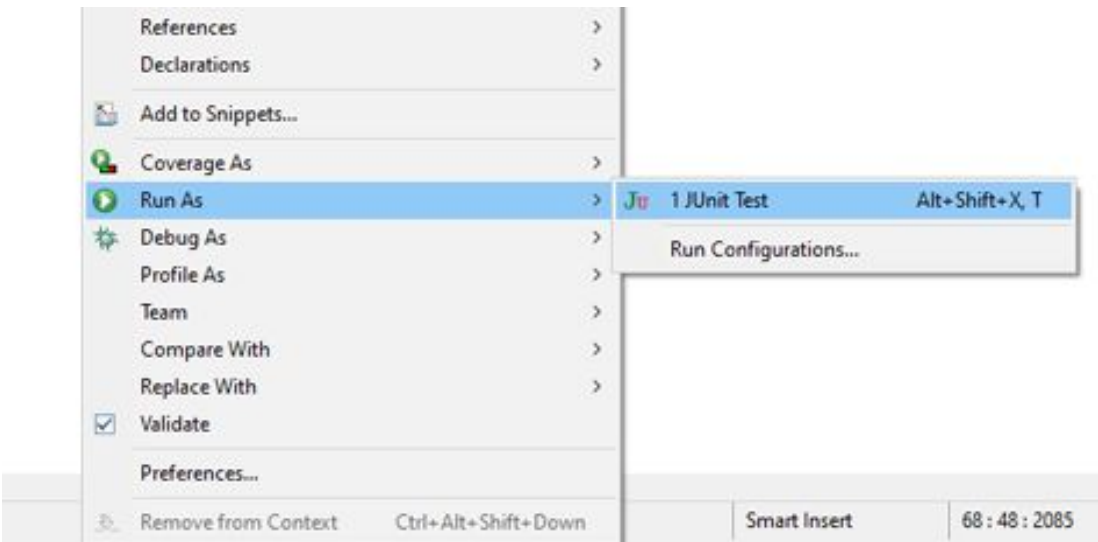
```
@Nested
@DisplayName("Unhappy Tests")
class unhappycases {

    @Test
    void justtest() {
        String name = "just testing";
        assertEquals(name, "just testing");
    }

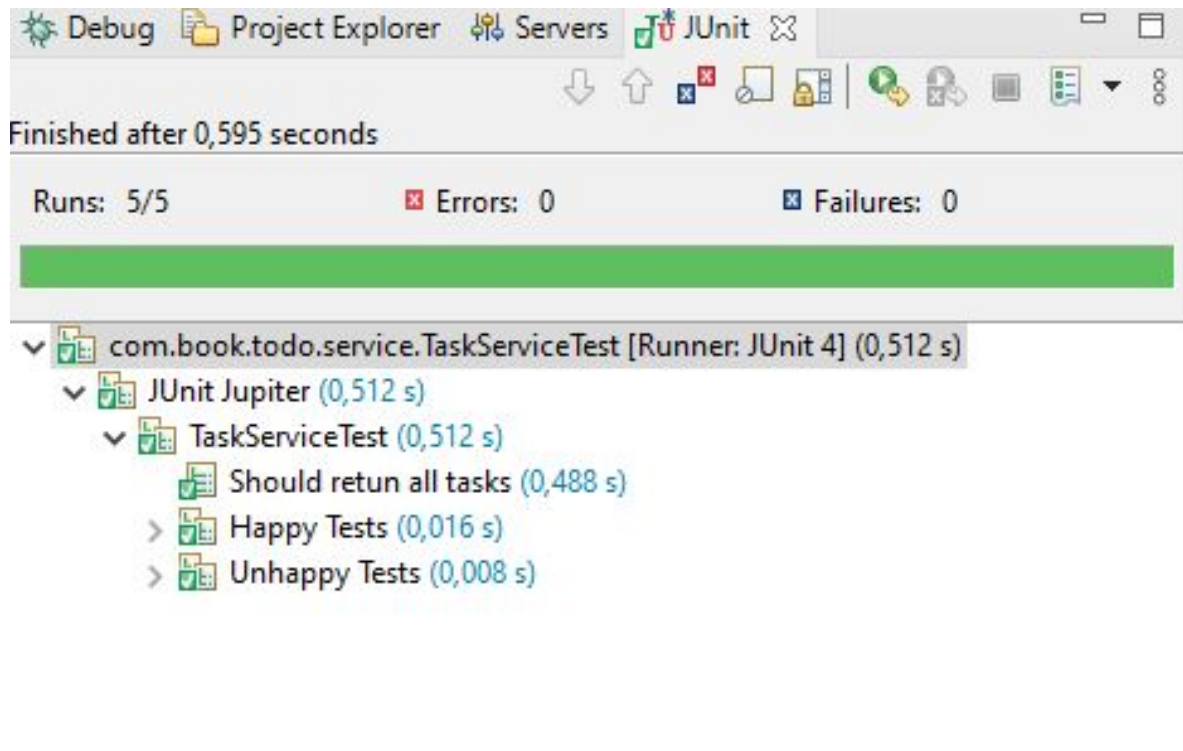
    @Test
    void justtest1() {
        String name = "just testing";
        assertEquals(name, "just testincc");
    }
}
```

```
}  
}  
  
}
```

Após a implementação do nosso teste unitário, podemos executar o teste clicando com o botão direito na própria classe de teste e fazendo Run As > JUnit Test.



Depois do teste ser executado, podemos ver o resultado:



Isso é o que queremos ver em todo e qualquer projeto em que trabalhemos, um teste rápido, bem divididos em diferentes cenários e de acordo com as regras de negócio do nosso sistema, e acima de tudo, sem falhas.

Para mais informações sobre a utilização do JUnit e do Mockito, segue os links para as documentações.

<https://junit.org/junit5/docs/current/user-guide/>

<https://site.mockito.org/>

## 6.3 TESTES DE INTEGRAÇÃO COM REST ASSURED

Nesta seção, iremos fazer alguns simples testes de integração, para garantir que outros sistemas consigam fazer uso do nosso Microserviço. Para isso, vamos utilizar a biblioteca Rest Assured, que nos auxilia na implementação desse tipo de teste. Basicamente o Rest Assured nos permite criar chamadas HTTP, simulando um cliente que estivesse acessando nossas APIs. Ele suporta todos os verbos HTTPs e podemos fazer validações das respostas.

Então para começar, vamos adicionar as dependências do Rest Assured ao nosso pom.xml.

```
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured</artifactId>
  <version>4.1.0</version>
  <scope>test</scope>
</dependency>
```

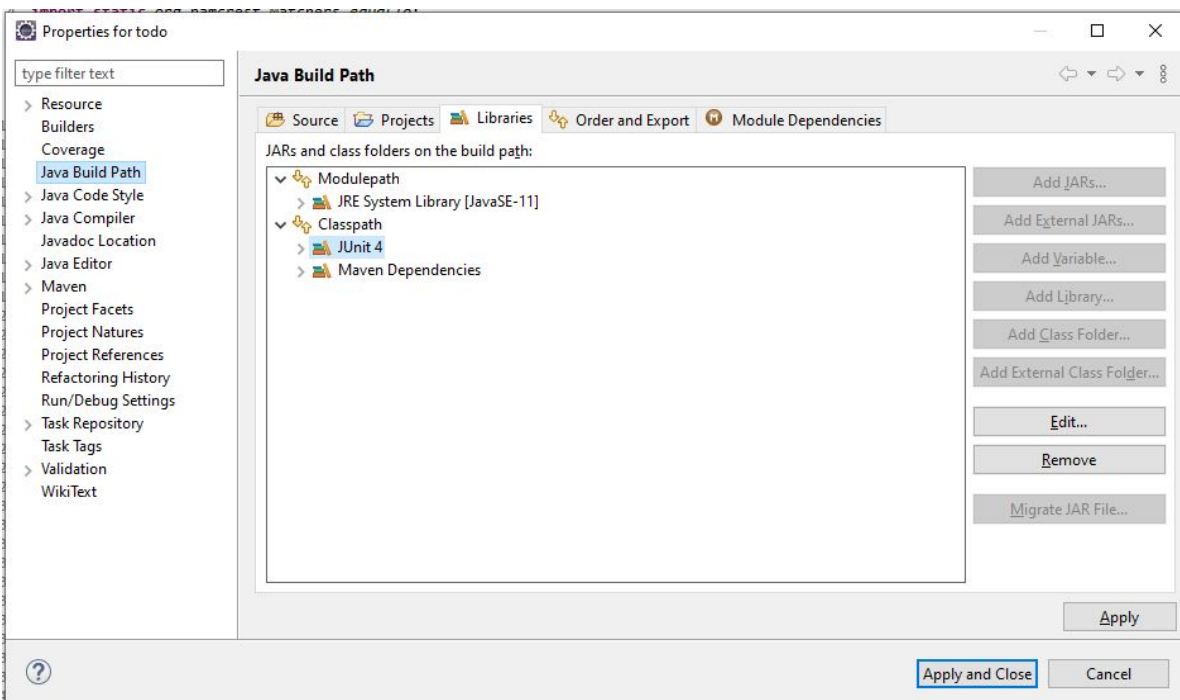
```
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured-all</artifactId>
  <version>4.0.0</version>
</dependency>
```

Também vamos utilizar a biblioteca do Hamcrest, que nos auxilia a criar regras de verificação de forma declarativa. A princípio, não precisamos de incluir a biblioteca, pois o JUnit já a traz consigo.

Um problema que é bastante comum, é que o Eclipse traz a biblioteca do JUnit com ele, com isso, ao utilizar os Matchers da biblioteca do Hamcrest, ocorre um erro com a mensagem:

“maven class "org.hamcrest.Matchers"'s signer information does not match signer information of other classes in the same package”

Para resolver isso, basta remover a biblioteca do JUnit que está no ClassPath do nosso projeto. Para isso, clique com o botão direito no projeto e faça: “Build Path > Configure Build Path...”. Em seguida, navegue para a tab Libraries.



Nesta tab, basta selecionar a biblioteca do JUnit 4 e clicar em Remove, em seguida Apply and Close.



Em seguida, vamos criar uma classe com o seguinte nome: `TasksControllerIntegrationTest.java`, ela ficará localizada no pacote: `com.book.todo.integration`.

```
package com.book.todo.integration;
```

```
import static io.restassured.RestAssured.get;  
import static org.hamcrest.Matchers.equalTo;
```

```
import org.junit.Before;  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.extension.ExtendWith;  
import org.junit.platform.runner.JUnitPlatform;  
import org.junit.runner.RunWith;  
import org.mockito.junit.jupiter.MockitoExtension;
```

```
import io.restassured.RestAssured;
```

```
@ExtendWith(MockitoExtension.class)
```

```
@RunWith(JUnitPlatform.class)
```

```
public class TasksControllerIntegrationTest {
```

```
    @Before
```

```
    public void setup() {
```

```
        RestAssured.baseURI = "http://localhost:8080";
```

```
        RestAssured.port = 8080;
```

```
    }
```

```
    @Test
```

```
    public void
```

```
givenUrl_whenSuccessOnGetsResponseAndJsonHasRequiredKV  
_thenCorrect() {
```

```

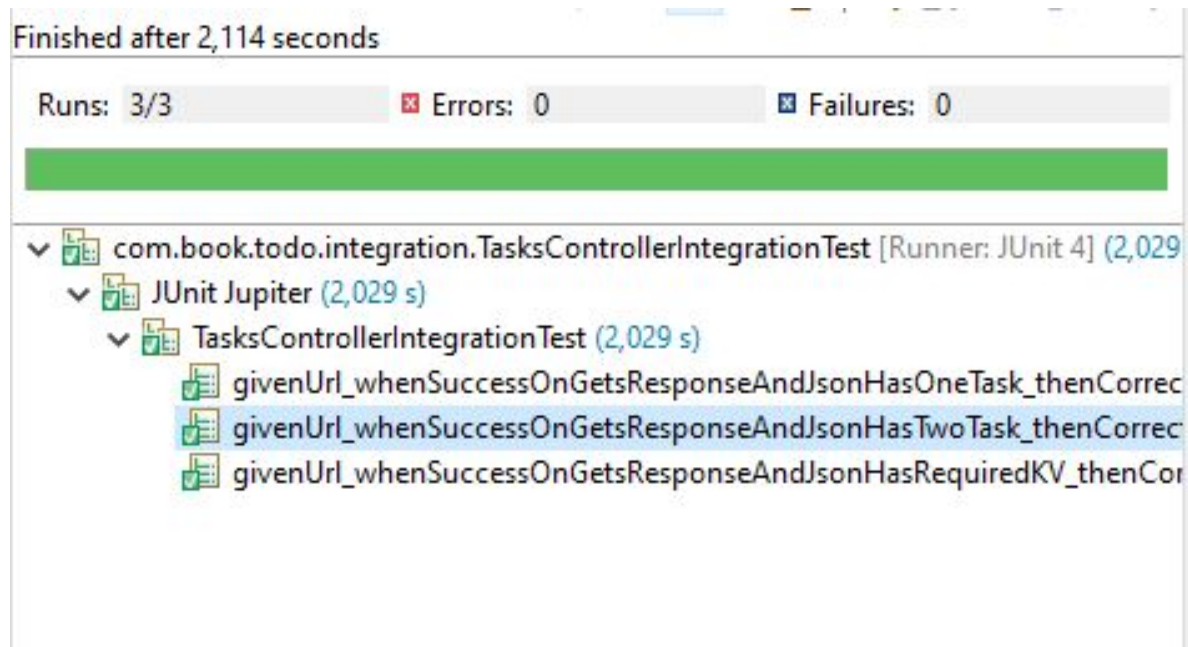
        get("/api/todo/tasks").then().statusCode(200);
    }

    @Test
    public void
givenUrl_whenSuccessOnGetsResponseAndJsonHasOneTask_thenCorrect() {
        get("/api/todo/task/1").then().statusCode(200)
            .assertThat().body("name",
                equalTo("name1"), "description",
                equalTo("Primeira tarefa"));
    }

    @Test
    public void
givenUrl_whenSuccessOnGetsResponseAndJsonHasTwoTask_thenCorrect() {
        get("/api/todo/task/2").then().statusCode(200)
            .assertThat().body("name",
                equalTo("name2"), "description",
                equalTo("Segunda tarefa"));
    }
}

```

Com o projeto sendo executado, podemos clicar na classe, da mesma forma que fizemos no teste unitário e fazer “Run as > JUnit Test”, assim veremos o resultado dos nossos testes.



Porém, se tentarmos com a nossa aplicação parada, fazer um mvn clean install, esses testes vão falhar, pois não teremos uma aplicação rodando para se conectar e realizar os testes.

Para que isso não aconteça, vamos adicionar 2 Annotations, na nossa classe **TasksControllerIntegrationTest.java**, são elas **@SpringBootTest** e **@ActiveProfiles**.

### **@SpringBootTest**

Essa Annotation informa ao Spring Boot que essa classe de testes precisa que uma Main Configuration Class esteja sendo executada, em outras palavras, precisa que alguma classe com a Annotation **@SpringBootApplication** esteja sendo executada, no nosso caso, vai ser a classe `TodoApplication.java`

### **@ActiveProfiles**

Essa classe serve para ativarmos um perfil específico no Spring Boot, no nosso caso, queremos garantir que vamos utilizar um perfil de testes ao iniciar o Spring Boot.

Então o trecho da declaração de nossa classe **TasksControllerIntegrationTest.java**, vai ficar assim:

```
@ExtendWith(MockitoExtension.class)
@RunWith(JUnitPlatform.class)
@SpringBootTest(classes = {TodoApplication.class},
webEnvironment =
SpringBootTest.WebEnvironment.DEFINED_PORT)
@ActiveProfiles("test")
public class TasksControllerIntegrationTest {
```

Com isso, já podemos fazer mvn clean install no nosso projeto sem nenhum problema.

# CAPÍTULO 7

## 7.1 DOCKER - PREPARANDO A APLICAÇÃO

Assim como no capítulo anterior, que falamos sobre testes, Docker não é o foco deste livro, mas por considerar que é uma peça importante para nossa aplicação, decidi dedicar esse capítulo a ele.

Atualmente, está se tornando cada vez mais comum a prática de “Dockerizar” uma aplicação, e é exatamente o que vamos fazer aqui. Isso significa, que vamos criar uma imagem Docker para nossa aplicação, e seguindo as regras da nossa arquitetura, essa imagem deve conter tudo que o nosso Microserviço necessita para funcionar. Essa imagem posteriormente poderá ser gerida por alguma engine de orquestração, como Docker Swarm ou Kubernetes, porém, não vamos abordar isso neste livro.

A primeira alteração que vamos fazer no nosso projeto então, será no pom.xml, vamos fazer uma alteração, para que ao executar o comando mvn clean install, o Maven gere um pacote .jar da nossa aplicação num diretório específico que vamos determinar. Ao gerar o pacote num diretório separado, nos traz a vantagem de não precisar enviar todo o código fonte da nossa aplicação para o daemon do Docker quando estivermos construindo nossa imagem.

Então no nosso pom.xml, vamos procurar pelo seguinte artefato:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
  <excludes>
```



```
<exclude>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
</exclude>
</excludes>
</configuration>
</plugin>
```

Em seguida, vamos modificá-lo para que fique dessa forma:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>repackage</goal>
      </goals>
      <configuration>
        <mainClass>com.book.todo.TODOApplication</mainClass>
        <outputDirectory>${project.basedir}/docker</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Nesse ponto, devemos ter uma atenção especial para a tag `mainClass`, que deve apontar para a nossa `MainClass`, no nosso caso, a `TODOApplication.java`. Em seguida, podemos executar o comando `mvn clean install` no nosso projeto, após o término da

execução, deverá ser possível encontrar o pacote gerado no diretório que indicamos na tag `outputDirectory`.

▼  docker  
     todo-0.0.1-SNAPSHOT.jar

## 7.2 DOCKER - DOCKERFILE

Com a nossa aplicação já preparada para gerar o pacote da forma que precisamos, agora vamos criar o nosso Dockerfile, que vai ser responsável por criar nossa imagem Docker.

No diretório docker do nosso projeto, vamos criar um arquivo chamado Dockerfile e seu conteúdo será o seguinte:

```
# Alpine Linux com OpenJDK 11 JRE
FROM azul/zulu-openjdk-alpine:11

# Copiar pacote .jar e renomear para echo.war
COPY todo-0.0.1-SNAPSHOT.jar /echo.war

# Executar a aplicação
CMD ["/usr/bin/java", "-jar", "/echo.war"]
```

Apesar de achar que é bastante auto explicativo, o que estamos fazendo nesse Dockerfile é definindo que vamos utilizar uma outra imagem como base, no nosso caso, um Alpine Linux, que vai ser o sistema operacional do nosso container Docker.

Essa imagem em específico, também vem com a JDK 11 instalada e configurada, então, é basicamente tudo o que precisamos para que seja possível executar nosso projeto. Em seguida, fazemos a cópia do nosso pacote e o renomeamos apenas para facilitar o processo e em seguida, executamos a aplicação.

O que precisamos fazer agora é utilizar essa imagem para criar um container no nosso Docker local. Para isso, abra o Git Bash e em seguida execute essa lista de comandos:



```
$ docker build -t task-api .
```

```
$ docker tag task-api book/task-api
```

```
$ docker run -d -p 8080:8080 --name taskapi task-api
```

Com isso, já teremos nosso container sendo executado com a imagem que criamos e já podemos fazer requisições para nossa aplicação normalmente.

# CAPÍTULO 8

## 8.1 CONCLUSÃO E CONSIDERAÇÕES FINAIS

Esse livro é apenas um guia básico de como as coisas poderiam ser feitas e para essa aplicação ir para produção, várias outras coisas ainda teriam de ser ajustadas, porém, para uma demonstração é mais que suficiente.

Como estamos utilizando o Java 11 no nosso projeto, é importante que sejam utilizados os recursos do mesmo, caso contrário, não faz sentido o seu uso, então sempre que possível, faça uso das novas funcionalidades introduzidas nas versões mais recentes, isso ajuda a melhorar as aplicações assim como melhorar nossas capacidades de desenvolvimento.

Já falei isso antes, mas, tenha a certeza de que está utilizando o verbo HTTP correto em suas aplicações para que isso não gere problemas posteriormente, isso é muito importante.

Sempre desenvolva a maior cobertura de testes possível para seu projeto, isso ajuda muito a melhorar a qualidade do seu código e te tornar um desenvolvedor melhor e que faz entregas com menos problemas.

Por fim, espero que esse livro tenha te ajudado não só a construir um Microsserviço, mas a ser um desenvolvedor melhor de alguma forma, por isso, até a próxima e muito obrigado por ter chegado até aqui.

Ah, antes que eu me esqueça, todo o código do nosso projeto pode ser encontrado aqui no GitHub:

<https://github.com/caioalan/microservice-task>



# REFERÊNCIAS BIBLIOGRÁFICAS

Documentação Oficial Spring Boot:

<https://docs.spring.io/spring-boot/docs/current/reference/html/index.html>

Documentação Oficial Model Mapper:

<http://modelmapper.org/>

Documentação Oficial Swagger:

<https://swagger.io/docs/>

Documentação Oficial JUnit:

<https://junit.org/junit5/docs/current/user-guide/>

Documentação Oficial Mockito:

<https://site.mockito.org/>

Documentação Oficial QueryDSL:

[http://www.querydsl.com/static/querydsl/4.4.0/reference/html\\_single/](http://www.querydsl.com/static/querydsl/4.4.0/reference/html_single/)

Documentação Oficial Docker:

<https://docs.docker.com/>

## SOBRE O AUTOR

CAIO COSTA é um Engenheiro de Software Sênior, especializado em Java, com uma vasta experiência no setor público e no mercado corporativo, tendo iniciado sua carreira no Brasil e mais recentemente convidado a atuar em Portugal.

Nos mais de 10 anos de experiência, agrega as equipes com que trabalha produtividade, agilidade e conhecimento de novas tecnologias, nunca abrindo mão da alta qualidade do Software.

Excelência e celeridade são sua filosofia para desempenhar papéis de liderança e destacar-se no setor de tecnologia.