NORMANDES JR

PRIMEIROS PASSOS COM SPRING MVC

algaworks

Primeiros Passos com Spring MVC
por Normandes Junior

1ª Edição, 10/02/2016

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!

WORKSHOP ONLINE

SPRING MVC

COMPRE AQUI ↓

Sobre o autor

Normandes José Moreira Junior

Sócio e instrutor da AlgaWorks, formado em Engenharia Elétrica pela Universidade Federal de Uberlândia e detentor das certificações LPIC-1, SCJP e SCWCD. Palestrante internacional, autor e co-autor de livros e instrutor de cursos de Java, JPA, TDD, Design Patterns, Spring, etc.

LinkedIn: https://www.linkedin.com/in/normandesjr

Twitter: @normandesjr

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 Java e OO, pode ser interessante estudar por nosso curso online.

Como obter ajuda?

Durante os estudos, é muito comum surgir várias dúvidas. Eu gostaria muito de te ajudar pessoalmente nesses problemas, mas infelizmente não consigo fazer isso com todos os leitores do livreto, afinal, ocupo 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, minha 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 livreto?

Se você encontrar algum erro no conteúdo desse livreto ou se tiver alguma sugestão para melhorar a próxima edição, vou ficar muito feliz se você puder me dizer.

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

Onde encontrar o código-fonte do projeto?

Neste livreto, 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 livreto.

Caso você tenha perdido esse link, acesse http://alga.works/livreto-primeiros-passos-spring-mvc/ 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 livreto para seus amigos que também se interessam por programação Java. Compartilhe no Facebook e Twitter!

Sumário

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 esse é o objetivo desse livreto, mostrar um caminho mais fácil e prazeroso para criar um projeto Java web.

Nós vamos criar uma aplicação simples com Spring MVC, Spring Boot 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.

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 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.

Mas mesmo com configuração via código Java, que trouxe benefícios, como evitar erros de digitação, pois a configuração agora precisa ser compilada, ainda assim precisávamos escrever muito código explicitamente.

Com toda essa configuração excessiva, o desenvolvedor perde o foco do mais importante: o desenvolvimento da aplicação, das regras de negócio e da entrega do software.

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 apenas dependências que agrupam outras, 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*.

2.4. O Spring MVC

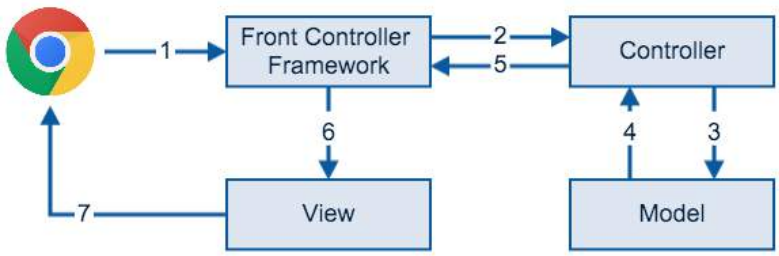
Dentre os projetos Spring, o Spring MVC é o framework que te ajuda no desenvolvimento de aplicações web robustas, flexíveis e com uma clara separação de responsabilidades nos papéis do tratamento da requisição.

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", clicamos no "Enter" e pronto, se nada der errado, uma página HTML será renderizada para nós.

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 de volta a ele.

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, 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.

2.5. 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`.

2.6. O Maven

O Maven é uma ferramenta muito importante no dia a dia do desenvolvedor Java, 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?

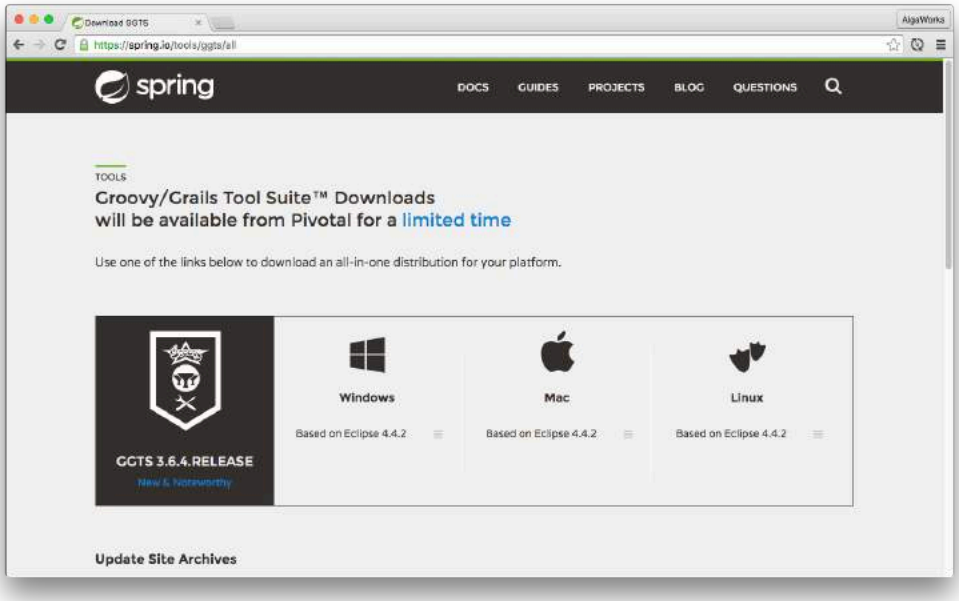
Para saber mais sobre como começar com o Apache Maven, recomendo você assistir esta videoaula gratuita no blog da AlgaWorks.

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

2.7. O Spring Tool Suite - STS

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

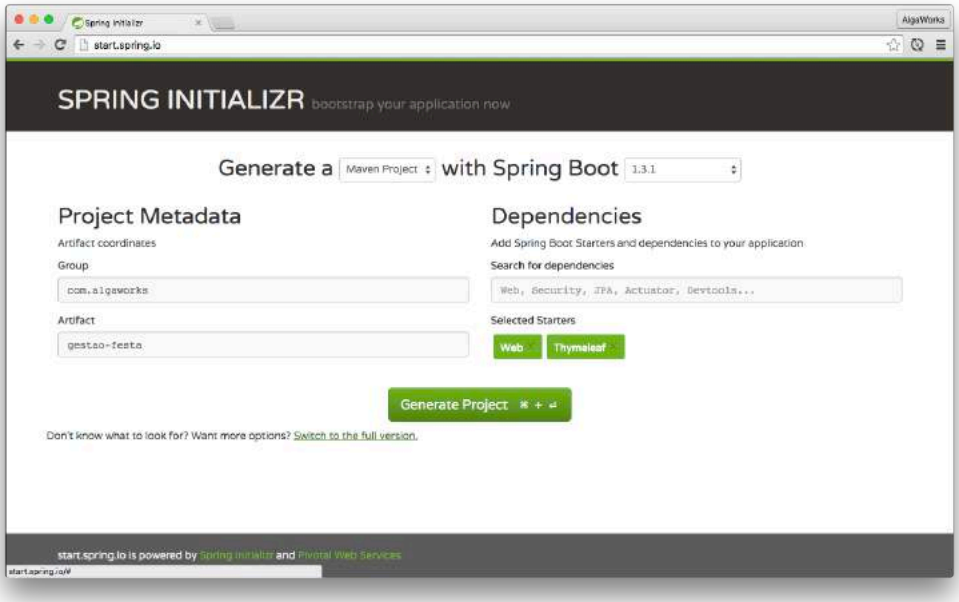
Existem versões para Windows, Mac e Linux em https://spring.io/tools/sts/all.



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 MVC.

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, 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

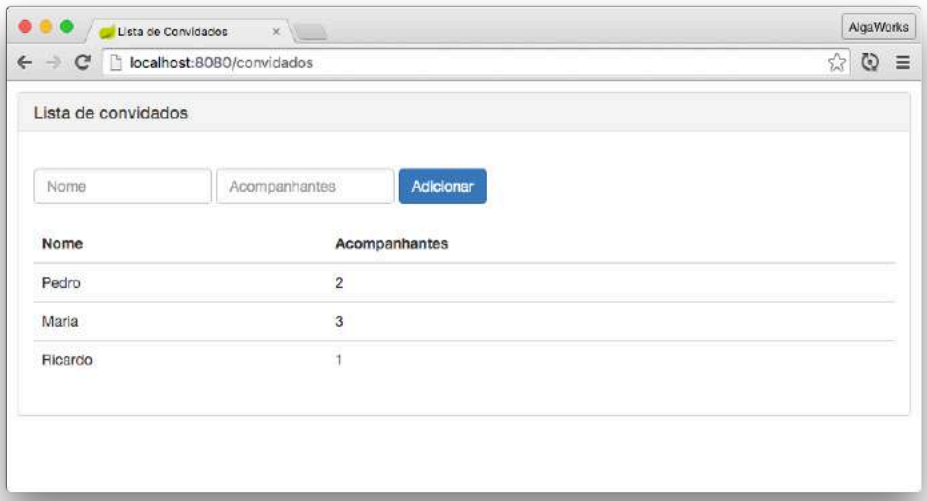
Bora colocar a mão na massa?

Vamos criar uma aplicação simples, do zero e passo a passo para você ver como o Spring Boot, o Spring MVC 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 vêm com ele.

Na aplicação, teremos uma única tela com dois campos de texto de entrada, 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 figura abaixo.

Vamos agora começar a configurar nossa aplicação.

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

- Para *Java Version*, selecione a versão do Java que está configurada para seu ambiente (recomendo que você use a versão 1.8)

- Em *Language*, claro, será Java

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

- Em *Group*, informe *com.algaworks*

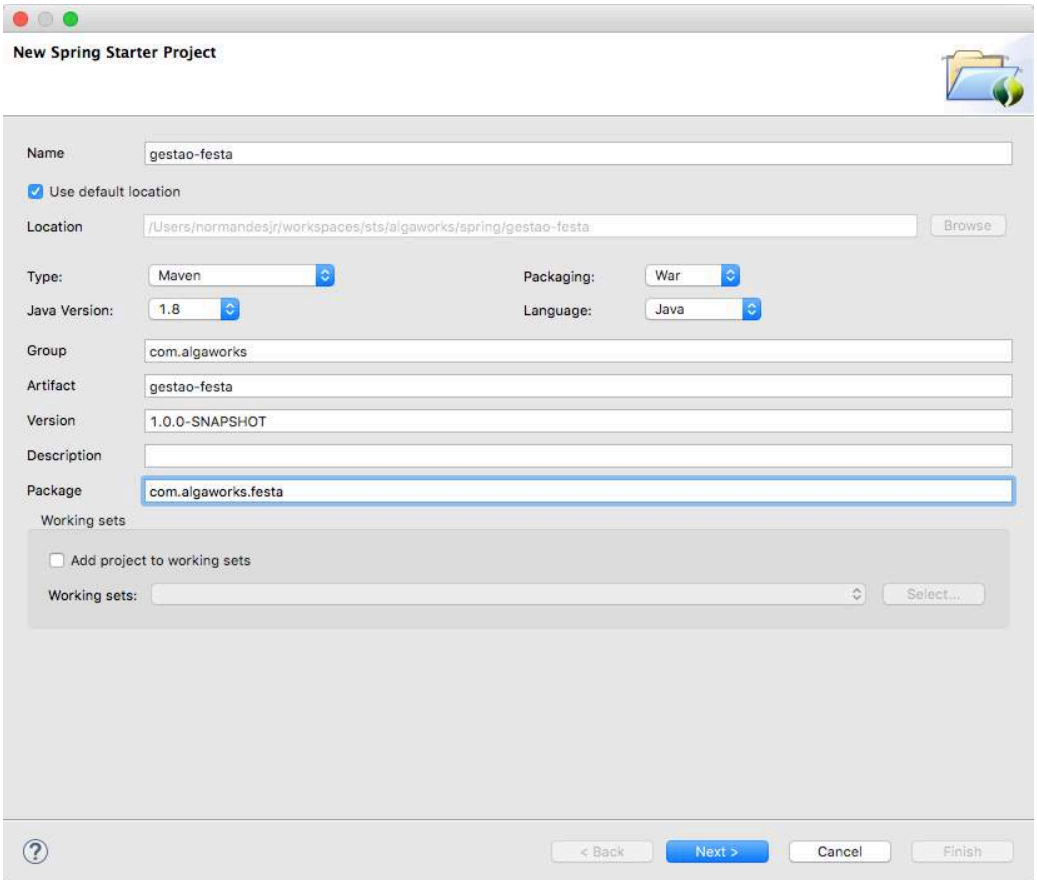
- Em *Artifact*, informe *gestao-festa*

- 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 ainda.

- 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*

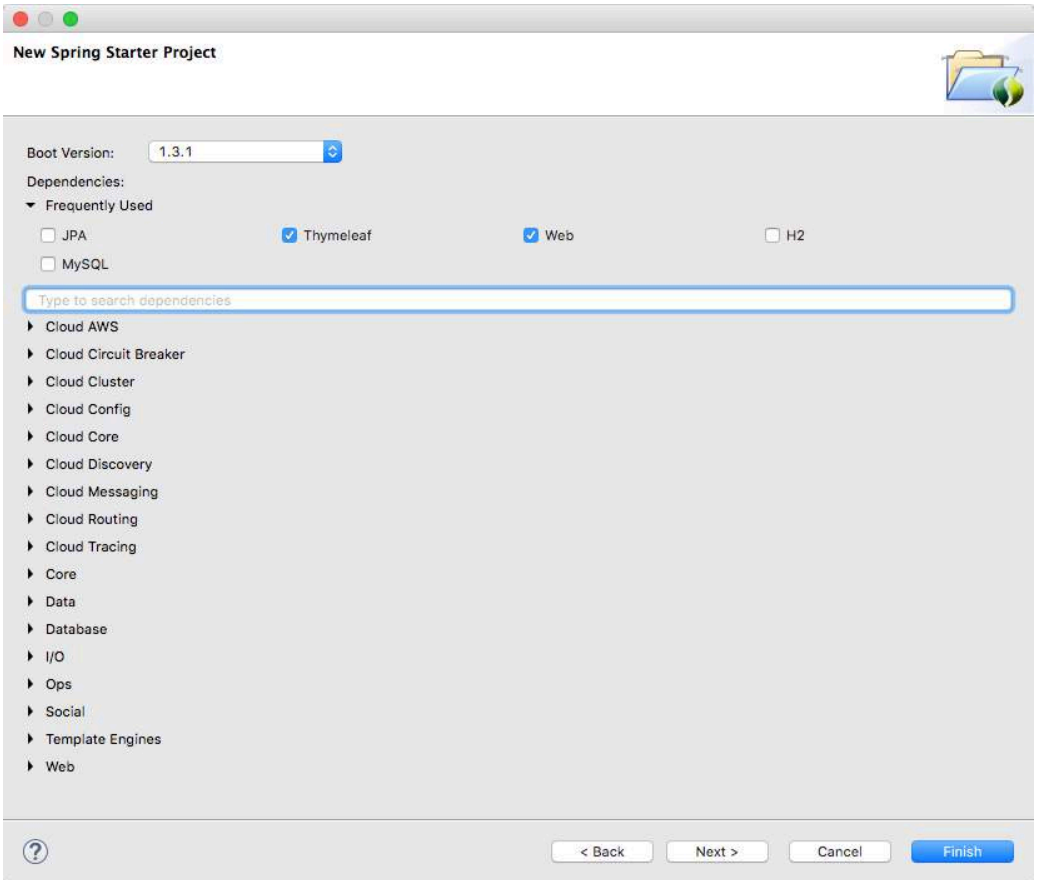
Veja na imagem abaixo a tela preenchida com as informações. Após tudo conferido, clique em *Next*.



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

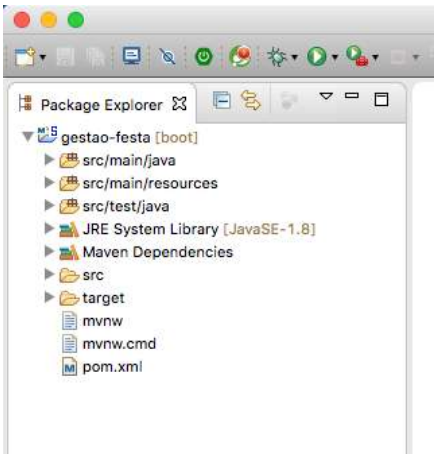
Essa tela é muito útil para iniciarmos o desenvolvimento da nossa aplicação, pois ela é a base para o Maven gerar o arquivo pom.xml, ou seja, ao invés de você ter que lembrar o nome completo das dependências do Spring MVC, pode apenas selecionar "Web".

Nossa aplicação só precisa das opções "Thymeleaf" e "Web" 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 mostrado na imagem abaixo.



Em `src/main/java` você encontra o pacote `com.algaworks.festa` com duas classes, `GestaoFestaApplication` e `ServletInitializer`.

Vamos analisar o código de `GestaoFestaApplication`:

```
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);
    }
}
```

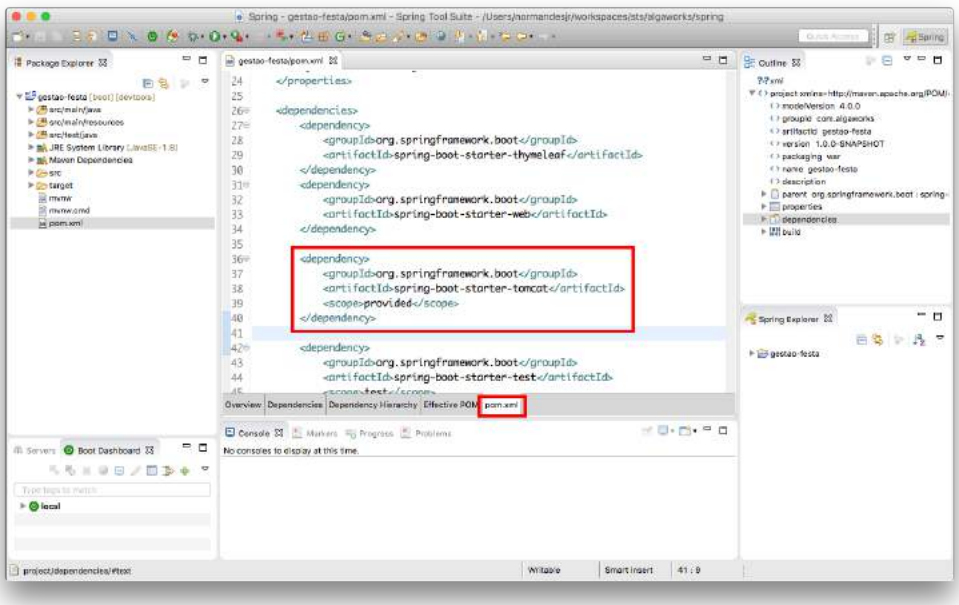
O método `main` inicia a nossa aplicação!

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

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

Acontece que o Spring Boot usa um Tomcat embarcado para facilitar o desenvolvimento, então para iniciar nossa aplicação, basta executarmos o método `main` da classe `GestaoFestaApplication`.

Dê uma olhada no arquivo `pom.xml` e confira que temos uma dependências para o Tomcat.



Voltando na classe `GestaoFestaApplication` e analisando o código com mais detalhes, vemos a anotação:

```
@SpringBootApplication
```

Ela diz que a classe faz parte da configuração do Spring. Poderíamos adicionar configurações customizadas, 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.

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).

Já a classe `ServletInitializer` será usada se nossa aplicação for executada em um servidor externo, como um Apache Tomcat em produção, por exemplo.

```
package com.algaworks.festa;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.context.web.SpringBootServletInitializer;

public class ServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(
                            SpringApplicationBuilder application) {
        return application.sources(GestaoFestaApplication.class);
    }

}
```

Essa classe estende de `SpringBootServletInitializer`, que em um container web que suporta a partir da versão 3.0 da especificação de Servlet, faz com que a aplicação seja inciada sem a necessidade do web.xml.

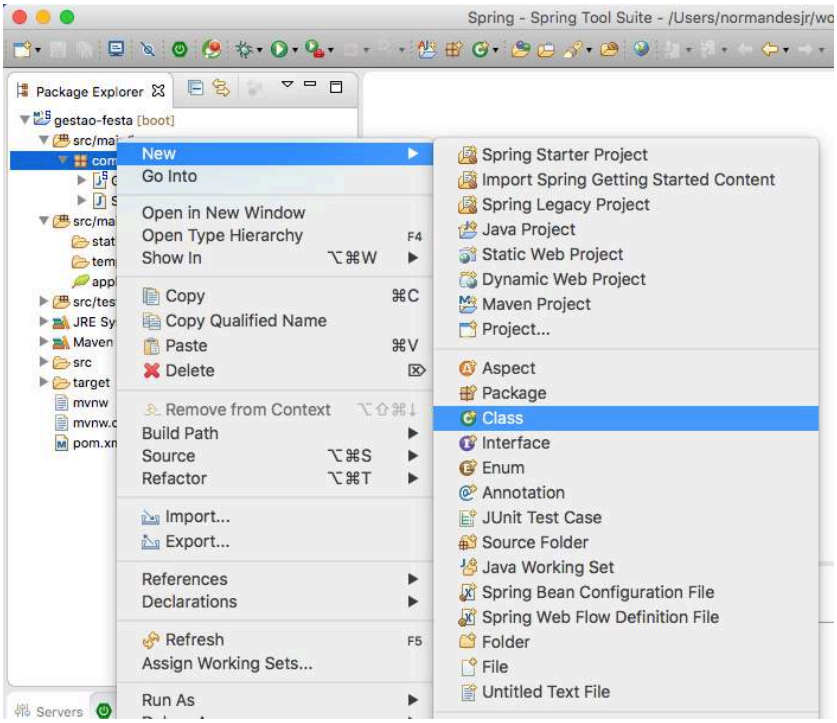
Repare que o código do método `configure()` usa a classe `GestaoFestaApplication`. Podemos então deixar as configurações sempre em `GestaoFestaApplication`, que em qualquer ambiente que nossa aplicação for instalada, sempre irá funcionar.

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 renderizados na página.

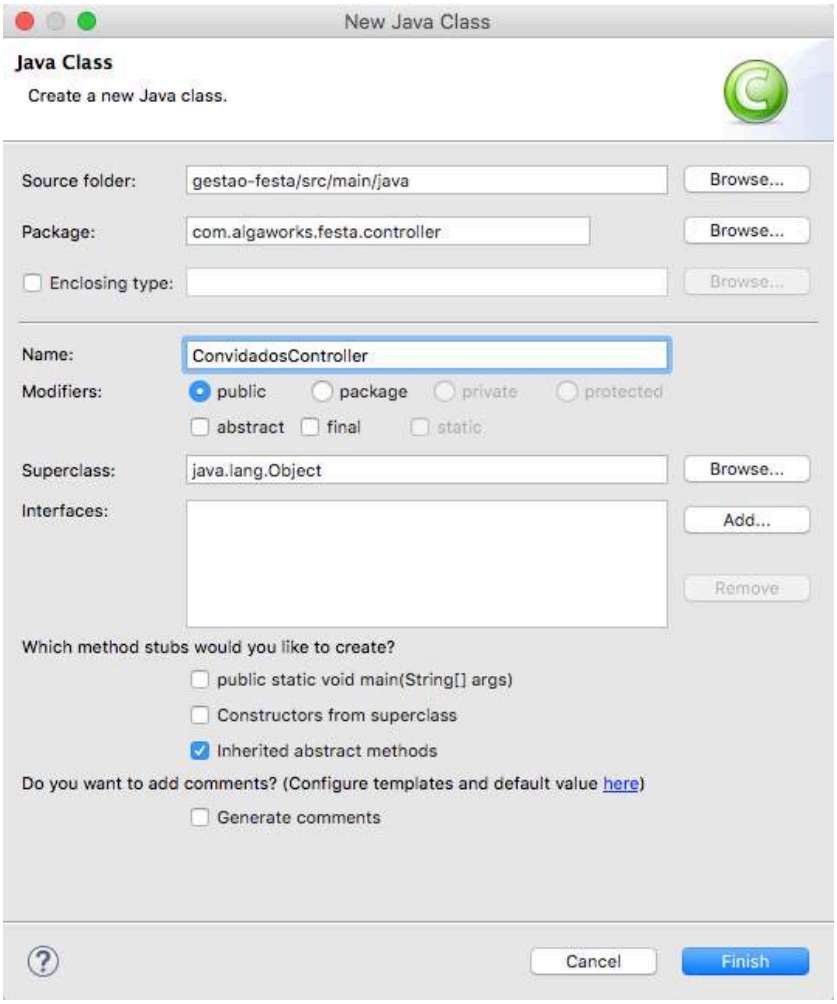
Vamos criar uma nova classe Java e começar a programá-la.



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".



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 `@RequestMapping`. Vamos mapear para que a requisição **/convidados** caia nesse método. Para isso, é só fazer como o código abaixo.

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

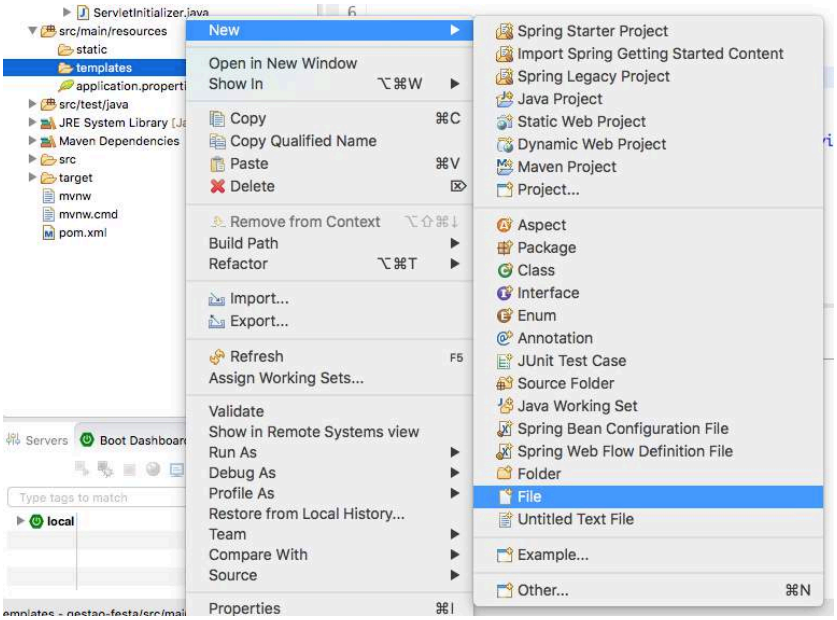
Pronto! Agora o método `listar()` é chamado quando acessarmos no browser a URL http://localhost:8080/convidados, e o Spring MVC saberá que a view *ListaConvidados* deve ser renderizada para o cliente.

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*.

Para vermos algo funcionando o mais rápido possível, vamos criar uma página simples com o Thymeleaf e usando também o Bootstrap, para deixar nosso sistema mais bonito.

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

<head>
    <meta charset="UTF-8"/>
    <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
    <meta name="viewport" content="width=device-width" />

    <title>Lista de Convidados</title>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
```

```
        crossorigin="anonymous"/>
</head>
<body>

    <h1>AlgaWorks!</h1>

</body>
</html>
```

Vamos destacar alguns pontos deste código para melhor entendimento.

Logo no início temos:

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

O atributo "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>
```

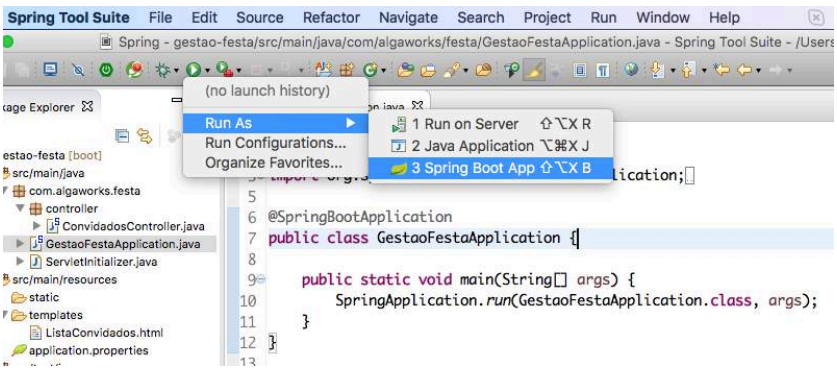
Já "xmlns:th" define que podemos usar as propriedades definidas pelo Thymeleaf, e você vai gostar delas, nas próximas seções veremos algumas.

Por fim vamos destacar a importação do Bootstrap, um framework HTML, CSS e JavaScript para desenvolvimento de aplicações responsivas pra web, e que nos ajuda a criar aplicações bem mais elegantes com menos esforço.

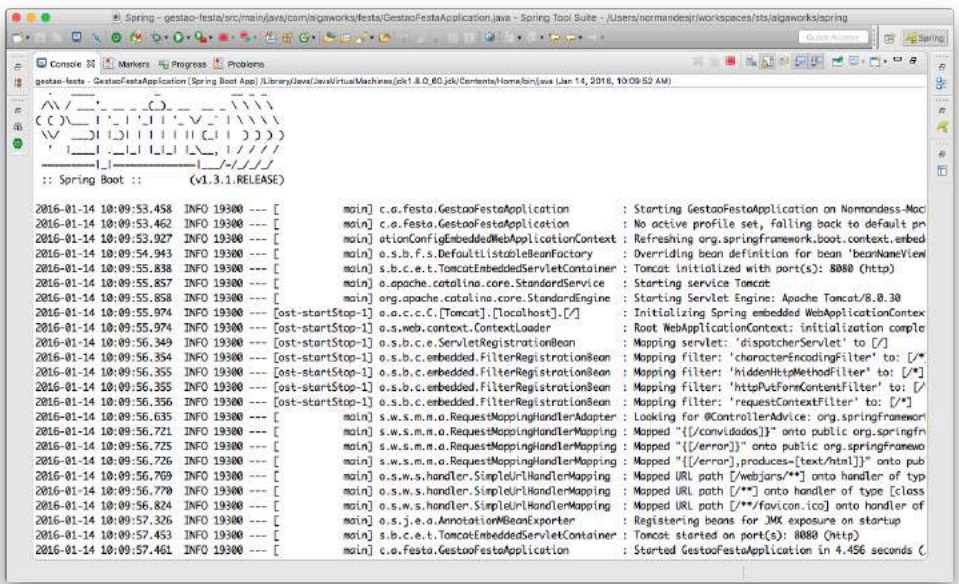
3.5. Rodando o projeto pela primeira vez

Já temos algo executável, nada funcional ainda, mas já podemos ver algo 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, 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 armazená-los e buscá-los.

Não iremos usar um banco de dados de verdade, porque foge do escopo deste livreto, vamos apenas adicionar e listar usando uma variável estática que fica em memória.

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

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

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

public class Convidado {
```

```
    private String nome;
    private Integer quantidadeAcompanhantes;

    public Convidado() {

    }

    public Convidado(String nome, Integer quantidadeAcompanhantes) {
        this.nome = nome;
        this.quantidadeAcompanhantes = quantidadeAcompanhantes;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public Integer getQuantidadeAcompanhantes() {
        return quantidadeAcompanhantes;
    }

    public void setQuantidadeAcompanhantes
            (Integer quantidadeAcompanhantes) {
        this.quantidadeAcompanhantes = quantidadeAcompanhantes;
    }

}
```

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

No código abaixo você pode ver que carregamos alguns convidados em um bloco estático, apenas para vermos algo na tela. O método `todos()` retorna a lista de convidados e o método `adicionar(Convidado)` nos permite adicionar um novo convidado à lista.

```
package com.algaworks.festa.repository;
```

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

import com.algaworks.festa.model.Convidado;

public class Convidados {

    private static final List<Convidado> LISTA_CONVIDADOS
        = new ArrayList<>();

    static {
        LISTA_CONVIDADOS.add(new Convidado("Pedro", 2));
        LISTA_CONVIDADOS.add(new Convidado("Maria", 3));
        LISTA_CONVIDADOS.add(new Convidado("Ricardo", 1));
    }

    public List<Convidado> todos() {
        return Convidados.LISTA_CONVIDADOS;
    }

    public void adicionar(Convidado convidado) {
        Convidados.LISTA_CONVIDADOS.add(convidado);
    }

}
```

Está quase pronto, precisamos apenas falar para o Spring que essa classe é um componente, e que é um repositório. Para isso, basta adicionarmos a anotação `@Repository` na classe.

```
package com.algaworks.festa.repository;

import org.springframework.stereotype.Repository;
// omitindo os outros imports

@Repository
public class Convidados {

    // omitindo o código

}
```

Essa simples anotação nos permite 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 à 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.web.servlet.ModelAndView;

import com.algaworks.festa.model.Convidado;
import com.algaworks.festa.repository.Convidados;

// omitindo alguns imports

@Controller
public class ConvidadosController {

    @Autowired
    private Convidados convidados;

    @RequestMapping("/convidados")
    public ModelAndView listar() {
        ModelAndView mv = new ModelAndView("ListaConvidados");
        mv.addObject("convidados", convidados.todos());
        return mv;
    }

}
```

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

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 listar os 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>1</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.

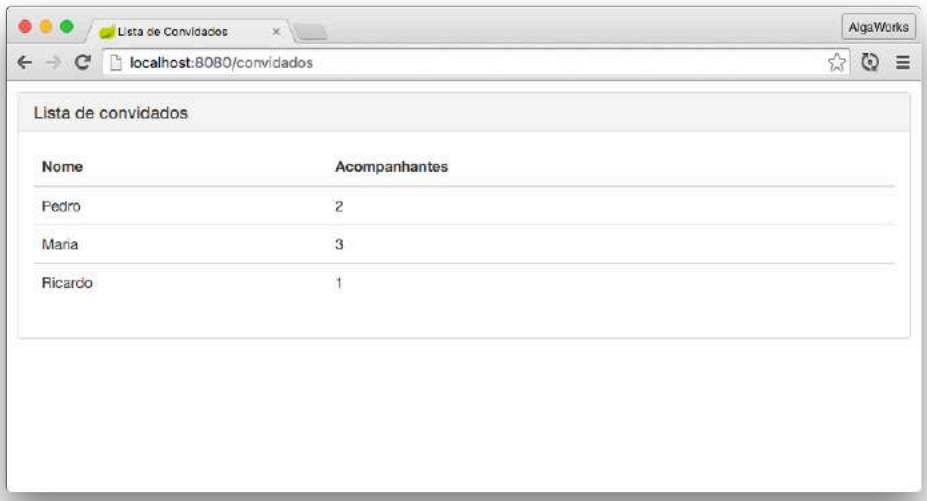
```
<tr th:each="convidado : ${convidados}">
    <td th:text="${convidado.nome}"></td>
    <td th:text="${convidado.quantidadeAcompanhantes}"></td>
</tr>
```

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.

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 às tags `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 `@{}` 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.

O método HTTP default que `@RequestMapping` define é GET, portanto vamos criar um outro método `salvar()` no nosso controller.

```
@RequestMapping(value = "/convidados", method = RequestMethod.POST)
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.

```
@RequestMapping(value = "/convidados", method = RequestMethod.POST)
public String salvar(Convidado convidado) {
    this.convidados.adicionar(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.

```
@RequestMapping(value = "/convidados", method = RequestMethod.POST)
public String salvar(Convidado convidado) {
```

```
    this.convidados.adicionar(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 `@RequestMapping` nos métodos `listar` e `salvar` e esteja perguntando: tem como melhorar?

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

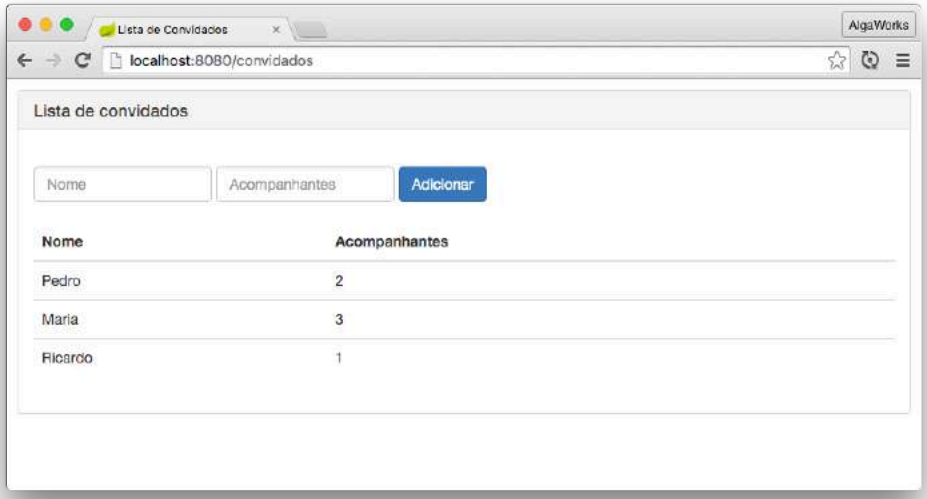
```
@Controller
@RequestMapping("/convidados")
public class ConvidadosController {

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

    @RequestMapping(method = RequestMethod.POST)
    public String salvar(Convidado convidado) {
        // ...
    }
}
```

Agora as anotações `@RequestMapping` nos métodos irão começar sempre com /convidados.

A aplicação final deve se parecer com a imagem abaixo.


Lista de Convidados
AlgaWorks
localhost:8080/convidados
Lista de convidados
Nome
Acompanhantes
Adicionar
Nome
Acompanhantes
Pedro
2
Maria
3
Ricardo
1

Capítulo 4

Conclusão

Que legal ter chegado ao final da leitura, estou feliz por você ter cumprido mais essa etapa na sua carreira.

Espero que tenha colocado em prática tudo que aprendeu. Não se contente em apenas ler esse livreto. 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 livreto, por favor, me ajude a manter esse trabalho. Recomende-o para seus amigos de trabalho, faculdade e/ou compartilhe no Facebook e Twitter.

4.1. Próximos passos

Embora esse livreto tenha te ajudado a criar uma aplicação do início ao fim com Spring MVC, Spring Boot 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 eu quero dizer é que há muito mais coisas para aprofundar.

Caso você tenha interesse em continuar seu aprendizado, recomendo que veja agora mesmo nosso workshop "Começando com Spring MVC" nesse link: http://alga.works/livreto-springmvc-cta/.

WORKSHOP ONLINE

SPRING MVC

COMPRE AQUI ↓

PRIMEIROS PASSOS COM SPRING MVC

NORMANDES JUNIOR

algaworks