

## Treinamento Springboot – Conteúdo Teórico/Prático

## Sumário

O que é Spring Boot?.....	3
Objetivos básicosdo Spring Boot.....	3
Principais recursos do Spring Boot.....	3
Estrutura de projeto padrão para projetos Spring Boot .....	7
Não use o “default” package.....	7
Layout Típico .....	7
Estrutura do Projeto - Primeira abordagem.....	8
Conteúdo Estático (static content).....	9
Conteúdo web dinâmico (templates).....	10
Estrutura do Projeto - Segunda abordagem .....	11
pom.xml.....	11
O que é o Spring Boot Starter Parent e como usá-lo? .....	12
Aplicativos web spring boot - usando JSP como visualizações .....	13
Arquitetura do projeto Spring Boot .....	14
Arquitetura de três camadas (três camadas).....	15
Arquitetura Spring Boot Flow (exemplo) .....	16
Java Spring Boot & Eclipse .....	19
Pré-Requisitos .....	19
Instalando o Eclipse.....	19
Spring Tools .....	25
Criando o projeto .....	25
Criando projeto Spring Boot com Spring Initializer.....	33
Web Service Spring Restful .....	38
Estrutura do Diretório do Projeto .....	40
Springboot Annotations .....	48
Projeto API de upload / download de arquivo de springboot .....	53
<b>Escrevendo APIs para upload e download de arquivos</b> .....	58
Spring Boot Thymeleaf CRUD.....	77
Etapas de Desenvolvimento.....	78
Construindo um aplicativo de chat com Spring Boot e WebSocket .....	90
Projeto Spring Boot integrado com Angular 9 .....	116
Arquitetura do projeto .....	117
Projeto Spring Boot com Angular eAPI Rest MongoDB .....	131

Criando back-end Spring Boot.....	132
Apendice.....	152
Referencias:.....	153

## O que é Spring Boot?

Spring Boot é basicamente uma extensão do framework Spring que eliminou as configurações clichê necessárias para configurar um aplicativo Spring.

Spring Boot é uma estrutura de opinião que ajuda os desenvolvedores a construir aplicativos baseados em Spring de forma rápida e fácil. **O principal objetivo do Spring Boot é criar rapidamente aplicativos baseados em Spring sem exigir que os desenvolvedores escrevam a mesma configuração clichê repetidamente.**

### Objetivos básicos do Spring Boot

- Fornecer uma experiência inicial radicalmente mais rápida e amplamente acessível para todo o desenvolvimento do Spring.
- Fornecer uma variedade de recursos não funcionais que são comuns a grandes classes de projetos (como servidores integrados, segurança, métricas, verificações de saúde e configuração externalizada).
- Absolutamente nenhuma geração de código e nenhum requisito para configuração XML.

### Principais recursos do Spring Boot

Abaixo, temos alguns recursos-chave da inicialização do Spring e abordaremos cada um deles:

1. Spring Boot starters
2. Spring Boot autoconfiguration
3. configuration management “elegante”
4. Spring Boot actuator
5. Easy-to-use embedded servlet container support

#### 1. Spring Boot Starters

Spring Boot oferece muitos módulos iniciais para começar rapidamente com muitas das tecnologias comumente usadas, como SpringMVC, JPA, MongoDB, Spring Batch, SpringSecurity, Solr, Elasticsearch, etc. Esses iniciadores são pré-configurados com as dependências de biblioteca mais comumente usadas. você não precisa procurar as versões de biblioteca compatíveis e configurá-las manualmente.

Por exemplo, o módulo *spring-boot-starter-data-jpa* starter inclui todas as dependências necessárias para usar Spring Data JPA, junto com as dependências da biblioteca Hibernate, já que Hibernate é a implementação JPA mais comumente usada.

Mais um exemplo, quando adicionamos a dependência *spring-boot-starter-web*, ela puxará por padrão todas as bibliotecas comumente usadas durante o desenvolvimento de aplicativos Spring MVC, como *spring-webmvc*, *jackson-json*, *validation-api* e *tomcat*.

O *spring-boot-starter-web* não apenas adiciona todas essas bibliotecas, mas também configura os beans comumente registrados como *DispatcherServlet*, *ResourceHandlers*, *MessageSource*, etc. com padrões razoáveis.

## 2. Configuração automática do Spring Boot (Spring Boot Autoconfiguration)

Spring Boot resolve o problema de que os aplicativos Spring precisam de configuração complexa, eliminando a necessidade de definir manualmente a configuração padrão.

Spring Boot tem uma visão “opinativa” da aplicação e configura vários componentes automaticamente, registrando os beans com base em vários critérios. Os critérios podem ser:

- Disponibilidade de uma classe particular em um classpath
- Presença ou ausência de spring bean
- Presença de uma propriedade do sistema
- Ausência de um arquivo de configuração

Por exemplo, se você tiver a dependência *spring-webmvc* em seu classpath, Spring Boot assume que você está tentando construir um aplicativo da web baseado em SpringMVC e tenta automaticamente registrar *DispatcherServlet* se ainda não estiver registrado. Se você tiver qualquer driver de banco de dados embutido no classpath, como H2 ou HSQL, e se você não configurou um bean *DataSource* explicitamente, o Spring Boot

registrará automaticamente um bean *DataSource* usando as configurações do banco de dados na memória.

### 3. Configuration Management elegante

O Spring oferece suporte à externalização de propriedades configuráveis usando a configuração [@PropertySource](#). O Spring Boot vai ainda mais longe, usando os padrões razoáveis e a poderosa associação de propriedade de tipo seguro às propriedades do bean. O Spring Boot oferece suporte a arquivos de configuração separados para perfis diferentes sem exigir muitas configurações.

### 4. Spring Boot Actuator (Atuador Spring Boot)

Ser capaz de obter os vários detalhes de um aplicativo em execução na produção é crucial para muitos aplicativos. O atuador Spring Boot fornece uma ampla variedade de recursos prontos para produção sem exigir que os desenvolvedores escrevam muitos códigos. Alguns dos recursos do atuador de springboot são:

- Possibilidade ver os detalhes de configuração do bean de aplicativo
- Possibilidade visualizar os mapeamentos de URL do aplicativo, detalhes do ambiente e valores de parâmetro de configuração
- Pode ver as métricas de verificação de saúde registradas

### 5. Suporte para contêiner de servlet embutido fácil de usar (Easy-to-Use Embedded Servlet Container Support)

Tradicionalmente, ao construir aplicativos da web, você precisa criar módulos do tipo *WAR* e, em seguida, implantá-los em servidores externos como *Tomcat*, *WildFly*, etc. Mas usando Spring Boot, você pode criar um módulo do tipo *JAR* e incorporar o contêiner de servlet no aplicativo muito facilmente para que o aplicativo seja uma unidade de implantação independente.

Além disso, durante o desenvolvimento, você pode facilmente executar o módulo do tipo Spring Boot JAR como um aplicativo Java a partir do IDE ou da linha de comando usando uma ferramenta de construção como [Maven](#) ou Gradle.

## Requisitos de sistema

Spring Boot 2+ requer Java 8 ou 9 e Spring Framework 5.1.0.RELEASE ou superior.

O suporte explícito de compilação é fornecido para as seguintes ferramentas de compilação:

Build Tool	Version
Maven	3.2+
Gradle	4.x

## Servlet Containers

Spring Boot oferece suporte aos seguintes contêineres de servlet incorporados:

Name	Servlet Version
Tomcat 8.5	3.1
Jetty 9.4	3.1
Undertow 1.4	3.1

Você também pode implantar aplicativos Spring Boot em qualquer contêiner compatível com Servlet 3.1+.

## Estrutura de projeto padrão para projetos Spring Boot

Neste passo, abordaremos a criação de uma estrutura de projeto de springboot padrão ou estrutura de *package*. Observaremos maneiras recomendadas de criar uma estrutura de projeto de springboot.

De acordo com a documentação do Spring boot, a equipe nos aponta que o Spring Boot não requer nenhum layout de código específico para funcionar. No entanto, existem algumas práticas recomendadas que podem nos ajudar com nossos projetos. É importante observarmos alguns pontos, por exemplo:

### Não use o “default” package

Quando uma classe não inclui uma declaração de pacote, ela é considerada no “pacote padrão” (default package). O uso do “pacote padrão” (default package) geralmente é desencorajado e deve ser evitado. Isso pode causar problemas específicos para aplicativos Spring Boot que usam as **annotations**:

- `@ComponentScan`
- `@EntityScan`
- `@SpringBootApplication`

já que todas as classes de todos os jar são lidas.

É recomendado que possamos seguir as convenções de nomenclatura de pacote recomendadas do Java e usemos um nome de domínio reverso (por exemplo, *com.example.projectname*).

### Layout Típico

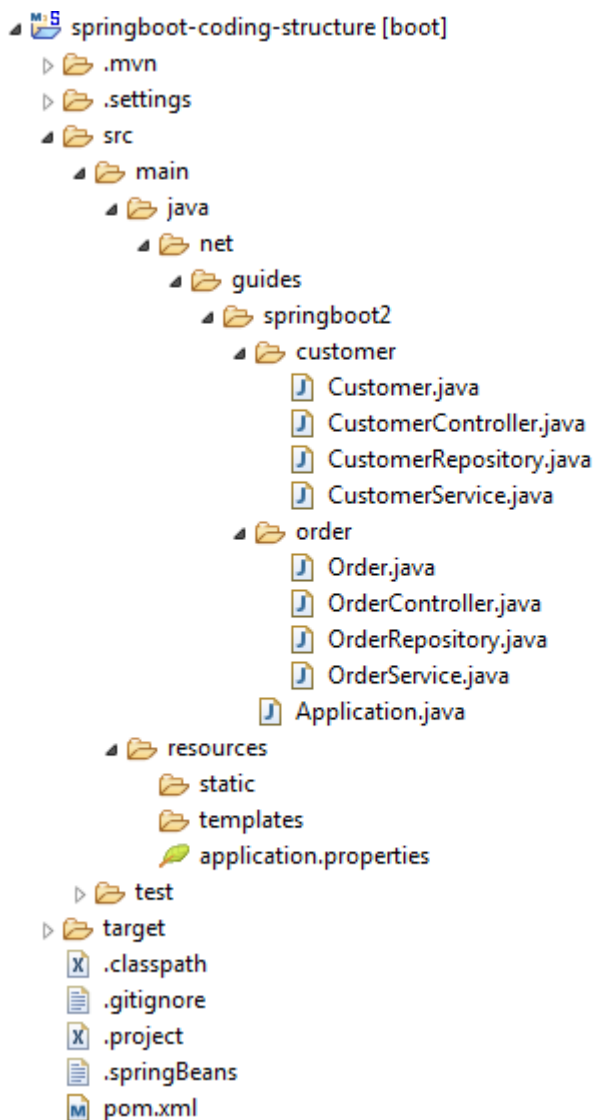
A equipe springboot geralmente recomenda que possamos localizar a classe de aplicativo principal em um pacote raiz acima de outras classes. A annotation **@SpringBootApplication** é freqüentemente colocada em sua classe principal e define implicitamente um “pacote de pesquisa” básico para certos itens. Por exemplo, se você estiver escrevendo um aplicativo JPA, o pacote da classe anotada **@SpringBootApplication** é usado para



procurar itens `@Entity`. O uso de um pacote raiz também permite que a varredura de componente se aplique apenas ao seu projeto.

### Estrutura do Projeto - Primeira abordagem

A seguinte estrutura de projeto de inicialização de spring mostra um layout típico recomendado pela equipe de inicialização de spring:



O arquivo *Application.java* declararia o método principal, junto com o **@SpringBootApplication** básico, da seguinte maneira:

```

import org.springframework.boot.SpringApplication ;
import
org.springframework.boot.autoconfigure.SpringBootApplication ;
@SpringBootApplication public class Application {
    public static void main ( String [] args ) {
        SpringApplication . run ( Application . class,
args);
    }
}

```

### Conteúdo Estático (static content)

Em um aplicativo web com springboot, arquivos estáticos como HTML, CSS, JS e IMAGE podem ser exibidos diretamente de qualquer um dos seguintes locais de *classpath* prontos para uso. Nenhuma configuração é necessária.

Vejamos a *resources folder* na estrutura do projeto acima. Este diretório, como o nome sugere, é dedicado a todos os recursos estáticos, modelos e arquivos de propriedades.

- *resources / static* - contém recursos estáticos como CSS, js e imagens.
- *resources / templates* - contém templates do lado do servidor que são renderizados pelo Spring.
- *resources / application.properties* - Este arquivo é muito importante. Ele contém propriedades de todo o aplicativo. Spring lê as propriedades definidas neste arquivo para configurar seu aplicativo. Você pode definir a porta padrão de um servidor, caminho de contexto do servidor, URLs de banco de dados, etc, neste arquivo.

Por padrão, o Spring boot fornece conteúdo estático de um dos seguintes locais no classpath:

- /static
- /public
- /resources
- / META-INF / resouces

Por padrão, os recursos são mapeados em / **\*\*** , mas você pode ajustar isso com a propriedade *spring.mvc.static-path-pattern* . Por exemplo, realocar todos os recursos para / *resources* / **\*\*** pode ser feito da seguinte forma:

```
spring.mvc.static-path-pattern = /resources/**
```

Você também pode personalizar os locais de recursos estáticos usando a propriedade `spring.resources.static-locations` (substituindo os valores padrão por uma lista de locais de diretório). O caminho de contexto do Servlet raiz, "/", também é adicionado automaticamente como um local.

### Conteúdo web dinâmico (templates)

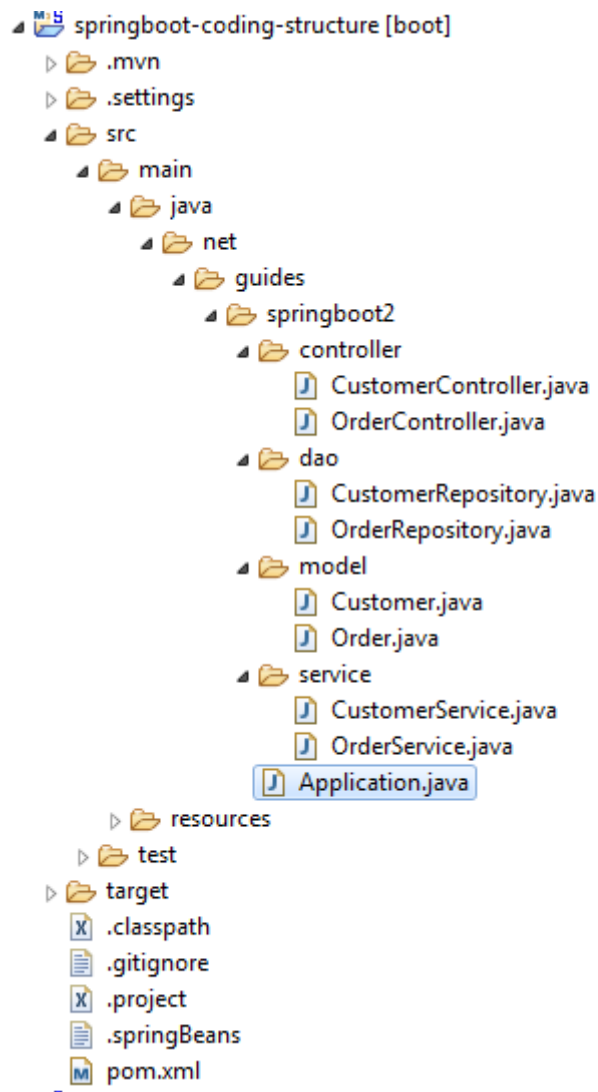
O Spring oferece suporte aos seguintes mecanismos de modelo por padrão. Esses modelos podem ser ativados usando iniciadores de inicialização de springboot apropriados.

- **FreeMarker** - spring-boot-starter-freemarker
- **Groovy** - spring-boot-starter-groovy
- **Thymeleaf** - spring-boot-starter-thymeleaf
- **Mustache** - spring-boot-starter-mustache

Todos esses mecanismos de modelo resolverão seus arquivos de modelo a partir do caminho src / *main* / *resources* / *template* .

## Estrutura do Projeto - Segunda abordagem

No entanto, a abordagem de layout típica acima funciona bem, mas alguns dos desenvolvedores preferem usar o seguinte pacote ou estrutura de projeto:



Pacotes separado para cada camada como um modelo, controller, dao e service, etc.

## pom.xml

Em um projeto spring boot, a maioria dos módulos pode ser habilitada ou desabilitada apenas adicionando um conjunto de iniciadores. Todos os projetos Spring Boot normalmente usam `spring-boot-starter-parent` como o pai no arquivo `pom.xml`.

## O que é o Spring Boot Starter Parent e como usá-lo?

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.3.3.RELEASE</version>
</parent>
```

**spring-boot-starter-parent** os permite gerenciar as seguintes coisas para vários projetos e módulos filho:

- Configuration - Versão Java e outras propriedades
- Dependency Management - versão das dependências
- Default Plugin Configuration

É necessário especificar apenas o número da versão do Spring Boot nesta dependência. Se você importar iniciadores adicionais, pode omitir com segurança o número da versão.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.4.RELEASE</version>
  <relativePath /> <!-- lookup parent from repository -->
</parent>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

**override default Java version from parent**

**specify only the Spring Boot version number on this dependency and omit the version number for additional starters**

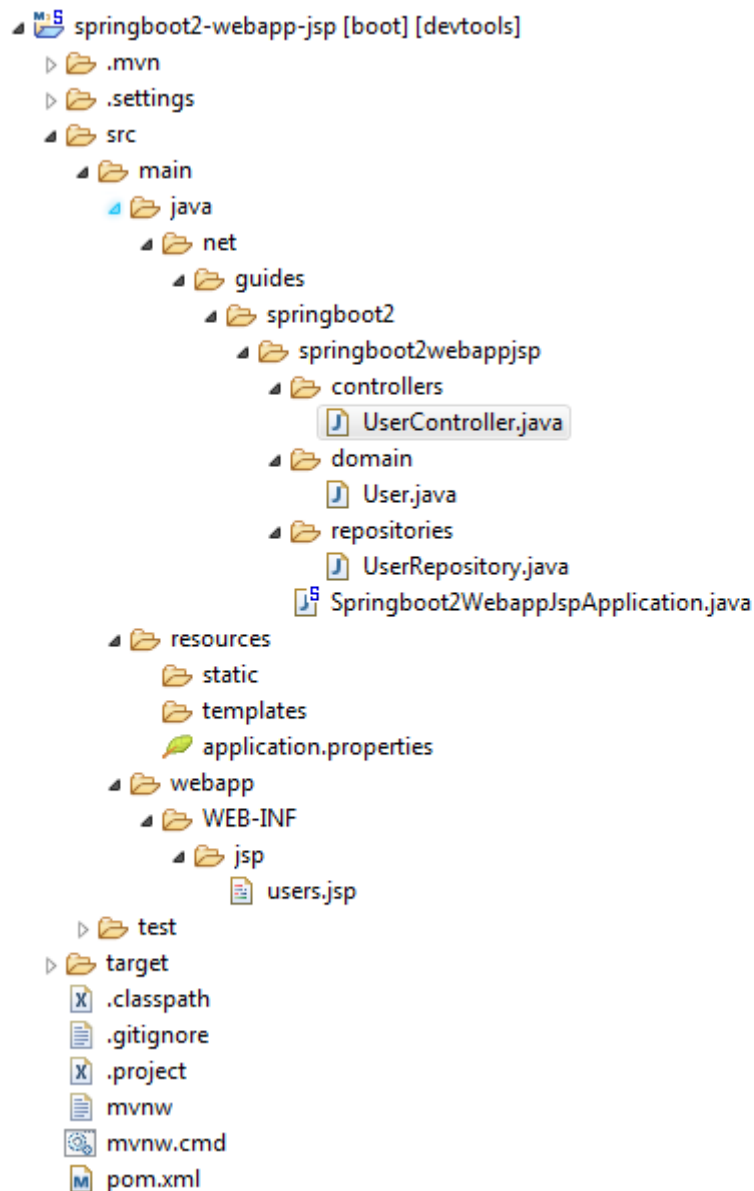
**omit version number**

Para um bom código devemos:

- Adicionar apenas os iniciadores necessários. Isso mantém o aplicativo mais leve. Um iniciador indesejado pode levar a grãos autowired extras .
- A maioria dos starters vem com suas próprias dependências transitivas. Portanto, talvez você nunca precise especificar as versões. A maioria dos IDE já destaca essas tags de versão indesejadas.
- Conhecer suas dependências iniciais. Dessa forma, você nunca precisará escrever configurações.

### **Aplicativos web spring boot - usando JSP como visualizações**

Se você estiver criando um aplicativo da web MVC Spring Boot usando JSP como visualizações, o diagrama a seguir mostra um layout de estrutura de projeto típico. Como prática recomendada, recomendo enfaticamente colocar seus arquivos JSP em um diretório no diretório 'WEB-INF' para que não haja acesso direto dos clientes.



## Arquitetura do projeto Spring Boot

Neste passo, observarmos a maneira de criar uma arquitetura de três camadas em projetos MVC típicos de Spring boot.

O Spring Boot facilita a criação de aplicativos baseados em Spring autônomos e de nível de produção que você pode "simplesmente executar". A arquitetura de três camadas (ou três camadas) é uma solução amplamente aceita para organizar a base de código. De acordo com essa arquitetura, a base de código

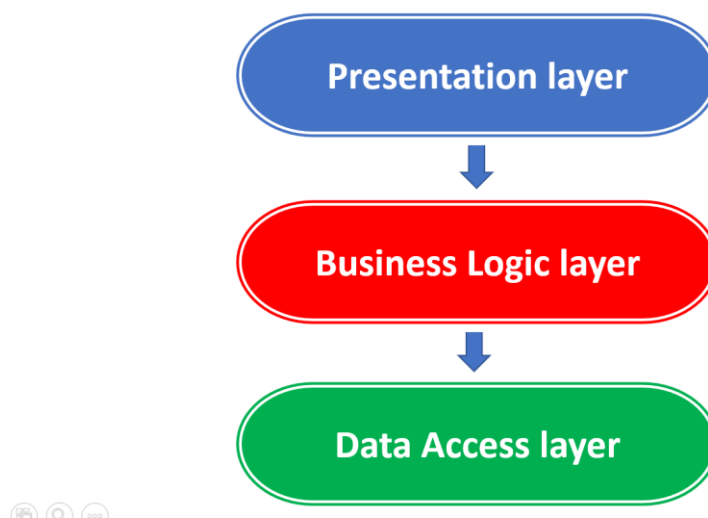
é dividida em três camadas separadas com responsabilidades distintas.

### Arquitetura de três camadas (três camadas)

Spring Boot segue uma arquitetura em camadas em que cada camada se comunica com a camada diretamente abaixo ou acima (estrutura hierárquica) dela:

## Three-Tier (or Three-Layer) Architecture

---



**Camada de apresentação:** é a interface do usuário do aplicativo que apresenta os recursos e os dados do aplicativo para o usuário.

**Camada de lógica de negócios (ou aplicativo):** essa camada contém a lógica de negócios que orienta as funcionalidades centrais do aplicativo. Como tomar decisões, cálculos, avaliações e processar os dados que passam entre as outras duas camadas.

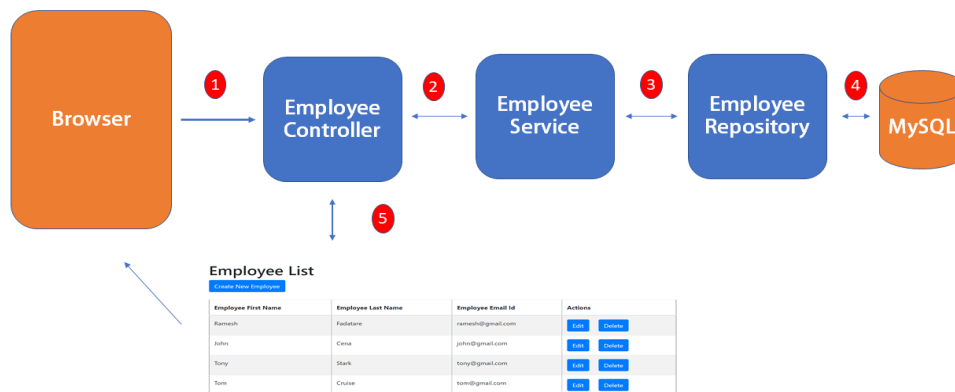
**Camada de acesso a dados (ou dados):** essa camada é responsável por interagir com os bancos de dados para salvar e recuperar dados do aplicativo.



## Arquitetura Spring Boot Flow (exemplo)

O diagrama abaixo mostra o fluxo de aplicativo típico de nosso aplicativo da web MVC Spring boot com Thymeleaf:

### Spring Boot Thymeleaf CRUD Database Real-Time Project



1. O controlador Spring MVC recebe uma solicitação HTTP do cliente (navegador).

2. O controlador Spring MVC processa a solicitação e envia essa solicitação para a camada de serviço.

3. Camada de serviço responsável por manter a lógica de negócios do aplicativo.

4. Camada de repositório responsável por interagir com bancos de dados para salvar e recuperar dados do aplicativo.

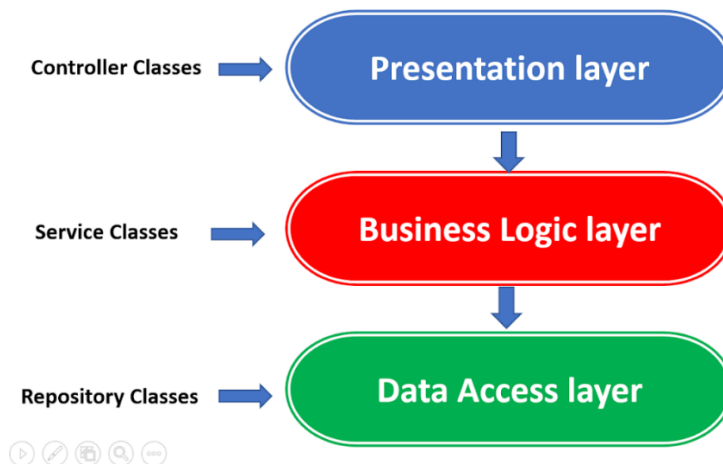
5. Visualização de retorno do controlador Spring MVC (JSP ou Thymeleaf) para renderizar no navegador.

## Como usar a arquitetura de três camadas em aplicativos web Spring Boot MVC.

Em um aplicativo web MVC Spring boot, as três camadas da arquitetura se manifestarão da seguinte maneira:

## Three-Tier (or Three-Layer) Architecture in Spring MVC

---



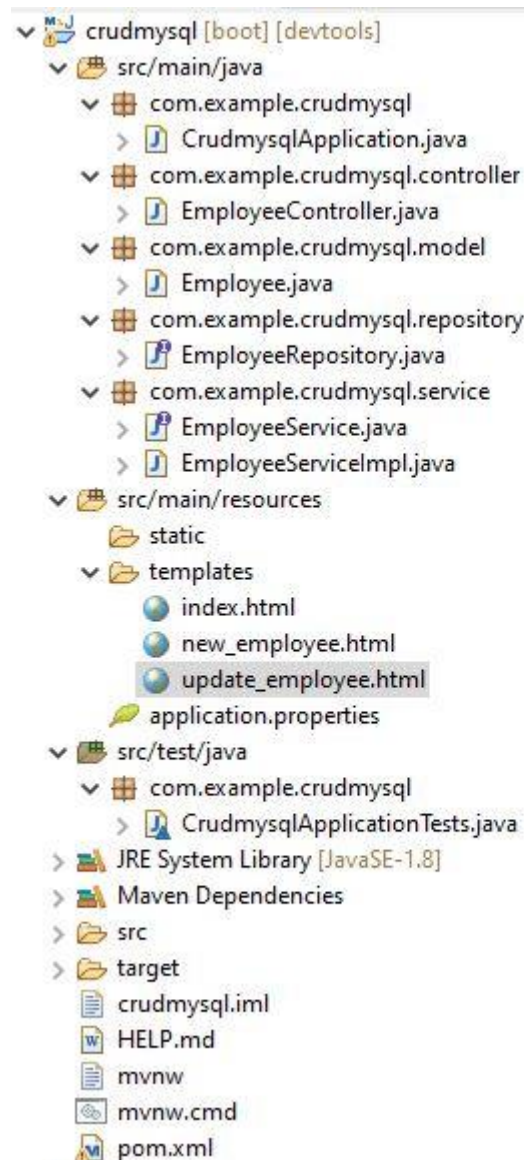
- **Classes de controlador (controller)** como camada de apresentação. Mantenha essa camada o mais fina possível e limitada à mecânica das operações MVC, por exemplo, receber e validar as entradas, manipular o objeto do modelo, retornar o objeto **ModelAndView** apropriado e assim por diante. Todas as operações relacionadas ao negócio devem ser feitas nas classes de serviço. As classes de controlador geralmente são colocadas em um pacote controller .
- **Classes de serviço (service)** como camada de lógica de negócios. Cálculos, transformações de dados, processos de dados e validações de registro cruzado (regras de negócios) geralmente são feitos nesta camada. Eles são chamados pelas classes do controlador e podem chamar repositórios ou outros serviços. As classes de serviço geralmente são colocadas em um pacote de serviço.
- **Classes de repositório(repository)** como camada de acesso a dados. A responsabilidade dessa camada é limitada às operações Criar, Recuperar, Atualizar e Excluir (CRUD) em uma fonte de dados, que geralmente é um banco de dados relacional ou não

relacional. As classes de repositório geralmente são colocadas em um pacote de repositório.

Considere a seguir o aplicativo da web Spring MVC usando Spring boot e Thymeleaf. Criamos uma arquitetura de três camadas e cada camada é mapeada para o pacote correspondente.

Por exemplo:

- Camada de apresentação - pacote **controlador (controller)**
- Camada de lógica de negócios - pacote de **serviços (service)**
- Camada de acesso a dados - pacote de **repositório (repository)**



## **Conclusão**

A arquitetura de três camadas (ou três camadas) é uma solução amplamente aceita para organizar a base de código. De acordo com essa arquitetura, a base de código é dividida e separada com responsabilidades distintas.

## **Java Spring Boot & Eclipse**

### **Pré-Requisitos**

Java JDK 8 ou superior

### **Instalando o Eclipse**

O site oficial do IDE Eclipse é [www.eclipse.org](http://www.eclipse.org), e na página de downloads podemos baixar a versão mais recente do IDE. Para preparar seu computador para desenvolvimento em Java, você precisará do JDK e de um ambiente de desenvolvimento integrado, como o Eclipse (ou outro, como o NetBeans). Vejamos os procedimentos necessários para instalar o Eclipse:

1 – Baixar e instalar o JDK (Java Development Kit) para que seja possível desenvolver aplicações em Java. Se você quiser usar o Eclipse para desenvolver em outras linguagens que não o Java, basta baixar e instalar o JRE (Java Runtime Environment) para que seja possível rodar o Eclipse, pois ele é escrito em Java também. A versão necessária é a 1.7.0 ou mais recente para rodar o Eclipse.

2 – Baixe o instalador do Eclipse para sua versão do Windows (32 ou 64 bits) no link a seguir:

**<http://www.eclipse.org/downloads/>**

Para o release Mars (versão 6) é possível usar um instalador, o que facilita um pouco o processo de instalação e configuração inicial do Eclipse para desenvolvimento em Java. Mesmo assim, isso não dispensa a etapa anterior, que consiste em baixar e instalar o JDK.

A imagem a seguir mostra a tela da página de downloads do site do Eclipse. Veja que também há versões para Mac OS X e Linux, além do Windows:



3 – Vamos proceder à instalação do Eclipse no Windows.

Execute o arquivo do instalador que foi baixado:



Na tela que se abrirá você deve escolher a primeira opção, “Eclipse IDE for Java Developers”, pois nosso objetivo é usar o IDE para aprender programação em Java.



Na tela seguinte, revise a pasta de destino da instalação, e os checkboxes para criação de entrada no menu iniciar e atalho no desktop – você pode desmarcar essas opções se não desejar esses atalhos. Clique então no botão “INSTALL” para iniciar o processo de instalação do Eclipse:



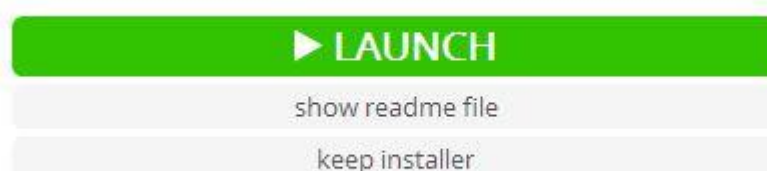
Aceite o Contrato de Licença clicando no botão “Accept Now”:



E aguarde enquanto o software é instalado em seu computador:



Após o término da instalação, clique no botão “LAUNCH” para iniciar o Eclipse pela primeira vez:

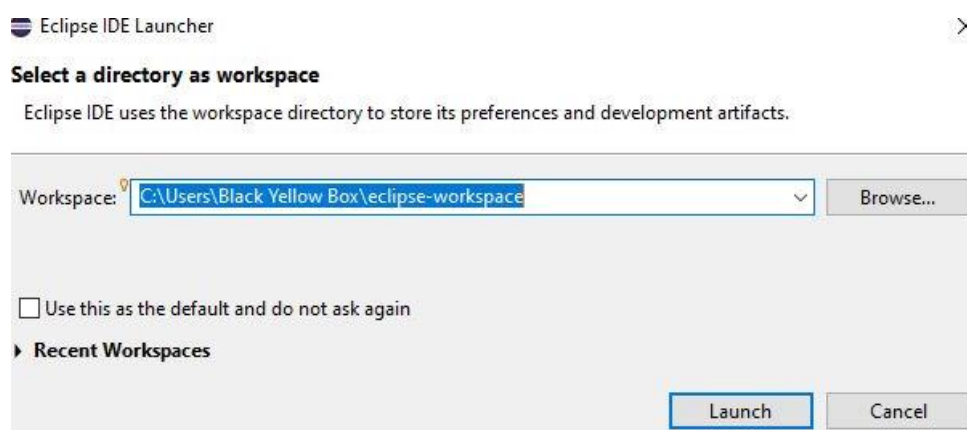


E aguarde enquanto o programa é carregado na memória:





Será perguntado onde seus projetos deverão ser armazenados. Para isso, o Eclipse cria uma pasta especial denominada **Workspace**. Escolha um local no seu disco, ou mantenha o padrão sugerido (dentro de seu diretório pessoal), marque a caixa “Use this as the default and do not ask again” para tornar a pasta padrão em futuros projetos e clique em **OK**:

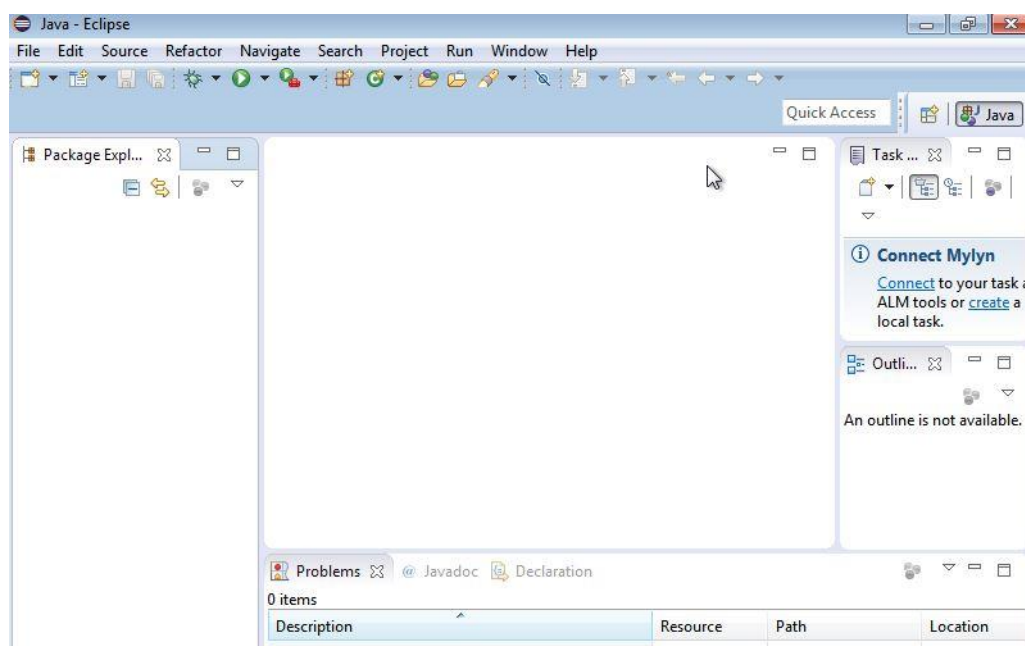


Você verá a tela de boas-vindas do **Eclipse IDE for Java**, onde há links para tutoriais, exemplos e outras informações sobre o software. Feche essa aba clicando no “x” ao lado da palavra Welcome, no lado superior esquerdo:





E aí está a tela principal do Eclipse.



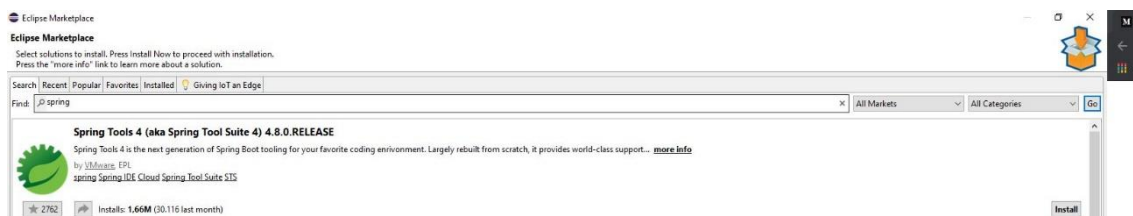
Vale lembrar que também é altamente válido instalar o [Spring Tool Suite \(STS\)](#) ao invés do eclipse puro. O STS é basicamente o eclipse configurado com o plugin [spring tools](#), otimizado para desenvolvimento com o spring framework

Utilize as seções abaixo de acordo com o sistema operacional que estiver utilizando:

## Configurando o eclipse

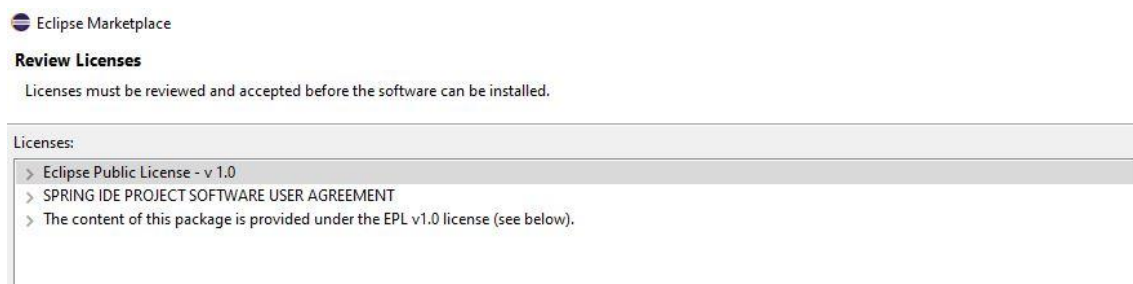
### Spring Tools

Para ter uma melhor experiência com desenvolvimento Spring, recomendo instalar o *Spring Tools*. Vá no menu *Help > Eclipse Marketplace...* e procure por spring:



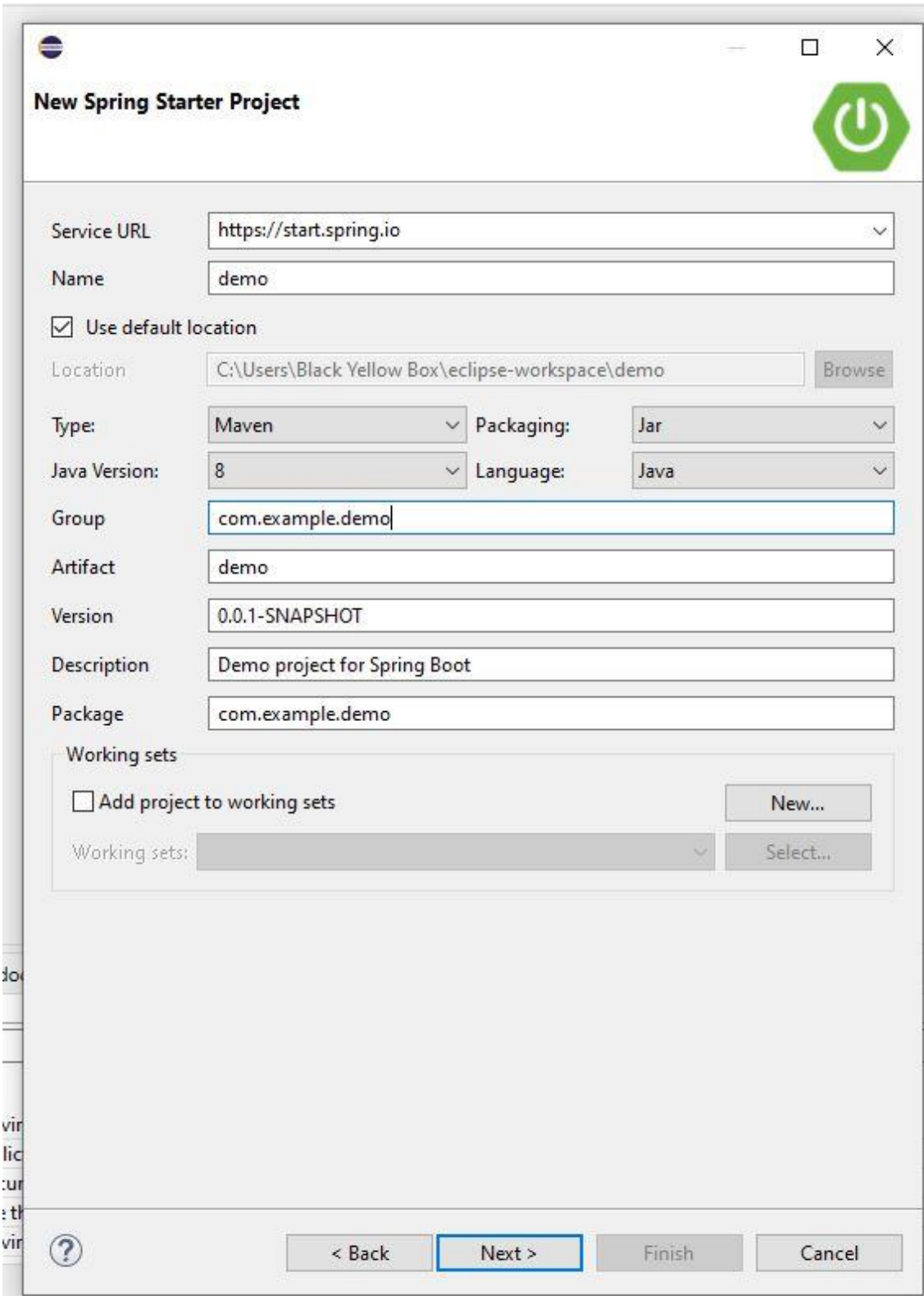
### Instalando Spring Tools

Instale a versão mais atual do *Spring Tools* (na data de escrita deste guia é a 4.2.1) e reinicie o eclipse.



### Criando o projeto

Com tudo configurado corretamente, hora de criarmos o projeto. Vá no menu: *File > New > Project...* e selecione a opção *Spring Starter Project* que está localizada abaixo do menu *Spring Boot*, conforme imagem abaixo:



The screenshot shows the 'New Spring Starter Project' dialog box in the Eclipse IDE. The dialog has a title bar with a question mark icon, a maximize button, and a close button. A green power button icon is in the top right corner. The main area contains several input fields and checkboxes. At the bottom, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'. The 'Next >' button is highlighted with a blue border.

**New Spring Starter Project**

Service URL:

Name:

☒ Use default location

Location:

Type:  Packaging:

Java Version:  Language:

Group:

Artifact:

Version:

Description:

Package:

Working sets

☐ Add project to working sets

Working sets:

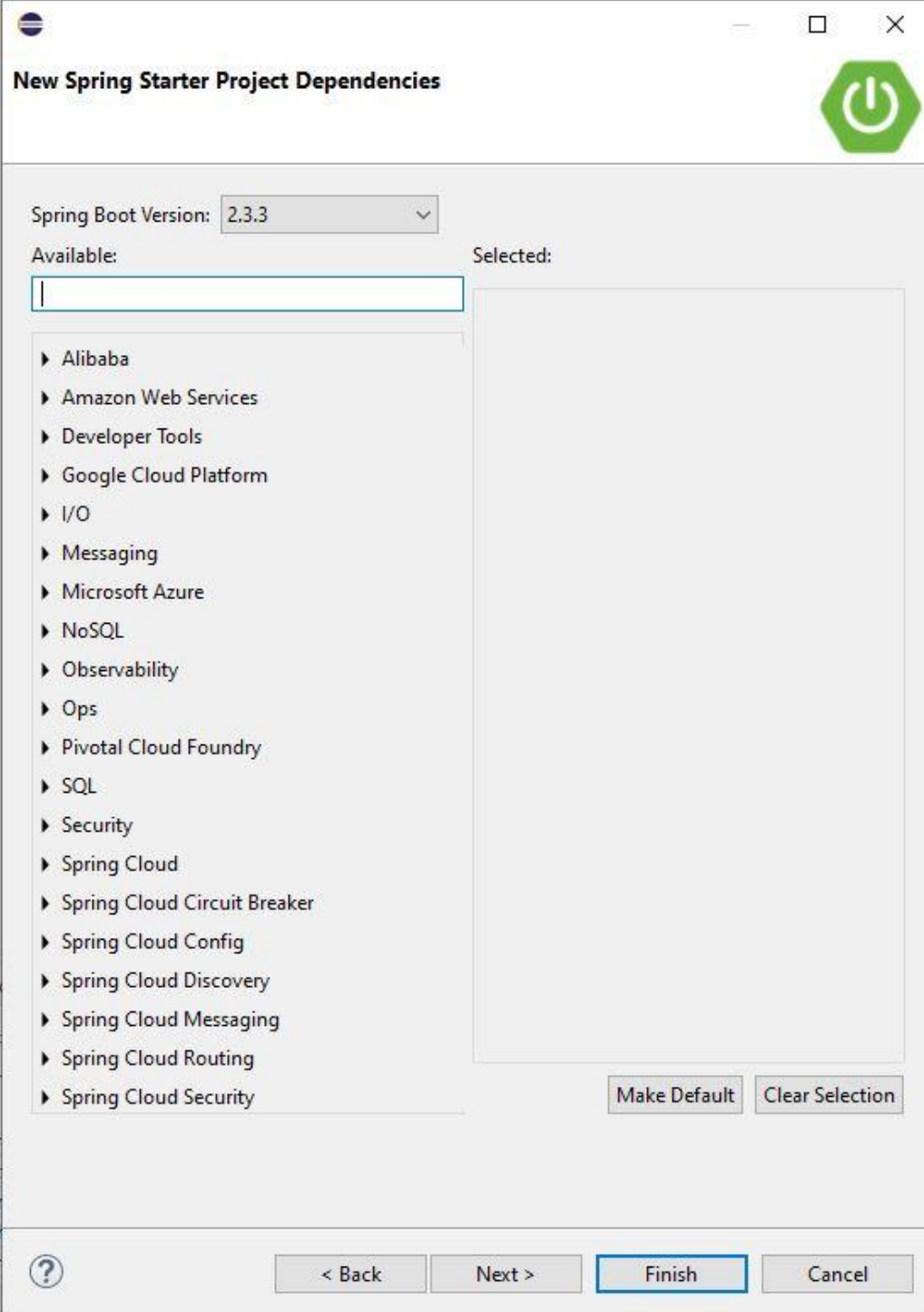
## Iniciando o projeto

Para este exemplo, utilizaremos como *build-tool* o [Maven](#).

Preencha os campos com o nome de seu artefato e o groupId, e selecione o campo *Type* como *Maven*

### **Inicialização do projeto utilizando o Maven**

Clicando no next, temos que especificar qual versão do *Spring* utilizar. É recomendável utilizar a última versão estável (na data de escrita deste guia é 2.1.4). Como dependência, selecione apenas a *Web* e clique em *Finish*.



The image shows a dialog box titled "New Spring Starter Project Dependencies". It features a green power button icon in the top right corner. The "Spring Boot Version" is set to "2.3.3". Below this, there are two columns: "Available:" and "Selected:". The "Available:" column contains a list of dependency categories, each preceded by a right-pointing triangle icon. The "Selected:" column is currently empty. At the bottom right of the dialog, there are two buttons: "Make Default" and "Clear Selection". At the very bottom, there is a row of four buttons: a help button (question mark in a circle), "< Back", "Next >", and "Finish" (which is highlighted with a blue border). A "Cancel" button is also present to the right of "Finish".

**New Spring Starter Project Dependencies**

Spring Boot Version: 2.3.3

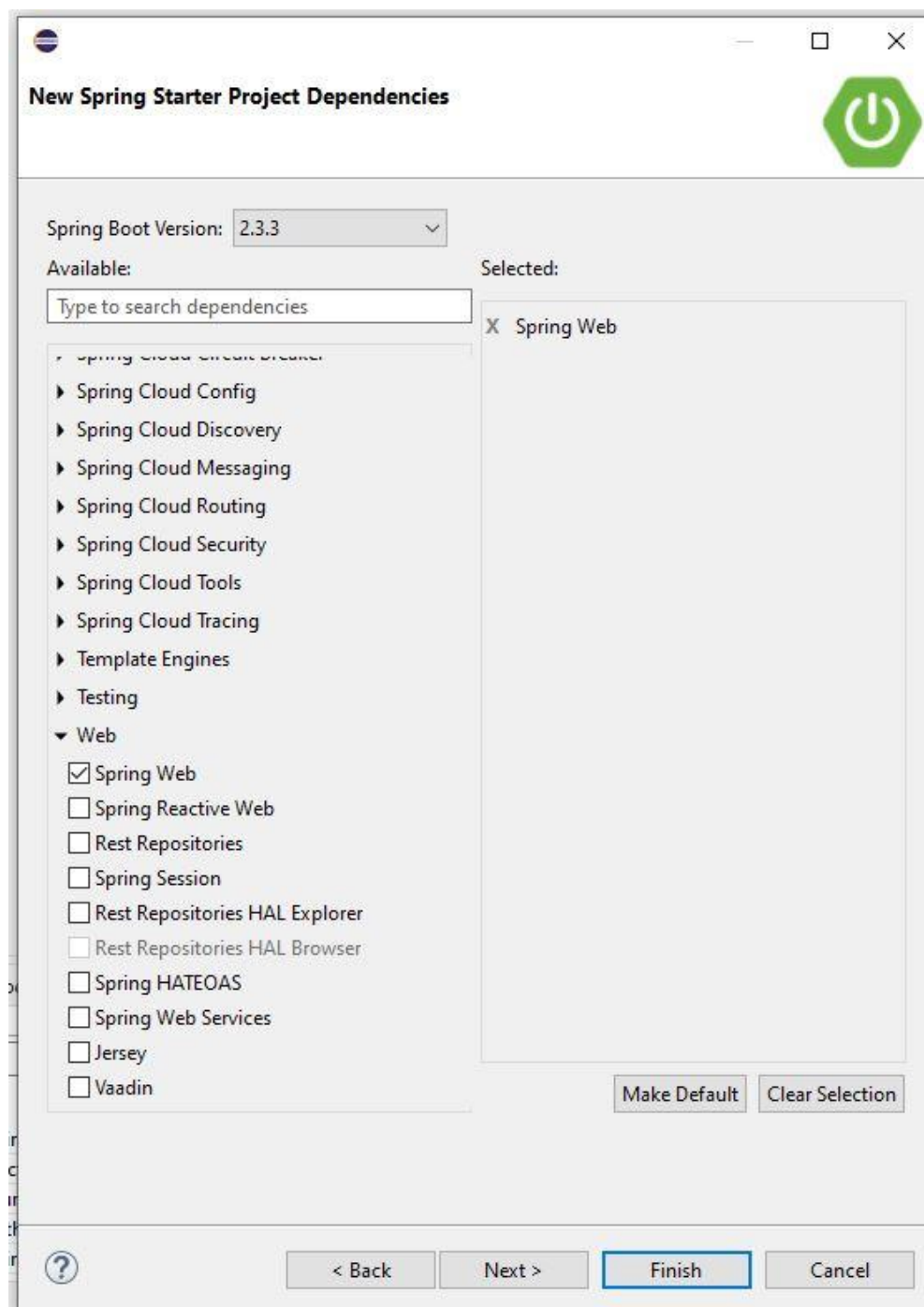
Available:

- ▶ Alibaba
- ▶ Amazon Web Services
- ▶ Developer Tools
- ▶ Google Cloud Platform
- ▶ I/O
- ▶ Messaging
- ▶ Microsoft Azure
- ▶ NoSQL
- ▶ Observability
- ▶ Ops
- ▶ Pivotal Cloud Foundry
- ▶ SQL
- ▶ Security
- ▶ Spring Cloud
- ▶ Spring Cloud Circuit Breaker
- ▶ Spring Cloud Config
- ▶ Spring Cloud Discovery
- ▶ Spring Cloud Messaging
- ▶ Spring Cloud Routing
- ▶ Spring Cloud Security

Selected:

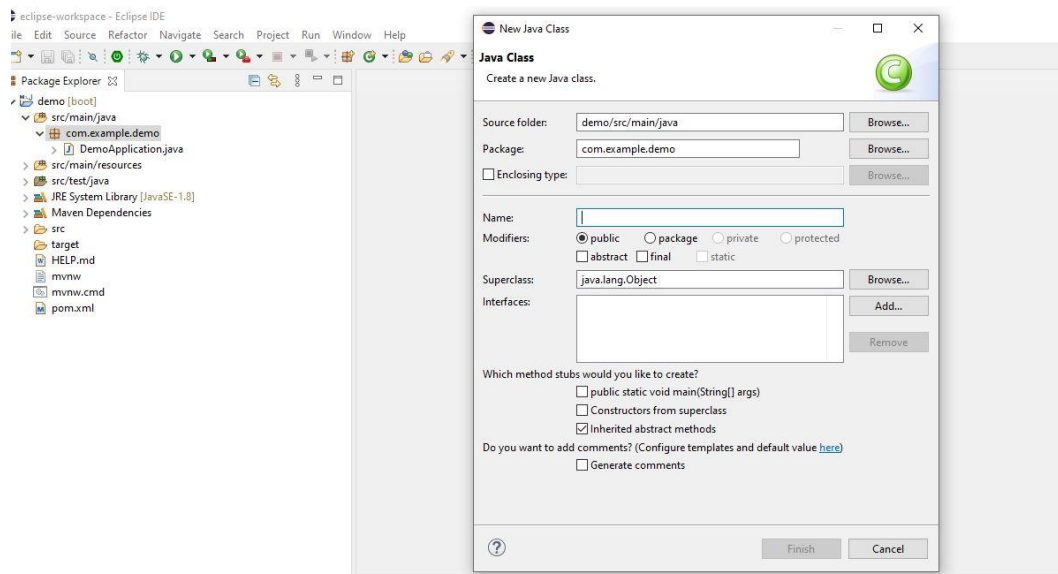
Make Default Clear Selection

? < Back Next > Finish Cancel



### Escolha da versão do Spring e dependências

Projeto criado, agora é hora de criar um *Controller* com um endpoint que irá retornar uma String fixa com o conteúdo: "Hello-World". Clique com o botão direito sobre o pacote principal e selecione *New > Class*



### Criando uma classe

E crie uma classe com o nome "ExampleController" e preencha o sufixo do pacote com ".controllers" assim irá ser criado um pacote *controllers* onde você poderá agrupar os controllers de seu projeto:

**New Java Class**

Create a new Java class.

Source folder: demo/src/main/java Browse...

Package: com.example.demo.controllers Browse...

☐ Enclosing type: Browse...

Name: ExampleController

Modifiers: ☒ public ☐ package ☐ private ☐ protected  
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add...  
Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)  
☐ Constructors from superclass  
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))  
☐ Generate comments

? Finish Cancel

Utilize o código abaixo para criar o endpoint que irá retornar "Hello World!":

```
package com.example.demo.controllers;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * ExampleController
 */
```



```

*
*/
@RestController
@RequestMapping("/api/example")
public class ExampleController {

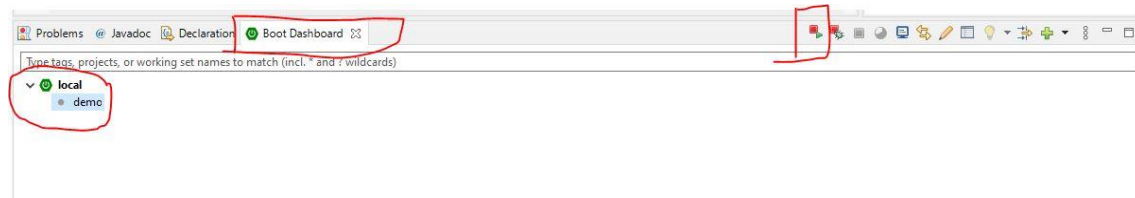
    @GetMapping("/hello-world")
    public ResponseEntity<String> get() {
        return ResponseEntity.ok("Hello World!");
    }
}

```

Execute o projeto, primeiramente clicando no botão *Boot Dashboard* e depois na guia *Boot Dashboard* na parte inferior da tela, selecione o nome do seu projeto e clique no botão de executar ao lado:



Após clicar no botão de executar, você deverá ver a saída do log de inicialização do spring na aba console:



```

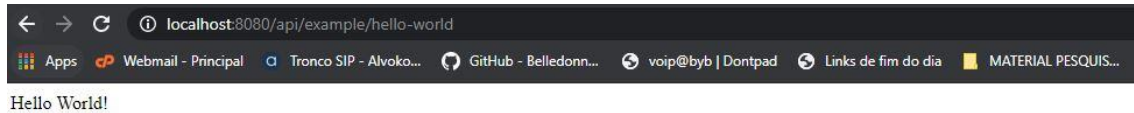
demo - DemoApplication [Spring Boot App] C:\Program Files\Java\jdk-1.8.0_221\bin\javaw.exe (16/09/2020 15:28:07)

:: Spring Boot ::
(v2.3.3.RELEASE)

2020-09-16 15:28:09.760 INFO 12180 --- [main] com.example.demo.DemoApplication : Starting DemoApplication on DESKTOP-1FQLTH with PID 12180 (C:\User
2020-09-16 15:28:09.763 INFO 12180 --- [main] com.example.demo.DemoApplication : No active profile set, falling back to default profiles: default
2020-09-16 15:28:10.735 INFO 12180 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-09-16 15:28:10.742 INFO 12180 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-09-16 15:28:10.743 INFO 12180 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.37]
2020-09-16 15:28:10.815 INFO 12180 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-09-16 15:28:10.815 INFO 12180 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1007 ms
2020-09-16 15:28:10.974 INFO 12180 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-09-16 15:28:11.161 INFO 12180 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2020-09-16 15:28:11.171 INFO 12180 --- [main] com.example.demo.DemoApplication : Started DemoApplication in 1.747 seconds (JVM running for 2.725)

```

Para testar o funcionamento basta abrir seu browser favorito e acessar: <http://localhost:8080/api/example/hello-world> e você deverá ver a mensagem de *Hello World*:



O Eclipse e também o Spring Tool Suite são IDEs ótimas para desenvolvimentos de aplicações robustas baseadas em java e spring, e devido a maturidade da ferramenta, acaba sendo super fácil o start de um projeto simples.

## Criando projeto Spring Boot com Spring Initializer

Neste passo, aprenderemos como criar um projeto simples de boot Spring com Spring Initializr .

Spring Initializr é uma ótima ferramenta desenvolvida pela equipe Spring para inicializar rapidamente seus projetos Spring Boot. Existem muitas maneiras de criar um aplicativo Spring Boot. A maneira mais simples é usar Spring Initializr em <http://start.spring.io/>, que é um gerador de aplicativos Spring Boot online.

Vamos criar um projeto simples Spring boot – seu nome será *hello world* - usando a ferramenta Spring Initializr e importar no Eclipse IDE.

### Criar projeto Spring Boot com Spring Initializr

A imagem abaixo mostra o processo passo a passo para criar um projeto de boot **Spring** usando **Spring Initializr** .

**Etapa 1.** Inicie o Spring Initializr usando o link <https://start.spring.io>

The screenshot shows the Spring Initializr web application interface. It features a sidebar with a hamburger menu and a Twitter icon. The main content area is divided into several sections:

- Project:** Includes radio buttons for **Maven Project** (selected), **Gradle Project**, and **Language** options: **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for various versions: **2.4.0 (SNAPSHOT)**, **2.4.0 (M2)**, **2.3.4 (SNAPSHOT)**, **2.3.3** (selected), **2.2.10 (SNAPSHOT)**, **2.2.9**, **2.1.17 (SNAPSHOT)**, and **2.1.16**.
- Project Metadata:** Includes input fields for **Group** (com.example), **Artifact** (demo), **Name** (demo), **Description** (Demo project for Spring Boot), and **Package name** (com.example.demo).
- Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
- Dependencies:** Includes a button **ADD DEPENDENCIES... CTRL + B** and the text *No dependency selected*.

At the bottom, there are three buttons: **GENERATE CTRL + G**, **EXPLORE CTRL + SPACE**, and **SHARE...**.

## Etapa 2. Especificar os detalhes do projeto

Observe o diagrama acima, faremos o preenchimento dos campos seguindo as especificamos e os seguintes detalhes:

- Project: Maven
- Language: Java
- Spring Boot: 2.3.3
- Java: 8
- Group: com.example
- Artifact: demo
- Name: demo
- Description: Demo Project for Spring Boot
- Package name: com.example.demo
- package: jar (este é o valor padrão)
- Dependencies: Web, JPA, MySQL

Depois que todos os detalhes forem inseridos, clique no botão Generate para gerar um projeto de spring initializr e baixá-lo. Em seguida, descompacte o arquivo zip baixado e importe-o em seu IDE favorito.

**Project**☒ Maven Project ☐ Gradle Project**Language**☒ Java ☐ Kotlin ☐ Groovy**Spring Boot**

☐ 2.4.0 (SNAPSHOT) ☐ 2.4.0 (M2) ☐ 2.3.4 (SNAPSHOT) ☒ 2.3.3  
☐ 2.2.10 (SNAPSHOT) ☐ 2.2.9 ☐ 2.1.17 (SNAPSHOT) ☐ 2.1.16

**Project Metadata**Group Artifact Name Description Package name Packaging ☒ Jar ☐ WarJava ☐ 14 ☐ 11 ☒ 8

|Web, Security, JPA, Actuator, Devtools... Press Ctrl for multiple adds

**DEVELOPER TOOLS**

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

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

**Spring Configuration Processor**  
Generate metadata for developers to offer contextual help and "code completion" when working with custom configuration keys (ex.application.properties/.yaml files).

**WEB**

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

**Spring Reactive Web**  
Build reactive web applications with Spring WebFlux and Netty.

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

**Spring Session**

**Dependencies** **ADD DEPENDENCIES... CTRL + B**

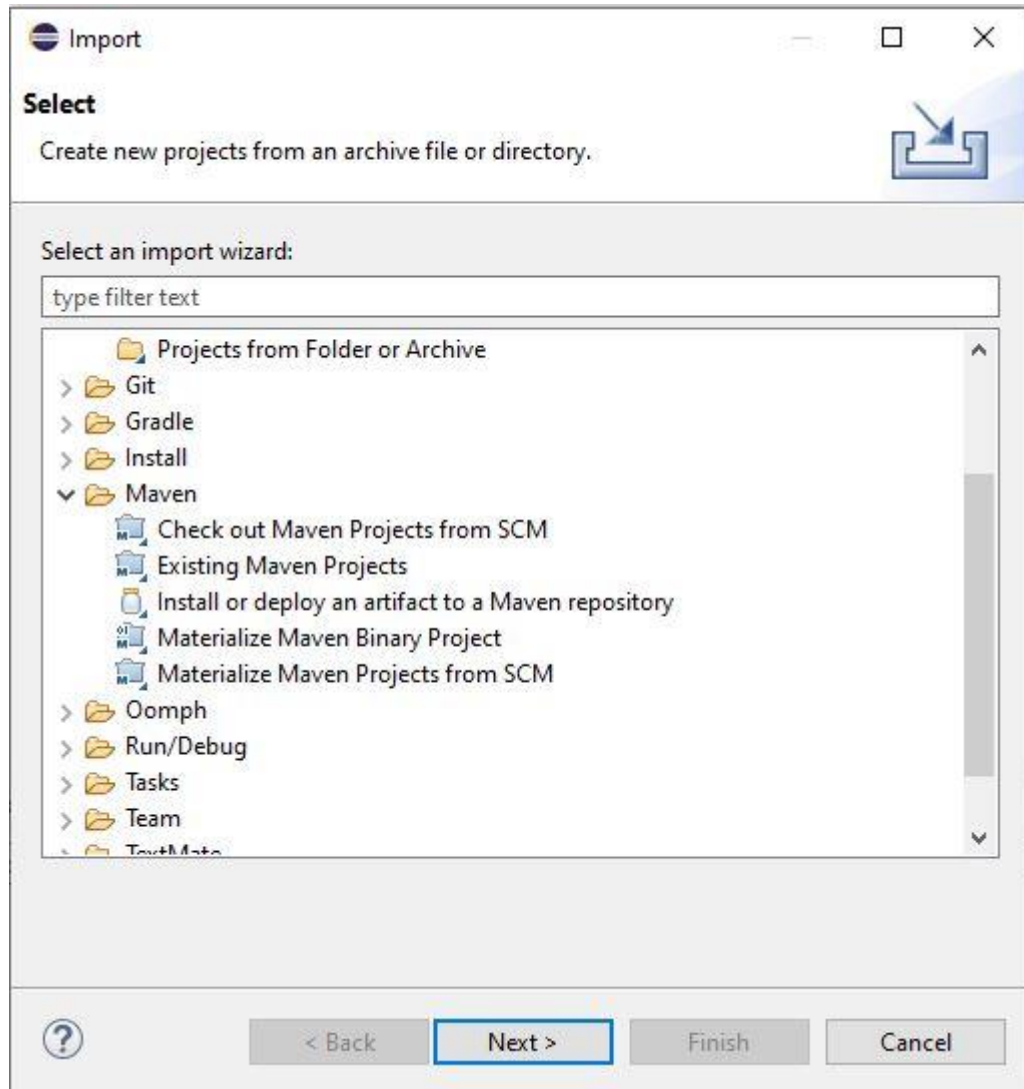
**Spring Web** **WEB**  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Agora, basta fazer o download do projeto clicando no botão GENERATE:

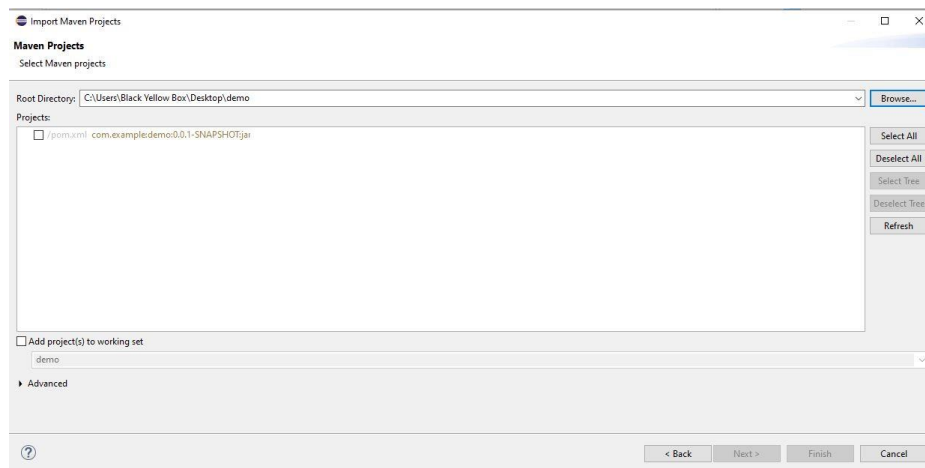
**GENERATE** CTRL + G **EXPLORE** CTRL + SPACE **SHARE...**

### 3. Importar projeto no Eclipse

No Eclipse, clique em File-> Import -> Projeto Maven Existente conforme mostrado abaixo.



Navegue ou digite o caminho da pasta para onde você extraiu o arquivo zip na próxima tela.



Depois de clicar em Finish, o Maven levará algum tempo para baixar todas as dependências e inicializar o projeto.

## Web Service Spring Restful

Neste projeto, aprenderemos como desenvolver um aplicativo de serviço da Web RESTful simples usando Spring Boot. Usamos o Maven para construir este projeto, pois a maioria dos IDEs o suportam.

Spring Boot a partir do 2.0.5.RELEASE requer Java 8 ou 9, Spring Framework 5.0.9.RELEASE ou superior e Maven 3.2+, portanto, antes de começarmos, abra um terminal e execute os seguintes comandos para garantir que você tenha versões válidas de Java e Maven instalado:

### Para verificar a versão do Java:

```
$ java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14,
mixed mode)
```

### Para verificar a versão do Maven:

```
$ mvn -v
Apache Maven 3.3.9
(b52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-
10T16:41:47+00:00)
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java version: 1.8.0_102, vendor: Oracle Corporation
```

### O que vamos construir?

Vamos construir um serviço que aceitará solicitações HTTP GET em:

```
http://localhost:8080/greeting
```

e responder com uma representação JSON de uma saudação:

```
{"id":1,"content":"Hello, World!"}
```

Podemos personalizar a saudação com um parâmetro de nome opcional na string de consulta:

```
http://localhost:8080/greeting?name=User
```

O valor do parâmetro name substitui o valor padrão de "World" e é refletido na resposta:

```
{"id":1,"content":"Hello, User!"}
```

### Criar projeto com Spring Initializr

Existem muitas maneiras de criar um aplicativo Spring Boot. A maneira mais simples é usar Spring Initializr em <http://start.spring.io/>, que é um gerador de aplicativos Spring Boot online.



The screenshot shows the Spring Initializr web interface. At the top left is the Spring logo and 'spring initializr' text. Below it, the 'Project' section has 'Maven Project' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '2.3.4' selected. The 'Project Metadata' section contains the following fields: Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), and Package name (com.example.demo). The 'Packaging' section has 'Jar' selected. At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'. A 'Dependencies' section on the right is empty with the text 'No dependency selected' and a button 'ADD DEPENDENCIES... CTRL + B'.

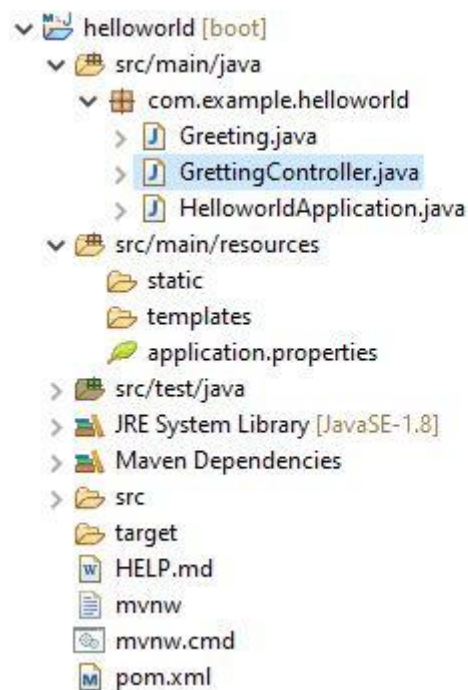
Observe o diagrama acima, especificamos os seguintes detalhes:

- Project: Maven
- Language: Java
- Spring Boot: 2.3.4
- Java: 8
- Group: com.example.
- Artifact: helloworld
- Name: helloworld
- Description: Demo Project for Spring Boot
- Package name: com.example.helloworld
- package: jar (este é o valor padrão)
- Dependencies: Web, JPA, MySQL

Depois que todos os detalhes forem inseridos, clique no botão Gerar projeto para gerar um projeto de inicialização de spring e baixá-lo. Em seguida, descompacte o arquivo zip baixado e importe-o em seu IDE favorito.

### Estrutura do Diretório do Projeto

A seguir está a estrutura de embalagem deste aplicativo para sua referência



### O arquivo pom.xml

Observe o arquivo pom.xml gerado com seu projeto. Faça as substituições, se necessárias, de acordo com o código abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId> com.example.helloworld</groupId>
  <artifactId> helloworld</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>helloworld<name>
  <description>Demo project for Spring
Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-
parent</artifactId>
    <version> 2.3.4.RELEASE</version>
    <relativePath/> <!-- lookup parent from
repository -->
  </parent>
```

```

    <properties>
      <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
      <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
      <java.version>1.8</java.version>
    </properties>

    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
web</artifactId>
      </dependency>

      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
test</artifactId>
        <scope>test</scope>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <plugin>

<groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-
plugin</artifactId>
        </plugin>
      </plugins>
    </build>
  </project>

```

No pom.xml acima, vamos entender alguns recursos importantes do Spring Boot.

### Plug-in Spring Boot Maven

O plugin Spring Boot Maven oferece muitos recursos convenientes:

- Ele coleta todos os jars no classpath e cria um único "über-jar" executável, que torna mais conveniente executar e transportar seu serviço.
- Ele procura o método public static void *main* () para sinalizar como uma classe executável.

- Ele fornece um resolvidor de dependências integrado que define o número da versão para corresponder às dependências do Spring Boot. Você pode substituir qualquer versão que desejar, mas o padrão será o conjunto de versões escolhido pelo Boot.

### **spring-boot-starter-parent**

Todos os projetos Spring Boot normalmente usam spring-boot-starter-parent como o pai em pom.xml.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-
parent</artifactId>
  <version> 2.3.4.RELEASE</version>
</parent>
```

Parent Poms permitem que você gerencie as seguintes coisas para vários projetos e módulos filhos:

- Configuração - Versão Java e outras propriedades
- Gerenciamento de dependências - versão das dependências
- Configuração de Plugin Padrão

### **Cria uma classe de representação de recurso - Greeting.java**

Agora que configuramos o projeto e o sistema de construção, podemos criar seu serviço da web.

Comece o processo pensando nas interações de serviço.

O serviço manipulará solicitações GET para / *greeting* , opcionalmente com um parâmetro de nome na string de consulta. A solicitação GET deve retornar uma resposta 200 OK com JSON no corpo que representa uma saudação. Deve ser parecido com isto:

```
{
    " id " : 1 ,
    " content " : " Hello, World! "
}
```

O campo *id* é um identificador exclusivo da saudação e o conteúdo é a representação textual da saudação.

Para modelar a representação da saudação, você cria uma classe de representação de recursos. Vamos implementar um objeto Java simples com campos, construtores e “acessadores” para os dados de id e de conteúdo:

```
public class Greeting {
    private final long id;
    private final String content;

    public Greeting(long id, String content) {
        this.id = id;
        this.content = content;
    }

    public long getId() {
        return id;
    }

    public String getContent() {
        return content;
    }
}
```

O Spring usa a biblioteca Jackson JSON para empacotar automaticamente instâncias do tipo *Greeting* em JSON.

A seguir, criamos um controlador (controller) de recursos que atenderá a essas saudações.

### **Criando um controlador de recurso - GreetingController.java**

Na abordagem do Spring para construir serviços da web RESTful, as solicitações HTTP são tratadas por um controlador. Esses componentes são facilmente identificados pela annotation *@RestController*, e

o *GreetingController* abaixo lida com solicitações GET para / *greeting* , retornando uma nova instância da classe *Greeting* :

```
import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping("/greeting")
    public Greeting greeting(@RequestParam(value = "name",
defaultValue = "World") String name) {
        return new Greeting(counter.incrementAndGet(),
String.format(template, name));
    }
}
```

Vamos entender o controlador acima passo a passo.

- A annotation *@RequestMapping* garante que as solicitações HTTP para / *greeting* sejam mapeadas para o método *greeting ()* .
- O exemplo acima não especifica GET vs. PUT, POST e assim por diante, porque *@RequestMapping* mapeia todas as operações HTTP por padrão. Use *@RequestMapping (method = GET)* para restringir esse mapeamento.
- *@RequestParam* vincula o valor do nome do parâmetro da string de consulta ao parâmetro *name* do método *greeting ()* . Se o parâmetro de nome estiver ausente na solicitação, o *defaultValue* de "World" é usado.
- A implementação do corpo do método cria e retorna um novo objeto *Greeting* com atributos *id* e *content* com base no próximo valor do contador e formata o nome fornecido usando o modelo de saudação.
- Uma diferença importante entre um controlador MVC tradicional e o controlador de serviço da web RESTful acima é a maneira como o corpo de resposta HTTP é criado. Em vez de depender de uma tecnologia de visualização para realizar a renderização do lado do

servidor dos dados de saudação para HTML, este controlador de serviço da Web RESTful simplesmente preenche e retorna um objeto de saudação. Os dados do objeto serão gravados diretamente na resposta HTTP como JSON.

- Este código usa a nova annotation *@RestController* do Spring 4, que marca a classe como um controlador onde cada método retorna um objeto de domínio em vez de uma visualização. É uma abreviação de *@Controller* e *@ResponseBody* juntos.
- O objeto *Greeting* deve ser convertido em JSON. Graças ao suporte ao conversor de mensagem HTTP do Spring, você não precisa fazer essa conversão manualmente. Como Jackson 2 está no classpath, *MappingJackson2HttpMessageConverter* do Spring é escolhido automaticamente para converter a instância de Saudação em JSON.

### O arquivo que faz com que nossa aplicação seja executável - HelloWorldApplication.java

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

@SpringBootApplication
public class HelloWorldApplication {

    public static void main(String[] args) {

SpringApplication.run(SpringbootHelloWorldApplication.class, args);
    }
}
```

*@SpringBootApplication* é uma annotation de conveniência que adiciona os seguintes elementos:

- *@Configuration*: marca a classe como uma fonte de definições de bean para o contexto do aplicativo.
- *@EnableAutoConfiguration*: diz ao Spring Boot para começar a adicionar beans com base nas configurações de classpath, outros beans e várias configurações de propriedade.

Normalmente você adicionaria `@EnableWebMvc` para um aplicativo Spring MVC, mas Spring Boot adiciona automaticamente quando vê *spring-webmvc* no classpath. Isso sinaliza o aplicativo como um aplicativo da web e ativa comportamentos-chave, como configurar um `DispatcherServlet`.

- `@ComponentScan`: diz ao Spring para procurar outros componentes, configurações e serviços no pacote `hello`, permitindo que ele encontre os controladores.

O método `main()` usa o método `SpringApplication.run()` do Spring Boot para iniciar um aplicativo. É possível observar que não há uma única linha de XML. Nenhum arquivo `web.xml` também. Esta aplicação web é 100% Java puro e você não teve que se preocupar com a configuração de qualquer infraestrutura.

### Executando o aplicativo

Neste projeto, as duas maneiras de iniciar o aplicativo autônomo de Spring Initializr.

1.No diretório raiz do aplicativo e digite o seguinte comando para executá-lo -

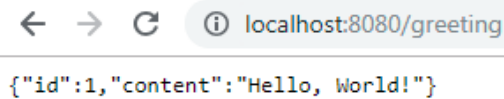
```
$ mvn spring-boot:run
```

Em seu IDE, execute o método `SpringbootHelloWorldApplication.main()` como uma classe Java autônoma que iniciará o servidor Tomcat incorporado na porta 8080 e aponte o navegador para <http://localhost:8080/>.

### Teste o serviço

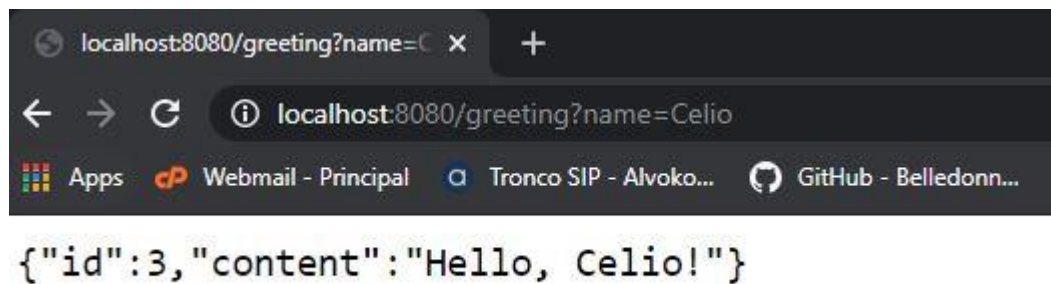
Agora que o serviço está ativo, visite <http://localhost:8080/greeting>, onde você verá:





```
{"id":1,"content":"Hello, World!"}
```

Forneça um parâmetro de string de consulta de nome com [http: // localhost: 8080 / greeting? Name = User](http://localhost:8080/greeting?name=User) . Observe como o valor do atributo content muda de "Hello, World!" para "Olá, usuário!":



```
{"id":3,"content":"Hello, Celio!"}
```

Essa alteração demonstra que a disposição *@RequestParam* em *GreetingController* está funcionando conforme o esperado. O parâmetro name recebeu o valor padrão "World", mas sempre pode ser substituído explicitamente por meio da string de consulta.

## Resumo

Você acabou de desenvolver um serviço da web RESTful com Spring.

## Springboot Annotations

Neste passo, vamos entender as annotations dos pacotes *org.springframework.boot.autoconfigure* e *org.springframework.boot.autoconfigure.condition*.

Como sabemos, o Spring Boot é uma nova estrutura da equipe da Pivotal, projetada para simplificar o bootstrap e o desenvolvimento de um novo aplicativo Spring. O framework adota uma abordagem opinativa para configuração, liberando os desenvolvedores da necessidade de definir uma configuração padrão. Ele fornece padrões para configuração de código e annotation para iniciar rapidamente novos projetos Spring em nenhum momento.

Um projeto Spring fornece muitas annotations para autoconfiguração dos pacotes

*org.springframework.boot.autoconfigure* e *org.springframework.boot.autoconfigure.condition*. Vamos listar todas as annotations desses pacotes.

## Spring Boot Annotations

1. @SpringBootApplication
2. @EnableAutoConfiguration
3. @ConditionalOnClass and @ConditionalOnMissingClass
4. @ConditionalOnBean and @ConditionalOnMissingBean
5. @ConditionalOnProperty
6. @ConditionalOnResource
7. @ConditionalOnWebApplication and @ConditionalOnNotWebApplication
8. @ConditionalExpression
9. @Conditional

### 1. @SpringBootApplication

A annotation @SpringBootApplication indica uma classe de configuração que declara um ou mais métodos @Bean e também aciona a configuração automática e a varredura de componente.

A annotation @SpringBootApplication é equivalente a usar @Configuration , @EnableAutoConfiguration e *@ComponentScan* com seus atributos padrão.



Exemplo de annotation *@SpringBootApplication* : Usamos esta annotation para marcar a classe principal de um aplicativo Spring Boot:

```

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

@SpringBootApplication // igual a @Configuration
@EnableAutoConfiguration @ComponentScan
public class Application {
    public static void main ( String [] args ) {
  
```

```

        SpringApplication . run ( Application . class, args);
    }
}

```

## 2. @EnableAutoConfiguration

A annotation `@EnableAutoConfiguration` diz ao Spring Boot para “adivinhar” como você deseja configurar o Spring, com base nas dependências jar que você adicionou. Como a dependência `spring-boot-starter-web` adicionada ao classpath leva à configuração do Tomcat e do Spring MVC, a configuração automática assume que você está desenvolvendo um aplicativo da web e configura o Spring de acordo.

Exemplo de annotation `@EnableAutoConfiguration`: vamos adicionar a annotation `@EnableAutoConfiguration` à classe da aplicação ou classe principal para habilitar um recurso de configuração automática.

```

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

@EnableAutoConfiguration
public class Application {
    public static void main ( String [] args ) {
        SpringApplication . run ( Application . class, args);
    }
}

```

## 3. @ConditionalOnClass e @ConditionalOnMissingClass

Essas annotations pertencem às condições de classe. Os `@ConditionalOnClass` e `@ConditionalOnMissingClass` annotations deixar configuração ser incluído com base na presença ou ausência de classes específicas.

**Exemplo:** No exemplo abaixo, usando essas condições, o Spring só usará o bean de configuração automática marcado se a classe no argumento da annotation estiver presente / ausente:

```

@Configuration
@ConditionalOnClass ( DataSource . Class)
class OracleAutoconfiguration {
    // ...
}

```

#### 4. @ConditionalOnBean e @ConditionalOnMissingBean

Os

annotations *@ConditionalOnBean* e *@ConditionalOnMissingBean* deixam de incluir um bean com base na presença ou ausência de bean específicos.

Exemplo de annotation *@ConditionalOnBean* : use quando quisermos definir condições com base na presença ou ausência de um bean específico:

```

@Bean
@ConditionalOnBean ( name = " dataSource " )
LocalContainerEntityManagerFactoryBean entityManagerFactory ()
{
    // ...
}

```

Exemplo de annotation *@ConditionalOnMissingBean* : quando colocado em um método *@Bean* , o tipo de destino é padronizado para o tipo de retorno do método, conforme mostrado no exemplo a seguir:

```

@Configuration
public class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public MyService myService () { ... }
}

```

No exemplo anterior, o bean *myService* será criado se nenhum bean do tipo *MyService* já estiver contido no *ApplicationContext* .

#### 5. @ConditionalOnProperty

A annotation `@ConditionalOnProperty` permite que a configuração seja incluída com base em uma propriedade `Spring Environment`.

Exemplo de annotation `@ConditionalOnProperty`: Com esta annotation, podemos fazer condições nos valores das propriedades:

```
@Bean
@ConditionalOnProperty (
    name = " usemysql " ,
    havingValue = " local "
)
DataSource dataSource () {
    // ...
}
```

## 6. @ConditionalOnResource

A annotation `@ConditionalOnResource` permite que a configuração seja incluída apenas quando um recurso específico estiver presente:

```
@ConditionalOnResource ( resources = " classpath: mysql.properties "
)
Propriedades additionalProperties () {
    // ...
}
```

## 7. @ConditionalOnWebApplication e @ConditionalOnNotWebApplication

Os

annotations `@ConditionalOnWebApplication` e `@ConditionalOnNotWebApplication` deixar de configuração ser incluídos dependendo se o aplicativo é uma “aplicação web”. Um aplicativo da web é um aplicativo que usa um Spring `WebApplicationContext`, define um escopo de sessão ou tem um `StandardServletEnvironment`.

Código de exemplo de annotation `@ConditionalOnWebApplication`: com essas annotations,

podemos criar condições com base em se o aplicativo atual é ou não um aplicativo da web:

```
@ConditionalOnWebApplication
HealthCheckController healthCheckController () {
    // ...
}
```

## 8. @ConditionalExpression

Podemos usar essa annotation em situações mais complexas. O Spring usará a definição marcada quando a *expressão SpEL* for avaliada como verdadeira:

```
@Bean
@ConditionalOnExpression ( " ${usemysql} && ${mysqlserver} ==
'local'} " )
DataSource dataSource () {
    // ...
}
```

## 9. @Conditional

Para condições ainda mais complexas, podemos criar uma classe avaliando a condição personalizada. Dizemos ao Spring para usar essa condição personalizada com *@Conditional* :

```
@Conditional ( HibernateCondition . Class)
Propriedades additionalProperties () {
    // ...
}
```

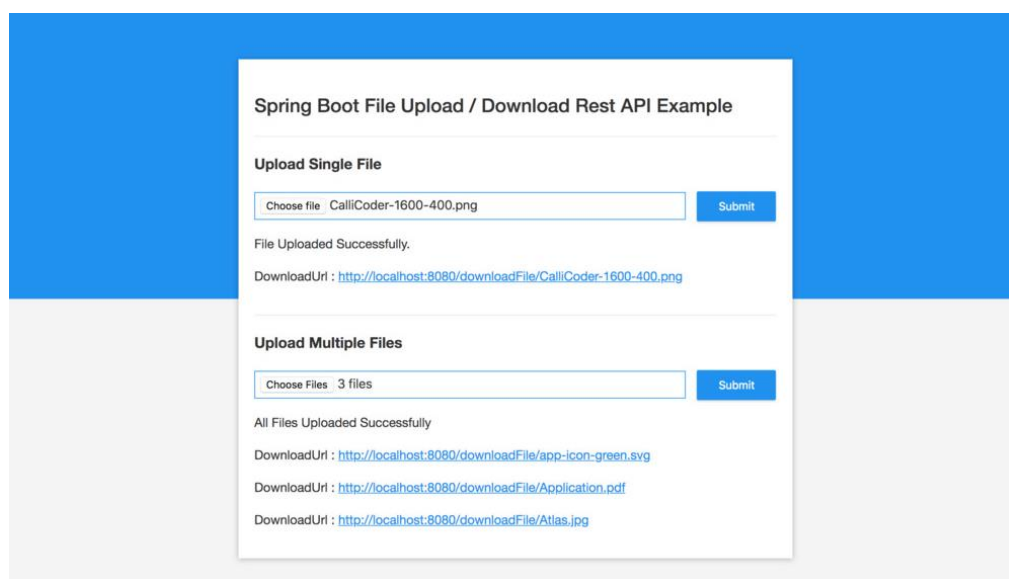
## Projeto API de upload / download de arquivo de springboot

Carregar e baixar arquivos são tarefas muito comuns para as quais os desenvolvedores precisam escrever código em seus aplicativos.

Neste projeto, você aprenderá a fazer upload e download de arquivos em um serviço da web de inicialização RESTful Spring.

Primeiro, construiremos as APIs REST para upload e download de arquivos e, em seguida, testaremos essas APIs usando o Postman. Também escreveremos código front-end em javascript para fazer upload de arquivos.

A seguir está a versão final do nosso aplicativo:



## Criando o aplicativo

Vamos acessar o spring Initializr <https://start.spring.io>

- Abra <http://start.spring.io>
- Insira o arquivo-demo no campo “Artefato”.
- Adicione Web na seção de dependências.
- Clique em Gerar para gerar e baixar o projeto.

Agora você pode descompactar o arquivo do aplicativo baixado e importá-lo para o seu IDE favorito.

## Configurando Propriedades de Servidor e Armazenamento de Arquivo

Vamos configurar nosso aplicativo Spring Boot para permitir uploads de arquivo multipartes e definir o tamanho máximo de arquivo que pode ser carregado. Também configuraremos o diretório no qual todos os arquivos carregados serão armazenados.

Abra o arquivo `src/main/resources/application.properties` e adicione as seguintes propriedades a ele:

```
## MULTIPART (MultipartProperties)

# Enable multipart uploads
spring.servlet.multipart.enabled=true
# Threshold after which files are written to disk.
spring.servlet.multipart.file-size-threshold=2KB
# Max file size.
spring.servlet.multipart.max-file-size=200MB
# Max Request Size
spring.servlet.multipart.max-request-size=215MB

## File Storage Properties

# All files uploaded through the REST API will be
stored in this directory

file.upload-dir=/Users/callicoder/uploads
```

Observação: altere a `file.upload-dir` propriedade para o caminho onde deseja que os arquivos carregados sejam armazenados.

Vinculando propriedades automaticamente a uma classe POJO

Spring Boot tem um recurso incrível chamado `@ConfigurationProperties`, o qual você pode vincular



automaticamente as propriedades definidas no arquivo `application.properties` a uma classe POJO.

Vamos definir uma classe POJO chamada `FileStorageProperties` dentro do pacote `com.example.filedemo.property` para vincular todas as propriedades de armazenamento de arquivo:

```
package com.example.filedemo.property;

import
org.springframework.boot.context.properties.ConfigurationPr
operties;

@ConfigurationProperties(prefix = "file")
public class FileStorageProperties {

    private String uploadDir;

    public String getUploadDir() {
        return uploadDir;
    }

    public void setUploadDir(String uploadDir) {
        this.uploadDir = uploadDir;
    }

}
```

A annotation `@ConfigurationProperties(prefix = "file")` faz seu trabalho na inicialização do aplicativo e vincula todas as propriedades com prefixo `file` aos campos correspondentes da classe POJO.

Se você definir `file` propriedades adicionais no futuro, pode simplesmente adicionar um campo correspondente na classe acima, e o spring boot vinculará automaticamente o campo ao valor da propriedade.

### Habilitar Propriedades de Configuração

Agora, para habilitar o `ConfigurationProperties` recurso, você precisa adicionar `@EnableConfigurationProperties` annotations a qualquer classe de configuração.

Abra a classe principal `src/main/java/com/example/filedemo/FileDemoApplication.java` e adicione a annotation `@EnableConfigurationProperties` a ela. Observe o código abaixo:

```
package com.example.filedemo;

import com.example.filedemo.property.FileStorageProperties;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.properties.EnableConfigurationProperties;

@SpringBootApplication
@EnableConfigurationProperties({
    FileStorageProperties.class
})
public class FileDemoApplication {
```

```

    public static void main(String[] args) {
        SpringApplication.run(FileDemoApplication.class, args);
    }
}

```

## Escrevendo APIs para upload e download de arquivos

Vamos agora escrever as APIs REST para upload e download de arquivos. Crie uma nova classe de controlador chamada `FileController` dentro

do pacote `com.example.filedemo.controller`

Aqui está o código completo para `FileController`:

```

package com.example.filedemo.controller;

import com.example.filedemo.payload.UploadFileResponse;
import com.example.filedemo.service.FileStorageService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

```

```

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

@RestController

public class FileController {

    private static final Logger logger =
LoggerFactory.getLogger(FileController.class);

    @Autowired

    private FileStorageService fileStorageService;

    @PostMapping("/uploadFile")

    public UploadFileResponse uploadFile(@RequestParam("file")
MultipartFile file) {

        String fileName = fileStorageService.storeFile(file);

        String fileDownloadUri =
ServletUriComponentsBuilder.fromCurrentContextPath()

            .path("/downloadFile/")

            .path(fileName)

            .toUriString();

        return new UploadFileResponse(fileName, fileDownloadUri,

            file.getContentType(), file.getSize());

    }

    @PostMapping("/uploadMultipleFiles")

    public List<UploadFileResponse>
uploadMultipleFiles(@RequestParam("files") MultipartFile[] files) {

        return Arrays.asList(files)

            .stream()

```

```

        .map(file -> uploadFile(file))

        .collect(Collectors.toList());

    }

    @GetMapping("/downloadFile/{fileName:.+}")
    public ResponseEntity<Resource> downloadFile(@PathVariable
String fileName, HttpServletRequest request) {

        // Load file as Resource

        Resource resource =
fileStorageService.loadFileAsResource(fileName);

        // Try to determine file's content type

        String contentType = null;

        try {

            contentType =
request.getServletContext().getMimeType(resource.getFile().getAbsolutePath());

        } catch (IOException ex) {

            logger.info("Could not determine file type.");

        }

        // Fallback to the default content type if type could
not be determined

        if(contentType == null) {

            contentType = "application/octet-stream";

        }

        return ResponseEntity.ok()

        .contentType(MediaType.parseMediaType(contentType))

            .header(HttpHeaders.CONTENT_DISPOSITION,
"attachment; filename=\"" + resource.getFilename() + "\"")

            .body(resource);

    }

```

```
}
```

A classe `FileController` usa o `FileStorageService` para armazenar arquivos no sistema de arquivos e recuperá-los. Ele retorna uma carga útil do tipo `UploadFileResponse` após a conclusão do upload. Vamos definir essas classes uma por uma.

### UploadFileResponse

Como o nome sugere, essa classe é usada para retornar a resposta das APIs `/uploadFile` e `/uploadMultipleFiles`.

Crie uma classe chamada `UploadFileResponse` dentro do pacote `com.example.filedemo.payload` com o seguinte conteúdo

```
package com.example.filedemo.payload;

public class UploadFileResponse {
    private String fileName;
    private String fileDownloadUri;
    private String fileType;
    private long size;

    public UploadFileResponse(String fileName, String
fileDownloadUri, String fileType, long size) {
        this.fileName = fileName;
        this.fileDownloadUri = fileDownloadUri;
        this.fileType = fileType;
        this.size = size;
    }
}
```

```
// Getters and Setters (Omitted for brevity)

}
```

## Serviço para armazenar arquivos no FileSystem e recuperá-los

Vamos agora escrever o serviço para armazenar arquivos no sistema de arquivos e recuperá-los. Crie uma nova classe chamada *FileStorageService.java* dentro do pacote *com.example.filedemo.service* com o seguinte conteúdo:

```
package com.example.filedemo.service;

import com.example.filedemo.exception.FileStorageException;
import com.example.filedemo.exception.MyFileNotFoundException;
import com.example.filedemo.property.FileStorageProperties;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.core.io.UrlResource;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;
import org.springframework.web.multipart.MultipartFile;
import java.io.IOException;
import java.net.MalformedURLException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;

@Service
public class FileStorageService {

    private final Path fileStorageLocation;
```

```

@Autowired

    public FileStorageService(FileStorageProperties
fileStorageProperties) {

        this.fileStorageLocation =
Paths.get(fileStorageProperties.getUploadDir())

            .toAbsolutePath().normalize();

        try {

            Files.createDirectories(this.fileStorageLocation);

        } catch (Exception ex) {

            throw new FileStorageException("Could not create the
directory where the uploaded files will be stored.", ex);

        }

    }

    public String storeFile(MultipartFile file) {

        // Normalize file name

        String fileName =
StringUtils.cleanPath(file.getOriginalFilename());

        try {

            // Check if the file's name contains invalid
characters

            if(fileName.contains("..")) {

                throw new FileStorageException("Sorry! Filename
contains invalid path sequence " + fileName);

            }

            // Copy file to the target location (Replacing
existing file with the same name)

            Path targetLocation =
this.fileStorageLocation.resolve(fileName);

            Files.copy(file.getInputStream(), targetLocation,
StandardCopyOption.REPLACE_EXISTING);

```



```

        return fileName;

    } catch (IOException ex) {

        throw new FileStorageException("Could not store file
" + fileName + ". Please try again!", ex);

    }

}

public Resource loadFileAsResource(String fileName) {

    try {

        Path filePath =
this.fileStorageLocation.resolve(fileName).normalize();

        Resource resource = new
UrlResource(filePath.toUri());

        if(resource.exists()) {

            return resource;

        } else {

            throw new MyFileNotFoundException("File not
found " + fileName);

        }

    } catch (MalformedURLException ex) {

        throw new MyFileNotFoundException("File not found "
+ fileName, ex);

    }

}

}

```

## Classes de exceção (exception)

A classe `FileStorageService` lança algumas exceções em caso de situações inesperadas. A seguir estão as definições dessas classes de exceção (todas as classes de exceção vão dentro do pacote `com.example.filedemo.exception`).

### 1. FileStorageException

é lançado quando ocorre uma situação inesperada durante o armazenamento de um arquivo no sistema de arquivos -

```
package com.example.filedemo.exception;

public class FileStorageException extends RuntimeException {

    public FileStorageException(String message) {

        super(message);

    }

    public FileStorageException(String message, Throwable cause)
    {

        super(message, cause);

    }

}
```

## 2. MyFileNotFoundException

É lançado quando um arquivo que o usuário está tentando baixar não é encontrado.

```
package com.example.filedemo.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class MyFileNotFoundException extends RuntimeException {

    public MyFileNotFoundException(String message) {

        super(message);

    }

}
```

```

    public MyFileNotFoundException(String message, Throwable
cause) {

        super(message, cause);

    }

}

```

Observe que implementamos uma annotation para a classe de exceção acima com `@ResponseStatus(HttpStatus.NOT_FOUND)`. Isso garante que o Spring boot responda com um status `404 Not Found` quando essa exceção for lançada.

### Desenvolvendo o Front-End

Nossas APIs de back-end estão construídas. Agora vamos escrever o código do front end para permitir que os usuários carreguem e baixem arquivos de nosso aplicativo da web.

Todos os arquivos do front-end irão para a pasta *src/main/resources/static*. A seguir, está a estrutura de diretório do nosso código de front-end -

```

static
├── css
│   └── main.css
├── js
│   └── main.js
└── index.html

```

### O template HTML

```

<!DOCTYPE html>

<html>

    <head>

```

```

        <meta name="viewport" content="width=device-width,
initial-scale=1.0, minimum-scale=1.0">

        <title>Spring Boot File Upload / Download Rest API
Example</title>

        <link rel="stylesheet" href="/css/main.css" />

    </head>

    <body>

        <noscript>

            <h2>Sorry! Your browser doesn't support
Javascript</h2>

        </noscript>

        <div class="upload-container">

            <div class="upload-header">

                <h2>Spring Boot File Upload / Download Rest API
Example</h2>

            </div>

            <div class="upload-content">

                <div class="single-upload">

                    <h3>Upload Single File</h3>

                    <form id="singleUploadForm"
name="singleUploadForm">

                        <input id="singleFileUploadInput"
type="file" name="file" class="file-input" required />

                        <button type="submit" class="primary
submit-btn">Submit</button>

                    </form>

                    <div class="upload-response">

                        <div id="singleFileUploadError"></div>

                        <div id="singleFileUploadSuccess"></div>

                    </div>

                </div>

                <div class="multiple-upload">

                    <h3>Upload Multiple Files</h3>

                    <form id="multipleUploadForm"
name="multipleUploadForm">

```

```

        <input id="multipleFileUploadInput"
type="file" name="files" class="file-input" multiple required />

        <button type="submit" class="primary
submit-btn">Submit</button>

    </form>

    <div class="upload-response">

        <div id="multipleFileUploadError"></div>

        <div
id="multipleFileUploadSuccess"></div>

    </div>

</div>

</div>

</div>

<script src="/js/main.js" ></script>

</body>

</html>

```

## O arquivo Javascript

```

'use strict';

var singleUploadForm =
document.querySelector('#singleUploadForm');

var singleFileUploadInput =
document.querySelector('#singleFileUploadInput');

var singleFileUploadError =
document.querySelector('#singleFileUploadError');

var singleFileUploadSuccess =
document.querySelector('#singleFileUploadSuccess');

var multipleUploadForm =
document.querySelector('#multipleUploadForm');

var multipleFileUploadInput =
document.querySelector('#multipleFileUploadInput');

var multipleFileUploadError =
document.querySelector('#multipleFileUploadError');

```

```

var multipleFileUploadSuccess =
document.querySelector('#multipleFileUploadSuccess');

function uploadSingleFile(file) {

    var formData = new FormData();

    formData.append("file", file);

    var xhr = new XMLHttpRequest();

    xhr.open("POST", "/uploadFile");

    xhr.onload = function() {

        console.log(xhr.responseText);

        var response = JSON.parse(xhr.responseText);

        if(xhr.status == 200) {

            singleFileUploadError.style.display = "none";

            singleFileUploadSuccess.innerHTML = "<p>File
Uploaded Successfully.</p><p>DownloadUrl : <a href='" +
response.fileDownloadUri + "' target='_blank'>" +
response.fileDownloadUri + "</a></p>";

            singleFileUploadSuccess.style.display = "block";

        } else {

            singleFileUploadSuccess.style.display = "none";

            singleFileUploadError.innerHTML = (response &&
response.message) || "Some Error Occurred";

        }

    }

    xhr.send(formData);

}

function uploadMultipleFiles(files) {

    var formData = new FormData();

    for(var index = 0; index < files.length; index++) {

        formData.append("files", files[index]);

```

```

    }

    var xhr = new XMLHttpRequest();

    xhr.open("POST", "/uploadMultipleFiles");

    xhr.onload = function() {

        console.log(xhr.responseText);

        var response = JSON.parse(xhr.responseText);

        if(xhr.status == 200) {

            multipleFileUploadError.style.display = "none";

            var content = "<p>All Files Uploaded  
Successfully</p>";

            for(var i = 0; i < response.length; i++) {

                content += "<p>DownloadUrl : <a href='" +  
response[i].fileDownloadUri + "' target='_blank'>" +  
response[i].fileDownloadUri + "</a></p>";

            }

            multipleFileUploadSuccess.innerHTML = content;

            multipleFileUploadSuccess.style.display = "block";

        } else {

            multipleFileUploadSuccess.style.display = "none";

            multipleFileUploadError.innerHTML = (response &&  
response.message) || "Some Error Occurred";

        }

    }

    xhr.send(formData);

}

singleUploadForm.addEventListener('submit', function(event) {

    var files = singleFileUploadInput.files;

    if(files.length === 0) {

```

```

        singleFileUploadError.innerHTML = "Please select a
file";

        singleFileUploadError.style.display = "block";

    }

    uploadSingleFile(files[0]);

    event.preventDefault();

}, true);

multipleUploadForm.addEventListener('submit', function(event) {

    var files = multipleFileUploadInput.files;

    if(files.length === 0) {

        multipleFileUploadError.innerHTML = "Please select at
least one file";

        multipleFileUploadError.style.display = "block";

    }

    uploadMultipleFiles(files);

    event.preventDefault();

}, true);

```

O código acima está usando XMLHttpRequest junto com o objeto FormData para fazer upload de arquivo (s) como multipart/form-data.

## O arquivo CSS

```

* {

    -webkit-box-sizing: border-box;

    -moz-box-sizing: border-box;

    box-sizing: border-box;

}

body {

    margin: 0;

```



```
padding: 0;

font-weight: 400;

font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;

font-size: 1rem;

line-height: 1.58;

color: #333;

background-color: #f4f4f4;

}

body:before {

    height: 50%;

    width: 100%;

    position: absolute;

    top: 0;

    left: 0;

    background: #128ff2;

    content: "";

    z-index: 0;

}

.clearfix:after {

    display: block;

    content: "";

    clear: both;

}

h1, h2, h3, h4, h5, h6 {

    margin-top: 20px;

    margin-bottom: 20px;

}
```

```
h1 {  
  font-size: 1.7em;  
}  
  
a {  
  color: #128ff2;  
}  
  
button {  
  box-shadow: none;  
  border: 1px solid transparent;  
  font-size: 14px;  
  outline: none;  
  line-height: 100%;  
  white-space: nowrap;  
  vertical-align: middle;  
  padding: 0.6rem 1rem;  
  border-radius: 2px;  
  transition: all 0.2s ease-in-out;  
  cursor: pointer;  
  min-height: 38px;  
}  
  
button.primary {  
  background-color: #128ff2;  
  box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.12);  
  color: #fff;  
}  
  
input {
```

```
        font-size: 1rem;
    }

    input[type="file"] {
        border: 1px solid #128ff2;
        padding: 6px;
        max-width: 100%;
    }

    .file-input {
        width: 100%;
    }

    .submit-btn {
        display: block;
        margin-top: 15px;
        min-width: 100px;
    }

    @media screen and (min-width: 500px) {
        .file-input {
            width: calc(100% - 115px);
        }

        .submit-btn {
            display: inline-block;
            margin-top: 0;
            margin-left: 10px;
        }
    }
}
```

```
.upload-container {  
    max-width: 700px;  
    margin-left: auto;  
    margin-right: auto;  
    background-color: #fff;  
    box-shadow: 0 1px 11px rgba(0, 0, 0, 0.27);  
    margin-top: 60px;  
    min-height: 400px;  
    position: relative;  
    padding: 20px;  
}  
  
.upload-header {  
    border-bottom: 1px solid #ececcec;  
}  
  
.upload-header h2 {  
    font-weight: 500;  
}  
  
.single-upload {  
    padding-bottom: 20px;  
    margin-bottom: 20px;  
    border-bottom: 1px solid #e8e8e8;  
}  
  
.upload-response {  
    overflow-x: hidden;  
    word-break: break-all;  
}
```

## Opção JQuery

Se você preferir usar jQuery em vez de javascript, pode fazer upload de arquivos usando um método `jQuery.ajax()` como este:

```
$('#singleUploadForm').submit(function(event) {

    var formElement = this;

    // You can directly create form data from the form element

    // (Or you could get the files from input element and append
    them to FormData as we did in vanilla javascript)

    var formData = new FormData(formElement);

    $.ajax({

        type: "POST",

        enctype: 'multipart/form-data',

        url: "/uploadFile",

        data: formData,

        processData: false,

        contentType: false,

        success: function (response) {

            console.log(response);

            // process response

        },

        error: function (error) {

            console.log(error);

            // process error

        }

    });

    event.preventDefault();

});
```

## Conclusão

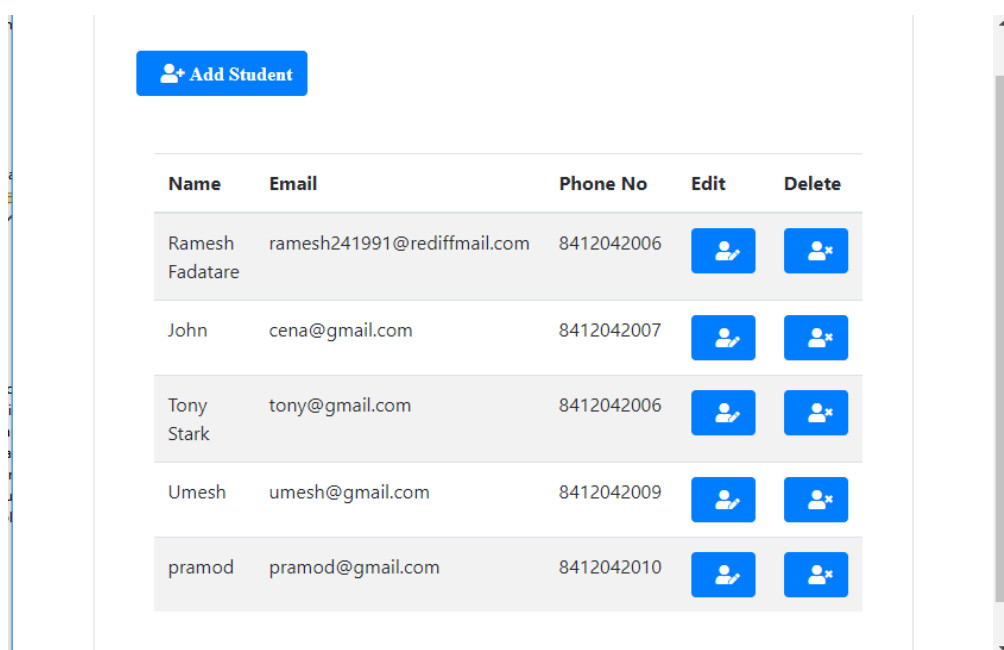
Neste projeto, aprendemos como fazer upload de arquivos únicos e também de vários arquivos por meio de APIs REST escritas em Spring Boot. Também aprendemos como baixar arquivos no Spring Boot. Por fim, escrevemos código para fazer upload de arquivos chamando as APIs por meio de javascript.











## Spring Boot Thymeleaf CRUD

Neste projeto, aprenderemos como desenvolver um aplicativo da web CRUD com Spring Boot e Thymeleaf.

### Premissa

Vamos construir um aplicativo Web CRUD para a entidade **Student** (criar student, listar students, atualizar student e excluir student) usando Springboot e Thymeleaf. Vamos usar um banco de dados H2 para configuração rápida e execução da nossa aplicação.



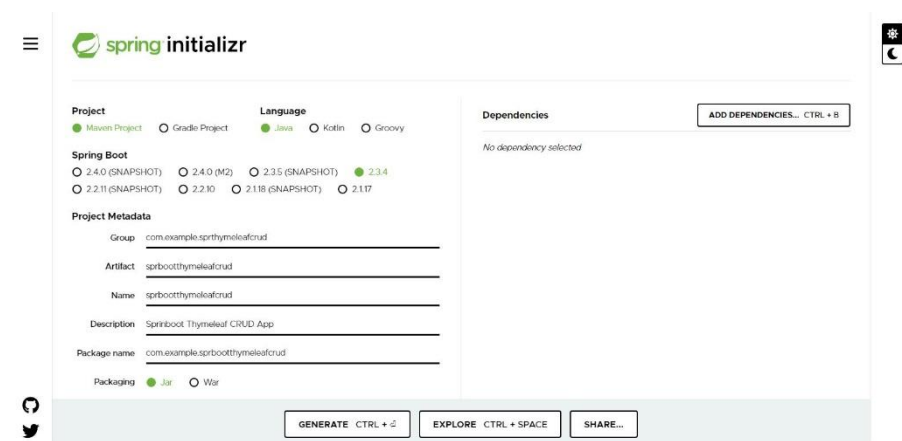
Name	Email	Phone No	Edit	Delete
Ramesh Fadatare	ramesh241991@rediffmail.com	8412042006		
John	cena@gmail.com	8412042007		
Tony Stark	tony@gmail.com	8412042006		
Umesh	umesh@gmail.com	8412042009		
pramod	pramod@gmail.com	8412042010		

## Etapas de Desenvolvimento

1. Criando um aplicativo Spring Boot
2. Estrutura do Projeto
3. Dependências Maven - Pom.xml
4. Camada de Domínio
5. A Camada de Repositório
6. A camada de controlador
7. A Camada de Visualização
8. Executando o aplicativo
9. Demo

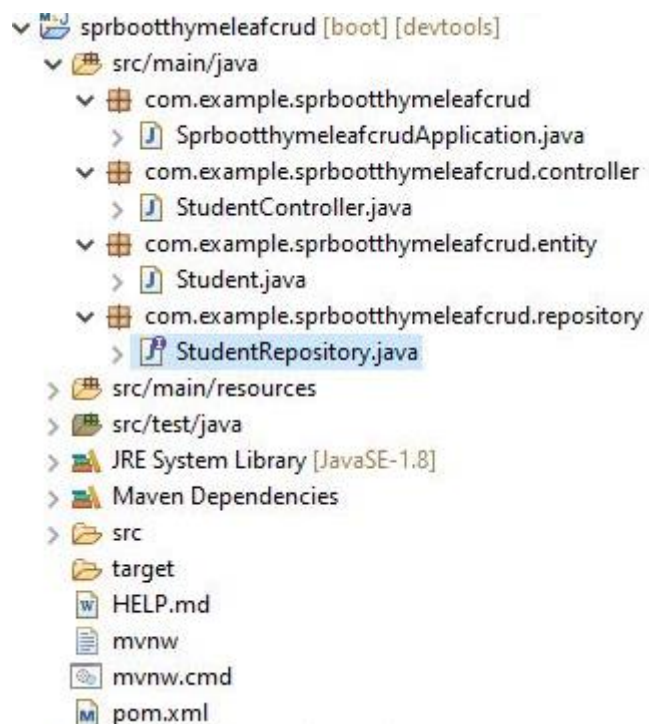
### 1. Criando um aplicativo Spring Boot

Existem muitas maneiras de criar um aplicativo Spring Boot. Podemos utilizar o Spring Initializr ou o STS do nosso IDE Eclipse. Vamos utilizar o Spring Initializr e repetir o que fizemos para criar outras aplicações. Aqui, o projeto está sendo criado com Spring Initializr:



### 2. Estrutura do Projeto

A seguir está o pacote ou estrutura do projeto:



### 3. Dependências do Maven - pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.4.RELEASE</version>
    <relativePath/>
    <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example.sprthymeleafcrud</groupId>
  <artifactId>sprbootthymeleafcrud</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>sprbootthymeleafcrud</name>
  <description>Sprinboot Thymeleaf CRUD App</description>
  <properties>
    <java.version>1.8</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-
jpa</artifactId>
    </dependency>
```



```

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-
thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>8.0.1</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-
plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>

```

Observe que estamos indo com os valores padrão, não adicionamos nada no arquivo de configuração **application.properties** .

#### 4. Camada de Domínio

A camada de domínio é uma coleção de objetos de entidade e lógicas de negócios relacionadas projetadas para representar o modelo de negócios da empresa.

Vamos criar uma entidade JPA **Student** :

```

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.NotBlank;

@Entity
public class Student {

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

    @NotBlank(message = "Name is mandatory")
    @Column(name = "name")
    private String name;

    @NotBlank(message = "Email is mandatory")
    @Column(name = "email")
    private String email;

    @Column(name = "phone_no")
    private long phoneNo;

    public Student() {}

    public Student(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public void setId(long id) {
        this.id = id;
    }

    public long getId() {
        return id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    public long getPhoneNo() {

```

```

        return phoneNo;
    }

    public void setPhoneNo(long phoneNo) {
        this.phoneNo = phoneNo;
    }
}

```

## 5. A Camada do Repositório

O Spring Data JPA nos permite implementar repositórios baseados em JPA (um nome sofisticado para a implementação do padrão DAO) com o mínimo de confusão.

O Spring Data JPA é um componente chave do *spring-boot-starter-data-jpa* do Spring Boot que torna mais fácil adicionar a funcionalidade CRUD por meio de uma camada poderosa de abstração colocada no topo de uma implementação JPA. Essa camada de abstração nos permite acessar a camada de persistência sem ter que fornecer nossas próprias implementações DAO desde o início.

Vamos criar uma classe **StudentRepository** com o seguinte código:

```

import java.util.List;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import net.javaguides.springboot.projeto.entity.Student;

@Repository
public interface StudentRepository extends CrudRepository
<Student, Long> {
    List<Student> findByName(String name);
}

```

## 6. A Camada Controller ou Camada Web

Esta camada é responsável por processar a entrada do usuário e retornar a resposta correta ao usuário. A camada da web também deve lidar com as exceções lançadas pelas outras camadas. Como a camada da web é o ponto de

entrada de nosso aplicativo, ela deve cuidar da autenticação e agir como a primeira linha de defesa contra usuários não autorizados.

No Spring, a classe do controlador depende de alguns dos principais recursos do Spring MVC. Vamos criar uma classe **StudentController** que invoca internamente a camada do repositório:

```
import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
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.RequestMapping;

import net.javaguides.springboot.projeto.entity.Student;
import net.javaguides.springboot.projeto.repository.StudentRepository;

@Controller
@RequestMapping("/students/")
public class StudentController {

    private final StudentRepository studentRepository;

    @Autowired
    public StudentController(StudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }

    @GetMapping("signup")
    public String showSignUpForm(Student student) {
        return "add-student";
    }

    @GetMapping("list")
    public String showUpdateForm(Model model) {
        model.addAttribute("students",
studentRepository.findAll());
        return "index";
    }

    @PostMapping("add")
    public String addStudent(@Valid Student student, BindingResult
result, Model model) {
        if (result.hasErrors()) {
            return "add-student";
        }

        studentRepository.save(student);
        return "redirect:list";
    }
}
```

```

        @GetMapping("edit/{id}")
        public String showUpdateForm(@PathVariable("id") long id,
Model model) {
            Student student = studentRepository.findById(id)
                .orElseThrow(() - > new
IllegalArgumentException("Invalid student Id:" + id));
            model.addAttribute("student", student);
            return "update-student";
        }

        @PostMapping("update/{id}")
        public String updateStudent(@PathVariable("id") long id,
@Valid Student student, BindingResult result,
Model model) {
            if (result.hasErrors()) {
                student.setId(id);
                return "update-student";
            }

            studentRepository.save(student);
            model.addAttribute("students",
studentRepository.findAll());
            return "index";
        }

        @GetMapping("delete/{id}")
        public String deleteStudent(@PathVariable("id") long id, Model
model) {
            Student student = studentRepository.findById(id)
                .orElseThrow(() - > new
IllegalArgumentException("Invalid student Id:" + id));
            studentRepository.delete(student);
            model.addAttribute("students",
studentRepository.findAll());
            return "index";
        }
    }
}

```

## 7. A Camada de Visualização

Na pasta *src / main / resources / templates*, precisamos criar os modelos HTML necessários para exibir o **student** que se inscreveu, atualizar o student e renderizar a lista de entidades de Student persistentes. Estaremos usando o Thymeleaf como o mecanismo de template subjacente para analisar os arquivos de template.



```

        <input type="submit" class="btn
btn-primary" value="Add Student">
    </div>

    <div class="form-group col-md-
8"></div>

    </div>
</form>
</div>
</div>
</div>
</div>
</body>
</html>

```

## Lista de students - index.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head>
    <meta charset="utf-8">
    <meta http-equiv="x-ua-compatible" content="ie=edge">
    <title>Users</title>
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap
.min.css" integrity="sha384-
MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
crossorigin="anonymous">
    <link rel="stylesheet"
href="https://use.fontawesome.com/releases/v5.4.1/css/all.css"
integrity="sha384-
5sAR7xN1Nv6T6+dtT2mhtzEpVJvfS3NScPQTrOxhvjIuvCA67KV2R5Jz6kr4abQsz"
crossorigin="anonymous">
    <!-- <link rel="stylesheet" href="../css/shards.min.css"> --
>
</head>

<body>
    <div class="container my-2">
        <div class="card">
            <div class="card-body">
                <div th:switch="{students}" class="container
my-5">

                    <p class="my-5">
                        <a href="/students/signup" class="btn
btn-primary"><i
class="fas fa-user-plus ml-2"> Add Student</i></a>
                    </p>
                    <div class="col-md-10">
                        <h2 th:case="null">No Students yet!</h2>
                        <div th:case="*">

```

```

<table class="table table-striped
table-responsive-md">
    <thead>
        <tr>
            <th>Name</th>
            <th>Email</th>
            <th>Phone No</th>
            <th>Edit</th>
            <th>Delete</th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="student :
${students}">
            <td>
                th:text="${student.name}"</td>
            <td>
                th:text="${student.email}"</td>
            <td>
                th:text="${student.phoneNo}"</td>
            <td><a
th:href="@{/students/edit/{id} (id=${student.id})}" class="btn btn-
primary"><i class="fas fa-user-edit ml-2"></i></a></td>
            <td><a
th:href="@{/students/delete/{id} (id=${student.id})}" class="btn btn-
primary"><i class="fas fa-user-times ml-2"></i></a></td>
        </tr>
    </tbody>
</table>
</div>
</div>
</div>
</div>
</div>
</div>
</body>
</html>

```

### Atualizar (update) student - update-student.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head>
    <meta charset="utf-8">
    <meta http-equiv="x-ua-compatible" content="ie=edge">
    <title>Update User</title>
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap
.min.css" integrity="sha384-
MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
crossorigin="anonymous">

```



```

    <link rel="stylesheet"
href="https://use.fontawesome.com/releases/v5.4.1/css/all.css"
integrity="sha384-
5sAR7xN1Nv6T6+dT2mhtzEpVJvfS3NScPQTrOxhwjIuvcA67KV2R5Jz6kr4abQsz"
crossorigin="anonymous">
  </head>

  <body>
    <div class="container my-5">
      <h3> Update Student</h3>
      <div class="card">
        <div class="card-body">
          <div class="col-md-8">
            <form action="#"
th:action="@{/students/update/{id} (id=${student.id})}"
th:object="${student}" method="post">
              <div class="row">
                <div class="form-group col-md-6">
                  <label for="name" class="col-
form-label">Name</label> <input type="text" th:field="*{name}"
class="form-control" id="name" placeholder="Name"> <span
th:if="${#fields.hasErrors('name')}" th:errors="*{name}" class="text-
danger"></span>
                  </div>
                  <div class="form-group col-md-8">
                    <label for="email" class="col-
form-label">Email</label> <input type="text" th:field="*{email}"
class="form-control" id="email" placeholder="Email"> <span
th:if="${#fields.hasErrors('email')}" th:errors="*{email}"
class="text-danger"></span>
                  </div>
                  <div class="form-group col-md-8">
                    <label for="phoneNo" class="col-
form-label">Phone No</label> <input type="text" th:field="*{phoneNo}"
class="form-control" id="phoneNo" placeholder="phoneNo"> <span
th:if="${#fields.hasErrors('phoneNo')}" th:errors="phoneNo"
class="text-danger"></span>
                  </div>
                  <div class="form-group col-md-8">
                    <input type="submit" class="btn
btn-primary" value="Update Student">
                  </div>
                </div>
              </form>
            </div>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>

```

## 8. Executando o aplicativo

Por fim, vamos definir o ponto de entrada do aplicativo. Como a maioria dos aplicativos Spring Boot, podemos fazer isso com um método main () simples:

```
import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;

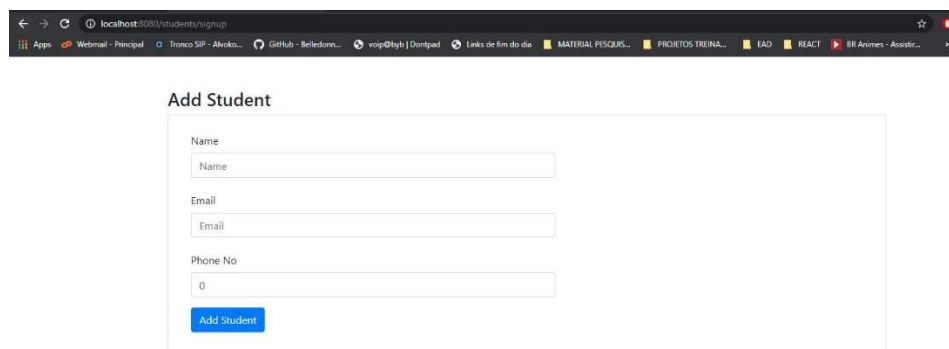
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

## 9. Executando a aplicação

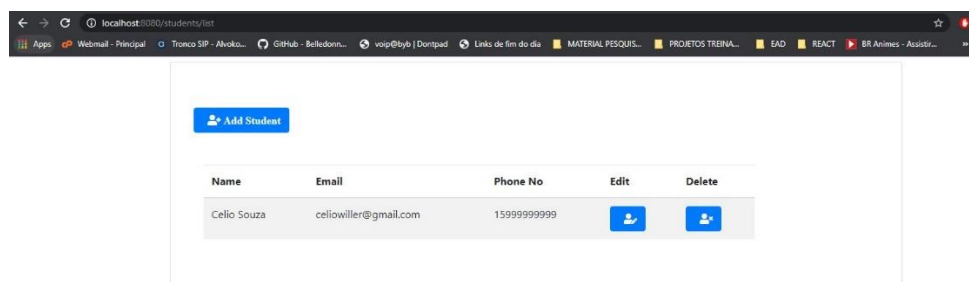
Vamos acessar o aplicativo da web implantado acima usando [http: // localhost: 8080](http://localhost:8080) Inicialmente, você não tem nenhum student na lista, então vamos adicionar um novo student primeiro.

### Tela Adicionar (add/create) student

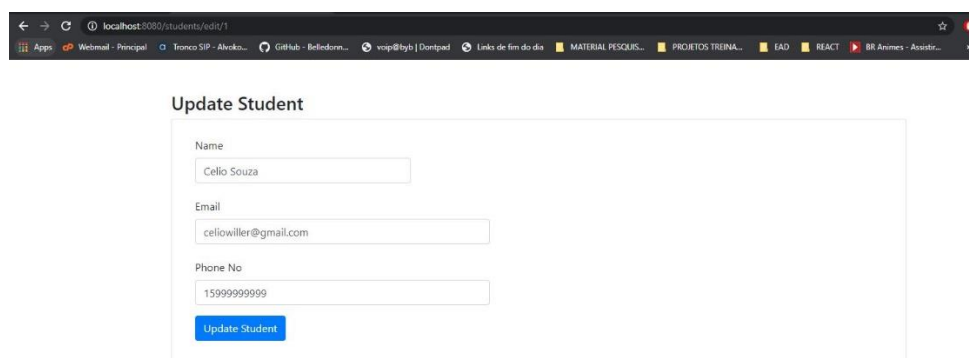


The screenshot shows a web browser window with the address bar displaying 'localhost:8080/student/signup'. The page title is 'Add Student'. The form contains three input fields: 'Name', 'Email', and 'Phone No'. The 'Phone No' field has the value '0' entered. Below the input fields is a blue button labeled 'Add Student'.

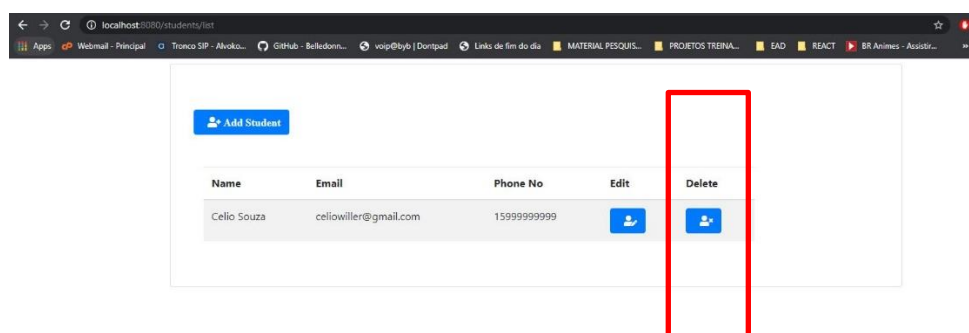
## Tela da Lista (read)Students



## Tela Atualizar (update) student



## Tela Excluir (delete) student

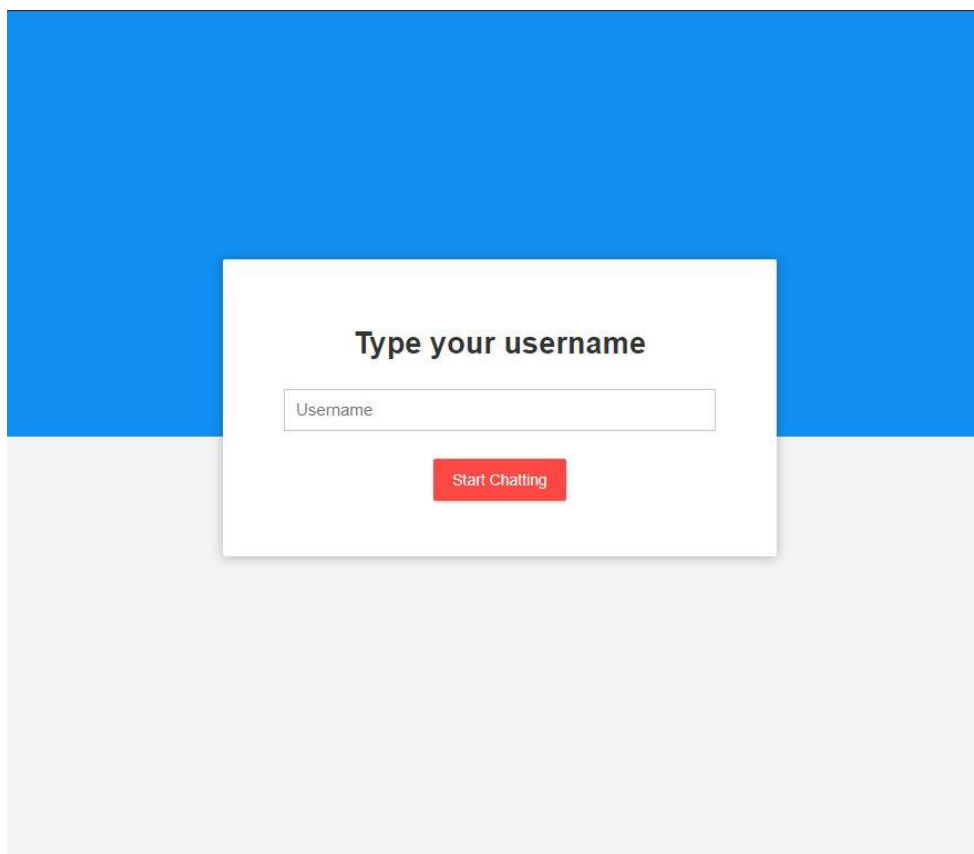


## Construindo um aplicativo de chat com Spring Boot e WebSocket

Neste projeto, vamos aprender a como usar a API WebSocket com Springboot e construir um aplicativo de chat simples.

Você pode abrir o aplicativo em duas abas, fazer o login com nomes de usuário diferentes e começar a enviar mensagens.

A seguir está uma captura de tela do aplicativo de bate-papo que construiremos neste projeto:



**WebSocket** é um protocolo de comunicação que permite estabelecer um canal de comunicação bidirecional entre um servidor e um cliente.

O WebSocket funciona estabelecendo primeiro uma conexão HTTP regular com o servidor e, em seguida, atualizando-a para uma conexão websocket bidirecional, enviando um header `Upgrade`.

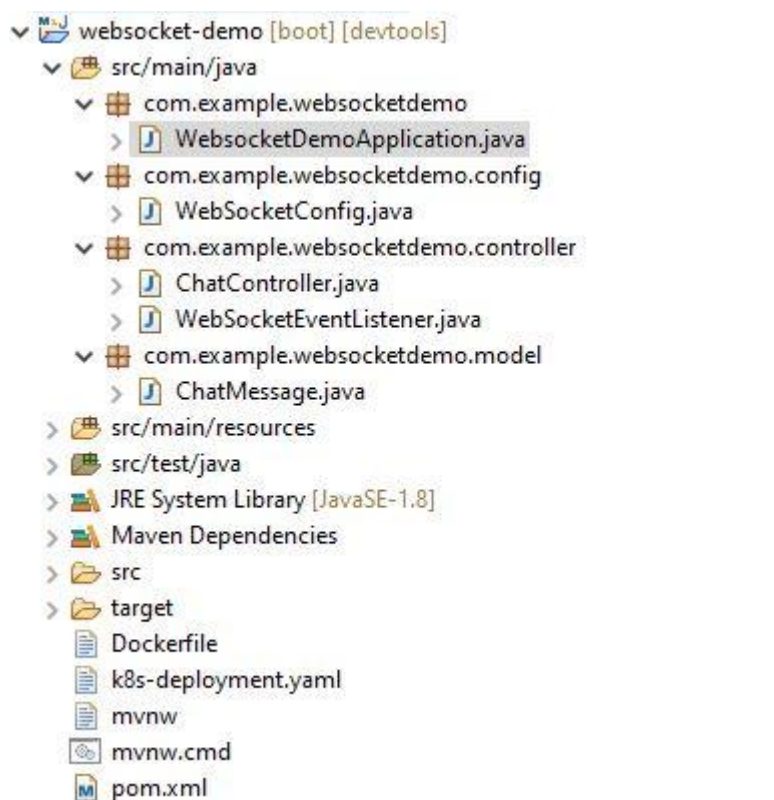
É compatível com a maioria dos navegadores web modernos e, para navegadores que não o suportam, existem bibliotecas que fornecem alternativas a partir de outras técnicas, como [comet](#) e [long-polling](#).

## Criando o aplicativo

Vamos utilizar a ferramenta da web Spring Initializer para gerar o projeto. Siga as etapas abaixo para gerar o projeto usando Spring Initializer -

- Vá para <http://start.spring.io/> .
- Insira o valor do Artifact como websocket-demo .
- Adicione WebSocket na seção de dependências.
- Clique em Gerar projeto para baixar o projeto.
- Extraia o arquivo zip baixado.

Após gerar o projeto, importe-o em seu IDE favorito. A estrutura do diretório do projeto deve ser semelhante a esta:



## Configuração WebSocket

A primeira etapa é configurar o endpoint do websocket e o message broker. Crie um novo pacote `config` dentro do pacote `com.example.websocketdemo` e, em seguida, crie uma nova classe `WebSocketConfig` dentro do pacote `config` com o seguinte conteúdo:

```

package com.example.websocketdemo.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.*;

@Configuration
@EnableWebSocketMessageBroker

public class WebSocketConfig implements
WebSocketMessageBrokerConfigurer {

    @Override

    public void registerStompEndpoints(StompEndpointRegistry
registry) {

        registry.addEndpoint("/ws").withSockJS();

    }

    @Override

    public void configureMessageBroker(MessageBrokerRegistry
registry) {

        registry.setApplicationDestinationPrefixes("/app");

        registry.enableSimpleBroker("/topic");

    }

}

```

O instrução `@EnableWebSocketMessageBroker` é usada para habilitar nosso servidor WebSocket. Implementamos a interface `WebSocketMessageBrokerConfigurer` e fornecemos implementação para alguns de seus métodos para configurar a conexão do websocket.

No primeiro método, registramos um endpoint de websocket que os clientes usarão para se conectar ao nosso servidor de websocket.

Observe o uso de `withSockJS()` com a configuração do terminal. **SockJS** é usado para habilitar opções de fallback para navegadores que não suportam websocket.

Você deve ter notado a palavra **STOMP** no nome do método. Esses métodos vêm da implementação STOMP de frameworks Spring. STOMP significa Simple Text Oriented Messaging Protocol. É um protocolo de mensagens que define o formato e as regras para troca de dados.

**Por que precisamos do STOMP?** WebSocket é apenas um protocolo de comunicação. Ele não define coisas como - Como enviar uma mensagem apenas para usuários que estão inscritos em um determinado tópico, ou como enviar uma mensagem para um determinado usuário. Precisamos do STOMP para essas funcionalidades.

No segundo método, estamos configurando um agente de mensagens que será usado para rotear mensagens de um cliente para outro.

A primeira linha define que as mensagens cujo destino começa com “/app” devem ser roteadas para métodos de tratamento de mensagens (definiremos esses métodos em breve).

E, a segunda linha define que as mensagens cujo destino começa com “/tópico” devem ser roteadas para o agente de mensagens. O agente de mensagens transmite mensagens para todos os clientes conectados que estão inscritos em um determinado tópico.

No exemplo acima, habilitamos um corretor de mensagens simples na memória. Também é possível usar qualquer outro corretor de mensagens completo, como [RabbitMQ](#) ou [ActiveMQ](#).

### Criação do *ChatMessageModel*

*ChatMessageModel* é a carga útil da mensagem que será trocada entre os clientes e o servidor. Crie um novo pacote chamado *model* dentro do pacote ***com.example.websocketdemo*** e, em seguida, crie a classe ***ChatMessage*** dentro do pacote *model* com o seguinte conteúdo:

```
package com.example.websocketdemo.model;
```

```
public class ChatMessage {  
    private MessageType type;  
    private String content;  
    private String sender;  
  
    public enum MessageType {  
        CHAT,  
        JOIN,  
        LEAVE  
    }  
  
    public MessageType getType() {  
        return type;  
    }  
  
    public void setType(MessageType type) {  
        this.type = type;  
    }  
  
    public String getContent() {  
        return content;  
    }  
  
    public void setContent(String content) {  
        this.content = content;  
    }  
  
    public String getSender() {  
        return sender;  
    }  
}
```



```

    public void setSender(String sender) {
        this.sender = sender;
    }
}

```

### Criação do controlller para enviar e receber mensagens

Vamos definir os métodos de tratamento de mensagens em nosso controlador. Esses métodos serão responsáveis por receber mensagens de um cliente e depois transmiti-las a outros.

Crie um novo pacote, dentro do pacote base e chame-o de **controller** dentro do pacote base e, em seguida, crie a classe *ChatController* com o seguinte conteúdo -

```

package com.example.websocketdemo.controller;

import com.example.websocketdemo.model.ChatMessage;
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.messaging.simp.SimpMessageHeaderAccessor;
import org.springframework.stereotype.Controller;

@Controller
public class ChatController {

    @MessageMapping("/chat.sendMessage")
    @SendTo("/topic/public")
    public ChatMessage sendMessage(@Payload ChatMessage chatMessage) {

```

```

        return chatMessage;
    }

    @MessageMapping("/chat.addUser")
    @SendTo("/topic/public")
    public ChatMessage addUser(@Payload ChatMessage chatMessage,
                               SimpMessageHeaderAccessor
headerAccessor) {
        // Add username in web socket session
        headerAccessor.getSessionAttributes().put("username",
chatMessage.getSender());

        return chatMessage;
    }
}

```

Se você se lembra da configuração do websocket, todas as mensagens enviadas de clientes com um destino começando com `/app` serão roteadas para esses métodos de tratamento de mensagens anotados com `@MessageMapping`.

Por exemplo, uma mensagem com destino `/app/chat.sendMessage` será roteada para o método `sendMessage()` e uma mensagem com destino `/app/chat.addUser` será roteada para o método `addUser()`.

### Adicionar listeners de evento WebSocket

Usaremos listeners de evento para “ouvir” eventos de conexão e desconexão de soquete para que possamos registrar esses eventos e também transmiti-los quando um usuário entrar ou sair da sala de chat:

```

package com.example.websocketdemo.controller;

import com.example.websocketdemo.model.ChatMessage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

```

```

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.event.EventListener;

import
org.springframework.messaging.simp.SimpMessageSendingOperations;

import
org.springframework.messaging.simp.stomp.StompHeaderAccessor;

import org.springframework.stereotype.Component;

import
org.springframework.web.socket.messaging.SessionConnectedEvent;

import
org.springframework.web.socket.messaging.SessionDisconnectEvent;


@Component

public class WebSocketEventListener {

    private static final Logger logger =
LoggerFactory.getLogger(WebSocketEventListener.class);


    @Autowired

    private SimpMessageSendingOperations messagingTemplate;


    @EventListener

    public void
handleWebSocketConnectListener(SessionConnectedEvent event) {

        logger.info("Received a new web socket connection");

    }


    @EventListener

    public void
handleWebSocketDisconnectListener(SessionDisconnectEvent event) {

        StompHeaderAccessor headerAccessor =
StompHeaderAccessor.wrap(event.getMessage());


        String username = (String)
headerAccessor.getSessionAttributes().get("username");

```

```

        if (username != null) {

            logger.info("User Disconnected : " + username);

            ChatMessage chatMessage = new ChatMessage();

            chatMessage.setType(ChatMessage.MessageType.LEAVE);

            chatMessage.setSender(username);

            messagingTemplate.convertAndSend("/topic/public",
chatMessage);

        }

    }

}

```

Já estamos transmitindo o evento de adesão do usuário no método `addUser()` definido dentro `ChatController`. Portanto, não precisamos fazer nada no evento `SessionConnected`.

No evento `SessionDisconnect`, escrevemos um código para extrair o nome do usuário da sessão do websocket e transmitir um evento de saída do usuário para todos os clientes conectados.

### Criação do front-end

Crie as seguintes pastas e arquivos dentro do diretório `src/main/resources`:

```

static
├── css
│   └── main.css
├── js
│   └── main.js
└── index.html

```

A pasta `src/main/resources/static` é o local padrão para arquivos estáticos no Spring Boot.

## 1. Criação do HTML - `index.html`

O arquivo HTML contém a interface do usuário para exibir as mensagens de bate-papo. Ele inclui as bibliotecas `sockjs` e `stomp` JavaScript.

SockJS é um cliente WebSocket que tenta usar WebSockets nativos e fornece opções de fallback inteligente para navegadores mais antigos que não oferecem suporte a WebSocket. STOMP JS é o cliente stomp para javascript.

A seguir está o código completo para `index.htm`:

```
<!DOCTYPE html>

<html>

  <head>

    <meta name="viewport" content="width=device-width,
initial-scale=1.0, minimum-scale=1.0">

    <title>Spring Boot WebSocket Chat Application</title>

    <link rel="stylesheet" href="/css/main.css" />

  </head>

  <body>

    <noscript>

      <h2>Sorry! Your browser doesn't support Javascript</h2>

    </noscript>

    <div id="username-page">

      <div class="username-page-container">

        <h1 class="title">Type your username</h1>

        <form id="usernameForm" name="usernameForm">

          <div class="form-group">

            <input type="text" id="name"
placeholder="Username" autocomplete="off" class="form-control" />
```

```

        </div>

        <div class="form-group">
            <button type="submit" class="accent
username-submit">Start Chatting</button>
        </div>

    </form>

</div>

</div>

<div id="chat-page" class="hidden">
    <div class="chat-container">
        <div class="chat-header">
            <h2>Spring WebSocket Chat Demo</h2>
        </div>
        <div class="connecting">
            Connecting...
        </div>
        <ul id="messageArea">

        </ul>
        <form id="messageForm" name="messageForm">
            <div class="form-group">
                <div class="input-group clearfix">
                    <input type="text" id="message"
placeholder="Type a message..." autocomplete="off" class="form-
control"/>
                    <button type="submit"
class="primary">Send</button>
                </div>
            </div>
        </form>
    </div>
</div>

```

```

        <script src="https://cdnjs.cloudflare.com/ajax/libs/sockjs-
client/1.1.4/sockjs.min.js"></script>

        <script
src="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.min.j
s"></script>

        <script src="/js/main.js"></script>

    </body>

</html>

```

## 2. JavaScript - `main.js`

Agora vamos adicionar o javascript necessário para se conectar ao endpoint do websocket e enviar e receber mensagens. Primeiro, adicione o seguinte código ao arquivo `main.js` e, em seguida, exploraremos alguns dos métodos importantes neste arquivo:

```

'use strict';

var usernamePage = document.querySelector('#username-page');
var chatPage = document.querySelector('#chat-page');
var usernameForm = document.querySelector('#usernameForm');
var messageForm = document.querySelector('#messageForm');
var messageInput = document.querySelector('#message');
var messageArea = document.querySelector('#messageArea');
var connectingElement = document.querySelector('.connecting');

var stompClient = null;
var username = null;

var colors = [
    '#2196F3', '#32c787', '#00BCD4', '#ff5652',
    '#ffc107', '#ff85af', '#FF9800', '#39bbb0'

```

```

];

function connect(event) {
    username = document.querySelector('#name').value.trim();

    if(username) {
        usernamePage.classList.add('hidden');
        chatPage.classList.remove('hidden');

        var socket = new SockJS('/ws');
        stompClient = Stomp.over(socket);

        stompClient.connect({}, onConnected, onError);
    }
    event.preventDefault();
}

function onConnected() {
    // Subscribe to the Public Topic
    stompClient.subscribe('/topic/public', onMessageReceived);

    // Tell your username to the server
    stompClient.send("/app/chat.addUser",
        {},
        JSON.stringify({sender: username, type: 'JOIN'})
    )

    connectingElement.classList.add('hidden');
}

```



```

function onError(error) {
    connectingElement.textContent = 'Could not connect to
WebSocket server. Please refresh this page to try again!';
    connectingElement.style.color = 'red';
}

```

```

function sendMessage(event) {
    var messageContent = messageInput.value.trim();
    if(messageContent && stompClient) {
        var chatMessage = {
            sender: username,
            content: messageInput.value,
            type: 'CHAT'
        };
        stompClient.send("/app/chat.sendMessage", {},
JSON.stringify(chatMessage));
        messageInput.value = '';
    }
    event.preventDefault();
}

```

```

function onMessageReceived(payload) {
    var message = JSON.parse(payload.body);

    var messageElement = document.createElement('li');

    if(message.type === 'JOIN') {
        messageElement.classList.add('event-message');
        message.content = message.sender + ' joined!';
    }
}

```

```

    } else if (message.type === 'LEAVE') {

        messageElement.classList.add('event-message');

        message.content = message.sender + ' left!';

    } else {

        messageElement.classList.add('chat-message');

        var avatarElement = document.createElement('i');

        var avatarText =
document.createTextNode(message.sender[0]);

        avatarElement.appendChild(avatarText);

        avatarElement.style['background-color'] =
getAvatarColor(message.sender);

        messageElement.appendChild(avatarElement);

        var usernameElement = document.createElement('span');

        var usernameText =
document.createTextNode(message.sender);

        usernameElement.appendChild(usernameText);

        messageElement.appendChild(usernameElement);

    }

    var textElement = document.createElement('p');

    var messageText = document.createTextNode(message.content);

    textElement.appendChild(messageText);

    messageElement.appendChild(textElement);

    messageArea.appendChild(messageElement);

    messageArea.scrollTop = messageArea.scrollHeight;

}

```

```
function getAvatarColor(messageSender) {
    var hash = 0;
    for (var i = 0; i < messageSender.length; i++) {
        hash = 31 * hash + messageSender.charCodeAt(i);
    }
    var index = Math.abs(hash % colors.length);
    return colors[index];
}

usernameForm.addEventListener('submit', connect, true)
messageForm.addEventListener('submit', sendMessage, true)
```

Aqui, fazemos uso da função `connect()`, `SocketJS` e `stomp` no cliente para se conectar ao endpoint `/ws` que configuramos na aplicação springboot.

Após a conexão bem-sucedida, o cliente assina o destino `/topic/public` e informa o nome do usuário ao servidor, enviando uma mensagem ao destino `/app/chat.addUser`.

A função `stompClient.subscribe()` usa um método de retorno de chamada que é chamado sempre que uma mensagem chega ao tópico inscrito.

O restante do código é usado para exibir e formatar as mensagens na tela.

### 3. Adicionando CSS - `main.css`

Finalmente, adicione os seguintes estilos ao arquivo `main.css`:

```
* {
    -webkit-box-sizing: border-box;
    -moz-box-sizing: border-box;
    box-sizing: border-box;
}
```

```
html,body {  
    height: 100%;  
    overflow: hidden;  
}  
  
body {  
    margin: 0;  
    padding: 0;  
    font-weight: 400;  
    font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;  
    font-size: 1rem;  
    line-height: 1.58;  
    color: #333;  
    background-color: #f4f4f4;  
    height: 100%;  
}  
  
body:before {  
    height: 50%;  
    width: 100%;  
    position: absolute;  
    top: 0;  
    left: 0;  
    background: #128ff2;  
    content: "";  
    z-index: 0;  
}  
  
.clearfix:after {  
    display: block;
```

```
        content: "";
        clear: both;
    }

    .hidden {
        display: none;
    }

    .form-control {
        width: 100%;
        min-height: 38px;
        font-size: 15px;
        border: 1px solid #c8c8c8;
    }

    .form-group {
        margin-bottom: 15px;
    }

    input {
        padding-left: 10px;
        outline: none;
    }

    h1, h2, h3, h4, h5, h6 {
        margin-top: 20px;
        margin-bottom: 20px;
    }

    h1 {
        font-size: 1.7em;
```

```
}

a {
  color: #128ff2;
}

button {
  box-shadow: none;
  border: 1px solid transparent;
  font-size: 14px;
  outline: none;
  line-height: 100%;
  white-space: nowrap;
  vertical-align: middle;
  padding: 0.6rem 1rem;
  border-radius: 2px;
  transition: all 0.2s ease-in-out;
  cursor: pointer;
  min-height: 38px;
}

button.default {
  background-color: #e8e8e8;
  color: #333;
  box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.12);
}

button.primary {
  background-color: #128ff2;
  box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.12);
  color: #fff;
}
```

```
}

button.accent {
    background-color: #ff4743;
    box-shadow: 0 2px 2px 0 rgba(0, 0, 0, 0.12);
    color: #fff;
}

#username-page {
    text-align: center;
}

.username-page-container {
    background: #fff;
    box-shadow: 0 1px 11px rgba(0, 0, 0, 0.27);
    border-radius: 2px;
    width: 100%;
    max-width: 500px;
    display: inline-block;
    margin-top: 42px;
    vertical-align: middle;
    position: relative;
    padding: 35px 55px 35px;
    min-height: 250px;
    position: absolute;
    top: 50%;
    left: 0;
    right: 0;
    margin: 0 auto;
    margin-top: -160px;
}
```

```
.username-page-container .username-submit {  
    margin-top: 10px;  
}  
  
#chat-page {  
    position: relative;  
    height: 100%;  
}  
  
.chat-container {  
    max-width: 700px;  
    margin-left: auto;  
    margin-right: auto;  
    background-color: #fff;  
    box-shadow: 0 1px 11px rgba(0, 0, 0, 0.27);  
    margin-top: 30px;  
    height: calc(100% - 60px);  
    max-height: 600px;  
    position: relative;  
}  
  
#chat-page ul {  
    list-style-type: none;  
    background-color: #FFF;  
    margin: 0;  
    overflow: auto;  
    overflow-y: scroll;  
    padding: 0 20px 0px 20px;  
    height: calc(100% - 150px);
```



```
}

#chat-page #messageForm {
    padding: 20px;
}

#chat-page ul li {
    line-height: 1.5rem;
    padding: 10px 20px;
    margin: 0;
    border-bottom: 1px solid #f4f4f4;
}

#chat-page ul li p {
    margin: 0;
}

#chat-page .event-message {
    width: 100%;
    text-align: center;
    clear: both;
}

#chat-page .event-message p {
    color: #777;
    font-size: 14px;
    word-wrap: break-word;
}

#chat-page .chat-message {
    padding-left: 68px;
```

```
        position: relative;
    }

    #chat-page .chat-message i {
        position: absolute;
        width: 42px;
        height: 42px;
        overflow: hidden;
        left: 10px;
        display: inline-block;
        vertical-align: middle;
        font-size: 18px;
        line-height: 42px;
        color: #fff;
        text-align: center;
        border-radius: 50%;
        font-style: normal;
        text-transform: uppercase;
    }

    #chat-page .chat-message span {
        color: #333;
        font-weight: 600;
    }

    #chat-page .chat-message p {
        color: #43464b;
    }

    #messageForm .input-group input {
        float: left;
```

```
        width: calc(100% - 85px);
    }

    #messageForm .input-group button {

        float: left;

        width: 80px;

        height: 38px;

        margin-left: 5px;
    }

    .chat-header {

        text-align: center;

        padding: 15px;

        border-bottom: 1px solid #ececec;
    }

    .chat-header h2 {

        margin: 0;

        font-weight: 500;
    }

    .connecting {

        padding-top: 5px;

        text-align: center;

        color: #777;

        position: absolute;

        top: 65px;

        width: 100%;
    }
```

```
@media screen and (max-width: 730px) {

    .chat-container {

        margin-left: 10px;

        margin-right: 10px;

        margin-top: 10px;

    }

}

@media screen and (max-width: 480px) {

    .chat-container {

        height: calc(100% - 30px);

    }

    .username-page-container {

        width: auto;

        margin-left: 15px;

        margin-right: 15px;

        padding: 25px;

    }

    #chat-page ul {

        height: calc(100% - 120px);

    }

    #messageForm .input-group button {

        width: 65px;

    }

    #messageForm .input-group input {

        width: calc(100% - 70px);

    }

}
```

```
}

.chat-header {
    padding: 10px;
}

.connecting {
    top: 60px;
}

.chat-header h2 {
    font-size: 1.1em;
}
}
```

### Executando o aplicativo

Você pode executar o aplicativo Spring Boot digitando o seguinte comando em seu terminal ou pelo IDE Eclipse. Pelo terminal, insira o comando abaixo na prompt:

```
$ mvn spring-boot:run
```

O aplicativo é iniciado na porta padrão **8080** do Spring Boot. Você pode navegar pelo aplicativo em <http://localhost:8080>.

## Projeto Spring Boot integrado com Angular 9

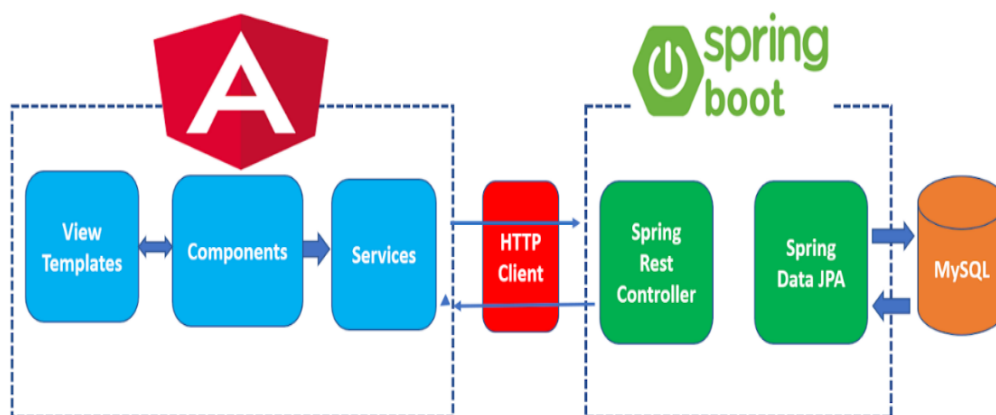
Neste projeto, aprenderemos como desenvolver um aplicativo da Web CRUD (Criar, Ler, Atualizar, Excluir) usando Angular 9 como front-end e Spring boot 2 restful API como back-end. Você desenvolverá seu

primeiro aplicativo FULL STACK com Angular 9 e Spring Boot.

### Por que usamos a versão Angular 9?

O novo recurso mais proeminente do Angular 9 é Ivy . Ivy é o novo compilador e renderizador do Angular. O renderizador é o mecanismo que pega seus componentes e modelos e os traduz em instruções que manipulam o DOM. Ivy é um componente interno, então você não interage diretamente com ele. No entanto, pode ter um impacto significativo em seu código, gerando pacotes JavaScript muito menores e aumentando o desempenho.

### Arquitetura do projeto



### Implementação de recursos

- Criar um employee
- Atualizar um employee
- Lista de employees
- Excluir employee

- Ver Employee

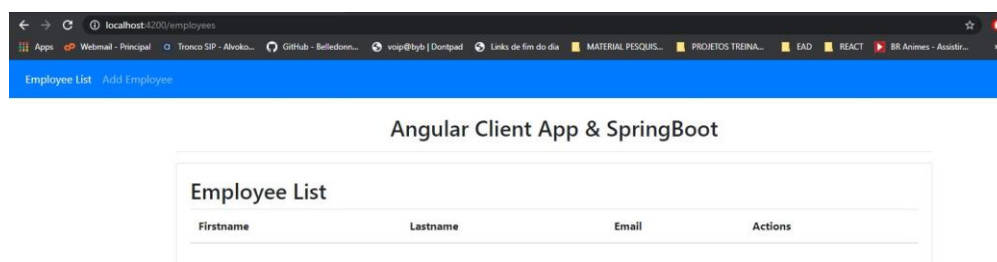
## O que vamos construir?

Basicamente, iremos criar dois projetos (Cliente e Servidor):

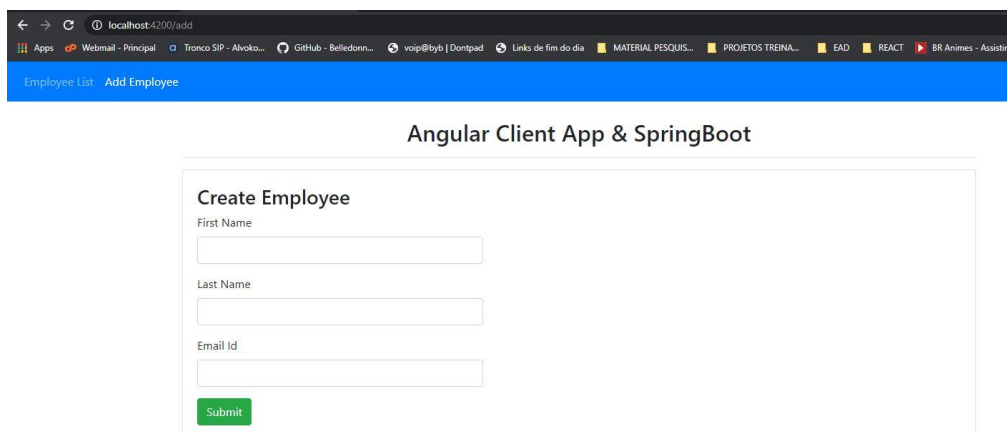
1. **springboot2-jpa-crud-example** : Este projeto é usado para desenvolver APIs CRUD RESTful para um **sistema de gerenciamento de employees** simples usando Spring Boot 2, JPA e MySQL como banco de dados.
2. **angular9-springboot-client** : Este projeto é usado para desenvolver um aplicativo de página única usando Angular 9 como tecnologia de front-end. Este aplicativo Angular 9 consome APIs CRUD Restful desenvolvidas e expostas por um projeto **springboot2-jpa-crud-example**.

Abaixo estão as capturas de tela que mostram a IU do nosso **aplicativo de sistema de gerenciamento de employees**:

### Página de lista de employees



### Página Adicionar(add) employee



Angular Client App & SpringBoot

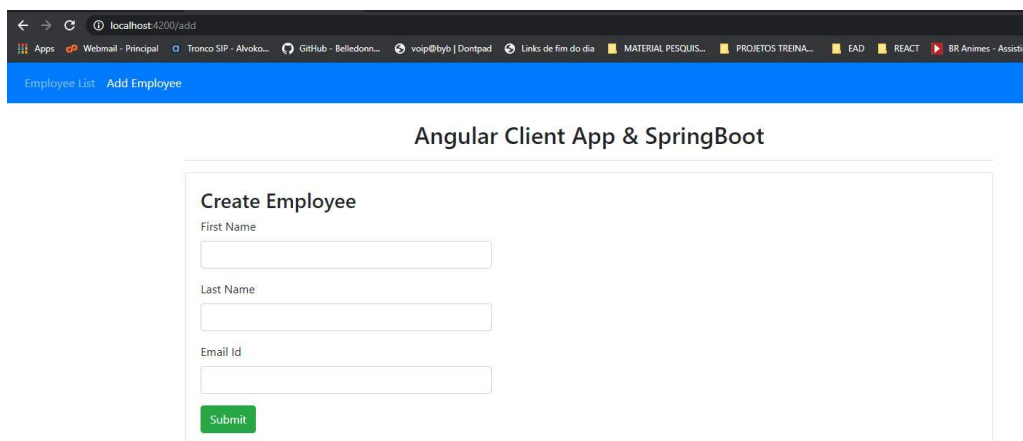
Create Employee

First Name

Last Name

Email Id

## Atualizar (update) página do employee



Angular Client App & SpringBoot

Create Employee

First Name

Last Name

Email Id

## página de detalhes (details) do employee

## Excluir employee

## Desenvolvimento de APIs Spring Boot CRUD Rest

A seguir estão cinco APIs REST (métodos manipuladores de controlador(controller)), que desenvolveremos para o recurso **Employee**.



Sr. No.	API Name	HTTP Method	Path	Status Code	Description
(1)	GET Employees	GET	/api/v1/employees	200 (OK)	All Employee resources are fetched.
(2)	POST Employee	POST	/api/v1/employees	201 (Created)	A new Employee resource is created.
(3)	GET Employee	GET	/api/v1/employees/{id}	200 (OK)	One Employee resource is fetched.
(4)	PUT Employee	PUT	/api/v1/employees/{id}	200 (OK)	Employee resource is updated.
(5)	DELETE Employee	DELETE	/api/v1/employees/{id}	204 (No Content)	Employee resource is deleted.

## 1. Criação e importação do projeto

Podemos usar o recurso **Spring Initializr** em <http://start.spring.io/>, que é um gerador de aplicativos Spring Boot online.

The screenshot shows the Spring Initializr web application interface. It includes sections for Project (Maven Project selected), Language (Java selected), Spring Boot (2.3.4 selected), Project Metadata (Group: com.example, Artifact: angularspringboot, Name: angularspringboot, Description: Springboot Angular Project, Package name: com.example.angularspringboot), and Dependencies (Spring Web selected). At the bottom, there are buttons for GENERATE, EXPLORE, and SHARE.

Observe o diagrama acima, especificamos os seguintes detalhes:

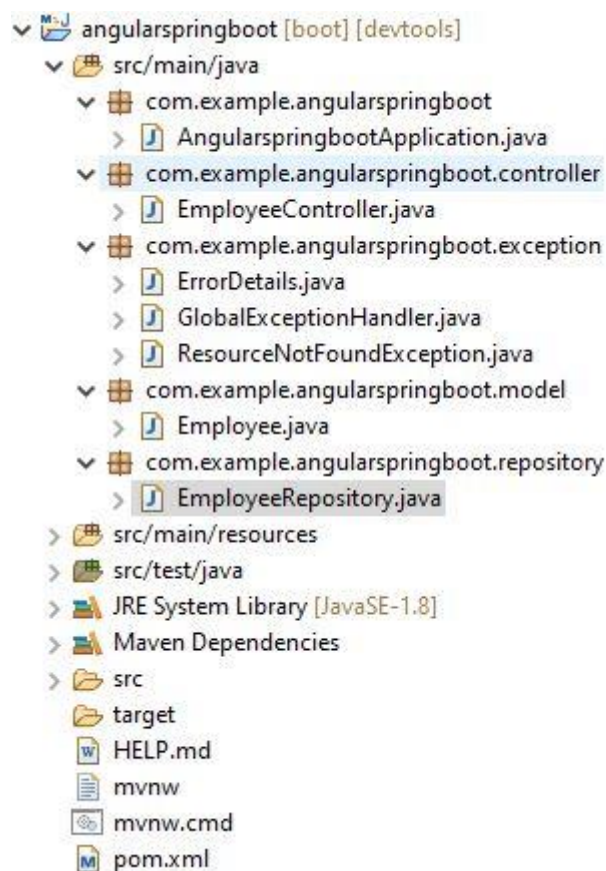
- **Project** : Maven
- **Versão Java** : 8
- **Spring Boot** : 2.3.4
- **Grupo** : com.example
- **Artifact** : angularspringboot
- **Name** : angularspringboot
- **Description**: Springboot Angular Project
- **PackageName** : com.example.amgularspringbooot
- **pacote** : jar (este é o valor padrão)

- **Dependencies** : Web, JPA, MySQL, DevTools

Depois que todos os detalhes forem inseridos, clique no botão Gerar projeto para gerar um projeto de inicialização de spring e baixá-lo. Em seguida, descompacte o arquivo zip baixado e importe-o em seu IDE favorito.

## 2. Estrutura de pasta e recurso do projeto

A seguir está a estrutura de embalagem do nosso **Sistema de Gestão de Employees**:



## 3. O arquivo pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>
<artifactId>angularspringboot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>angularspringboot</name>
<description>Sprinboot Angular Project</description>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.3.4.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
  <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-
jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-
plugin</artifactId>
    </plugin>
  </plugins>

```

```
        </build>  
</project>
```

#### 4. Configurando o banco de dados (H2 ou MySQL)

Neste projeto, usaremos o banco de dados H2 para configurar e executar rapidamente um projeto de springboot sem instalar bancos de dados.

Observe que adicionamos a dependência abaixo no arquivo *pom.xml* :

```
<dependency>  
    <groupId>com.h2database</groupId>  
    <artifactId>h2</artifactId>  
    <scope>runtime</scope>  
</dependency>
```

Se você usar o banco de dados H2, não precisamos configurar as propriedades relacionadas ao banco de dados no arquivo *application.properties* .

Usando o banco de dados H2 in-memory, não será necessário criar um banco de dados e tabelas, o spring boot faz isso automaticamente para você.

Neste projeto, vamos usar o caminho de contexto para o projeto de springboot, então vamos adicionar a propriedade abaixo no arquivo *application.properties* :

```
server.servlet.context-path=/springboot-crud-rest
```

#### 5. Criar Entidade JPA - Employee.java

```
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;
```

```

import javax.persistence.Table;

@Entity
@Table(name = "employees")
public class Employee {

    private long id;
    private String firstName;
    private String lastName;
    private String emailId;

    public Employee() {

    }

    public Employee(String firstName, String lastName, String
emailId) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.emailId = emailId;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }

    @Column(name = "first_name", nullable = false)
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @Column(name = "last_name", nullable = false)
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Column(name = "email_address", nullable = false)
    public String getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }

    @Override
    public String toString() {
        return "Employee [id=" + id + ", firstName=" + firstName
+ ", lastName=" + lastName + ", emailId=" + emailId
+ "]";
    }
}

```

```
}
```

## 6. Criar um Spring Data Repository - EmployeeRepository.java

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import
net.guides.springboot2.springboot2jpacrudexample.model.Employee;

@Repository
public interface EmployeeRepository extends
JpaRepository<Employee, Long>{

}
```

## 7. Criar controlador Spring Rest - EmployeeController.java

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
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.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import
net.guides.springboot2.springboot2jpacrudexample.exception.ResourceNot
FoundException;
import
net.guides.springboot2.springboot2jpacrudexample.model.Employee;
import
net.guides.springboot2.springboot2jpacrudexample.repository.EmployeeRe
pository;

@RestController @CrossOrigin(origins = "http://localhost:4200")
@RequestMapping("/api/v1")
public class EmployeeController {
    @Autowired
    private EmployeeRepository employeeRepository;

    @GetMapping("/employees")
```

```

        public List<Employee> getAllEmployees() {
            return employeeRepository.findAll();
        }

        @GetMapping("/employees/{id}")
        public ResponseEntity<Employee>
        getEmployeeById(@PathVariable(value = "id") Long employeeId)
            throws ResourceNotFoundException {
            Employee employee =
            employeeRepository.findById(employeeId)
                .orElseThrow(() -> new
            ResourceNotFoundException("Employee not found for this id :: " +
            employeeId));
            return ResponseEntity.ok().body(employee);
        }

        @PostMapping("/employees")
        public Employee createEmployee(@Valid @RequestBody Employee
        employee) {
            return employeeRepository.save(employee);
        }

        @PutMapping("/employees/{id}")
        public ResponseEntity<Employee>
        updateEmployee(@PathVariable(value = "id") Long employeeId,
            @Valid @RequestBody Employee employeeDetails) throws
            ResourceNotFoundException {
            Employee employee =
            employeeRepository.findById(employeeId)
                .orElseThrow(() -> new
            ResourceNotFoundException("Employee not found for this id :: " +
            employeeId));

            employee.setEmailId(employeeDetails.getEmailId());
            employee.setLastName(employeeDetails.getLastName());
            employee.setFirstName(employeeDetails.getFirstName());
            final Employee updatedEmployee =
            employeeRepository.save(employee);
            return ResponseEntity.ok(updatedEmployee);
        }

        @DeleteMapping("/employees/{id}")
        public Map<String, Boolean>
        deleteEmployee(@PathVariable(value = "id") Long employeeId)
            throws ResourceNotFoundException {
            Employee employee =
            employeeRepository.findById(employeeId)
                .orElseThrow(() -> new
            ResourceNotFoundException("Employee not found for this id :: " +
            employeeId));

            employeeRepository.delete(employee);
            Map<String, Boolean> response = new HashMap<>();
            response.put("deleted", Boolean.TRUE);
            return response;
        }
    }

```

## Habilitar CORS no servidor

Para habilitar o CORS no servidor, adicione uma annotation `@CrossOrigin` ao `EmployeeController` como :

```
@CrossOrigin(origins = "http://localhost:4200")
@RestController
@RequestMapping("/api/v1")
public class EmployeeController {
    // ....
}
```

## Recurso não presente

Aqui está o que acontece quando você dispara uma solicitação para um recurso não encontrado: <http://localhost:8080/some-dummy-url>

```
{
  "timestamp": 1512713804164,
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/some-dummy-url"
}
```

Essa é uma resposta de erro. Ele contém todos os detalhes normalmente necessários.

## O que acontece quando lançamos uma exceção?

Vamos ver o que Spring Boot faz quando uma exceção é lançada de um `Resource`. podemos especificar o Status de Resposta para uma exceção específica junto com a definição da Exceção com a annotation `'@ResponseStatus'`.



Vamos criar uma classe `ResourceNotFoundException.java`.

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends Exception{

    private static final long serialVersionUID = 1L;

    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

### Personalização da estrutura de resposta a erros

A resposta de erro padrão fornecida pelo Spring Boot contém todos os detalhes que são normalmente necessários.

No entanto, você pode desejar criar uma estrutura de resposta independente de estrutura para sua organização. Nesse caso, você pode definir uma estrutura de resposta de erro específica.

Vamos definir um bean de resposta de erro simples.

```
import java.util.Date;

public class ErrorDetails {
    private Date timestamp;
    private String message;
    private String details;

    public ErrorDetails(Date timestamp, String message, String
details) {
        super();
        this.timestamp = timestamp;
        this.message = message;
        this.details = details;
    }

    public Date getTimestamp() {
        return timestamp;
    }

    public String getMessage() {
        return message;
    }

    public String getDetails() {
```

```

        return details;
    }
}

```

Para usar `ErrorDetails` para retornar a resposta de erro, vamos criar uma classe `GlobalExceptionHandler` anotada com a annotation `@ControllerAdvice`. Esta classe trata exceções específicas e globais em um único lugar.

```

import java.util.Date;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;

@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<?>
resourceNotFoundException(ResourceNotFoundException ex, WebRequest
request) {
        ErrorDetails errorDetails = new ErrorDetails(new
Date(), ex.getMessage(), request.getDescription(false));
        return new ResponseEntity<>(errorDetails,
HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<?> globalexceptionHandler(Exception
ex, WebRequest request) {
        ErrorDetails errorDetails = new ErrorDetails(new Date(),
ex.getMessage(), request.getDescription(false));
        return new ResponseEntity<>(errorDetails,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

```

## 9. Aplicação em execução

Este aplicativo springboot possui uma classe Java de ponto de entrada chamada `Application.java` com o método `public static void main (String [] args)`, que você pode executar para iniciar o aplicativo.

```

import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

```

O método `main ()` usa o método `SpringApplication.run ()` do Spring Boot para iniciar um aplicativo. Ou você pode iniciar o aplicativo spring boot via linha de comando usando `mvn spring-boot: run` command.

### Teste de APIs REST

Use os endpoints abaixo Rest para testar CRUD Rest APIs e no aplicativo Angular.

#### Obtenha todos os employees:

Método HTTP: GET

```

http: // localhost: 8080 / springboot-crud-rest / api / v1
/ workers

```

Obter employee por id:

Método HTTP GET

```

http: // localhost: 8080 / springboot-crud-rest / api / v1
/ employees / {employeeId}

```

**Criar employee:**

Método HTTP - POST

```
http: // localhost: 8080 / springboot-crud-rest / api / v1  
/ employees
```

**Atualização (update) Employee**

Método HTTP - POST

```
http: // localhost: 8080 / springboot-crud-rest / api / v1  
/ employees / {employeeId}
```

**Excluir (delete )employee por id:**

Método HTTP - DELETE

```
http: // localhost: 8080 / springboot-crud-rest / api / v1  
/ employees / {employeeId}
```

Isso conclui o desenvolvimento das APIs do Spring boot CRUD Rest.

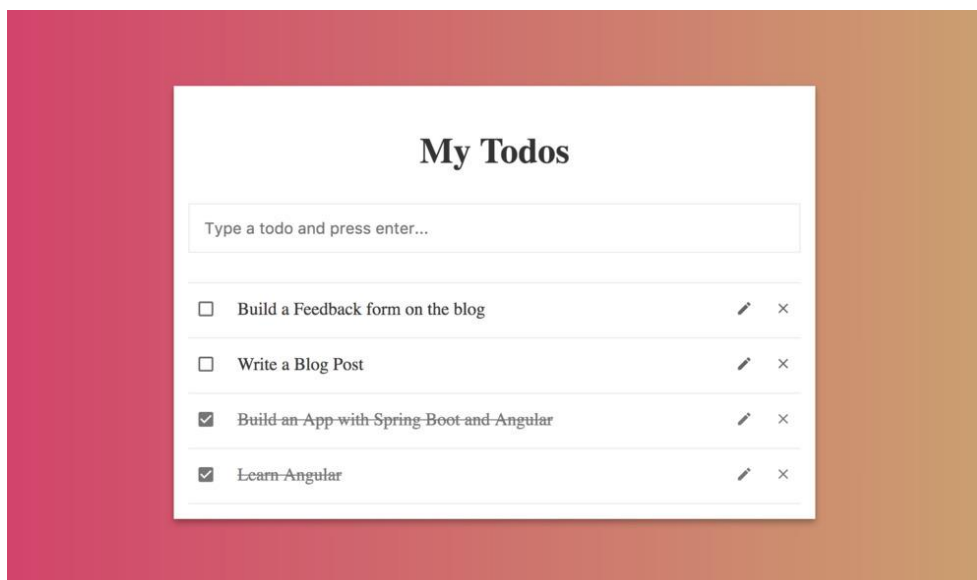
**Projeto Spring Boot com Angular eAPI Rest MongoDB**

**Angular** é uma das estruturas de JavaScript mais populares com ferramentas, velocidade e desempenho incríveis.

Se você quiser aprender como usar o Angular e construir um aplicativo full stack com Spring Boot no back-end e Angular no front-end, esta postagem do blog é para você.

Neste post, vamos construir um aplicativo de tarefas completo com Spring Boot e MongoDB no back-end e Angular no front-end.

A seguir está uma captura de tela do todo-app que iremos construir neste projeto.



Vamos construir esse projeto passo-a-passo como construir o aplicativo do zero.

## Criando back-end Spring Boot

### 1. Inicializando o projeto

Usaremos o aplicativo da web Spring Initializer para gerar nosso projeto de springboot.

- Vá para <http://start.spring.io/> .
- Insira o valor do artefato como **todoapp** .
- Adicione **Spring Web** e **Spring Data MongoDB** na seção de dependências.
- Clique em **Gerar** para gerar e baixar o projeto.

Depois de fazer o download do projeto, descompacte-o e importe-o em seu IDE favorito.

## 2. Configurando o banco de dados MongoDB

O Spring Boot tenta configurar automaticamente a maioria das coisas para você com base nas dependências que você adicionou no arquivo `pom.xml`.

Como adicionamos a dependência `spring-boot-starter-mongodb`, Spring Boot tenta construir uma conexão com MongoDB lendo a configuração do banco de dados do arquivo `application.properties`.

Abra o `application.properties` arquivo e adicione as seguintes propriedades mongodb:

```
# MONGODB (MongoProperties)
spring.data.mongodb.uri=mongodb://localhost:27017/todoapp
```

Observe que, para que a configuração acima funcione, você precisa ter o MongoDB instalado em seu sistema.

Saída do [doc oficial mongodb](#) para instalar MongoDB em seu sistema.

Se você não deseja instalar o MongoDB localmente, pode usar um dos provedores de banco de dados como serviço gratuitos mongodb, como o [MongoLab](#).

[O MongoLab](#) oferece 500 MB de dados em seu plano gratuito. Você pode criar um banco de dados com seu plano gratuito. Depois de criar um banco de dados, você obterá um uri de conexão mongodb no seguinte formato:

```
mongodb://<dbuser>:<dbpassword>@ds161169.mlab.com:61169/testdatabase
```

Basta adicionar o uri de conexão ao arquivo `application.properties` e você estará pronto para prosseguir.

### 3. Criação do *TodoModel*

Vamos agora criar o *TodoModel* que será mapeado para um Documento no banco de dados mongodb. Crie um novo pacote chame-o de `models` dentro `com.example.todoapp` e adicione um arquivo `Todo.java` dentro dele com o seguinte conteúdo:

```
package com.example.todoapp.models;

import java.util.Date;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.index.Indexed;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection="todos")
@JsonIgnoreProperties(value = {"createdAt"}, allowGetters =
true)

public class Todo {

    @Id

    private String id;
```

```
@NotBlank

@Size(max=100)

@Indexed(unique=true)

private String title;


private Boolean completed = false;


private Date createdAt = new Date();


public Todo() {
    super();
}


public Todo(String title) {
    this.title = title;
}


public String getId() {
    return id;
}


public void setId(String id) {
    this.id = id;
}


public String getTitle() {
    return title;
}


public void setTitle(String title) {
```



```

        this.title = title;
    }

    public Boolean getCompleted() {
        return completed;
    }

    public void setCompleted(Boolean completed) {
        this.completed = completed;
    }

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
        this.createdAt = createdAt;
    }

    @Override
    public String toString() {
        return String.format(
            "Todo[id=%s, title='%s', completed='%s']",
            id, title, completed);
    }
}

```

Anotamos `title` com a `@Indexed` annotations e marcamos como único. Isso cria um índice exclusivo no campo de título.

Além disso, certificamo-nos de que o título da tarefa não está em branco anotando-o com uma `@NotBlank` annotation.

A `@JsonIgnoreProperties` annotation é usada para ignorar o campo `createdAt` durante a desserialização. Não queremos que a aplicação-client enviem o valor `createdAt`. Se enviar um valor para este campo, iremos simplesmente ignorá-lo.

#### 4. Criação `TodoRepository` para acessar o banco de dados

Em seguida, precisamos criar `TodoRepository` para acessar dados do banco de dados.

**Primeiro**, crie um novo pacote `repositories` dentro `com.example.todoapp`.

**Em seguida**, crie uma interface chamada `TodoRepository` dentro do `repositories` pacote com o seguinte conteúdo:

```
package com.example.todoapp.repositories;

import com.example.todoapp.models.Todo;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface TodoRepository extends MongoRepository<Todo,
String> {

}
```

Nós estendemos `TodoRepository` a interface `MongoRepository` fornecida por `spring-data-mongodb`. A `MongoRepository` interface define métodos para todas as operações CRUD no documento como `findAll()`, `findOne()`, `save()`, `delete()` etc.

O Spring Boot se conecta automaticamente a uma implementação de `MongoRepository` interface chamada `SimpleMongoRepository` em tempo de

execução. Portanto, todos os métodos CRUD definidos por `MongoRepository` estão prontamente disponíveis para você, sem fazer nada.

## 5. Criação das APIs - `TodoController`

Por fim, vamos criar as APIs que serão expostas ao client. Crie um novo pacote `controllers` dentro `com.example.todoapp` e adicione um arquivo `TodoController.java` dentro do pacote `controllers` com o seguinte código:

```
package com.example.todoapp.controllers;

import javax.validation.Valid;
import com.example.todoapp.models.Todo;
import com.example.todoapp.repositories.TodoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/api")
@CrossOrigin("*")
public class TodoController {

    @Autowired
    TodoRepository todoRepository;

    @GetMapping("/todos")
    public List<Todo> getAllTodos() {
        Sort sortByCreatedAtDesc = new Sort(Sort.Direction.DESC,
"createdAt");
        return todoRepository.findAll(sortByCreatedAtDesc);
    }

    @PostMapping("/todos")
    public Todo createTodo(@Valid @RequestBody Todo todo) {
        todo.setCompleted(false);
        return todoRepository.save(todo);
    }

    @GetMapping(value="/todos/{id}")
    public ResponseEntity<Todo> getTodoById(@PathVariable("id")
String id) {
        return todoRepository.findById(id)
            .map(todo -> ResponseEntity.ok().body(todo))
            .orElse(ResponseEntity.notFound().build());
    }

    @PutMapping(value="/todos/{id}")
    public ResponseEntity<Todo> updateTodo(@PathVariable("id")
String id,
```

```

                                                                    @Valid @RequestBody
    Todo todo) {
        return todoRepository.findById(id)
            .map(todoData -> {
                todoData.setTitle(todo.getTitle());
                todoData.setCompleted(todo.getCompleted());
                Todo updatedTodo =
            todoRepository.save(todoData);
            return
            ResponseEntity.ok().body(updatedTodo);
            }).orElse(ResponseEntity.notFound().build());
    }

    @DeleteMapping(value="/todos/{id}")
    public ResponseEntity<?> deleteTodo(@PathVariable("id")
    String id) {
        return todoRepository.findById(id)
            .map(todo -> {
                todoRepository.deleteById(id);
                return ResponseEntity.ok().build();
            }).orElse(ResponseEntity.notFound().build());
    }
}

```

A annotation **@CrossOrigin** no controlador acima é usada para habilitar solicitações de origem cruzada. Isso é necessário porque acessaremos as apis do servidor front-end do angular.

Nosso trabalho de back-end está concluído. Você pode executar o aplicativo usando:

```
mvn spring-boot:run
```

O aplicativo será iniciado na porta 8080. Você pode testar o backend apis usando o carteiro ou qualquer outro cliente restante de sua escolha.

Vamos agora trabalhar no frontend Angular.

Criação do front-end angular

1. Gerando o aplicativo usando angular-cli

**Angular-cli** é uma ferramenta de linha de comando para criar aplicativos angulares prontos para produção. Ajuda na criação, teste, agrupamento e implementação de um projeto angular.

Você pode instalar o angular digitando o seguinte comando em seu terminal:

```
$ npm install -g @angular/cli
```

Depois de instalado, você pode gerar um novo projeto usando o comando `ng new`

```
$ ng new angular-frontend
```

## 2. Executando o aplicativo

Vamos primeiro executar o aplicativo gerado pelo angular-cli e depois alterar e adicionar os arquivos necessários para o nosso aplicativo. Para executar o aplicativo, vá para o diretório raiz do aplicativo e digite `ng serve`.

```
$ cd angular-frontend  
$ ng serve --open
```

A `--open` opção é usada para abrir automaticamente o aplicativo em seu navegador padrão. O aplicativo será carregado em seu navegador padrão em <http://localhost:4200> e exibirá uma mensagem de boas-vindas:

## 3. Alterar o modelo para AppComponent `(app.component.html)`

O modelo para a página de boas-vindas padrão que você vê é definido em `app.component.html`. Remova tudo neste arquivo e adicione o seguinte código:

```
<main>

  <todo-list></todo-list>

</main>
```

A `<todo-list>` tag será usada para exibir o componente `TodoList` nesta página. Definiremos o `TodoListComponent` em breve.

#### 4. Criar classe `Todo` `todo.ts`

Antes de definir o `TodoListComponent`, vamos definir uma classe `Todo` para trabalhar com `Todos`. Crie um novo arquivo `todo.ts` dentro da pasta `src/app` e adicione o seguinte código a ele:

```
export class Todo {
  id: string;
  title: string;
  completed: boolean;
  createdAt: Date;
}
```

#### 5. Crie `TodoListComponent` `todo-list.component.ts`

Vamos agora definir o `TodoListComponent`. Ele será usado para exibir uma lista de tarefas, criar uma nova tarefa, editar e atualizar uma tarefa.

Crie um novo arquivo `todo-list.component.ts` dentro do diretório `src/app` e adicione o seguinte código a ele:

```
import { Component, OnInit } from '@angular/core';
import { Todo } from '../todo';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'todo-list',
  templateUrl: '../todo-list.component.html'
})
```

```

    })

    export class TodoListComponent implements OnInit {
        todos: Todo[];
        newTodo: Todo = new Todo();
        editing: boolean = false;
        editingTodo: Todo = new Todo();

        ngOnInit(): void {
            this.getTodos();
        }

        getTodos(): void {

        }

        createTodo(todoForm: NgForm): void {

        }

        deleteTodo(id: string): void {

        }

        updateTodo(todoData: Todo): void {

        }

        toggleCompleted(todoData: Todo): void {

        }

        editTodo(todoData: Todo): void {

        }

        clearEditing(): void {

        }
    }
}

```

O seletor para `TodoListComponent` é `todo-list`. Lembre-se, usamos a tag `<todo-list>` no `app.component.html` para nos referir a `TodoListComponent`.

Implementaremos todos os métodos declarados no componente na próxima seção. Mas, antes disso, precisamos declarar o `TodoListComponent` dentro `app.module.ts`.

Basta importar `TodoListComponent` e adicioná-lo ao array de declarações dentro `@NgModule`:

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';

import { AppComponent } from './app.component';

import { TodoListComponent } from './todo-list.component';

@NgModule({
  declarations: [
    AppComponent,
    TodoListComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Observe que, adicionamos `FormsModule` e `HttpClientModule` dentro `app.module.ts` porque vamos precisar deles para lidar com a vinculação de formulários e chamar as APIs restantes, respectivamente.

## 6. Crie um modelo para `TodoListComponent` `todo-list.component.html`

A seguir, vamos definir o modelo para `TodoListComponent`. Crie um novo arquivo `todo-list.component.html` dentro do diretório `src/app` e adicione o seguinte código a ele:

```

<div class="todo-content">
  <h1 class="page-title">My Todos</h1>
  <div class="todo-create">
    <form #todoForm="ngForm" (ngSubmit)="createTodo(todoForm)" "
novalidate>
      <input type="text" id="title" class="form-control"
placeholder="Type a todo and press enter..."
      required
      name="title" [(ngModel)]="newTodo.title"
      #title="ngModel" >
      <div *ngIf="title.errors && title.dirty"
class="alert alert-danger">

```



```

        <div [hidden]="!title.errors.required">
            Title is required.
        </div>
    </div>
</form>
</div>
<ul class="todo-list">
    <li *ngFor="let todo of todos"
[class.completed]="todo.completed === true" >
        <div class="todo-row" *ngIf="!editing || editingTodo.id
!= todo.id">
            <a class="todo-completed"
(click)="toggleCompleted(todo)">
                <i class="material-icons toggle-completed-
checkbox"></i>
            </a>
            <span class="todo-title">
                {{todo.title}}
            </span>
            <span class="todo-actions">
                <a (click)="editTodo(todo)">
                    <i class="material-icons edit">edit</i>
                </a>
                <a (click)="deleteTodo(todo.id)">
                    <i class="material-icons delete">clear</i>
                </a>
            </span>
        </div>
        <div class="todo-edit" *ngIf="editing && editingTodo.id
=== todo.id">
            <input class="form-control" type="text"
[(ngModel)]="editingTodo.title" required>
            <span class="edit-actions">
                <a (click)="updateTodo(editingTodo)">
                    <i class="material-icons">done</i>
                </a>
                <a (click)="clearEditing()">
                    <i class="material-icons">clear</i>
                </a>
            </span>
        </div>
    </li>
</ul>
<div class="no-todos" *ngIf="todos && todos.length == 0">
    <p>No Todos Found!</p>
</div>
</div>

```

O modelo contém o código para exibir uma lista de todos e métodos de chamada para editar um todo, deletar um todo e marcar um todo como concluído.

## 7. Adicionar Estilos `styles.css`

Vamos adicionar alguns estilos para fazer nosso modelo parecer bom. Observe que estou adicionando todos os estilos no arquivo

global `styles.css`. Mas, você pode definir `styles` relacionado a um componente em um arquivo css separado e fazer referência a esse css definindo `styleUrls` no `@Component` *decorator*.

Abra o arquivo `styles.css` localizado na pasta `src` e adicione os seguintes estilos:

```
/* You can add global styles to this file, and also import other
style files */

body {
  font-size: 18px;
  line-height: 1.58;
  background: #d53369;
  background: -webkit-linear-gradient(to left, #cbad6d,
#d53369);
  background: linear-gradient(to left, #cbad6d, #d53369);
  color: #333;
}

h1 {
  font-size: 36px;
}

* {
  box-sizing: border-box;
}

i {
  vertical-align: middle;
  color: #626262;
}

input {
  border: 1px solid #E8E8E8;
}

.page-title {
  text-align: center;
}

.todo-content {
  max-width: 650px;
  width: 100%;
  margin: 0 auto;
  margin-top: 60px;
  background-color: #fff;
  padding: 15px;
  box-shadow: 0 0 4px rgba(0,0,0,.14), 0 4px 8px
rgba(0,0,0,.28);
}

.form-control {
  font-size: 16px;
  padding-left: 15px;
}
```

```

    outline: none;
    border: 1px solid #E8E8E8;
}

.form-control:focus {
    border: 1px solid #626262;
}

.todo-content .form-control {
    width: 100%;
    height: 50px;
}

.todo-content .todo-create {
    padding-bottom: 30px;
    border-bottom: 1px solid #e8e8e8;
}

.todo-content .alert-danger {
    padding-left: 15px;
    font-size: 14px;
    color: red;
    padding-top: 5px;
}

.todo-content ul {
    list-style: none;
    margin: 0;
    padding: 0;
    max-height: 450px;
    padding-left: 15px;
    padding-right: 15px;
    margin-left: -15px;
    margin-right: -15px;
    overflow-y: scroll;
}

.todo-content ul li {
    padding-top: 10px;
    padding-bottom: 10px;
    border-bottom: 1px solid #E8E8E8;
}

.todo-content ul li span {
    display: inline-block;
    vertical-align: middle;
}

.todo-content .todo-title {
    width: calc(100% - 160px);
    margin-left: 10px;
    overflow: hidden;
    text-overflow: ellipsis;
}

.todo-content .todo-completed {
    display: inline-block;
    text-align: center;
    width: 35px;
    height: 35px;
    cursor: pointer;
}

```

```

}

.todo-content .todo-completed i {
  font-size: 20px;
}

.todo-content .todo-actions, .todo-content .edit-actions {
  float: right;
}

.todo-content .todo-actions i, .todo-content .edit-actions i {
  font-size: 17px;
}

.todo-content .todo-actions a, .todo-content .edit-actions a {
  display: inline-block;
  text-align: center;
  width: 35px;
  height: 35px;
  cursor: pointer;
}

.todo-content .todo-actions a:hover, .todo-content .edit-actions
a:hover {
  background-color: #f4f4f4;
}

.todo-content .todo-edit input {
  width: calc(100% - 80px);
  height: 35px;
}

.todo-content .edit-actions {
  text-align: right;
}

.no-todos {
  text-align: center;
}

.toggle-completed-checkbox:before {
  content: 'check_box_outline_blank';
}

li.completed .toggle-completed-checkbox:before {
  content: 'check_box';
}

li.completed .todo-title {
  text-decoration: line-through;
  color: #757575;
}

li.completed i {
  color: #757575;
}

```

## 8. Adicione ícones de materiais em `index.html`

Estamos usando ícones de materiais para exibir o botão de edição, o botão de exclusão e a caixa de seleção. Basta adicionar a tag `<link>` ao arquivo `src/index.html`:

```
<link rel="stylesheet"
href="https://fonts.googleapis.com/icon?family=Material+Icons">
```

## 9. Crie `TodoService` `todo.service.ts`

O `TodoService` será usado para obter os dados do backend chamando `spring boot apis`. Crie um novo arquivo `todo.service.ts` dentro do diretório `src/app` e adicione o seguinte código a ele:

```
import { Injectable } from '@angular/core';
import { Todo } from '../todo';
import { Headers, Http } from '@angular/http';
import 'rxjs/add/operator/toPromise';

@Injectable()
export class TodoService {
  private baseUrl = 'http://localhost:8080';

  constructor(private http: Http) { }

  getTodos(): Promise<Todo[]> {
    return this.http.get(this.baseUrl + '/api/todos/')
      .toPromise()
      .then(response => response.json() as Todo[])
      .catch(this.handleError);
  }

  createTodo(todoData: Todo): Promise<Todo> {
    return this.http.post(this.baseUrl + '/api/todos/', todoData)
      .toPromise().then(response => response.json() as Todo)
      .catch(this.handleError);
  }

  updateTodo(todoData: Todo): Promise<Todo> {
    return this.http.put(this.baseUrl + '/api/todos/' +
      todoData.id, todoData)
      .toPromise()
      .then(response => response.json() as Todo)
      .catch(this.handleError);
  }

  deleteTodo(id: string): Promise<any> {
    return this.http.delete(this.baseUrl + '/api/todos/' + id)
      .toPromise()
      .catch(this.handleError);
  }
}
```

```

    }

    private handleError(error: any): Promise<any> {
      console.error('Some error occurred', error);
      return Promise.reject(error.message || error);
    }
  }
}

```

Precisamos declarar `TodoService` dentro `app.module.ts` para podermos usá-lo em nossos componentes.

Abra `app.module.ts` e adicione a seguinte instrução de importação:

```

// Inside app.module.ts

import { TodoService } from './todo.service';

```

Em seguida, adicione `TodoService` dentro do array de providers:

```

// Inside app.module.ts

providers: [TodoService]

```

## 10. Implementar métodos `TodoListComponent`

Finalmente, implementaremos métodos para criar, recuperar, atualizar e excluir todos em nosso `TodoListComponent`.

Certifique-se de que seu `todo-list.component.ts` arquivo corresponda ao seguinte:

```

import { Component, Input, OnInit } from '@angular/core';
import { TodoService } from './todo.service';
import { Todo } from './todo';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'todo-list',
  templateUrl: './todo-list.component.html'
})

export class TodoListComponent implements OnInit {
  todos: Todo[];

```

```

newTodo: Todo = new Todo();
editing: boolean = false;
editingTodo: Todo = new Todo();

constructor(
  private todoService: TodoService,
) {}

ngOnInit(): void {
  this.getTodos();
}

getTodos(): void {
  this.todoService.getTodos()
    .then(todos => this.todos = todos );
}

createTodo(todoForm: NgForm): void {
  this.todoService.createTodo(this.newTodo)
    .then(createTodo => {
      todoForm.reset();
      this.newTodo = new Todo();
      this.todos.unshift(createTodo)
    });
}

deleteTodo(id: string): void {
  this.todoService.deleteTodo(id)
    .then(() => {
      this.todos = this.todos.filter(todo => todo.id !== id);
    });
}

updateTodo(todoData: Todo): void {
  console.log(todoData);
  this.todoService.updateTodo(todoData)
    .then(updatedTodo => {
      let existingTodo = this.todos.find(todo => todo.id ===
updatedTodo.id);
      Object.assign(existingTodo, updatedTodo);
      this.clearEditing();
    });
}

toggleCompleted(todoData: Todo): void {
  todoData.completed = !todoData.completed;
  this.todoService.updateTodo(todoData)
    .then(updatedTodo => {
      let existingTodo = this.todos.find(todo => todo.id ===
updatedTodo.id);
      Object.assign(existingTodo, updatedTodo);
    });
}

editTodo(todoData: Todo): void {
  this.editing = true;
  Object.assign(this.editingTodo, todoData);
}

clearEditing(): void {
  this.editingTodo = new Todo();
}

```

```
        this.editing = false;  
    }  
}
```

### Executar servidores de back-end e front-end e testar o aplicativo

Você pode executar o servidor back-end de springboot digitando `mvn spring-boot:run` no terminal. Ele será executado na porta 8080.

```
$ cd todoapp  
$ mvn spring-boot:run
```

O front-end Angular pode ser executado digitando `ng serve` command. Vai começar na porta 4200:

```
$ cd angular-frontend  
$ ng serve --open
```



## Apendice

### Especificar o diretório para upload de arquivos

Abra – a partir da aplicação upload/download de arquivos springboot - o arquivo `src/main/resources/application.properties` e altere a propriedade `file.upload-dir` para o path onde você quer que os arquivos sejam armazenados.

```
file.upload-dir=/Users/seuusuario/uploads
```

## Referencias:

<https://www.javaguides.net/2018/09/getting-started-with-spring-boot.html>

<http://www.bosontreinamentos.com.br/java/programacao-em-java-instalando-o-ide-eclipse/>

<https://www.javaguides.net/2018/10/spring-boot-annotations.html>

<https://www.callicoder.com/spring-boot-file-upload-download-rest-api-example/>

<https://www.javaguides.net/2019/04/spring-boot-thymeleaf-crud-example-projeto.html>

<https://www.callicoder.com/spring-boot-websocket-chat-example/>

<https://www.javaguides.net/2020/01/spring-boot-angular-9-crud-example-projeto.html>

<https://www.callicoder.com/spring-boot-mongodb-angular-js-rest-api-projeto/>