Passo 1 (Sobre o Spring Boot) O Spring Boot é uma ferramenta que facilita a configuração de projetos que utilizam o Spring. Em geral, muitos projetos utilizam configurações idênticas ou muitas parecidas. O Spring Boot, em função dos recursos que o projeto utiliza, procura fazer as configurações automaticamente de como que o programador não tenha de se preocupar com isso. Isso inclui a especificação de dependências para o Maven. O Spring Boot se encarrega de escrever uma versão do arquivo pom.xml e de atualizá-la conforme as necessidades do projeto. Um bom IDE para desenvolvimento de aplicações com Spring MVC e Spring Boot é o Spring Tools Suite (STS), que pode ser obtido no Link 1.1. Trata-se de uma versão do Eclipse com diversos plugins apropriados para o uso de recursos do Spring.

Link 1.1
https://spring.io/tools3/sts/all

Apesar da existência do STS, iremos utilizar o Eclipse padrão. O Link 1.2 fornece uma ferramenta que permite a geração do arquivo pom.xml da mesma forma como o STS faria. Nessa ferramenta podemos especificar se queremos um projeto Maven ou Gradle, as versões desejadas, bibliotecas etc. Depois de gerado o arquivo, basta importá-lo no Eclipse.

Link 1.2
https://start.spring.io/

Passo 2 (Criando um projeto Maven com o Spring Boot) Aqui iremos utilizar a ferramenta do Link 1.2 para gerar um projeto inicial Maven, usando o Spring Boot. As bibliotecas que utilizaremos serão as seguintes
- DevTools (para valores de configuração padronizados, restart automático, LiveReload (Refresh no navegador automático etc)

- Web (Inclui o Tomcat e a implementação do Spring)

- H2 (um banco de dados na memória, somente para alguns testes iniciais)

- JPA (Java Persistence API, para fazer operações de persistência sem ter de escrever código JDBC diretamente)

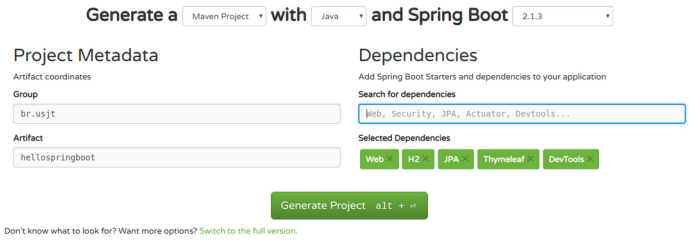
- Thymeleaf (Biblioteca de TAGS para a camada de visualização. Uma boa alternativa ao uso de JSPs)

A seguir, especifique os seguintes valores:

Group: br.com.bossini
Artifact: hellospringboot

O resultado deve ser aquele exibido pela Figura 2.1.

Figura 2.1



Clique em Generate Project, o que fará com que um arquivo .zip contendo o projeto criado seja baixado. Descompacte o arquivo.

Passo 3 (Importando o projeto no Eclipse) A seguir, no Eclipse (certifique-se de estar na perspectiva Java EE) clique em File >> Import Maven >> Existing Maven Projects. Navegue até a pasta em que o arquivo foi descompactado e realize a importação.

Passo 4 (Inspecionando o código fonte, executando e talvez trocando a porta) Expanda a pasta src/main/java para encontrar o pacote principal de sua aplicação e, dentro dele, uma classe criada automaticamente. Ela deve se parecer com aquela exibida pela Listagem 4.1.

Listagem 4.1

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

Note que há um método main e que esse não se parece com o código de uma aplicação WEB.

O que ocorre é que há um um container de servlets (o Tomcat, por padrão) embarcado e, quando o método main for executado, isso dará origem à criação de uma aplicação WEB que será hospedada nele.

Execute o método main. Isso colocará o Tomcat em funcionamento e hospedará a aplicação nele.

Nota: O Tomcat tentará usar a porta 8080 por padrão. Caso ela esteja ocupada, o console irá exibir um erro, como mostra a Figura 4.1. Para ajusta isso basta trocar a porta que seu Tomcat usará. Para tal, abra o arquivo src/main/resources/application.properties e adicione a configuração da Listagem 4.1. A porta que você vai usar não importa muito. Só é preciso usar uma porta que já não esteja em uso por outra aplicação.

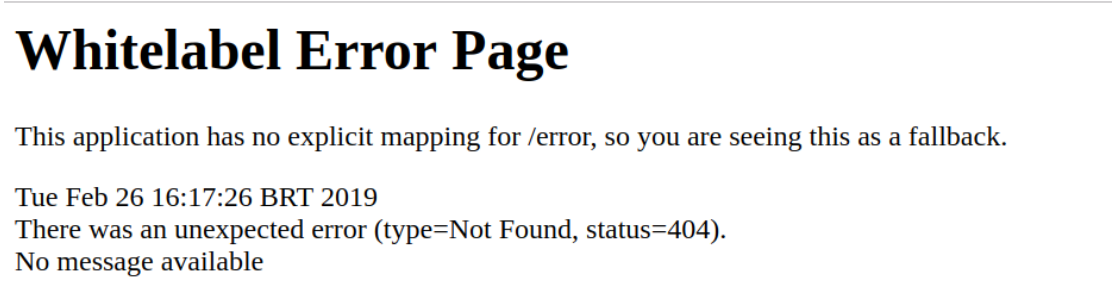
Listagem 4.1

```
server.port=9090
```

Figura 4.1

Você pode visitar o link localhost:8080 (ou usando a porta que você configurou) agora. Se tudo deu certo, você deverá ver a mensagem de erro exibida pela Figura 4.2. Ela indica que o recurso solicitado não foi encontrado e a aplicação não possui nenhum recurso mapeado para essa situação.

Figura 4.2



A anotação SpringBootApplication é fundamental. Ela tem os seguintes efeitos (entre outros):

- Indicar que esse método deverá ser executado por uma aplicação no container embarcado
- Indicar o pacote principal da aplicação o que faz com que ele e todos os seus subpacotes sejam considerados pelo Spring para encontrar componentes gerenciáveis.

Também é possível adicionar anotações para alterações configurações como o idioma, para fazer redirecionamentos etc.

Passo 5 (Criando um controller) Agora iremos criar um primeiro controller para a aplicação. Ele será responsável por atender requisições em um determinado padrão de URL. Para tal, clique com o direito na pasta src/main/java e escolha New >> Class. O nome da classe será **HelloSpringBootController** e ele ficará no pacote br.com.bossini.hellospringboot.controller. Iremos colocar todos os nossos controllers nesse pacote. O próximo passo é anotar a classe com @Controller, como mostra a Listagem 5.1.

Listagem 5.1

```
package br.com.bossini.hellospringboot.controller;
import org.springframework.stereotype.Controller;

@Controller
public class HelloSpringBootController {

}
```

Passo 6 (Adicionando um método ao controller) Quando a aplicação receber uma requisição, desejamos que um método Java entre em execução para que alguma regra de negócio seja executada. A seguir,desejamos direcionar o resultado para uma página que possa ser renderizada e exibida. Neste exemplo ainda não temos uma regra muito complexa. Na verdade, escreveremos somente um método que devolve o nome da página a ser exibida. Para especificar qual o padrão de URL que faz com que ele entre em execução, podemos usar algumas anotações como:

- **GetMapping** (para requisições HTTP GET)
- **PostMapping** (para requisições HTTP POST)
- **RequestMapping** (para requisições HTTP em geral)

O método fica como mostra a Listagem 6.1.

Listagem 6.1

```
package br.com.bossini.hellospringboot.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HelloSpringBootController {

	@GetMapping ("/hello")
	public String hello() {
		return "hello_spring_boot";
	}
}
```

Note que o nome do método não importa. O que define o momento em que ele entrará em execução é a string definida na anotação aplicada a ele. Note também que ele, de fato, devolve uma String. Esse é o nome da página que deverá ser renderizada e exibida depois que ele terminar a sua execução.

Passo 7 (Criando a página) Por padrão, o Spring irá procurar as páginas na pasta src/main/resources/templates e esperará que elas tenham a extensão .html. Lembre-se de que o método do controller devolve o nome **hello_spring_boot**, portanto o nome da página deverá ser hello_spring_boot.html. Para simplificar, vamos criar um arquivo simples e adicionar tags HTML a ele. Clique com o direito na pasta src/main/resources/templates e escolha New >> General >> File. O conteúdo do arquivo deve ser o exibido pela Listagem 7.1.

Listagem 7.1

```
<!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>Hello Spring Boot</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>Hello Spring Boot!</h1>
	</body>
</html>
```

Note os seguintes detalhes:
- definimos um namespace para o thymeleaf chamado th (o nome th poderia ser qualquer outro)
- usando uma biblioteca chamada Webjars, importamos os arquivos do bootstrap.

Para que o Webjars funcione, precisamos declarar sua dependência no pom.xml. Veja a Listagem 7.2. No Maven, cada tag dependency deve ser filha da tag dependencies.

Listagem 7.2

```
<dependency>
	<groupId>org.webjars</groupId>
	<artifactId>webjars-locator</artifactId>
	<version>0.36</version>
</dependency>
<dependency>
	<groupId>org.webjars</groupId>
	<artifactId>bootstrap</artifactId>
	<version>3.3.7</version>
</dependency>
```

Passo 8 (Executando a aplicação novamente) Execute o método main novamente e acesse o endereço localhost:sua_porta. Você deverá a mensagem que colocou em seu arquivo html.

Passo 9 (Criando uma entidade para testar o banco H2 e as anotações JPA) Vamos fazer com que nossa aplicação gerencie uma lista de alunos. A partir daqui, lidaremos com recursos sobre os quais falaremos mais detalhadamente nas próximas aulas. Na hora devida, eles serão explicados detalhadamente. Crie uma classe chamada Aluno no pacote br.com.bossini.hellospringboot.model. Ela deve ficar como mostra a Listagem 9.1.

Listagem 9.1

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
@Entity
public class Aluno implements Serializable{
	private static final long serialVersionUID = 1L;
	@Id
```

```
        @GeneratedValue
        private Long id;

        private String nome;

        private Double mediaNotas;

        //getters/setters
}
```

A classe Aluno faz uso de anotações JPA, que é uma especificação que permite automatizar o processo de mapeamento objeto/relacional. Ao utilizá-la, evitamos escrever código JDBC repetivo, pois ele será gerado automaticamente. As anotações que utilizamos têm o seguinte significado.

@Entity – Essa classe será mapeada para uma tabela no banco, por padrão com o mesmo nome.

@Id – Esse campo será mapeado como chave primária da tabela

@GeneratedValue – O valor de chave primária será gerado automaticamente (tal qual o AUTO_INCREMENT do MySQL, por exemplo)

Passo 10 (Criando uma interface para as operações de persistência) Naturalmente precisaremos cadastrar, consultar etc dados de alunos na base. Para isso, iremos criar uma interface na qual essas operações serão definidas. É comum que classes assim sejam chamadas de Repository, por isso ela será criada num pacote com esse nome. Porém, utilizaremos um recurso que nos permite simplesmente especificar uma interface que nos dará a implementação completa de métodos mais comum para operações no banco. Seu código aparece na Listagem 10.1. O simples fato de herdarmos de JPARepository (especificando o tipo envolvido e o tipo de sua chave primária) já faz com que métodos como findAll, findOne, save, exists sejam implementados automaticamente.

Listagem 10.1

```
package br.com.bossini.hellospringboot.repository;

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

import br.com.bossini.hellospringboot.model.Aluno;

public interface AlunosRepository extends JpaRepository<Aluno, Long>{

}
```

Passo 11 (Listando os alunos) Agora iremos criar um controller que permite fazer a listagem de todos os alunos (ok, ainda não tem nenhum, mas logo passaremos a cadastrá-los também). Lembre-se de que estamos usando, implicitamente, o banco de dados H2, que funciona diretamente na memória e que, a menos da dependência do Maven, não requer qualquer outra configuração. Os passos são os seguintes.

11.1 Crie uma nova classe chamada AlunosController no pacote br.com.bossini.hellospringboot.controller e configure-a como um novo controller. Veja a Listagem 11.1.

Listagem 11.1

```
package br.com.bossini.hellospringboot.controller;

import org.springframework.stereotype.Controller;

@Controller
public class AlunosController {

}
```

11.2 Para realizar as operações de acesso à base, precisaremos de um AlunosRepository. Não iremos instanciá-lo explicitamente. Iremos usar o recurso mais importante do Spring que é a injeção de dependência. Nosso código ***depende*** da existência de uma instância de AlunoRepository para funcionar, certo? Iremos pedir que o Spring crie a instância e injete na classe AlunosController. Veja a Listagem 11.2.

Listagem 11.2

```
@Controller
public class AlunosController {
	//a injeção de dependência ocorre aqui
	@Autowired
	private AlunosRepository alunosRepo;
}
```

11.3 De alguma forma o controller deverá disponibilizar a lista de alunos para que a view possa exibir. Ele pode fazer isso utilizando um objeto do tipo ModelAndView. Um objeto desse tipo permite que a view de destino seja especificada e ele carrega uma coleção de objetos que ficarão disponíveis para a view. Veja a Listagem 11.3.

Listagem 11.3

```
@GetMapping ("/alunos")
public ModelAndView listarAlunos () {
	//passe o nome da página ao construtor
	ModelAndView mv = new ModelAndView ("lista_alunos");

	//Busque a coleção com o repositório
	List <Aluno> alunos = alunosRepo.findAll();

	//adicione a coleção ao objeto ModelAndView
	mv.addObject("alunos", alunos);

	//devolva o ModelAndView
	return mv;
}
```

Passo 12 (Criando a página para listar alunos) Na pasta src/main/resources/templates, crie o arquivo chamado lista_alunos.html. Veja seu conteúdo na Listagem 12.1. Execute a aplicação e acesse o link localhost:sua_porta para ver o resultado.

Listagem 12.1

```
<!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>Hello Spring Boot</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>
		<div class="panel panel-default" style="margin: 10px">
			<div class="panel-heading">
				<h1 class="panel-title">Alunos</h1>
			</div>
			<div class="panel-body">
				<table class="table">
					<thead>
						<tr>
							<th>Nome</th>
							<th>Média</th>
						</tr>
					</thead>
					<tbody>
						<tr>
							<td>Ana</td>
							<td>10</td>
						</tr>
					</tbody>
				</table>
			</div>
		</div>
	</body>
</html>
```

Passo 13 (Pegando os objetos do banco) Note que a página lista_alunos tem somente um aluno e que seus dados estão fixos ali. Iremos agora utilizar tags do Thymeleaf para pegar todos os alunos que estão na coleção colocada no objeto ModelAndView pelo Controller. Ajuste o código da página como mostra a Listagem 13.1.

Listagem 13.1

```
<tbody>
	<tr th:each="aluno : ${alunos}">
		<td th:text="${aluno.nome}"></td>
		<td th:text="${aluno.mediaNotas}"></td>
	</tr>
</tbody>
```

Passo 14 (Inserindo dados no H2) Se a página for acessada agora ela mostrará um painel vazio. A razão é óbvia. Não há dados ainda na base de dados. Para fazer a inserção de dados no H2, basta criar um arquivo chamado import.sql na pasta src/main/resources e colocar as instruções de inserção nele. Automaticamente o spring irá executar o script a cada execução. Vá em frente e crie o arquivo, colocando como conteúdo o que exibe a Listagem 14.1.

Listagem 14.1

```
insert into aluno (id, nome, media_notas) values (1, 'Ana', 10);
insert into aluno (id, nome, media_notas) values (2, 'Maria', 3);
insert into aluno (id, nome, media_notas) values (3, 'Ricardo', 7);
```

Execute novamente a aplicação para ver o resultado!

Passo 15 (Form para adicionar alunos) Na mesma página de listagem, logo após o heading da tabela, adicione o formulário da Listagem 15.1 que permitirá que novos alunos sejam inseridos.

Listagem 15.1

```
<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="Média"/>
        <button type="submit" class="btn btn-primary">Adicionar</button>
        </div>
</form>
```

Passo 16 (Adicionando um objeto para modelar o formulário) Precisamos adicionar um objeto do tipo Aluno ao objeto ModelAndView. As propriedades desse objeto serão preenchidas automaticamente pelo Spring usando os campos de entrada de dados do formulário. Adicione o código da Listagem 16.1 ao método listarAlunos.

Listagem 16.1

```
//para modelar o formulário
mv.addObject(new Aluno());
```

Passo 17 (Indicando ao Thymeleaf qual objeto usar no form) Para que o Thymeleaf possa preencher os campos desse objeto com os valores dos campos de entrada do formulário, vamos usar sua tag “object”. Ela deve ser aplicada ao form, como mostra a Listagem 17.1.

Listagem 17.1

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

Além disso, será necessário associar cada campo do formulário a cada propriedade do objeto correspondente. Veja a Listagem 17.2.

Listagem 17.2

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

Passo 18 (Action do form) Para onde a requisição deve ser enviada quando o botão Adicionar for clicado. Podemos especificar isso usando a propriedade action do Thymeleaf. Veja a Listagem 18.1.

Listagem 18.1

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

Acesse a página localhost:sua_porta agora para ver o erro da Figura 18.1. Sempre que tiver um erro, pare para ler com atenção. Veja que, neste caso, ele é muito simples. Estamos enviando o formulário por meio do método POST do protocolo HTTP, porém não temos nenhum componente capaz de atender essa requisição. Embora tenhamos um método mapeado com o padrão /alunos, ele usa a anotação @GetMapping, o que diz que só é ativado quando o método HTTP GET é utilizado.

Figura 18.1

Passo 19 (Criando o método para salvar um aluno) No mesmo controller de alunos, crie um novo método responsável por persistir os dados de aluno vindos do formulário. A Listagem 19.1 mostra o código. Note que ele é anotado com PostMapping e que responde ao mesmo padrão: alunos. O objeto aluno recebido está preenchido com os dados vindos do form. O retorno indica que desejamos ver a mesma página quando a operação terminar. O qualificador redirect indica ao navegador que ele deve fazer nova requisição ao servidor para obter a nova página, que já conterá o novo aluno inserido.

Listagem 19.1

```
@PostMapping
public String salvar (Aluno aluno) {
        alunosRepo.save(aluno);
        return "redirect:/alunos";
}
```

Pronto!

Atividade

1. Crie um novo projeto com Spring Boot, usando as mesmas bibliotecas deste material.

2. Crie uma classe para representar um Veiculo. Veiculos têm id, modelo, marca, ano de fabricação e cor.

3. Crie um repositório do Spring Data para lidar com veículos.

4. Crie um Controller que leva o usuário a uma página que lista todos os veículos existentes no banco.

5. Insira alguns veículos no banco e veja o resultado.