Introdução ao Spring Tool Suite 4

Prof. Julio Cesar Nardi.

Baseado no livro “Produtividade no Desenvolvimento de Aplicações Web com Spring Boot”, 3ª Edição, AlgaWorks, 2017, de Alexandre Afonso.

Aplicação introdutória usando Spring Boot, o Spring MVC, o Spring Data JPA e o Thymeleaf, além do Spring Tool Suite e o Maven. Tal aplicação gerencia títulos de livros de uma biblioteca pessoal. Teremos a classe *Livro* que possui os seguintes atributos: *nome* e *número de exemplares*.

Criando um projeto

No Spring Tool Suite 4, escolha *File → New → Spring Starter Project.*

Após isso, aparecerá a tela de configuração da aplicação com os respectivos campos, conforme apresentado na Figura 1.

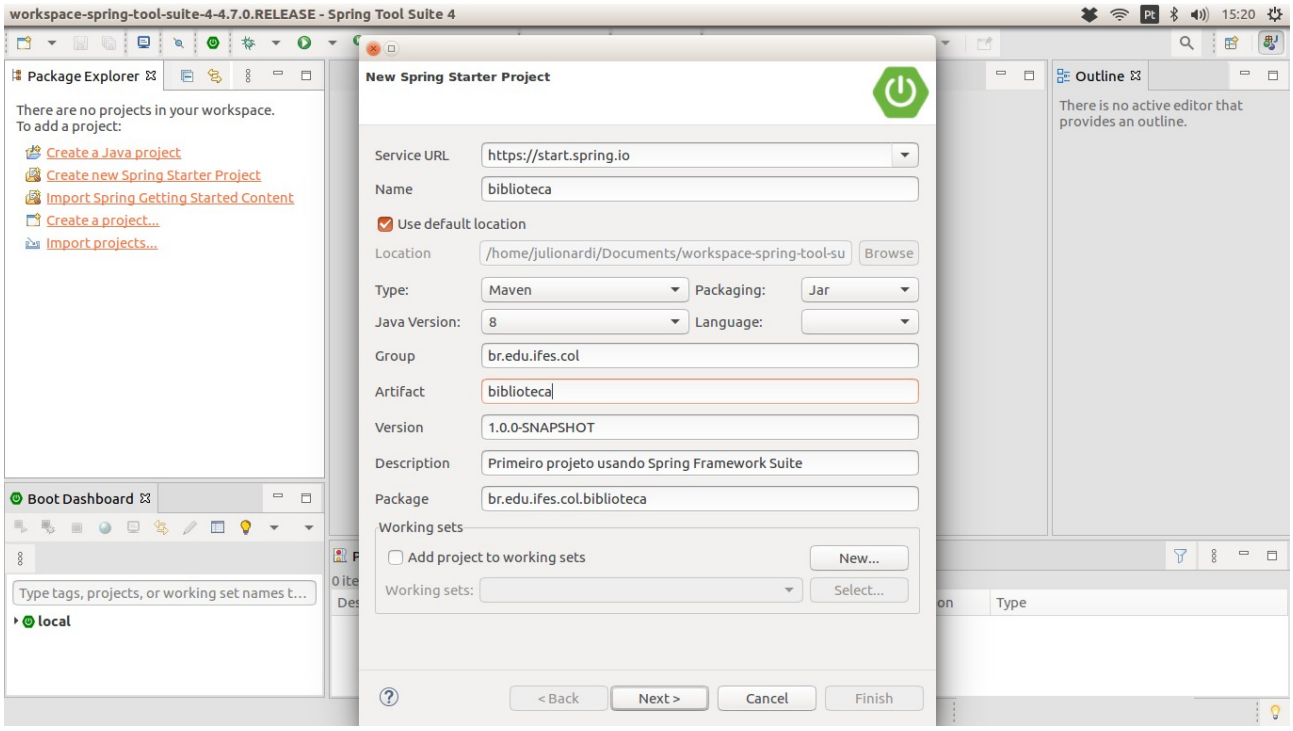

Figura 1: Tela de configuração da aplicação.

Vamos, agora, selecionar os *frameworks* e bibliotecas necessários à execução do projeto.

A Figura 2 apresenta a tela de configuração correspondente, já com os pacortes necessários ao nosso projeto. Observe que, por meio desta tela, podemos selecionar os pacotes necessários a qualquer aplicação e, assim, o STS gerará, automaticamente, o arquivo *pom.xml*, o qual contem as

ligações de dependência entre pacotes. No contexto deste projeto, vamos utilizar DevTools, Web, Thymeleaf, JPA e H2.

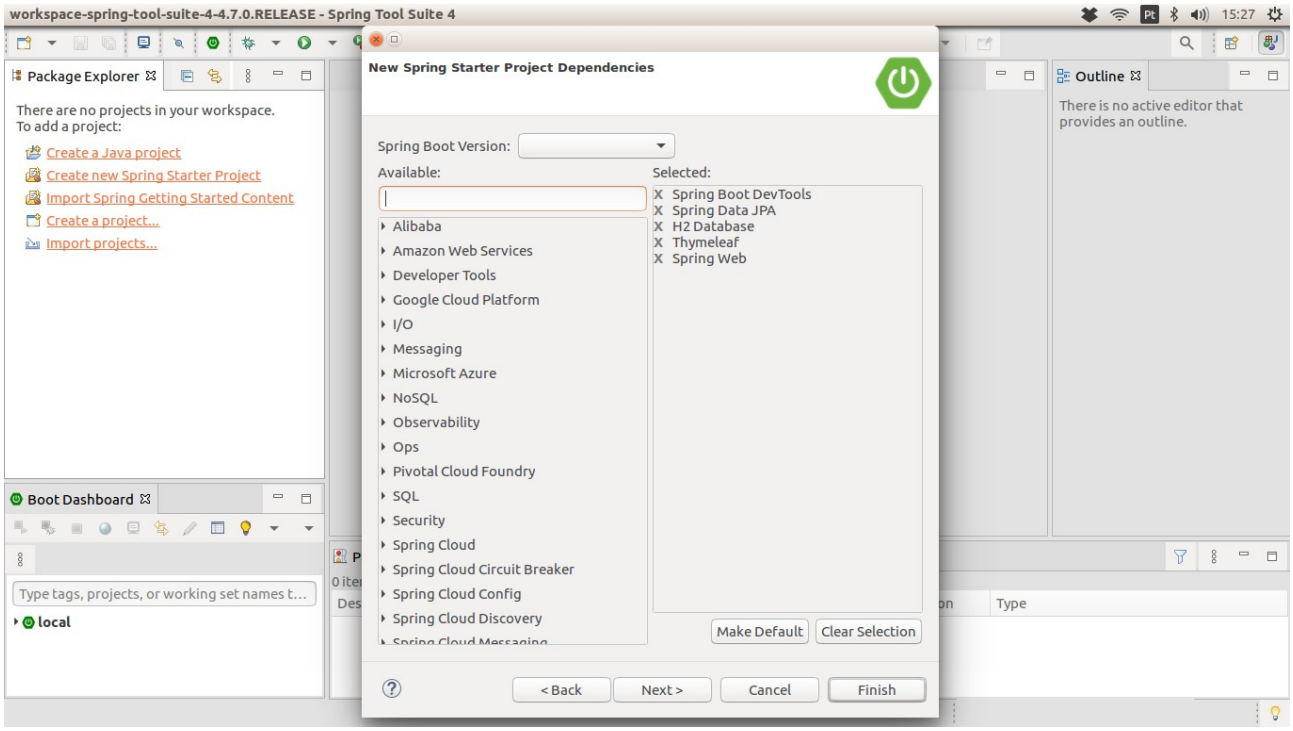

Figura 2: Tela de seleção de pacotes necessários à aplicação.

Ao clicar em *Finish*, o STS criará o projeto já com as configurações e toda a estrutura de pacotes, como pode ser observado na Figura 3.

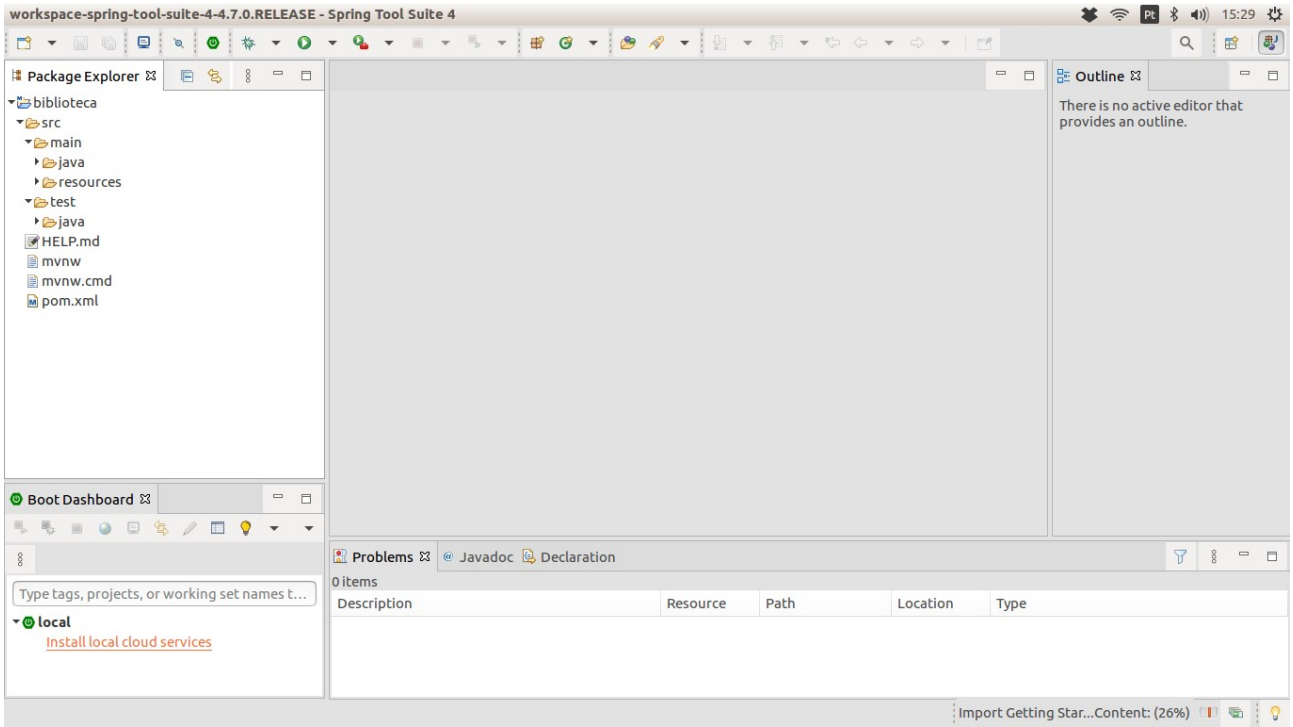

Figura 3: Tela inicial do projeto já criado.

Em *src/main/java*, dentro do pacote *br.edu.ifes.col.biblioteca* você encontra a classe *BibliotecaApplication.java*, a qual é o ponto de início da nossa aplicação. Basta executar, portanto, o método *main* desta classe. Assim, a classe *BibliotecaApplication.java* é anotada com *@SpringBootApplication*.

Testando o projeto criado

Para testar a configuração do projeto e verificar se a aplicação inicia sem erros, *clique com o direito na classe BibliotecaApplication* e selecione *Run As → 3 Spring Boot App*. Verifique no console se houve alguma mensagem de erro ou se o Spring iniciou corretamente, conforme mostra a Figura 4.

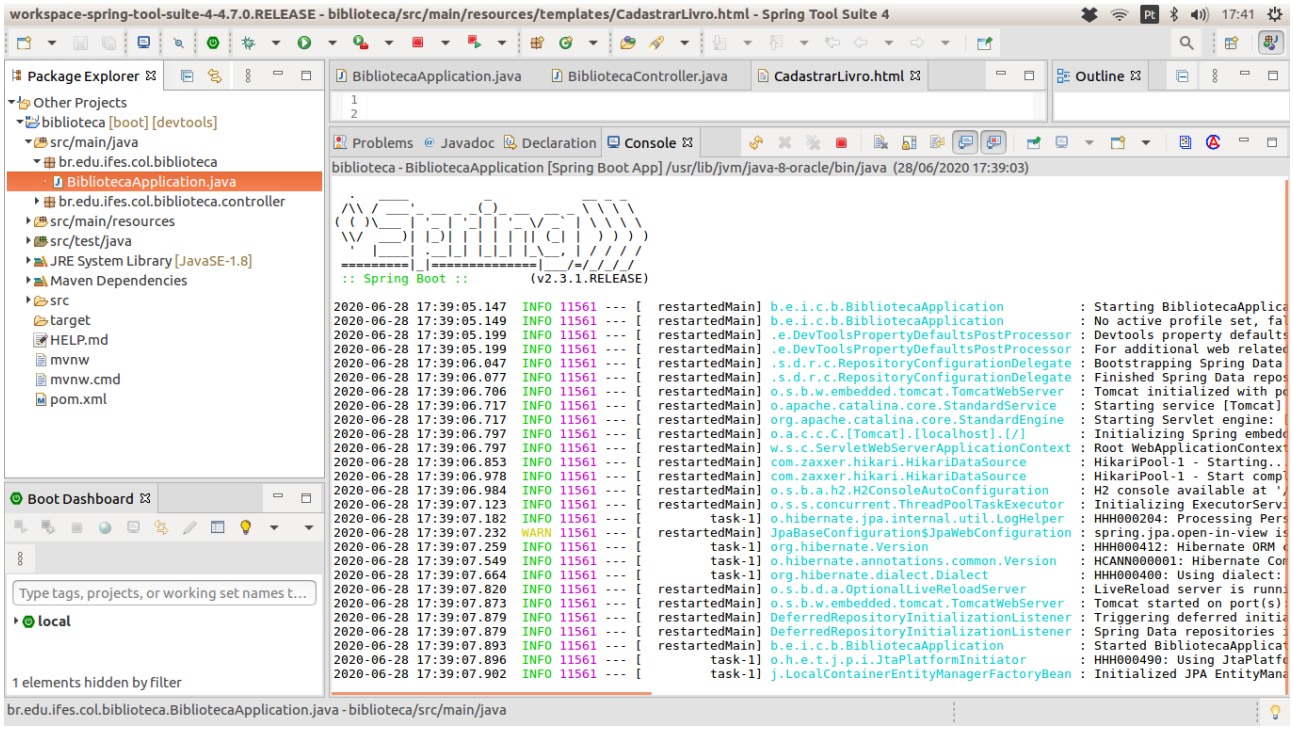

Figura 4: Tela do console onde são apresentadas mensagens de sucesso ou erro.

Criando o controlador

Crie uma nova classe chamada *BibliorecaController* por meio do menu *File → New → Class*. A Figura 5 exibe a janela de criação da classe que usaremos como controlador da nossa aplicação. Observe os campos de criação da classe como, por exemplo, *Package* e *Name*, dentre outros, e crie seu controlador.

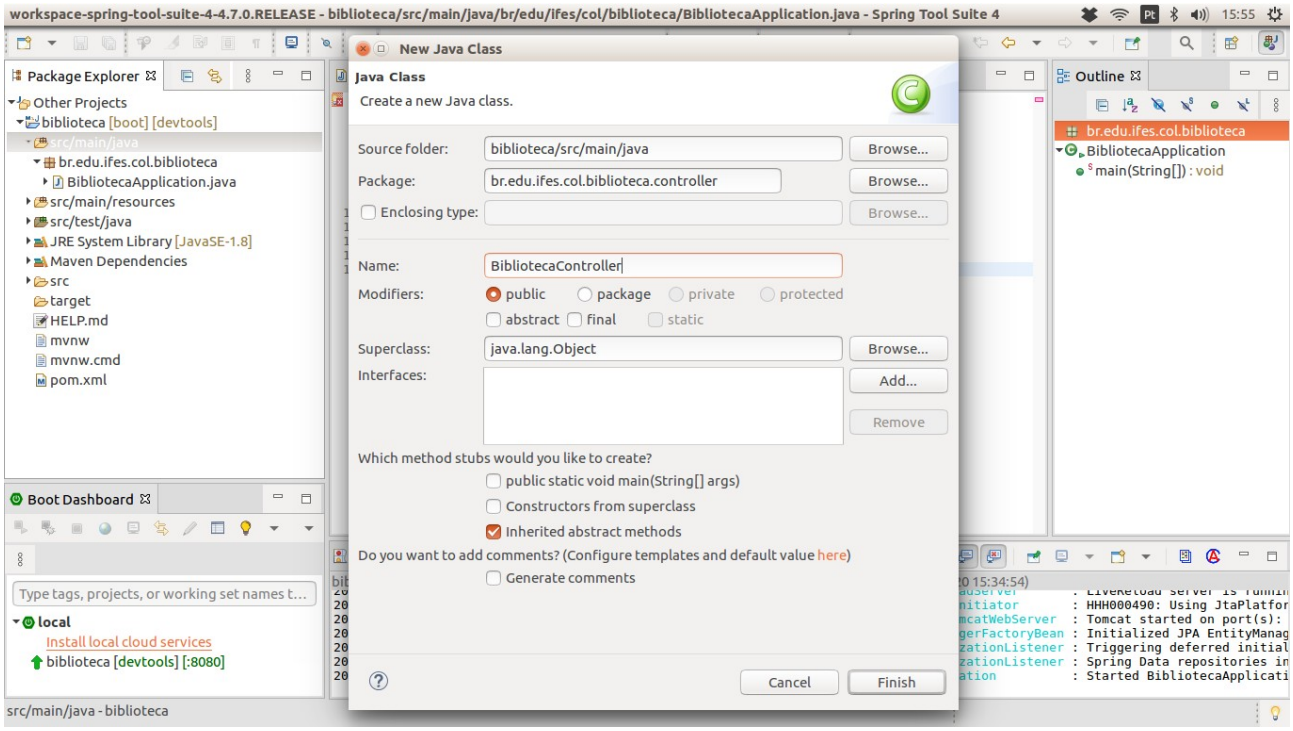

Figura 5: Tela de criação da classe BibliotecaController.

Uma vez criada a classe BibliotecaController, para que ela seja, de fato, identificada como um controlador pelo Spring Boot, temos que anotá-la com *@Controller*, conforme mostra a Figura 6.

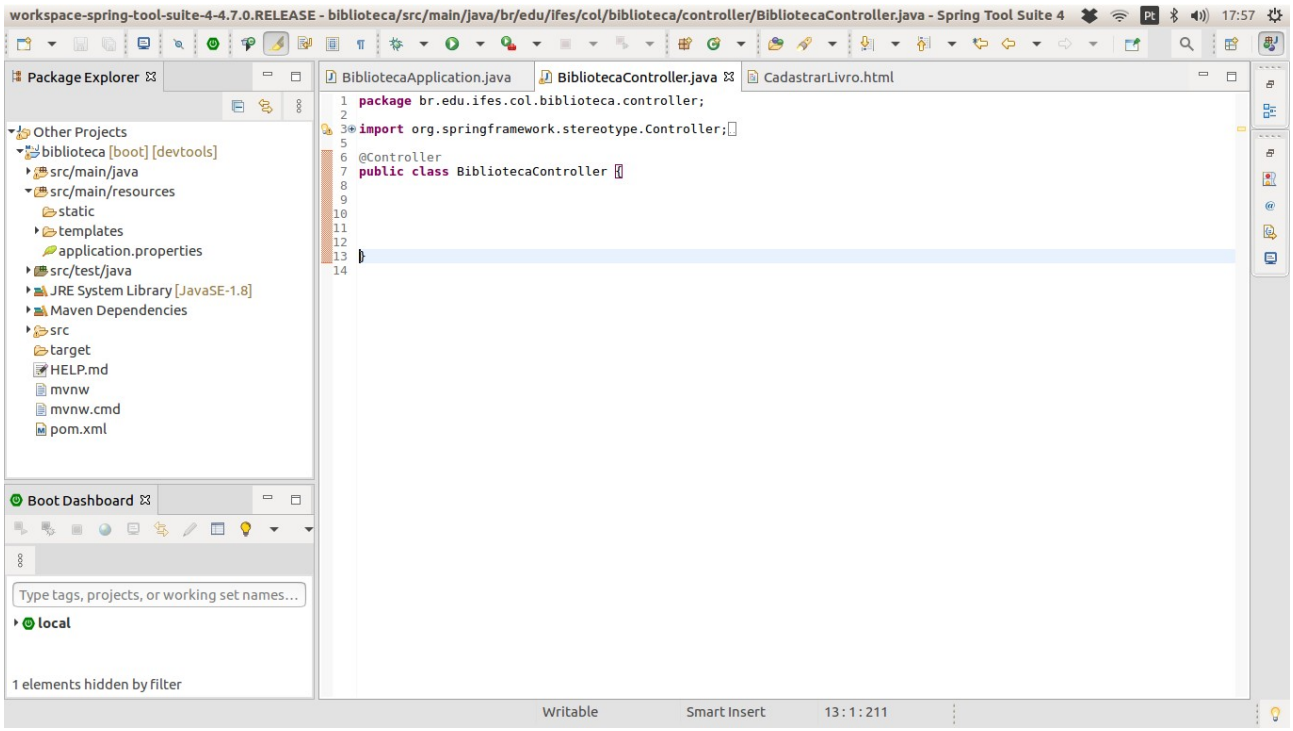

Figura 6: Tela apresentando anotação @Controller na classe BibliotecaController.

Criando a view

Considerando que configuramos nosso projeto para usar o Thymeleaf, por padrão, as páginas de visão serão organizadas em *src/main/resources/templates*. Assim, crie o arquivo *CadastrarLivro.html* em *src/main/resources/templates.* Para tanto, clique em *File → New → File*.

Vamos iniciar um com código bem simples para nossa página CadastrarLivro, conforme apresenta a Figura 7.

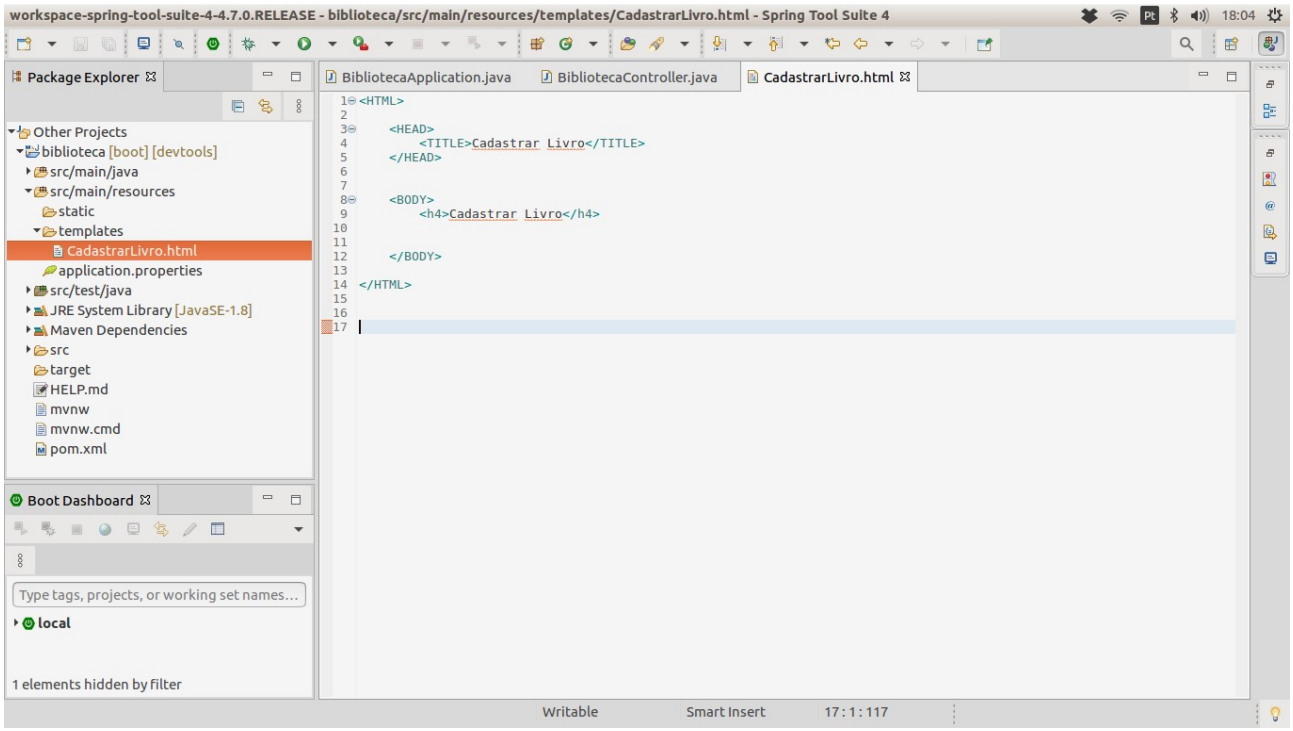

Figura 7: Código-fonte da página CadastrarLivro.html.

Na sequência, vamos configurar o Thymeleaf, por meio do arquivo *src/main/resources/application.properties* e usando as seguintes propriedades:

- *spring.thymeleaf.mode=html* (altera para HTML o modo de *templates* do Thymeleaf)
- *spring.thymeleaf.cache=false* (indica para não realizar cache das páginas)

A Figura 8 apresenta o arquivo de configuração editado.

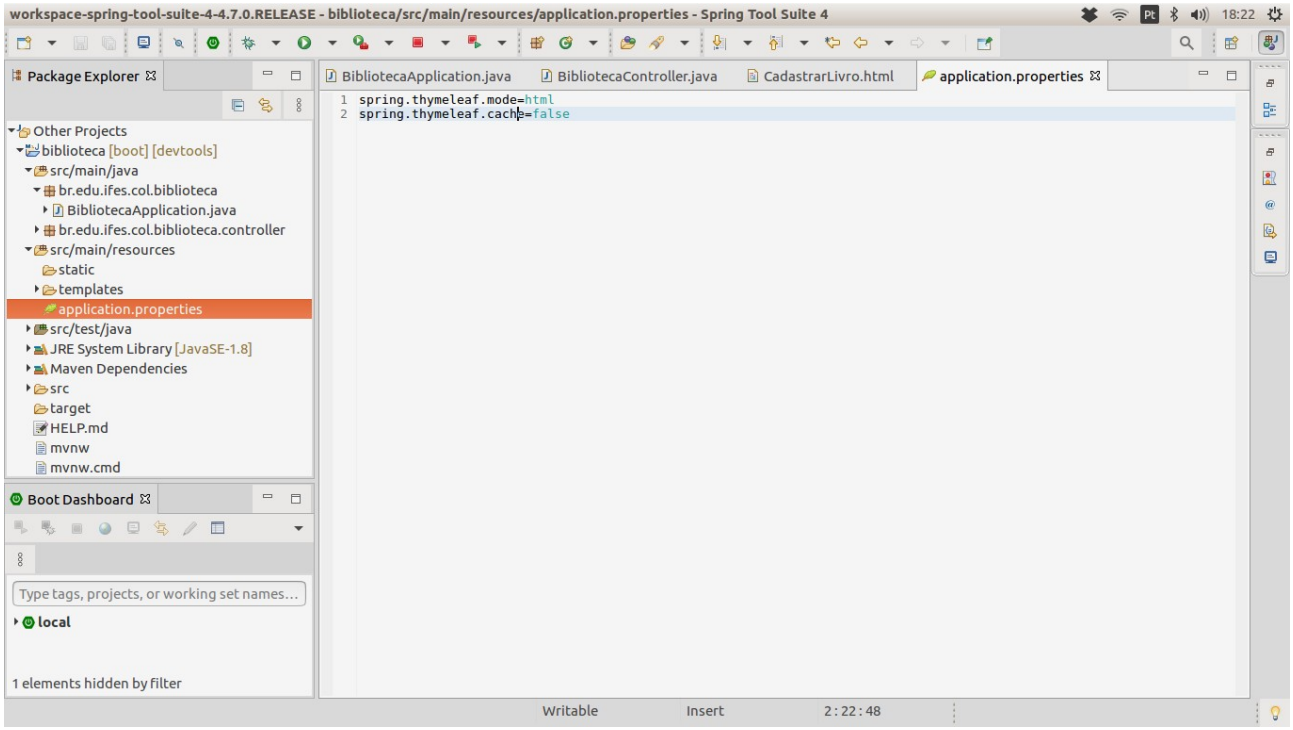

Figura 8: Tela de configuração do arquivo src/main/resources/application.properties

Configurando a comunicação View - Controller

Na classe BibliotecaController, adicione o código-fonte indicado abaixo:

```
package br.edu.ifes.col.biblioteca.controller;

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

@Controller
public class BibliotecaController {

	@GetMapping("/CadastrarLivro")
	public String getViewCadastrarLivro() {

		return "CadastrarLivro";

	}
}
```

Observe que criamos o método *getViewCadastrarLivro* e que acima desse método usamos a anotação *@GetMapping*, por meio da qual, é configurado o caminho de acesso à pagina *CadastrarLivro.html*. Em outras palavras, quando no browser for indicado o caminho da aplicação *http://localhost:8080/CadastrarLivro*, isso será repassado ao controller *BibliotecaController*, o qual retornará o nome da página de visão correspondente a ser exibida, a qual, portanto, será encaminhada ao *browser*.

Para realizar o teste dessas configurações:

1. Inicie a aplicação por meio de *clique com o direito* na classe *BibliotecaApplication → Run As → 3 Spring Boot App.*
2. Vá até o browser e digite *http://localhost:8080/CadastrarLivro*

Se tudo estiver funcionando corretamente, na tela do *browser* deve ser apresentado o resultado conforme exibido pela Figura 9.

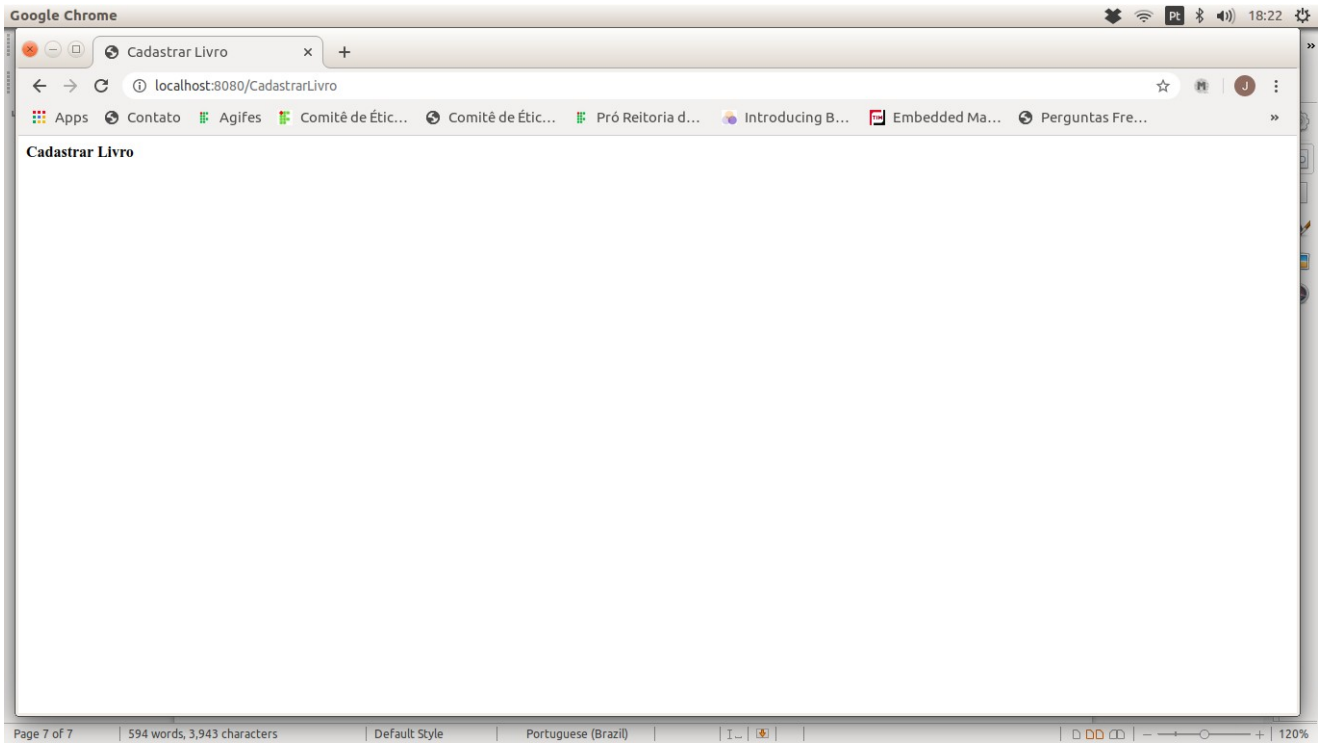

Figura 9: Tela com o resultado do acesso à página CadastrarLivro.

Criando o Model

Crie a classe *Livro* dentro do pacote *br.edu.ifes.col.biblioteca.model*, conforme o código abaixo.

```
package br.edu.ifes.col.biblioteca.model;

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

@Entity
public class Livro {

	@Id
	@GeneratedValue
	private Long id;

	private String titulo;

	private Integer numeroExemplares;

	public Livro() {
	}
...
}
```

Crie, agora, a interface *LivroRepository*, conforme o código abaixo. Para facilitar, vamos coloque tal interface no mesmo pacote em que foi criada a classe *Livro* (embora isso não seja obrigatório).

```
package br.edu.ifes.col.biblioteca.model;

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

public interface LivroRepository extends JpaRepository <Livro, Long>{


}
```

Observe que a interface *LivroRepository* herda de *JpaRepository*, ou seja, é um tipo de repositório no padrão JPA e se comportará como tal, inclusive para questões de injeção de dependências.

Passando parâmetros/objetos entre Controller e View

Além de retornar o nome/referência de páginas de visão a serem exibidas, um controlador pode retornar também objetos para uma determinada página de visão, a qual utilizará esses objetos para realizar algum processamento e, então, ser exibida.

Para não alterarmos o código do controlador *BibliotecaController*, vamos criar um novo controlador chamado *LivrosController*, a fim de demonstrar a passagem de parâmetros entre controlador e visão, conforme código abaixo.

```
package br.edu.ifes.col.biblioteca.controller;

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

import br.edu.ifes.col.biblioteca.model.Livro;
import br.edu.ifes.col.biblioteca.model.LivroRepository;

@Controller
public class LivrosController {

	@Autowired
	private LivroRepository livrosCadastrados;


	@GetMapping("/obterLivrosCadastrados")
	public ModelAndView obterLivrosCadastrados() {

		ModelAndView modelAndView = new ModelAndView("CadastrarLivro");
		modelAndView.addObject("var_livros_cadastrados",
livrosCadastrados.findAll());
		modelAndView.addObject(new Livro());
		return modelAndView;

	}
}
```

Observe a anotação *@Autowired* sobre o atributo *livrosCadastrados*, a qual indicará um ponto de injeção de dependências tomando como base a interface *LivroRepository*. Ademais, observe que ao criar um objeto da classe *ModelAndView*, passamos como parâmetro o nome da view para a qual o objeto que será encaminhado.

Processando objetos recebidos pela view

Alteraremos o código da página *CadastrarLivro.html* de modo que abaixo dos campos de entrada de dados, possam ser exibidos os livros já cadastrados. Observe o código abaixo, em especial, a parte em destaque (negrito).

```
<HTML>
	<HEAD>
		<TITLE>Cadastrar Livro</TITLE>
	</HEAD>

	<BODY>

		<h4>Cadastrar Livro</h4>

		<form>
			Título:
			<br>
			<input type=text name='txt_titulo'>
			<br>
			Número de exemplares:
			<br>
			<input type=text name='num_exemplares'>
			<br>
			<br>
			<input type='submit' value='OK'> <input type='reset'
value='Cancelar'>

		</form>

		<br>
		<br>

		<h4> Livros Cadastrados </h4>

		<table border=1>

			<tr>
				<td>Título</td>
				<td>Nº exemplares</td>
			</tr>

			<tr th:each="aux_livro : ${var_livros_cadastrados}">
				<td th:text="${aux_livro.titulo}"></td>
				<td th:text="${aux_livro.numeroExemplares}"></td>
			</tr>

		</table>

	</BODY>
</HTML>
```

Observe o nome da variável *var_livros_cadastrados* e compare com aquela usada pelo controlador *LivrosController* para passar o objeto para a view correspondente. Temos uma espécie de chave-valor, associando o objeto passado com parâmetro à chave de acesso/recuperação. Assim, esse código escrito em Thymeleaf irá percorrer a lista de livros recebida e criará a tabela correspondente em HTML, gerando, assim, dinamicamente a página resultante.

Para testarmos o resultado, vamos usar um artifício de criar algumas instâncias de Livros no repositório por meio do arquivo *main/resources/import.sql*. Localize este arquivo na árvore de seu projeto e inclua as seguintes linhas de código:

```
insert into livro (id, titulo, numero_exemplares) values (1, 'Livro 1', 2);
insert into livro (id, titulo, numero_exemplares) values (2, 'Livro 2', 5);
insert into livro (id, titulo, numero_exemplares) values (3, 'Livro 3', 7);
```

Em seguida, inicie sua aplicação e acione a página por meio da URL http://localhost:8080/obterLivrosCadastrados. O resultado é aquele apresentado pela Figura 10.

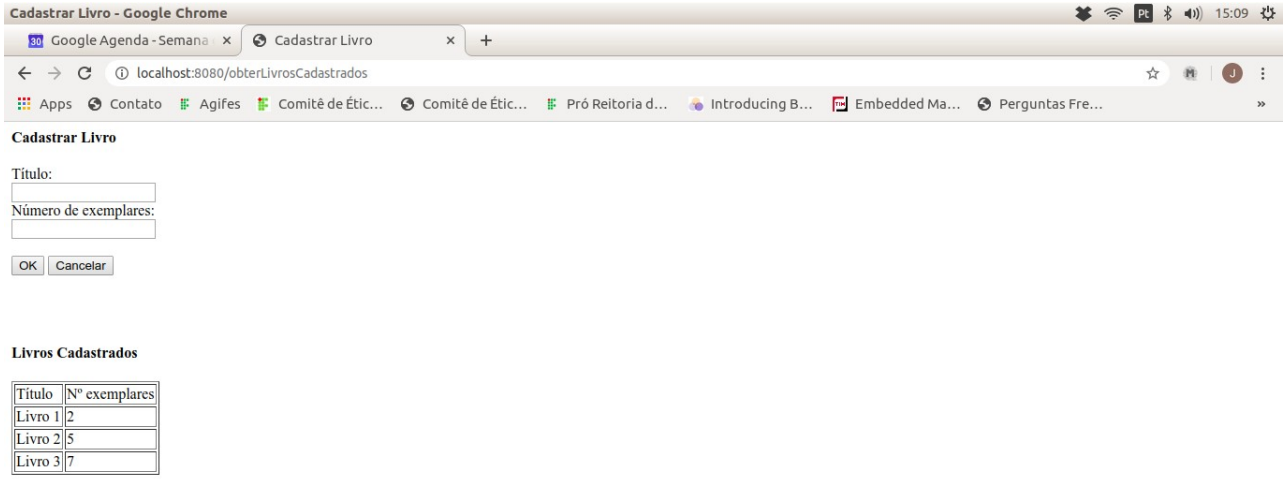

Figura 10: Tela com a listagem de livros cadastrados já implementada.

Adicionando um livro

Para adicionar um livro, vamos utilizar o formulário para entrada de dados existente na página *CadastrarLivro.html*.

Ademais, vamos modificar o método *obterLivrosCadastrados*, da classe *LivrosController*, deixando-o da seguinte forma:

```
@GetMapping("/obterLivrosCadastrados")
public ModelAndView obterLivrosCadastrados() {

            ModelAndView modelAndView = new ModelAndView("CadastrarLivro");
            modelAndView.addObject("var_livros_cadastrados",
livrosCadastrados.findAll());
            modelAndView.addObject(new Livro());
            return modelAndView;

}
```

Observe que adicionamos um objeto do tipo *Livro* no *ModelAndView*. Esse objeto será utilizado para manupular os valores das *tags input* do formulário HTML.

No formulário, vamos ajustar o código-fonte para poder utilizar tal objeto por meio do Thymeleaf. Assim, vamos adicionar o atributo *th:object*. Vamos também definir a *tag th:action* para indicar o destino dos dados a serem enviados. Nos campos de entrada, vamos associar o nome dos atributos de classe (no caso, da classe Livro) aos *inputs* usando *th:field*.

```
...

<h4>Cadastrar Livro</h4>

<form method="POST" th:object="${livro}" th:action="@{/salvarLivros}">

      Título:
      <br>
      <input type=text th:field="*{titulo}">
      <br>
      Número de exemplares:
      <br>
      <input type=text th:field="*{numeroExemplares}">
      <br>
      <br>
      <input type='submit' value='OK'> <input type='reset' value='Cancelar'>

</form>

...
```

Vamos, agora, ajustar o controlador *LivrosController* para que possa tratar a requisição */salvarLivros* definida no formulário HTML. Vamos inserir o seguinte método na classe:

```
@PostMapping("/salvarLivros")
public String salvar(Livro livro) {
      this.livrosCadastrados.save(livro);
      return "redirect:/obterLivrosCadastrados";
}
```

Observe que o método *salvar(...)* recebe como parâmetro o objeto a ser salvo, o qual foi criado e preparado pelo Spring MVC usando os dados informados no formulário HTML. Depois de salvar o objeto, a linha *return "redirect:/obterLivrosCadastrados"* permite que a página com o formulário seja recarregada e novos dados inseridos sejam apresentados.

Alterando o banco de dados

Até o momento utilizamos o banco de dados H2, o qual permite que os dados sejam armazenados e manipulados apenas na memória principal, não sendo, assim, persistidos no disco.

Para que possamos realizar a persistência definitiva, vamos passar a utilizar outro banco de dados. No caso, usaremos o MySQL. Assim, vamos, primeiramente, adicionar a dependência do driver JDBC no arquivo pom.xml do projeto, utilizando o código abaixo:

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

Em seguida, vamos definir as propriedades de acesso ao banco de dados no arquivo *src/main/resources/application.properties*, conforme abaixo. A propriedade “ddl-auto” indica que o banco de dados será recriado todas as vezes que o projeto se iniciar.

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