

PRODUTIVIDADE NO DESENVOLVIMENTO
DE APLICAÇÕES WEB COM
SPRING BOOT



Produtividade no Desenvolvimento de Aplicações Web com Spring Boot

3ª Edição, 22/09/2017

© 2017 AlgaWorks Softwares, Treinamentos e Serviços Ltda. Todos os direitos reservados.

Nenhuma parte deste livro pode ser reproduzida ou transmitida em qualquer forma, seja por meio eletrônico ou mecânico, sem permissão por escrito da AlgaWorks, exceto para resumos breves em revisões e análises.

AlgaWorks Softwares, Treinamentos e Serviços Ltda

www.algaworks.com

contato@algaworks.com

+55 (11) 2626-9415

Siga-nos nas redes sociais e fique por dentro de tudo!

A blue square button with the word "Facebook" in white text.

Facebook

A red square button with the word "YouTube" in white text.

YouTube

A purple square button with the word "Instagram" in white text.

Instagram

CURSO ONLINE



**DESENVOLVIMENTO WEB
COM SPRING**

COMPRA AQUI



Sobre o autor



Alexandre Afonso

Instrutor Java na AlgaWorks, graduado em Sistemas de Informação, está no mercado de programação Java há mais de 8 anos, principalmente no desenvolvimento de sistemas corporativos.

Antes de começar...

Antes que você comece a ler esse livro, eu gostaria de combinar algumas coisas com você, para que tenha um excelente aproveitamento do conteúdo. Vamos lá?

O que você precisa saber?

Você só conseguirá absorver o conteúdo desse livro se já conhecer pelo menos o básico de Java, Orientação a Objetos e HTML. Caso você ainda não domine esses assuntos, pode ser interessante estudar por [nossos cursos online](#).

Como obter ajuda?

Durante os estudos, é muito comum surgir várias dúvidas. Nós gostaríamos muito de te ajudar pessoalmente nesses problemas, mas infelizmente não conseguimos fazer isso com todos os leitores do livro, afinal, ocupamos grande parte do dia ajudando os alunos de cursos online na AlgaWorks.

Então, quando você tiver alguma dúvida e não conseguir encontrar a solução no Google ou com seu próprio conhecimento, nossa recomendação é que você poste na nossa Comunidade Java no Facebook. É só acessar:

<http://alga.works/comunidadejava/>

Como sugerir melhorias ou reportar erros sobre este livro?

Se você encontrar algum erro no conteúdo desse livro ou se tiver alguma sugestão para melhorar a próxima edição, vamos ficar muito felizes se você puder nos dizer.

Envie um e-mail para livros@algaworks.com.

Onde encontrar o código-fonte do projeto?

Neste livro, nós vamos desenvolver um pequeno projeto passo a passo. O link para baixar o código-fonte foi enviado para seu e-mail quando você se inscreveu para receber o livro.

Caso você tenha perdido esse link, acesse <http://alga.works/livro-spring-boot/> para recebê-lo novamente.

Ajude na continuidade desse trabalho

Escrever um livro (mesmo que pequeno, como esse) dá muito trabalho, por isso, esse projeto só faz sentido se muitas pessoas tiverem acesso a ele.

Ajude a divulgar esse livro para seus amigos que também se interessam por programação Java. Compartilhe no Facebook e Twitter!



Sumário

1 Introdução

2 O Spring

2.1	Introdução	12
2.2	Spring vs Java EE	12
2.3	O Spring Framework	13
2.4	Injeção de dependências com Spring	14
2.5	O Spring MVC	15
2.6	O Spring Data JPA	17
2.7	O Spring Security	18
2.8	O Spring Boot	19
2.9	O Thymeleaf	20
2.10	O Maven	21
2.11	O Spring Tool Suite - STS	21

3 O projeto de gestão de convidados

3.1	Funcionamento	24
3.2	Criando o projeto no STS	25
3.3	Criando o controller	30
3.4	Criando a página	33
3.5	Rodando o projeto pela primeira vez	37
3.6	Repositório de convidados	39

3.7	Enviando um objeto do Controller para a View	41
3.8	Listando objetos com o Thymeleaf	42
3.9	Adicionando um convidado	45
3.10	Escolhendo o banco de dados	49
3.11	Deixando a aplicação segura	50

4 Publicando a aplicação

4.1	Introdução	53
4.2	Usando o Postgres	54
4.3	Alteração de estratégia para chave primária	55
4.4	Criando o arquivo de inicialização	56
4.5	Instalando o Git	56
4.6	Configurando a aplicação no Heroku	57
4.7	Enviando a aplicação	59

5 Conclusão

5.1	Próximos passos	62
-----	-----------------------	----

Capítulo 1

Introdução

Às vezes, a parte mais difícil para quem está começando uma nova aplicação Java web, mesmo se esse programador já conhece a linguagem, é justamente começar!

Você precisa criar a estrutura de diretórios para os vários arquivos, além de criar e configurar o *build file* com as dependências.

Se você já é programador Java para web, sempre que precisa criar um novo projeto, o que você faz? É possível que sua resposta seja: “eu crio um novo projeto e vou copiando as configurações de outro que já está funcionando”.

Se você é iniciante, seu primeiro passo será procurar algum tutorial que te ensine a criar o projeto do zero, e então copiar e colar todas as configurações no seu ambiente de desenvolvimento até ter o “hello world” funcionando.

Esses são cenários comuns no desenvolvimento web com Java, quando estamos usando ferramentas como Eclipse e Maven simplesmente, mas existem alternativas a este “castigo inicial” da criação de um novo projeto, e neste livro você vai aprender um caminho mais fácil e prazeroso para criar um projeto web em Java com Spring Framework.

Nós vamos criar uma aplicação simples com Spring Boot, Spring MVC, Spring Data JPA, Spring Security e Thymeleaf, usando o Spring Tool Suite (STS), uma IDE baseada no Eclipse que vem com o Spring Initializr (não está escrito errado, é Initializr mesmo), uma ferramenta muito útil para criar nossos projetos com Spring.

Capítulo 2

O Spring

2.1. Introdução

O Spring não é um framework apenas, mas um conjunto de projetos que resolvem várias situações do cotidiano de um programador, ajudando a criar aplicações Java com simplicidade e flexibilidade.

Existem muitas áreas cobertas pelo ecossistema Spring, como Spring Data, para acesso a banco de dados, Spring Security, para prover segurança, e diversos outros projetos que vão de *cloud computing* até *big data*.

O Spring surgiu como uma alternativa ao Java EE, e seus criadores sempre se preocuparam para que ele fosse o mais simples e leve possível.

Desde a sua primeira liberação, em Outubro de 2002, o Spring tem evoluído muito, com diversos projetos maduros, seguros e robustos para utilizarmos em produção.

Os projetos Spring são *Open Source*. Você pode ver o código-fonte no [GitHub](https://github.com).

2.2. Spring vs Java EE

O Spring não chega a ser 100% concorrente do Java EE, até porque, com Spring, você também usa tecnologias que estão dentro do Java EE.

Mas existem programadores que preferem trabalhar com os projetos do Spring, e outros que preferem trabalhar com as especificações do Java EE, sem Spring.

Como o Spring é independente de especificação, novos projetos são lançados e testados muito mais rapidamente.

De fato, existem vários projetos que são lançados em beta para a comunidade, e caso exista uma aceitação geral, ele vai pra frente, com o apoio de uma grande massa de desenvolvedores.

2.3. O Spring Framework

É fácil confundir todo o ecossistema Spring com apenas o Spring Framework. Mas, o Spring Framework é apenas um, dentre o conjunto de projetos, que o Spring possui.

Ele é o projeto do Spring que serve de base para todos os outros, talvez por isso haja essa pequena confusão.

O Spring Framework foi pensado para que nossas aplicações pudessem focar mais na regra de negócio e menos na infraestrutura.

Dentre suas principais funcionalidades, podemos destacar:

- O Spring MVC
- Suporte para JDBC, JPA
- Injeção de dependências (Dependency injection – DI)

O Spring MVC, que falaremos melhor mais a frente, é um framework para criação de aplicações web e serviços RESTful. Ele é bastante conveniente, pois muitas das aplicações que construímos hoje em dia precisam atender requisições web.

Com o mesmo pensamento do Spring MVC, temos o suporte para JDBC e JPA. Isso porque persistência também é um recurso muito utilizado nas aplicações. É difícil conceber uma aplicação hoje que não grave ou consulte algum banco de dados.

Por último, assim como o Spring Framework é a base do ecossistema, a injeção de dependências é a base do Spring Framework. Imagino que os projetos Spring seriam quase que uma bagunça caso esse conceito não tivesse sido implementado.

2.4. Injeção de dependências com Spring

Injeção de dependências (ou Dependency Injection – DI) é um tipo de inversão de controle (ou Inversion of Control – IoC) que dá nome ao processo de prover instâncias de classes que um objeto precisa para funcionar.

A grande vantagem desse conceito é que nós conseguimos programar voltados para interfaces e, com isso, manter o baixo acoplamento entre as classes de um mesmo projeto.

Com certeza, essa característica é uma grande vantagem para a arquitetura do seu sistema, assim como é para o próprio Spring.

Como foi dito mais no início, essa funcionalidade é a base de todo o ecossistema Spring. É difícil pensar em Spring sem a injeção de dependências. Sendo assim, esse é um conceito que merece a atenção da sua parte.

Para entender um pouco de como é na prática, ao invés de você fazer isso:

```
public class ServicoCliente {  
  
    // Nesse exemplo, estou supondo que "RepositorioCliente" é uma  
    // interface. Mas poderia ser uma classe abstrata ou  
    // mesmo uma classe concreta  
    private RepositorioCliente repositorio = new RepositorioClienteImpl();  
  
    ...  
}
```

Você vai fazer isso:

```
public class ServicoCliente {  
  
    @Autowired
```

```
private RepositorioCliente repositorio;  
  
...  
}
```

A anotação `@Autowired` avisa ao Spring Framework para injetar uma instância de alguma implementação da interface `RepositorioCliente` na propriedade `repositorio`.

A priori, pode parecer insignificante, mas o Spring não para por aí. Tem várias outras funcionalidades que são possíveis ou, pelo menos, facilitadas por causa da injeção de dependências.

2.5. O Spring MVC

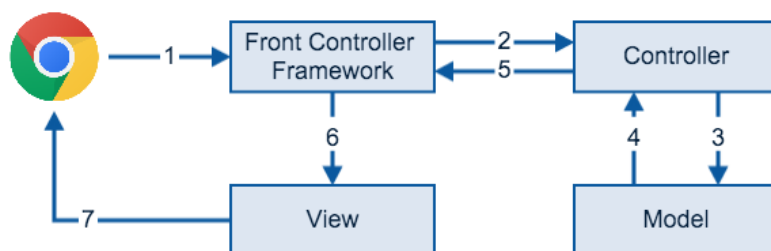
Dentre os projetos Spring, o Spring MVC é o framework que te ajuda no desenvolvimento de aplicações web robustas, flexíveis e com uma clara separação de responsabilidades nos papéis do tratamento da requisição.

MVC é acrônimo de Model, View e Controller, e entender bem o que cada um deve fazer na aplicação é importante para termos uma aplicação bem escrita e fácil para dar manutenção.

Vamos parar um pouco e pensar no que fazemos todos os dias quando estamos na internet.

Primeiro abrimos um browser (Chrome, Safari, Firefox), digitamos um endereço na “barra de endereços”, damos um “Enter” e pronto. Se nada der errado, uma página HTML será renderizada.

Mas, o que acontece entre o “Enter” e a página HTML ser renderizada? Claro que existem centenas de linguagens de programação e frameworks diferentes, mas nós vamos pensar no contexto do Spring MVC.



1. Acessamos uma URL no browser, que envia a requisição HTTP para o servidor que roda a aplicação web com Spring MVC. Esse servidor pode ser o [Apache Tomcat](#), por exemplo. Perceba que quem recebe a requisição é o controlador do framework, o Spring MVC.
2. O controlador do framework irá procurar qual classe é responsável por tratar essa requisição, entregando a ela os dados enviados pelo browser. Essa classe faz o papel do *controller*.
3. O *controller* passa os dados para o *model*, que por sua vez executa todas as regras de negócio, como cálculos, validações e acesso ao banco de dados.
4. O resultado das operações realizadas pelo *model* é retornado ao *controller*.
5. O *controller* retorna o nome da *view*, junto com os dados que ela precisa para renderizar a página.
6. O framework encontra a *view*, que processa os dados, transformando o resultado em um HTML.
7. Finalmente, o HTML é retornado ao browser do usuário.

Pare um pouco e volte na figura acima, leia mais uma vez todos os passos, desde a requisição do browser até a página ser renderizada.

Como você deve ter notado, temos o *Controller* tratando a requisição. Ele é o primeiro componente que nós vamos programar para receber os dados enviados pelo usuário.

Mas é muito importante estar atento e não cometer erros adicionando regras de negócio, acessando banco de dados ou fazendo validações nessa camada, precisamos passar essa responsabilidade para o *Model*.

No *Model*, pensando em algo prático, é o local certo para usarmos o *JPA/Hibernate* para salvar ou consultar algo no banco de dados, é onde iremos calcular o valor do frete para entrega de um produto, por exemplo.

A *View* irá “desenhar”, renderizar e transformar em HTML esses dados, para que o usuário consiga visualizar a informação, pois enquanto estávamos no *Controller* e no *Model*, estávamos programando em classes Java, e não em algo visual para o browser exibir ao usuário.

Essa é a ideia do MVC, separar claramente a responsabilidade de cada componente dentro de uma aplicação. Por quê? Para facilitar a manutenção do seu código, temos baixo acoplamento, e isso é uma boa prática de programação.

2.6. O Spring Data JPA

Spring Data JPA é um dos projetos do Spring Data.

O Spring Data é um projeto que tem o objetivo de simplificar o acesso a tecnologias de armazenamento de dados, sejam elas relacionais (MySQL, PostgreSQL, etc) ou não (MongoDB, Redis, etc).

O Spring Data já possui vários projetos dentro dele, como:

- Spring Data Commons
- Spring Data JPA
- Spring Data Gemfire
- Spring Data KeyValue
- Spring Data LDAP
- Spring Data MongoDB
- Spring Data REST
- Spring Data Redis
- Spring Data for Apache Cassandra

- Spring Data for Apache Solr

Mas aqui nós vamos focar no mais utilizado, que é o Spring Data JPA.

O suporte do Spring Framework para JPA já é muito bom, mas o projeto Spring Data JPA vai bem além. Ele ajuda os desenvolvedores padronizando o uso de algumas funcionalidades, e isso faz com que tenhamos menos esforço para implementar as mesmas coisas.

Um exemplo disso seria a implementação padrão que ele já nos dá em repositórios, incluindo métodos como `save`, `delete`, `findOne`, entre outros.

Esse exemplo não foi dado à toa. Mais para frente, faremos o uso da interface `JpaRepository`, que nos ajudará com essa funcionalidade.

2.7. O Spring Security

Como o próprio nome diz, esse projeto trata da segurança em nível de aplicação. Ele tem um suporte excelente para autenticação e autorização.

Spring Security torna bem simples a parte de autenticação. Com algumas poucas configurações, já podemos ter uma autenticação via banco de dados, LDAP ou mesmo por memória. Sem falar nas várias integrações que ele suporta e na possibilidade de criar as suas próprias.

Quanto a autorização, ele é bem flexível também. Através das permissões que atribuímos aos usuários autenticados, podemos proteger as requisições web (como as telas do nosso sistema, por exemplo), a simples invocação de um método e até a instância de um objeto.

Sem contar que, por tabela, nós já protegemos as nossas aplicações de diversos ataques como o Session Fixation e o Cross Site Request Forgery.

2.8. O Spring Boot

Enquanto os componentes do Spring eram simples, sua configuração era extremamente complexa e cheia de XMLs. Depois de um tempo, a partir da versão 3.0, a configuração pôde ser feita através de código Java.

A configuração via código Java trouxe benefícios, como evitar erros de digitação, porque a classe de configuração precisa ser compilada. Mas nós precisamos escrever muito código explicitamente.

Com toda essa configuração excessiva, o desenvolvedor perde o foco na coisa mais importante: desenvolver as regras de negócio da aplicação.

Talvez a maior revolução e o maior acerto dos projetos Spring, foi o Spring Boot. Com ele você alcança um novo paradigma para desenvolver aplicações Spring com pouco esforço.

O Spring Boot trouxe agilidade, e te possibilita focar nas funcionalidades da sua aplicação com o mínimo de configuração.

Vale destacar que, toda essa “mágica” que o Spring Boot traz para o desenvolvimento Spring, não é realizado com geração de código. Não, o Spring Boot não gera código! Ele simplesmente analisa o projeto e automaticamente o configura.

É claro que é possível customizar as configurações, mas o Spring Boot segue o padrão que a maioria das aplicações precisa, então, muitas vezes não é preciso fazer nada.

Se você já trabalha com o [Maven](#), sabe que precisamos adicionar várias dependências no arquivo *pom.xml*, que pode ficar extenso, com centenas de linhas de configuração, mas com o Spring Boot podemos reduzir muito com os *starters* que ele fornece.

Os *starters* são dependências que agrupam outras dependências, assim, se você precisa trabalhar com JPA e Hibernate, por exemplo, basta adicionar uma única entrada no *pom.xml*, que todas as dependências necessárias serão adicionadas ao *classpath*.

2.9. O Thymeleaf

A *view* irá retornar apenas um HTML para o browser do cliente, mas isso deixa uma dúvida: Como ela recebe os objetos Java, enviados pelo *controller*, e os transforma em HTML?

Nessa hora que entra em ação o Thymeleaf!

Teremos um código HTML misturado com alguns atributos do Thymeleaf, que após processado, será gerado apenas o HTML para ser renderizado no browser do cliente.

O Thymeleaf não é um projeto Spring, mas uma biblioteca que foi criada para facilitar a criação da camada de *view*, com uma forte integração com o Spring, e uma boa alternativa ao JSP.

O principal objetivo do Thymeleaf é prover uma forma elegante e bem formatada para criarmos nossas páginas. O dialeto do Thymeleaf é bem poderoso, como você verá no desenvolvimento da aplicação, mas você também pode estendê-lo para customizar, de acordo com suas necessidades.

Para você ver como ele funciona, vamos analisar o código abaixo.

```
<tr th:each="convidado : ${convidados}">
  <td th:text="${convidado.nome}"></td>
  <td th:text="${convidado.quantidadeAcompanhantes}"></td>
</tr>
```

A expressão `${}` interpreta variáveis locais ou disponibilizadas pelo *controller*.

O atributo `th:each` itera sobre a lista `convidados`, atribuindo cada objeto na variável local `convidado`. Isso faz com que vários elementos `tr` sejam renderizados na página.

Dentro de cada `tr` existem 2 elementos `td`. O texto que eles irão exibir vem do atributo `th:text`, junto com a expressão `${}`, lendo as propriedades da variável local `convidado`.

2.10. O Maven

O Maven é uma ferramenta muito importante no dia a dia do desenvolvedor Java, porque com ele nós conseguimos automatizar uma série de tarefas.

Mas talvez o que mais fez o Maven ter sucesso, foi o gerenciamento de dependências. É muito bom poder escrever algumas linhas e já ter a biblioteca disponível para o nosso projeto.

Como começar com Apache Maven?

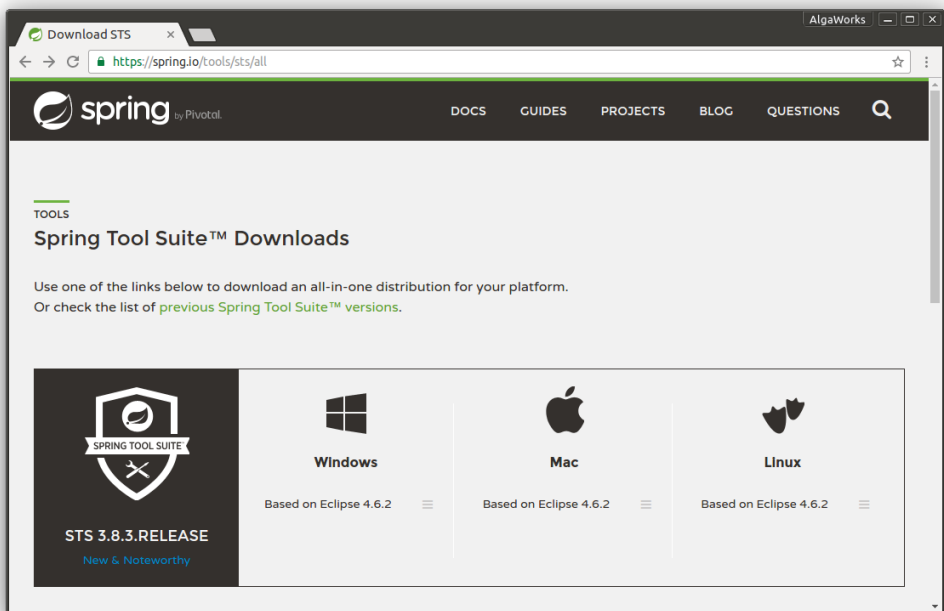
Para saber mais sobre como começar com o Apache Maven, recomendo você assistir esta videoaula gratuita no blog da AlgaWorks.

<http://blog.algaworks.com/comecando-com-apache-maven-em-projetos-java>

2.11. O Spring Tool Suite - STS

O Spring Tool Suite, ou STS, é um Eclipse com vários plugins úteis para o trabalho com o Spring.

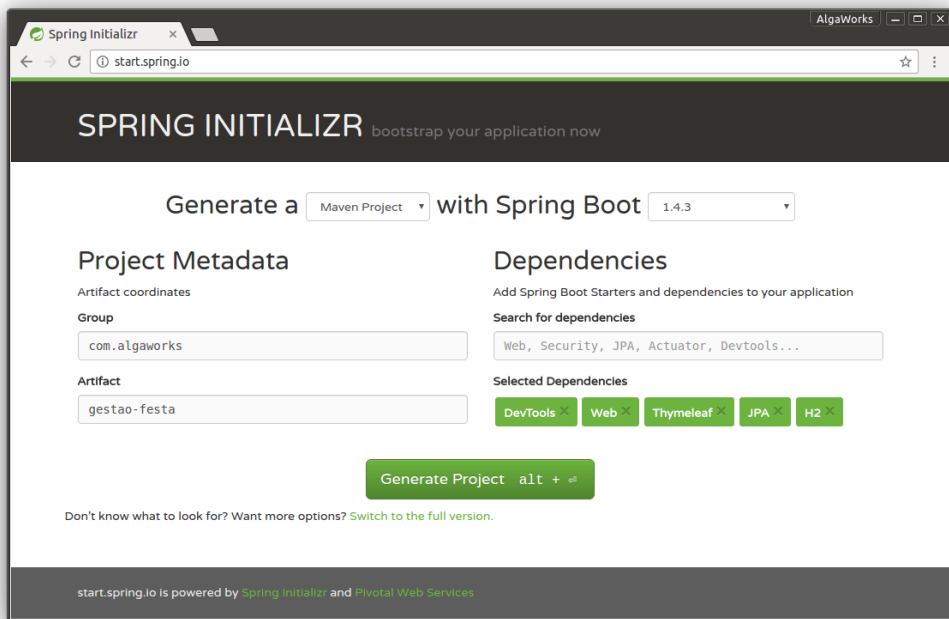
Existem versões para Windows, Mac e Linux em <https://spring.io/tools/sts/all>.



A instalação é como a do Eclipse, basta baixar e descompactar.

Nós vamos criar um pequeno projeto utilizando o STS, para você aprender os primeiros passos com o Spring.

Mas se você já tem uma certa experiência e gosta de outra IDE, não se preocupe, existe uma alternativa para a criação dos seus projetos Spring. Basta acessar o Spring Initializr online em <http://start.spring.io>.



Nesse site você consegue quase a mesma facilidade que vamos alcançar utilizando o STS. Nele, você informa os dados do projeto, frameworks e bibliotecas que deseja ter como dependência, e então um projeto Maven será gerado para ser importado na sua IDE.

O que informar e o que selecionar, vamos ver nos próximos capítulos.

Capítulo 3

O projeto de gestão de convidados

3.1. Funcionamento

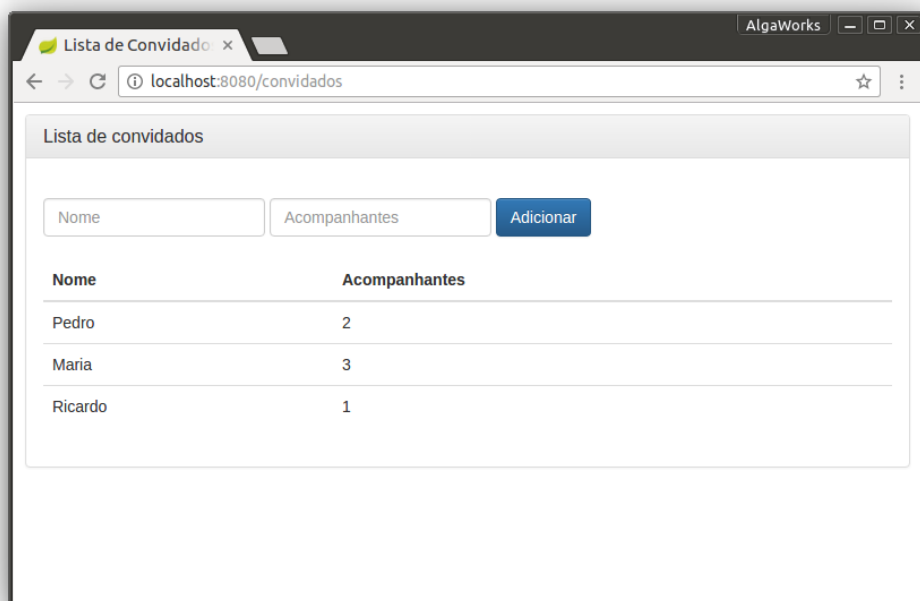
Bora colocar a mão na massa?

Vamos criar uma aplicação simples, do zero e passo a passo, para você ver como o Spring Boot, o Spring MVC, o Spring Data JPA, o Spring Security e o Thymeleaf funcionam juntos, e para isso, vamos usar o Spring Tool Suite e o Maven.

Nossa aplicação será útil para a gestão de convidados em uma festa. Precisamos do nome do convidado principal e a quantidade de acompanhantes que vem com ele.

Na aplicação, teremos uma única tela com dois campos de entrada de texto, um para informar o nome do convidado e o outro para dizer a quantidade de acompanhantes. Por exemplo, podemos cadastrar que o João levará 2 acompanhantes.

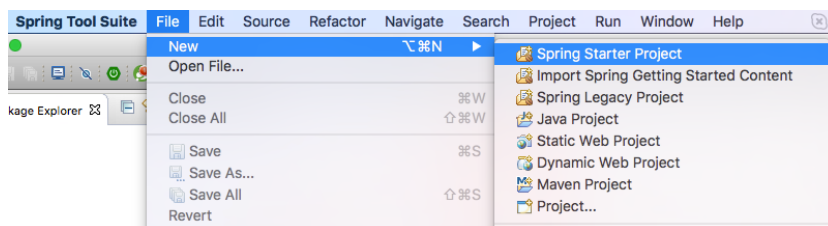
Também teremos um botão “Adicionar” e uma tabela para mostrar o que já foi cadastrado. Veja como será a versão final na imagem abaixo.



3.2. Criando o projeto no STS

Vamos começar criando o projeto no STS.

Com o STS aberto, clique em *File -> New -> Spring Starter Project*, como mostra a imagem abaixo.



Agora, vamos começar a configurar nossa aplicação.

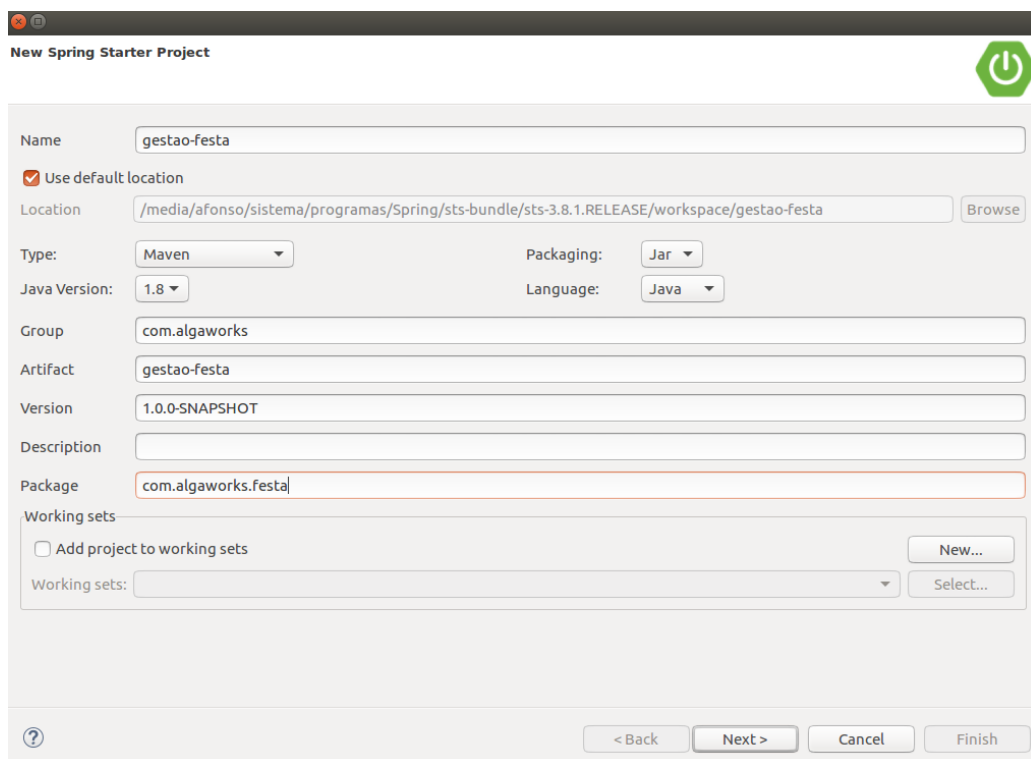
- No campo *Name*, informe o nome do projeto, que será *gestao-festa*
- Em *Type*, selecione *Maven*

- Em *Packaging*, deixe a opção *Jar*
- Para *Java Version*, selecione a versão do Java que está configurada para seu ambiente (recomendo que você use a versão 1.8 ou superior)
- Em *Language*, claro, será Java

Group, *Artifact* e *Version* são informações do Maven para identificar nosso projeto.

- Em *Group*, informe *com.algaworks*
- Em *Artifact*, informe *gestao-festa*
- Em *Version*, informe *1.0.0-SNAPSHOT*. A palavra *SNAPSHOT*, no contexto de um projeto, significa que estamos em uma versão de desenvolvimento, e que se gerarmos um *jar* ou *war* dele, teremos apenas um *momento* do projeto, e não uma versão final.
- Se quiser adicionar uma descrição sobre o que é o projeto, fique a vontade para fazer no campo *Description*.
- E em *Package*, definimos o nome do pacote que deve ser gerado para nossa aplicação. Informe *com.algaworks.festa*

Veja na imagem abaixo a tela preenchida com as informações. Após tudo conferido, clique em *Next*.



New Spring Starter Project

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

Package:

Working sets

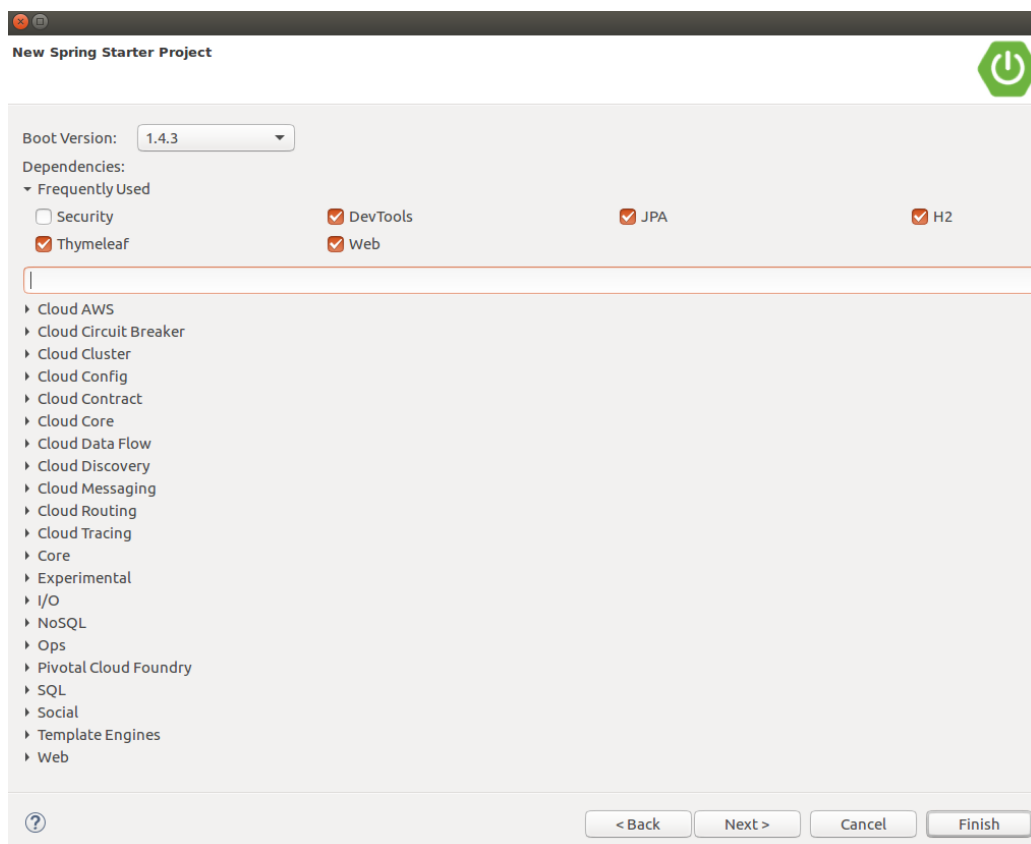
☐ Add project to working sets

Working sets:

Agora é hora de selecionarmos os frameworks ou bibliotecas que nosso sistema precisa. Navegue um pouco pelas opções, clicando nas pequenas setas, como em *Database*, e veja as opções disponíveis.

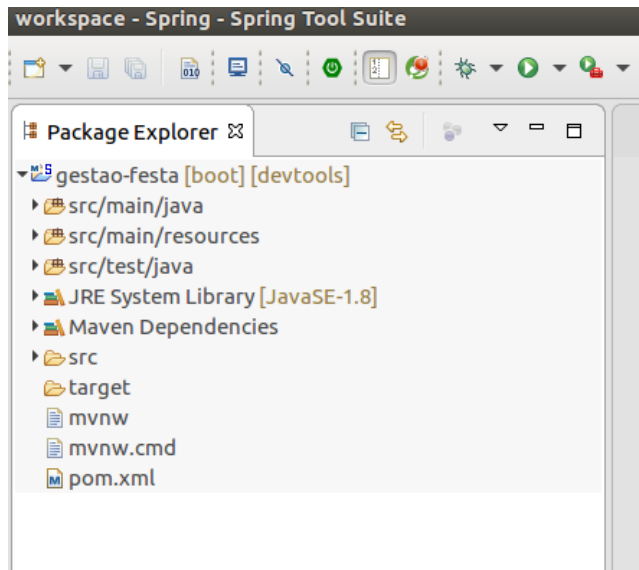
Essa tela é muito útil para iniciarmos o desenvolvimento da nossa aplicação, pois ela é a base para o STS gerar o arquivo pom.xml, ou seja, ao invés de você ter que lembrar o nome completo das dependências do Spring MVC, pode apenas selecionar “Web”.

Nossa aplicação só precisa das opções “DevTools”, “Web”, “Thymeleaf”, “JPA” e “H2” selecionadas. Confira na imagem abaixo, e logo em seguida clique em *Finish*.



Atenção: Se essa for a primeira vez que você faz este procedimento, pode demorar um pouco, pois muitas bibliotecas serão baixadas da internet.

Depois de criado, você verá no “Package Explorer”, posicionado do lado esquerdo no STS, uma estrutura como mostrada na imagem abaixo.



Em `src/main/java`, você encontra o pacote `com.algaworks.festa`, com a classe `GestaoFestaApplication`. Vamos analisar o código dela agora:

```
package com.algaworks.festa;

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

@SpringBootApplication
public class GestaoFestaApplication {

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

O método `main` inicia a nossa aplicação!

Talvez você tenha ficado com uma dúvida agora, pensando assim: “mas esse livro não iria me ensinar sobre Spring MVC, que é uma aplicação web? Cadê o servidor web para executar, algo como o Apache Tomcat?”.

Tudo bem se você pensou isso, mas é sim uma aplicação web.

Acontece que o Spring Boot sugere o uso de um container embarcado (por padrão é o Tomcat) para facilitar o desenvolvimento, então, para iniciar nossa aplicação, basta executarmos o método `main`, da classe `GestaoFestaApplication`.

Analizando o código com mais detalhes, vemos a anotação:

@SpringBootApplication

Essa anotação diz que a classe faz parte da configuração do Spring. Poderíamos adicionar configurações customizadas, como por exemplo, definir o idioma ou até fazer redirecionamentos, caso não encontre uma página, mas como já vimos, o Spring Boot define muitos comportamentos padronizados, e não precisaremos alterar nada para ter a aplicação funcionando.

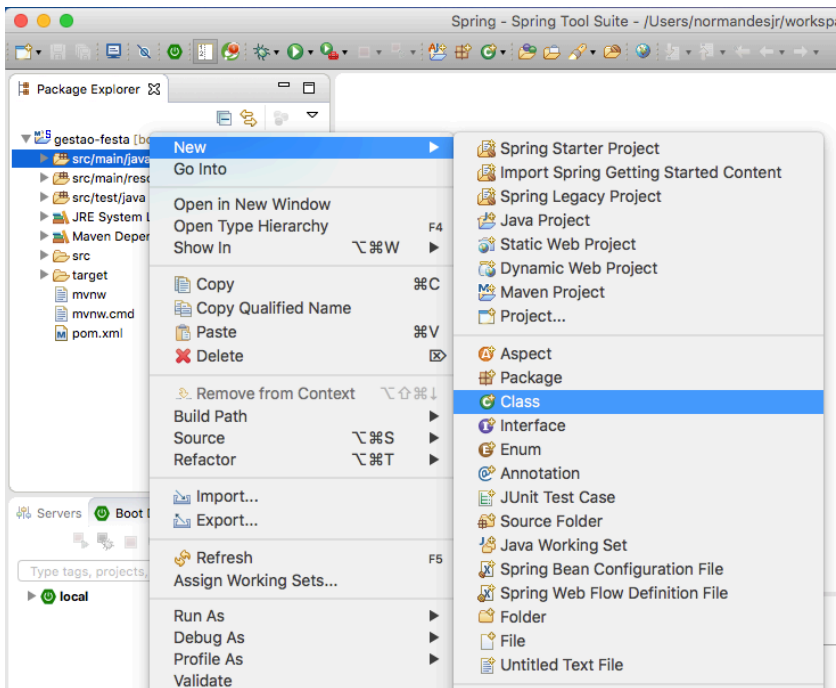
A anotação também define o ponto de partida para a procura dos demais componentes da aplicação, ou seja, todas as classes dos pacotes descendentes de `com.algaworks.festa` serão escaneadas e, se algum componente Spring for encontrado, será gerenciado, e isso facilitará muito a nossa vida (você verá no desenvolvimento da aplicação).

3.3. Criando o controller

Lembra quando vimos que em um framework *action based*, a requisição é entregue ao *controller*? Então, agora é o momento para criarmos essa classe, que receberá a requisição e dirá o nome da *view* ao framework, para então ser renderizada de volta ao browser do cliente.

Para começarmos bem devagar, esse primeiro controller só retornará o nome da *view*, depois vamos incrementar um pouco mais, adicionando objetos para serem usados no template na página.

Vamos criar uma nova classe Java e começar a programá-la.



É importante estar atento ao pacote que a classe irá ficar.

Para deixar nosso código organizado, todo *controller* deverá ficar dentro do pacote `com.algaworks.festa.controller`.

O nome da classe será `ConvidadosController`. Colocando o sufixo *Controller* nos ajuda a lembrar que ela é um controlador, então o nome completo nos faz pensar que essa classe “é o controlador de convidados”.

New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

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

Superclass:

Interfaces:

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

O código é muito simples. Primeiro, vamos anotar a classe com `@Controller`, para dizer que ela é um componente Spring e que é um *controller*.

```
package com.algaworks.festa.controller;
```

```
import org.springframework.stereotype.Controller;
```

```
@Controller
```

```
public class ConvidadosController {
```

```
}
```

Agora, podemos criar o método que receberá a requisição e retornará o nome da *view*.

Vamos chamar este método de `listar()`, pois ele será responsável por listar os convidados para mostrarmos na *view*, mais a frente.

Esse método pode retornar uma `String`, que é o nome da *view* que iremos criar daqui a pouco, chamada de *ListaConvidados*.

```
public String listar() {  
    return "ListaConvidados";  
}
```

Ok. Mas agora surge uma dúvida: qual URL que podemos digitar no browser para esse método ser chamado?

Aí que entra o papel da anotação `@GetMapping`. Vamos mapear para que a requisição `/convidados` caia nesse método. Para isso, é só fazer como o código abaixo.

```
@GetMapping("/convidados")  
public String listar() {  
    return "ListaConvidados";  
}
```

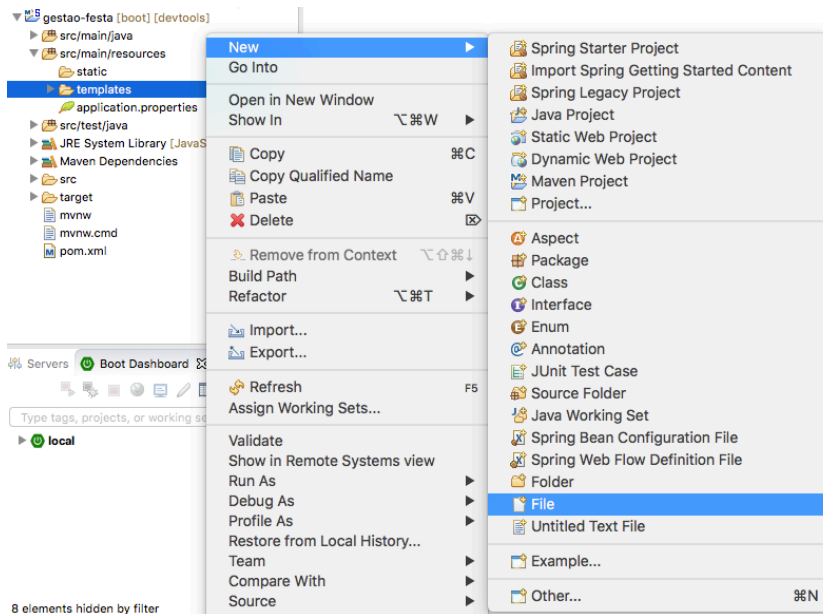
Pronto! Agora o método `listar()` é chamado quando acessarmos no browser a URL <http://localhost:8080/convidados>, e o Spring MVC saberá que a *view* *ListaConvidados* deve ser renderizada para o cliente.

3.4. Criando a página

Na seção anterior fizemos o *controller* retornar o nome da *view* *ListaConvidados* para a requisição.

A configuração default do Spring Boot com Thymeleaf, define que a *view* deve ficar em `src/main/resources/templates`, e o sufixo do arquivo ser `.html`.

Portanto, vamos criar um arquivo simples e transformá-lo em uma página HTML. Lembre-se de salvar em `src/main/resources/templates`.



Importante: o nome da view faz parte do nome do arquivo, informe *ListaConvidados.html* em *File Name*.

Antes de criarmos nossa página, vamos alterar a versão padrão do Thymeleaf que o Spring Boot usa para uma versão mais recente. Fazemos isso dentro da tag *properties*, do arquivo *pom.xml*. Veja:

```
<properties>
  <!-- ... -->
  <thymeleaf.version>3.0.2.RELEASE</thymeleaf.version>
  <thymeleaf-layout-dialect.version>2.0.4</thymeleaf-layout-dialect.version>
</properties>
```

Também vamos adicionar duas propriedades do Thymeleaf dentro do arquivo *src/main/resources/application.properties*:

```
spring.thymeleaf.mode=html
spring.thymeleaf.cache=false
```

A primeira, altera para HTML o modo de templates que o Thymeleaf irá trabalhar e a segunda é para que ele não faça cache das páginas, pelo menos, enquanto estivermos desenvolvendo o projeto.

Para vermos algo funcionando o mais rápido possível, vamos criar uma página simples com o Thymeleaf.

Para deixar nosso sistema mais bonito, vamos usar o [Bootstrap](#). Iremos usar também a biblioteca [WebJars](#), que vai nos permitir gerenciar os recursos (CSS e JS) necessários do Bootstrap, através do nosso arquivo *pom.xml*.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

<head>
  <meta charset="UTF-8"/>
  <meta name="viewport" content="width=device-width" />

  <title>Lista de Convidados</title>

  <link th:href="@{/webjars/bootstrap/css/bootstrap.min.css}"
        rel="stylesheet"/>
  <link th:href="@{/webjars/bootstrap/css/bootstrap-theme.min.css}"
        rel="stylesheet"/>
</head>
<body>

  <h1>AlgaWorks!</h1>

</body>
</html>
```

Vamos destacar alguns pontos deste código para melhor entendimento.

Logo no início temos a tag *html* com dois atributos:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
```

O “xmlns” especifica o namespace XML para nosso documento. Isso significa que devemos sempre abrir e fechar as tags HTML, nunca poderemos ter uma tag assim:

```
<input type="text">
```

O código acima funciona normalmente em um HTML comum, mas não em um XHTML, portanto, lembre-se sempre de abrir e fechar suas tags:

```
<input type="text"></input>
```

Já o atributo “xmlns:th” define que podemos usar as propriedades definidas pelo Thymeleaf, e você vai gostar delas.

Por fim, vamos destacar a importação do [Bootstrap](#), um framework HTML, CSS e JavaScript para desenvolvimento de aplicações responsivas para web, e que nos ajuda a criar páginas bem mais elegantes e com menos esforço:

```
<link th:href="@{/webjars/bootstrap/css/bootstrap.min.css}"
      rel="stylesheet"/>
<link th:href="@{/webjars/bootstrap/css/bootstrap-theme.min.css}"
      rel="stylesheet"/>
```

Temos duas coisas a reparar na importação acima.

Primeiro é que, na importação do Bootstrap, utilizamos a biblioteca [WebJars](#). Isso é para que não tenhamos que fazer o download do Bootstrap para dentro do nosso projeto.

O CSS do Bootstrap que usamos fica dentro do JAR do WebJars, que foi colocado como dependência do projeto através do Maven.

Por isso, foi necessária a dependência *org.webjars.bootstrap*. Somada a ela, tem a dependência *org.webjars.webjars-locator*, que serve para que não tenhamos que incluir a versão do Bootstrap na URL usada para importar o CSS dele.


Veja as duas:

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>webjars-locator</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
```

CORRIGIR

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>webjars-locator</artifactId>
  <version>0.36</version>
</dependency>
```



```
<version>3.3.7</version>
</dependency>
```

O Bootstrap tem uma integração com o jQuery que, para nosso projeto, não será necessária, mas caso queira deixar tudo preparado para utilizar essa integração, então adicione mais essa dependência aqui:

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>jquery</artifactId>
  <version>2.1.1</version>
</dependency>
```

E logo antes de fechar o elemento *body* da página, inclua essas duas importações de JavaScript:

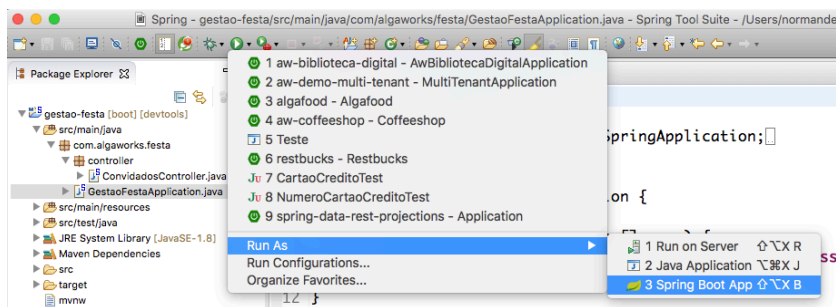
```
<script th:src="@{/webjars/jquery/jquery.min.js}"></script>
<script th:src="@{/webjars/bootstrap/js/bootstrap.min.js}"></script>
```

A segunda coisa que podemos reparar, tanto na importação dos arquivos CSS quanto dos arquivos JavaScript, é que fizemos o uso dos atributos `th:href` e `th:src` do Thymeleaf. Eles foram utilizados para podermos tirar proveito da expressão `@{}`. A vantagem dessa expressão é que ela coloca o contexto da aplicação nos links do projeto. Ainda veremos mais sobre essa expressão.

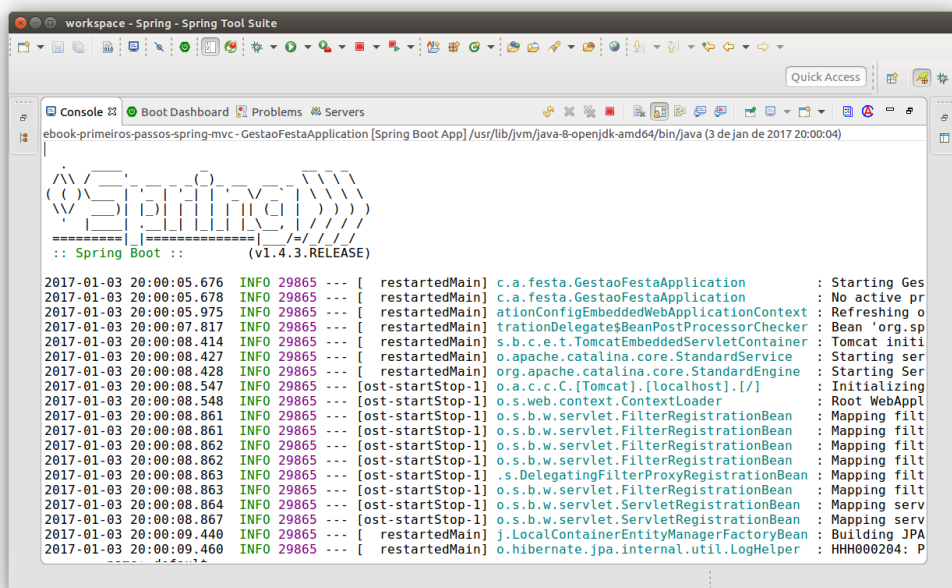
3.5. Rodando o projeto pela primeira vez

Já temos algo executável, nada funcional ainda, mas já podemos ver alguma coisa rodando no browser e ficar felizes por termos nossa aplicação funcionando até agora.

Com a classe `GestaoFestaApplication` aberta, clique na pequena seta ao lado do *Run* e selecione *Run As -> Spring Boot App*, conforme mostra a imagem abaixo.



O console mostrará alguns logs com várias mensagens, e depois de alguns segundos nossa aplicação estará no ar.



Vamos testar?

Abra o browser e digite <http://localhost:8080/convidados>.

Essa URL fará com que o Spring MVC chame o método `listar()` do *controller* `ConvidadosController`, que por sua vez retorna “ListaConvidados”, que é o nome da *view* “ListaConvidados.html”, que enviará para o cliente a página HTML.



3.6. Repositório de convidados

Começamos bem simples, mas em poucos minutos já conseguimos criar uma aplicação com Spring MVC e Thymeleaf e ver algo funcionando no browser. E isso é muito legal, pois não precisamos ficar procurando outras aplicações de exemplo para copiar e colar, fizemos tudo muito simples e rápido.

Agora é hora de sofisticar um pouco nossa aplicação. Vamos criar uma classe que representará cada convidado e um repositório para tratar do armazenamento dos dados.

O primeiro passo é criar a classe que representa a entidade de convidado. Lembre-se que um convidado tem um nome, a quantidade de acompanhantes que ele levará à festa e um identificador.

Crie a classe `Convidado` no pacote `com.algaworks.festa.model`.

```
package com.algaworks.festa.model;
```

```
import java.io.Serializable;
```

```
import javax.persistence.Entity;
```

```

import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Convidado implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue ← @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String nome;

    private Integer quantidadeAcompanhantes;

    // getters e setters omitidos

}

```

Repare que utilizamos três anotações do JPA. Não se assuste, você não vai precisar fazer mais do que isso para deixar a persistência funcionando em nosso projeto.

A anotação `@Entity` torna a nossa classe como uma entidade de banco de dados.

As anotações `@Id` e `@GeneratedValue` são para marcar a propriedade `id` como identificador da entidade (ou seja, a chave primária no banco de dados) e para pedir que o Hibernate (nossa implementação por trás do JPA) gere o identificador para nós, respectivamente.

Vamos criar agora uma interface chamada `Convidados`, que representará um repositório de convidados, ou seja, um lugar onde podemos listar ou adicionar convidados.

O projeto Spring Data JPA, através da interface `JpaRepository`, vai tornar essa tarefa uma das mais fáceis do nosso projeto. Veja o nosso repositório:

```

package com.algaworks.festa.repository;

import org.springframework.data.jpa.repository.JpaRepository;

```



```
import com.algaworks.festa.model.Convidado;

public interface Convidados extends JpaRepository<Convidado, Long> {

}
```

Não precisamos nem mesmo adicionar a anotação `@Repository`, que geralmente fazemos em repositórios quando utilizamos Spring.

Só de implementar a interface `JpaRepository`, já poderemos injetar um objeto do tipo `Convidados` no nosso *controller*, por exemplo.

E, para o leitor mais atento, essas duas classes, `Convidado` e `Convidados`, fazem parte do *Model* no padrão MVC.

3.7. Enviando um objeto do Controller para a View

Agora que já temos o repositório pronto, podemos enviar a lista de convidados do *controller* para a *view*, e essa tarefa é muito simples.

Ao invés de retornar uma `String` com o nome da *view*, podemos retornar um objeto do tipo `ModelAndView`, que nos permite, além de informar o nome da *view*, adicionar objetos para serem usados no HTML.

```
package com.algaworks.festa.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.servlet.ModelAndView;

import com.algaworks.festa.repository.Convidados;

@Controller
public class ConvidadosController {

    @Autowired
    private Convidados convidados;

    @GetMapping("/convidados")
```

```

public ModelAndView listar() {
    ModelAndView modelAndView = new ModelAndView("ListaConvidados");

    modelAndView.addObject("convidados", convidados.findAll());

    return modelAndView;
}
}

```

Repare que, com `@Autowired`, podemos injetar o repositório no *controller*, e isso nos livra da preocupação de como receber esse objeto na classe.

Veja também a anotação `@GetMapping`, que anota o nosso método `listar()`. Ela diz que nosso método irá responder a requisição HTTP do tipo GET para `/convidados`.

Já dentro de `listar()`, o construtor de `ModelAndView` recebe o nome da *view*, e com o método `addObject()`, podemos adicionar objetos para a *view*.

3.8. Listando objetos com o Thymeleaf

Agora que já temos o *controller* recuperando os dados do repositório e adicionando no `ModelAndView`, para ser usado na *view*, vamos editar o arquivo `ListaConvidados.html` e adicionar uma listagem de convidados.

Como já importamos o Bootstrap, vamos usá-lo para deixar a página mais bonita. Para este exemplo, vamos utilizar o [Panel with heading](#), que consiste em um painel com um cabeçalho e um corpo.

```

<body>
  <div class="panel panel-default" style="margin: 10px">
    <div class="panel-heading">
      <h1 class="panel-title">Lista de convidados</h1>
    </div>

    <div class="panel-body">
      <table class="table">
        <thead>

```

```

        <tr>
            <th>Nome</th>
            <th>Acompanhantes</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>João</td>
            <td>3</td>
        </tr>
    </tbody>
</table>
</div>
</div>
</body>

```

Repare que na tag `tbody` existe uma linha `tr`, mas o que queremos é iterar sobre a lista que o *controller* disponibilizou para a *view*, ao invés de deixar fixo como está.

Precisamos de algo que itere e gere várias linhas (`tr`) e, nas colunas (`td`), permita inserir o nome e a quantidade de acompanhantes do convidado.

Agora que o Thymeleaf entra em ação! Vamos usar dois atributos, o `th:each` e o `th:text`. O primeiro para iterar sobre a lista e o segundo para mostrar as propriedades do objeto nas colunas.

```

<tr th:each="convidado : ${convidados}">
    <td th:text="${convidado.nome}"></td>
    <td th:text="${convidado.quantidadeAcompanhantes}"></td>
</tr>

```

No `th:each`, usamos a expressão `${convidados}` para recuperar o objeto adicionado pelo *controller*.

Lembra do nome que usamos para adicionar no `ModelAndView`? Foi “convidados”, né?

Então, esse é o nome que usamos na expressão `${}`. Lembre-se que nessa variável temos uma lista de convidados, portanto, podemos iterar nela.

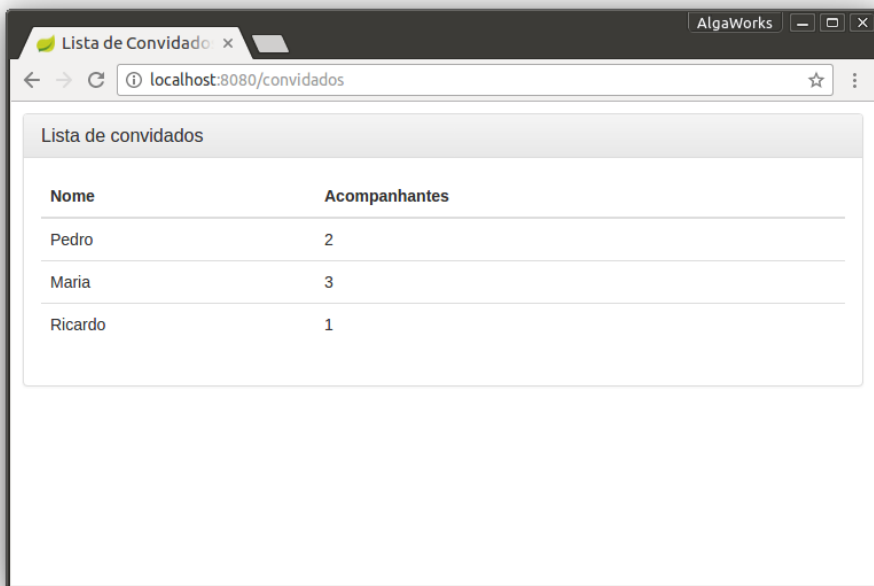
Antes dos dois pontos, criamos uma variável local para podermos usar na iteração. Repare que nas colunas, no atributo `th:text`, usamos `${convidado.nome}` para mostrar o nome do convidado no conteúdo da coluna.

Para vermos a propriedade `th:each` em ação agora, vamos criar o arquivo `src/main/resources/import.sql` e adicionar algumas inserções em SQL para testes dentro dele:

```
insert into convidado (id, nome, quantidade_acompanhantes)
  values (1, 'Pedro', 2);
insert into convidado (id, nome, quantidade_acompanhantes)
  values (2, 'Maria', 3);
insert into convidado (id, nome, quantidade_acompanhantes)
  values (3, 'Ricardo', 1);
```

Só precisamos criar o arquivo `import.sql` e mais nada. Isso porque o Spring Boot vai se encarregar de repassar esse arquivo ao Hibernate (nossa implementação de JPA) para que ele o execute.

Reinicie o servidor e acesse novamente a URL <http://localhost:8080/convidados>. Você deverá ver a lista dos convidados.



3.9. Adicionando um convidado

Já estamos listando, mas e se quisermos adicionar um novo convidado?

Vamos adicionar na mesma *view* um formulário para preenchermos o nome e a quantidade de acompanhantes de um convidado.

O formulário ficará dentro do painel, logo acima da tabela. Primeiro, vamos só adicionar o HTML para criarmos o protótipo, sem salvar no repositório ainda.

```
<form class="form-inline" method="POST" style="margin: 20px 0">
  <div class="form-group">
    <input type="text" class="form-control"
      placeholder="Nome"/>
    <input type="text" class="form-control"
      placeholder="Acompanhantes"/>
    <button type="submit"
      class="btn btn-primary">Adicionar</button>
```

```
</div>
</form>
```

Agora que nosso HTML está pronto, vamos começar as modificações para o Thymeleaf e o Spring conseguirem salvar um novo convidado.

A primeira alteração será no método `listar()` do *controller*. Vamos adicionar um objeto do tipo `Convidado` no `ModelAndView`.

Esse objeto é chamado de *command object*, que é o objeto que modela o formulário, ou seja, é ele que será setado com os valores das tags `input` da página.

Adicione simplesmente a linha abaixo no método `listar()`, da classe `ConvidadosController`.

```
mv.addObject(new Convidado());
```



Para o Thymeleaf usar este objeto no formulário, adicione o atributo `th:object` no `form`.

```
<form class="form-inline" method="POST" th:object="${convidado}"
      style="margin: 20px 0">
```

E nos campos de entrada, vamos usar as propriedades do objeto “convidado” nos *inputs*, usando `th:field`.

```
<input type="text" class="form-control"
        placeholder="Nome"
        th:field="*{nome}"/>
<input type="text" class="form-control"
        placeholder="Acompanhantes"
        th:field="*{quantidadeAcompanhantes}"/>
```

Repare que usamos a expressão `*{}` para selecionar a propriedade do objeto “convidado”.

Nesse momento o formulário está recebendo um novo objeto do tipo `Convidado` e suas propriedades `nome` e `quantidadeAcompanhantes` estão ligadas aos elementos `input` do `form`.

Para finalizar nosso formulário, precisamos apenas dizer para qual endereço ele deve enviar os dados. Vamos fazer isso usando o atributo `th:action`.

```
<form class="form-inline" method="POST" th:object="${convidado}"  
      th:action="@{/convidados}" style="margin: 20px 0">
```

A expressão `@{}`, como comentamos anteriormente, é muito útil quando queremos utilizar links no nosso HTML, pois ela irá resolver o *context path* da aplicação automaticamente.

Nosso formulário está pronto, podemos ver que ele será enviado via POST para o endereço `/convidados`.

No nosso controller não existe um método capaz de receber uma requisição POST em `/convidados`. Vamos criá-lo agora:

```
@PostMapping("/convidados")  
public String salvar(Convidado convidado) {  
  
}
```

Observe que o método `salvar()` recebe como parâmetro um objeto do tipo `Convidado`. O Spring MVC já vai criá-lo e definir os valores enviados pelo formulário neste objeto, facilitando muito nosso trabalho.

Com o objeto pronto, podemos simplesmente adicioná-lo ao repositório.

```
@PostMapping("/convidados")  
public String salvar(Convidado convidado) {  
    this.convidados.save(convidado);  
}
```

Depois de salvar o convidado, seria interessante recarregar a página para que a pesquisa fosse executada novamente, e consequentemente a tabela com a lista de convidados atualizada.

É muito simples fazer isso com o Spring MVC. Ao invés de retornarmos o nome da view que queremos renderizar, podemos retornar uma URL para redirecionar a requisição, usando `"redirect:"` na String.

```

@PostMapping("/convidados")
public String salvar(Convidado convidado) {
    this.convidados.save(convidado);
    return "redirect:/convidados";
}

```

No método acima, a string `redirect:/convidados` faz com que o browser faça uma nova requisição GET para `/convidados`, fazendo com que a tabela seja atualizada com a nova pesquisa.

Talvez você esteja pensando que ficou repetido o mapeamento em `@GetMapping` e `@PostMapping` nos métodos `listar` e `salvar`, e esteja perguntando: tem como melhorar?

A resposta é sim, podemos adicionar o `@RequestMapping` na classe do controller.

```

@Controller
@RequestMapping("/convidados")
public class ConvidadosController {

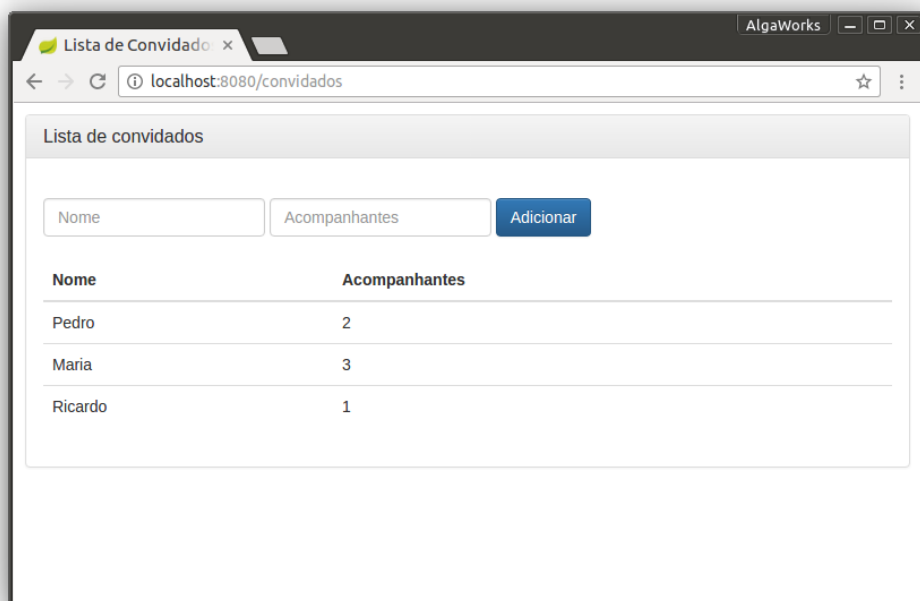
    @GetMapping
    public ModelAndView listar() {
        // ...
    }

    @PostMapping
    public String salvar(Convidado convidado) {
        // ...
    }
}

```

Agora as anotações `@GetMapping` e `@PostMapping` irão começar sempre com `/convidados`.

A aplicação final deve se parecer com a imagem abaixo.



3.10. Escolhendo o banco de dados

Como estamos fazendo o uso do JPA (através do Spring Data JPA), configurar e até mesmo trocar o banco de dados se torna uma tarefa bem simples.

Mais no início, quando criamos o nosso projeto, você deve ter visto que adicionamos a dependência do banco de dados [H2](#). Isso facilitou nosso trabalho até aqui.

Ele é um banco de dados em memória e foi escolhido para simplificar o desenvolvimento do nosso projeto, por dois motivos.

Primeiro porque ele não precisa ser instalado, e segundo porque não precisamos de configuração alguma para utilizá-lo com Spring Boot. Muito bom, não é mesmo?

Agora, a gente sabe que muitos vão querer utilizar um banco de dados diferente, e vamos fazer isso agora. Iremos configurar o nosso projeto com o [MySQL](#).

Além dessas pequenas configurações que serão feitas a partir de agora ajudarem aqueles que querem o próprio MySQL, vão servir como exemplo para quem precisar da configuração para outros SGBDs também.

A primeira coisa é adicionar a dependência do driver JDBC no *pom.xml*:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

Depois, você configura a URL, o usuário e a senha do seu banco no arquivo *src/main/resources/application.properties*:

```
spring.datasource.url=jdbc:mysql://localhost/festa
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create-drop
```

O nome do banco de dados escolhido para o nosso projeto foi “festa”. O usuário ficou como “root” e sem senha.

Por último, configuramos a propriedade “ddl-auto”, para recriar o banco de dados todas as vezes que o projeto se iniciar. Isso é apenas por uma questão didática. Você não pode deixar uma propriedade dessas em produção!

Pronto! Só com isso já podemos rodar o projeto com o MySQL.

3.11. Deixando a aplicação segura

Temos uma aplicação bem completa já funcionando. Falta somente deixá-la segura. Faremos isso com o Spring Security.

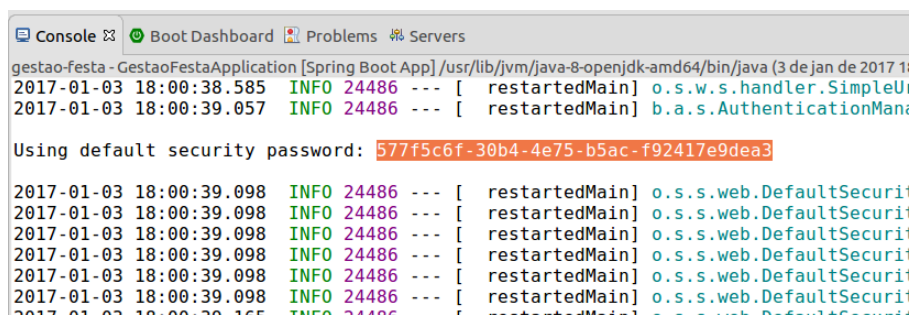
Como foi dito no início, ele simplifica bem o nosso trabalho. E, se estiver fazendo parceria com o Spring Boot (como é o nosso caso), então temos mais facilidades ainda.

Só de adicionarmos o Spring Security no *pom.xml*, o Spring Boot já entra em ação. Inclusive, poderíamos tê-lo adicionado logo na construção do projeto. Isso não foi feito para não termos que ficar logando a todo momento, quando ainda estávamos bem no início do desenvolvimento.

Faremos isso agora. Para isso, precisamos incluir somente dessa dependência em nosso *pom.xml*:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Só de adicioná-la, já é criado o usuário *user* com uma senha que temos que copiar do console toda vez que subimos a aplicação.



```
gestao-festa - GestaoFestaApplication [Spring Boot App] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (3 de jan de 2017 11:18:00)
2017-01-03 18:00:38.585 INFO 24486 --- [ restartedMain] o.s.w.s.handler.SimpleUserDetailsService
2017-01-03 18:00:39.057 INFO 24486 --- [ restartedMain] b.a.s.AuthenticationManager

Using default security password: 577f5c6f-30b4-4e75-b5ac-f92417e9dea3

2017-01-03 18:00:39.098 INFO 24486 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain
2017-01-03 18:00:39.098 INFO 24486 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain
2017-01-03 18:00:39.098 INFO 24486 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain
2017-01-03 18:00:39.098 INFO 24486 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain
2017-01-03 18:00:39.098 INFO 24486 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain
2017-01-03 18:00:39.098 INFO 24486 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain
2017-01-03 18:00:39.155 INFO 24486 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain
```

Já é impressionante termos segurança aplicada com tão pouco. Mas vamos dar mais um passo e adicionar nossos próprios usuários.

Para isso, basta criar a seguinte classe:

```
package com.algaworks.festa.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

@Configuration
public class InMemorySecurityConfig {

    @Autowired
```

```

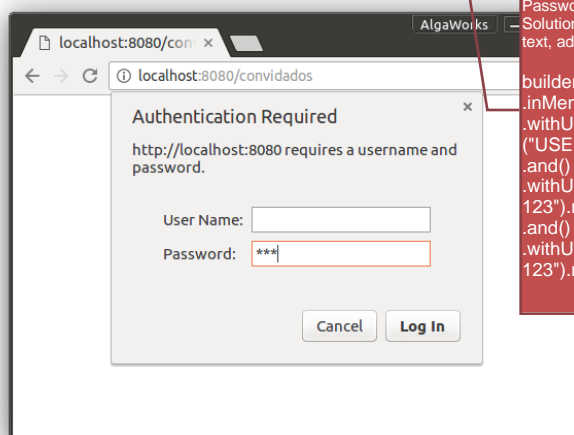
public void configureGlobal(AuthenticationManagerBuilder builder)
    throws Exception {
    builder
        .inMemoryAuthentication()
        .withUser("joao").password("123").roles("USER")
        .and()
        .withUser("alexandre").password("123").roles("USER")
        .and()
        .withUser("thiago").password("123").roles("USER");
    }
}

```

Observe que a anotação `@Configuration` torna a nossa classe `InMemorySecurityConfig` uma classe que poderia abrigar quaisquer outras configurações do Spring. Para melhor organizar, iremos deixar aqui somente o que for relacionado a autenticação de usuários.

O método `configureGlobal`, anotado com `@Autowired`, poderia ter qualquer outro nome, mas esse é uma convenção adotada para quando queremos configurar nosso `AuthenticationManagerBuilder`.

Dentro do método temos a configuração de três usuários que podemos agora utilizar para nos autenticar no sistema. Reinicie a aplicação e tente acessá-la novamente.



Prior to Spring Security 5.0 the default PasswordEncoder was NoOpPasswordEncoder which required plain text passwords. In Spring Security 5, the default is DelegatingPasswordEncoder, which required Password Storage Format.

Solution 1 – Add password storage format, for plain text, add {noop}

```

builder
    .inMemoryAuthentication()
    .withUser("joao").password("{noop}123").roles("USER")
    .and()
    .withUser("alexandre").password("{noop}123").roles("USER")
    .and()
    .withUser("thiago").password("{noop}123").roles("USER");

```

Capítulo 4

Publicando a aplicação

4.1. Introdução

Esse capítulo será como a cereja do bolo. Veremos como fazer a publicação do nosso projeto na nuvem, simulando que estamos colocando ele em produção.

Iremos utilizar a nuvem do [Heroku](#), e teremos a vantagem de ter toda a plataforma configurada pelo mesmo. Só precisamos enviar nossa aplicação.

Mas para isso, precisamos realizar algumas pequenas adaptações para compatibilizar o nosso projeto.

Basicamente, teremos que preparar nosso projeto para funcionar com o banco de dados [Postgres](#). Depois iremos precisar de duas instalações sobre as quais rodaremos os comandos para publicação do projeto.

Talvez você se pergunte: “Por que utilizar o Postgres? No Heroku não tem MySQL?”.

O Heroku tem sim o MySQL, mas para colocar ele em nossa aplicação, será preciso fazer a confirmação da nossa conta.

Essa confirmação é feita com a inclusão do cartão de crédito. Como muitos não tem ou não querem cadastrar seu cartão, então vamos remover essa barreira colocando o Postgres do Heroku, que não exige uma conta confirmada.

Quem quiser confirmar a conta para utilizar o MySQL, pode ficar tranquilo que não será cobrado imediatamente por isso. A cobrança virá somente se for feito um upgrade explícito no *dashboard* do Heroku ou através de comandos de alteração dos tipos de hospedagem.

Para os que preferirem o MySQL, iremos também, nesse capítulo, ver sobre as adaptações que serão necessárias.

4.2. Usando o Postgres

Da mesma forma que foi com o MySQL, será simples utilizar o [Postgres](#) em nossa aplicação.

Vamos precisar incluir a dependência dele no *pom.xml*:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

Só que não vamos mexer no arquivo *application.properties* para fazer configurações de URL, usuário e senha. Como só vamos utilizar o Postgres no Heroku, então criaremos um arquivo especial para ele.

O arquivo será o *src/main/resources/heroku-db.properties*. Ele irá sobrescrever algumas das propriedades que estão no *application.properties*. Veja:

```
spring.datasource.url=${JDBC_DATABASE_URL}
spring.datasource.username=
spring.datasource.password=
```

A propriedade *url* ganha o valor referente à variável de ambiente *JDBC_DATABASE_URL*. Ela é configurada pelo Heroku e, portanto, não precisamos nos preocupar.

As propriedades *username* e *password* devem ficar vazias, sobrescrevendo o arquivo *application.properties*. Esses dois valores já estão junto com a *url*.

Uma última coisa aqui, é que será necessário o arquivo `heroku-db.properties`, mesmo para quem for utilizar o MySQL no Heroku. Obviamente, para essas pessoas, não será preciso a dependência do Postgres.

4.3. Alteração de estratégia para chave primária

Vamos precisar de uma pequena alteração na entidade `Convidado`. Vou explicar.

Nossa entidade está utilizando a estratégia padrão para geração do código da nossa *primary key* do banco de dados. Veja a anotação `@GeneratedValue`:

```
@Id
@GeneratedValue
private Long id;
```

Com a configuração acima, é delegada para o Hibernate (nossa implementação de JPA) a missão de gerar o identificador. Soma-se a isso a informação de que, para o Postgres, a estratégia que o Hibernate usa é a criação de uma *sequence*, chamada `hibernate_sequence`.

Sabemos que uma *sequence* começa com o valor 1. E é aqui que vem o problema do nosso projeto, pois nós já temos o código igual a 1 no arquivo `import.sql`.

Esse vai ser o motivo de alterarmos a estratégia da geração do nosso identificador para a seguinte:

```
@Id
@GeneratedValue(generator = "increment")
@GenericGenerator(name = "increment", strategy = "increment")
private Long id;
```

Essa nova estratégia vai, simplesmente, fazer uma consulta na base de dados do tipo `max(id) + 1`, para buscar o próximo código do convidado. Com isso o problema anterior é eliminado.

Essa foi a forma escolhida para resolver essa questão, mas existem algumas outras.

Para quem for utilizar o MySQL no Heroku, esse tópico é opcional.

4.4. Criando o arquivo de inicialização

Para subir o projeto no Heroku, vamos precisar criar um arquivo chamado `Procfile`, que ficará na raiz do projeto.

Nesse arquivo, iremos incluir o comando que o Heroku vai utilizar para iniciar nossa aplicação.

```
web: java $JAVA_OPTS -Dserver.port=$PORT -jar target/*.jar
      --spring.config.location=classpath:heroku-db.properties
```

O mais importante do comando acima é o arquivo `heroku-db.properties`, que está sendo passado como parâmetro. Não podemos esquecer dele, pois ele carrega a nossa configuração para o banco de dados.

4.5. Instalando o Git

Para subir o projeto para o Heroku, vamos precisar do [Git](https://git-scm.com/downloads). Você pode baixá-lo no site <https://git-scm.com/downloads> e a instalação é bem simples.

O Git é o gerenciador de código-fonte mais utilizado no mundo. É através dele que o Heroku recebe nosso código para depois compilar e publicar.

Nós não precisaremos de muitos comandos do Git, mas as pessoas que desejarem se aprofundar, podem [começar pela documentação](#) do mesmo, que é bem ampla.

Depois da instalação, vamos precisar iniciar um repositório e fazer o *commit* de nossos arquivos.

Obviamente, aqueles que clonarem com o Git o código-fonte que disponibilizamos já terão esse repositório iniciado. Não será necessário iniciar e nem mesmo fazer commit - a menos, é claro, que tiverem feito alguma alteração no mesmo.

De qualquer forma, os comandos que vou apresentar agora deverão ser dados pelo terminal (ou Prompt de Comando do Windows), mais especificamente, de dentro da pasta do projeto.

```
$ cd /pasta/do/projeto/gestao-festa
```

Para iniciar nosso repositório local, vamos utilizar o comando:

```
$ git init
```

Depois, iremos adicionar os arquivos que desejamos que sejam incluídos no *commit*. Fazemos isso assim:

```
$ git add .
```

O comando acima irá incluir todos os arquivos, exceto aqueles que estiverem listados em um arquivo especial, chamado `.gitignore`, que você pode criar e incluir na raiz do projeto. O nosso ficará assim:

```
target/  
.settings/  
.classpath  
.project
```

O próximo passo é o *commit*:

```
$ git commit -m "Primeiro commit."
```

Ainda será dado mais um comando com o Git, mas antes precisamos configurar nossa aplicação no Heroku.

4.6. Configurando a aplicação no Heroku

Primeiramente é preciso ir até o [site do Heroku](https://heroku.com/), clicar em algum botão de *Sign up* ou entrar diretamente em <https://signup.heroku.com/>, para fazer o seu cadastro. Ele é bem simples.

Feito o cadastro, já podemos instalar o Heroku CLI. Os detalhes dessa instalação para cada sistema operacional, você encontra em <https://devcenter.heroku.com/articles/heroku-cli>.

Agora sim, depois da instalação do Heroku CLI, podemos rodar os comandos que vão configurar nossa aplicação. Abra novamente o terminal e vá até a pasta do projeto.

```
$ cd /pasta/do/projeto/gestao-festa
```

Obrigatoriamente, para configurar nossa aplicação, precisamos estar logados no Heroku pelo terminal. Fazemos isso com o comando `login`.

```
$ heroku login
```

Será pedido um usuário e senha para você. São os mesmos que você informou no cadastro dentro do site.

Agora podemos criar nossa aplicação lá dentro do Heroku.

```
$ heroku create aw-gestao-festa
```

Usamos o comando `create` para isso. O parâmetro passado é o nome que nossa aplicação irá ter no Heroku. O nome não é obrigatório e se não for passado, um aleatório será gerado para você.

O nome de uma aplicação no Heroku deve ser único em todo o mundo! Caso você utilize um que já existe, receberá uma mensagem de erro. A dica é você ter um prefixo seu, assim como foi feito acima. Veja que foi utilizado o prefixo “aw”, de AlgaWorks, juntamente com o nome do projeto, que é “gestao-festa”.

Criamos a aplicação, mas o banco de dados ainda precisa ser adicionado. Como você já sabe, nós vamos adicionar o Postgres:

```
$ heroku addons:create heroku-postgresql:hobby-dev
```

Para quem confirmou a conta e quer [utilizar o MySQL](#), basta adicionar o *addon*:

```
$ heroku addons:create cleardb:ignite
```

Até aqui temos a aplicação e o banco configurados, basta enviar ela agora com o comando push do Git. Já vamos fazer isso.

4.7. Enviando a aplicação

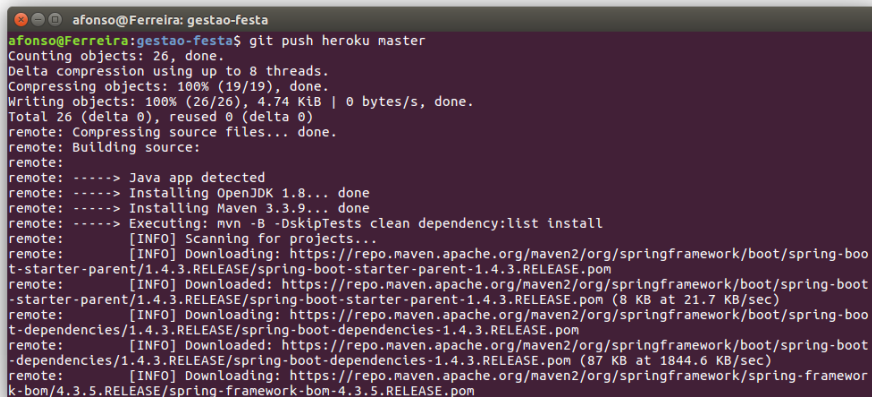
Se você seguiu todos os passos até aqui, então está pronto para enviar o código da sua aplicação para o Heroku.

O envio propriamente dito é feito com a execução de um comando do Git. O comando é o seguinte:

```
$ git push heroku master
```

No momento em que criamos a nossa aplicação com `heroku create`, foi adicionado um repositório remoto, de nome *heroku*, nas configurações do nosso repositório local do Git.

Logo depois desse comando, o que você deve ver no seu terminal é algo parecido com a imagem abaixo:



```
afonso@Ferreira:gestao-festa$ git push heroku master
Counting objects: 26, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (19/19), done.
Writing objects: 100% (26/26), 4.74 KiB | 0 bytes/s, done.
Total 26 (delta 0), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: ----> Java app detected
remote: ----> Installing OpenJDK 1.8... done
remote: ----> Installing Maven 3.3.9... done
remote: ----> Executing: mvn -B -DskipTests clean dependency:list install
remote: [INFO] Scanning for projects...
remote: [INFO] Downloading: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boo
t-starter-parent/1.4.3.RELEASE/spring-boot-starter-parent-1.4.3.RELEASE.pom
remote: [INFO] Downloaded: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boo
t-starter-parent/1.4.3.RELEASE/spring-boot-starter-parent-1.4.3.RELEASE.pom (8 KB at 21.7 KB/sec)
remote: [INFO] Downloading: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boo
t-dependencies/1.4.3.RELEASE/spring-boot-dependencies-1.4.3.RELEASE.pom
remote: [INFO] Downloaded: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boo
t-dependencies/1.4.3.RELEASE/spring-boot-dependencies-1.4.3.RELEASE.pom (87 KB at 1844.6 KB/sec)
remote: [INFO] Downloading: https://repo.maven.apache.org/maven2/org/springframework/spring-framewor
k-bom/4.3.5.RELEASE/spring-framework-bom-4.3.5.RELEASE.pom
```

Espere o processo acima encerrar e acesse o endereço <https://nome-da-aplicacao.herokuapp.com/convidados>. Para facilitar, você pode executar o

comando open do Heroku CLI, que ele já abre o browser no endereço correto para você.

```
$ heroku open
```

Lembrando que, depois que o browser for aberto, você ainda deve informar o caminho /convidados na URL.

É interessante conhecer também outros dois comandos do Heroku. O primeiro é logs --tail e o segundo é o ps.

O comando logs --tail, bem intuitivo, serve para exibir os logs, a medida que forem ocorrendo dentro da sua aplicação.

```
$ heroku logs --tail
```

Por último, o comando ps serve para que você veja alguns detalhes sobre a sua hospedagem. Ao executá-lo:

```
$ heroku ps --app aw-gestao-festa
```

Você tem acesso a alguns detalhes como na imagem abaixo:

A terminal window with a dark purple background. The title bar shows 'afonso@Ferreira: gestao-festa'. The prompt is 'afonso@Ferreira:gestao-festa\$'. The command 'heroku ps --app aw-gestao-festa' has been executed. The output shows 'Free dyno hours quota remaining this month: 548h 49m (99%)' and a link to Heroku's dyno sleeping article. Below that, it lists the dyno type 'web (Free)' with its command 'java \$JAVA_OPTS -Dserver.port=\$PORT -jar target/*.jar --spring.config.location=classpath:heroku-db.properties (1)'. The status is 'up' and it shows the last update time '2017/01/04 17:57:20 -0200 (~ 15m ago)'. The prompt is now 'afonso@Ferreira:gestao-festa\$' with a cursor.

```
afonso@Ferreira: gestao-festa
afonso@Ferreira:gestao-festa$ heroku ps --app aw-gestao-festa
Free dyno hours quota remaining this month: 548h 49m (99%)
For more information on dyno sleeping and how to upgrade, see:
https://devcenter.heroku.com/articles/dyno-sleeping

=== web (Free): java $JAVA_OPTS -Dserver.port=$PORT -jar target/*.jar
--spring.config.location=classpath:heroku-db.properties (1)
web.1: up 2017/01/04 17:57:20 -0200 (~ 15m ago)
afonso@Ferreira:gestao-festa$
```

As informações mais importantes mostradas acima são: o tipo de hospedagem que estamos utilizando (*free*, no caso) e a quantidade de horas que temos disponíveis para utilizar no mês corrente, com o tipo gratuito de hospedagem.

Capítulo 5

Conclusão

Que legal ter chegado ao final da leitura. Estamos felizes por você ter cumprido mais essa etapa na sua carreira.

Esperamos que tenha colocado em prática tudo que aprendeu. Não se contente em apenas ler esse livro. Pratique, programe, implemente cada detalhe, caso contrário, em algumas semanas já terá esquecido grande parte do conteúdo.

Afinal de contas, nada melhor do que colocar a mão na massa, não é mesmo?! :)

Se você gostou desse livro, por favor, ajude a manter esse trabalho. Recomende para seus amigos de trabalho, faculdade e/ou compartilhe no Facebook e Twitter.

5.1. Próximos passos

Embora esse livro tenha te ajudado a criar uma aplicação do início ao fim com Spring Boot, Spring MVC, Spring Data JPA, Spring Security e Thymeleaf, o que você aprendeu nele é só a ponta do iceberg!

É claro que você não perdeu tempo com o que acabou de estudar, o que nós queremos dizer é que há muito mais coisas para aprofundar.

Caso você tenha interesse em continuar seu aprendizado, recomendo que veja agora um curso online que temos sobre Spring, nesse link: <http://alga.works/livro-spring-boot-cta/>.

CURSO ONLINE



**DESENVOLVIMENTO WEB
COM SPRING**

COMPRAQUI



PRODUTIVIDADE NO DESENVOLVIMENTO
DE APLICAÇÕES WEB COM
SPRING BOOT