Passo 1 (Injeção de dependência no Spring – Definição inicial) A injeção de dependência é um tipo de inversão de controle. No contexto do Spring, ele fica responsável por disponibilizar os objetos dos quais um objeto depende para funcionar, ao invés de o próprio objeto instanciar suas dependências. Daí o nome inversão de controle.

Passo 2 (Inspecionando o código atual) Abra o projeto da aula passada e inspecione o arquivo AlunosController.java. Note a presença da anotação @Autowired. Ela foi utilizada para solicitar ao Spring que faça a injeção dessa dependência no controller. Ou seja, ele se encarrega se instanciar a classe e disponibilizar essa instância. Note que em momento algum utilizamos o operador **new** para construir esse objeto.

Passo 3 (Testando anotações em construtores) Também é possível anotar um construtor para que o Spring faça a injeção de dependência. Comente a anotação @Autowired do campo alunosRepo e adicione um construtor anotado, como mostra a Listagem 3.1, a fim de testar esse recurso.

Listagem 3.1

```
//@Autowired
private AlunosRepository alunosRepo;


//anotando um construtor
@Autowired
public AlunosController (AlunosRepository alunosRepo) {
	this.alunosRepo = alunosRepo;
}
```

Passo 4 (Testando anotações em métodos setter) Assim como em campos e construtores, também podemos aplicar a injeção de dependência em métodos modificadores. Faça o teste como mostra a Listage 4.1.

Listagem 4.1

```
// @Autowired
private AlunosRepository alunosRepo;
```

```
// anotando um método setter

@Autowired

public void setAlunosRepo(AlunosRepository alunosRepo) {

	this.alunosRepo = alunosRepo;

}
```

Passo 5 (Averiguando as interfaces do Spring que tornam objetos candidatos à injeção de dependência) As interfaces da Tabela 5.1 podem ser utilizadas para realizar a injeção de dependência por meio de anotações.

Tabela 5.1

Interface	Finalidade
@Component	Indica que a classe anotada é um "componente". É a anotação base para que elas possam se tornar auto detectáveis e injetáveis pelo Spring.
@Repository	Indica que a classe anotada é um Repositório, como Eric Evans [1] define : "Um mecanismo para encapsular armazenamento, recuperação, e comportamentos de busca que simulam uma coleção de objetos.
@Service	Indica que a classe anotada é um Serviço, como Eric Evans [1] define: "Uma operação oferecida como uma interface que se sobressai do modelo, sem estado encapsulado".
@Controller	Para que a classe anotada seja auto detectável por meio do escaneando do classpath feito pelo Spring. Geralmente usada em conjunto com @RequestMapping ou alguma de suas especializações.

Passo 6 (Adicionando a camada de serviço ao nosso projeto) Agora iremos criar a interface de serviço de nossa aplicação.

6.1 Comece criando a classe AlunosService, como mostra a Listagem 6.1. Coloque-a num pacote chamado service que deve ser subpacote do pacote principal da aplicação.

Listagem 6.1

```
@Service
public class AlunosService {
	@Autowired
	private AlunosRepository alunosRepo;

	public void salvar (Aluno aluno) {
		alunosRepo.save(aluno);
	}

	public List <Aluno> listarTodos (){
		return alunosRepo.findAll();
	}
}
```

6.2 Agora adapte o controller para injetar o componente de serviço. Veja a Listagem 6.2.

Listagem 6.2

```
//não faça mais essa injeção

// @Autowired

//private AlunosRepository alunosRepo;

//injete um serviço

@Autowired

private AlunosService alunosService;
```

6.3 Ajuste os métodos listar e salvar para usar o service. Veja a Listagem 6.3.

Listagem 6.3

```
@GetMapping("/alunos")

public ModelAndView listarAlunos() {

	// passe o nome da página ao construtor

	ModelAndView mv = new ModelAndView("lista_alunos");

	// para modelar o formulário

	mv.addObject(new Aluno());

	// Busque a coleção com o service

	List<Aluno> alunos = alunosService.listarTodos();

	// adicione a coleção ao objeto ModelAndView

	mv.addObject("alunos", alunos);

	// devolva o ModelAndView

	return mv;

}

@PostMapping("/alunos")

public String salvar(Aluno aluno) {

	//salvando com o service

	alunosService.salvar(aluno);

	return "redirect:/alunos";

}
```

Passo 7 (Injetando objetos quaisquer) O que ocorre se tivermos uma classe com funcionalidade de interesse e que não faz parte de nosso projeto ou da qual nem mesmo temos o código fonte? Neste exemplo, iremos inserir uma calculadora de média e ilustrar como permitir que o Spring a injete em um de nossos componentes mesmo que ela não faça parte do projeto.

7.1 Crie a classe calculadora, como mostra a Listagem 7.1. Ela faz parte do pacote model. Note que ela não é anotada.

Listagem 7.1

```
public class Calculadora {

	public double calculaMedia (double...notas) {

		double m = 0;

		for (Double d : notas) {

			m += d;

		}

		return m / notas.length;

	}

}
```

7.2 Crie uma classe que servirá para abrigar um método produtor, responsável por devolver uma instância de Calculadora. Ambas a classe e o método são anotadas com Configuration e Bean, respectivamente. Veja a Listagem 7.2. O nome da classe não tem nada de especial, pode ser qualquer um. Coloque-a no pacote principal da aplicação.

Listagem 7.2

```
@Configuration
public class AppConfig {
	@Bean
	public Calculadora getCalcudora() {
		return new Calculadora();
	}
}
```

7.3 Ajuste a classe Aluno para que cada um possua duas notas. Veja a Listagem 7.3.

Listagem 7.3

```
@Entity
public class Aluno implements Serializable{
	private static final long serialVersionUID = 1L;
	@Id
	@GeneratedValue (strategy = GenerationType.IDENTITY)
	private Long id;
	private String nome;
	private Double nota1;
	private Double nota2;
	private Double mediaFinal;
	//getters/setters

}
```

7.4 Ajuste o formulário para permitir o cadastro de duas notas por aluno bem como a exibição da média final de cada um. Veja a Listagem 7.4.

Listagem 7.4

```
<!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">
				<form class="form-inline" method="post" th:object="$
{aluno}" th:action="@{/alunos}" style="margin: 20px 0">
					<div class="form-group">
					<input type="text" class="form-control"
placeholder="Nome" th:field="*{nome}"/>
					<input type="text" class="form-control"
placeholder="Nota 1" th:field="*{nota1}"/>
					<input type="text" class="form-control"
placeholder="Nota 2" th:field="*{nota2}"/>
					<button type="submit" class="btn btn-
primary">Adicionar</button>
					</div>
				</form>
				<table class="table">
					<thead>
						<tr>
							<th>Nome</th>
							<th>Média</th>
						</tr>
					</thead>
					<tbody>
						<tr th:each="aluno : ${alunos}">
							<td th:text="${aluno.nome}"></td>
							<td th:text="${aluno.mediaFinal}"></
td>
						</tr>
					</tbody>
				</table>
			</div>
		</div>
	</body>
</html>
```

7.5 Injete a dependência de Calculadora no Service, como mostra a Listagem 7.5.

Listagem 7.5

```
@Service
public class AlunosService {
	@Autowired
	private AlunosRepository alunosRepo;

	@Autowired
	private Calculadora calculadora;

	public void salvar (Aluno aluno) {
		alunosRepo.save(aluno);
	}

	public List <Aluno> listarTodos (){
		List <Aluno> alunos =  alunosRepo.findAll();
		for (Aluno aluno : alunos)

		aluno.setMediaFinal(calculadora.calculaMedia(aluno.getNota1(),
							aluno.getNota2()));
		return alunos;
	}
}
```

Passo 8 (Controlando acesso por meio de uma página de login) Agora iremos configurar uma página para controle de login. A fim de cadastrar e listar alunos, o usuário deverá fazer login antes.

8.1 Crie a página de login da Listagem 8.1. Esse conteúdo deve ser colocado num arquivo chamado login.html armazenado no diretório resources/templates.

Listagem 8.1

```
<!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">Login</h1>

			</div>

			<div class="panel-body">

				<form class="form-inline" method="post" th:object="${usuario}"
th:action="@{/fazerLogin}" style="margin: 20px 0">

					<div class="form-group">

						<input required type="text" class="form-control"
placeholder="Usuário" th:field="*{login}"/>

						<input required type="password" class="form-
control" placeholder="Senha" th:field="*{senha}"/>

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

					</div>

				</form>

			</div>

		</div>

	</body>

</html>
```

8.2 No pacote model, crie a classe da Listagem 8.2. Ela representa usuários do sistema.

Listagem 8.2

```
@Entity
public class Usuario {
	@Id
	@GeneratedValue (strategy = GenerationType.IDENTITY)
	private Long id;
	private String login;
	private String senha;
	public String getLogin() {
		return login;
	}
	public void setLogin(String login) {
		this.login = login;
	}
	public String getSenha() {
		return senha;
	}
	public void setSenha(String senha) {
		this.senha = senha;
	}
}
```

8.3 Precisaremos fazer uma consulta no banco para verificar a existência de um usuário cujos login e senha sejam iguais aos digitados no formulário. Usando o Spring Data JPA, basta criar a interface da Listagem 8.3 exatamente como fizemos anteriormente, porém adicionando o método de interesse. Veja a padronização pelo nome. A query a ser executada será inferida pelo mecanismo de persistência. A interface deve, evidentemente, ser parte do pacote repository.

Listagem 8.3

```
public interface UsuarioRepository extends JpaRepository<Usuario, Long>{


	public Usuario findOneByLoginAndSenha (String login, String senha);


}
```

8.4 Agora vamos criar a classe de serviço para as atividades de Login. Ela depende de um UsuarioRepository que será injetado pelo container (o Spring, lembra?) e oferece um método para verificar a existência do usuário cujos dados foram informados no formulário. Veja a Listagem 8.4.

Listagem 8.4

```
@Service
public class LoginService {
	@Autowired
	UsuarioRepository usuarioRepo;
	public boolean logar (Usuario usuario) {
		return usuarioRepo.findOneByLoginAndSenha(usuario.getLogin(),
							usuario.getSenha()) != null;
	}
}
```

8.5 A classe LoginController é exibida na Listagem 8.5. Ela deve ser colocada no pacote de controllers. Note que o método login lida com dois padrões de URL diferentes. O usuário pode acessá-la acessando a raiz da aplicação ou digitando o texto login: localhost:sua_porta ou localhost:sua_porta/login devem direcionar para esse mesmo método. Além disso, o método fazerLogin é utilizado no action do formulário da tela de login.

Listagem 8.5

```java
@Controller
public class LoginController {
	@Autowired
	private LoginService loginService;
	@GetMapping (value = {"/login", "/"})
	public ModelAndView login () {
		ModelAndView mv = new ModelAndView ("login");
		mv.addObject(new Usuario());
		return mv;
	}
	@PostMapping ("/fazerLogin")
	public String fazerLogin (Usuario usuario) {
		if (loginService.logar(usuario)) {
			return "redirect:alunos";
		}
		else {
			return "login";
		}
	}
}
```

8.6 Insira um novo usuário no H2 editando o arquivo import.sql. Veja a Listagem 8.6.

Listagem 8.6

```
insert into aluno (id, nome, nota1, nota2) values (1, 'Ana', 2, 1);

insert into aluno (id, nome, nota1, nota2) values (2, 'Maria', 3, 10);

insert into aluno (id, nome, nota1, nota2) values (3, 'Ricardo', 7, 6);

--adicione um usuário

insert into usuario (id, login, senha) values (1, 'admin', 'admin')
```

Agora coloque o sistema em execução e certifique-se de que é possível listar os alunos digitando usuário e senha iguais a admin.

Passo 9 (Interceptando uma requisição) Embora tenhamos adicionado uma página de login ao sistema, nada impede que o usuário tente acessar recursos quaisquer (como uma outra página) do sistema digitando seu endereço diretamente no navegador. De fato isso é possível: faça o teste e acesse localhost:sua_porta/alunos e verifique que é possível listar os alunos sem ter de passar pela página de login. Iremos, agora, utilizar um recurso do Spring conhecido como **Interceptor**. Ele nos permitirá interceptar uma requisição e realizar algum pré-processamento antes de ela ser atendida de fato. Ou seja, é muito parecido com os filtros de JSP & Servlets. Neste caso, antes de entregar ao usuário um recurso que ele tenha solicitado, iremos verificar se ele já realizou login. Para isso usaremos também o recurso conhecido como **sessão**, aquele mesmo que conhecemos nas aulas de JSP & Servlets.

9.1 Crie a classe da Listagem 9.1. Note que utilizando a sessão para verificar se o usuário ainda não fez login. Caso não tenha feito, direcionamos a requisição para a página de login. Caso contrário, deixamos a requisição passar e chegar no controller, que irá interagir com o service para verificar se o usuário existe na base, por meio do repository.

Listagem 9.1

```
public class LoginInterceptor extends HandlerInterceptorAdapter{

	@Override

	public boolean preHandle(HttpServletRequest request,

	    HttpServletResponse response, Object handler)

	            throws Exception {

	        //pega a sessão

	        HttpSession session = request.getSession();

	        //se ainda não logou, manda para a página de login

	        if (session.getAttribute("usuarioLogado") == null)

	            response.sendRedirect("login");

	        //se já logou, deixa a requisição passar e chegar no

	                controller

	        return true;

	}

}
```

9.2 Agora iremos adaptar o LoginController (mais especificamente seu método fazerLogin) de modo que o Spring injete a sessão por meio do parâmetro do método. Isso permitirá que coloquemos o usuário na sessão caso ele seja encontrado na base. Veja a Listagem 9.2.

Listagem 9.2

```
@PostMapping ("/fazerLogin")
public String fazerLogin (HttpServletRequest request,
                                          Usuario usuario) {
     if (loginService.logar(usuario)) {
          request.getSession().setAttribute("usuarioLogado",
                                             usuario);
          return "redirect:alunos";
     }
     else {
          return "login";
     }
}
```

9.3 A fim de informar ao Spring sobre a existência do novo Interceptor, iremos utilizar a classe AppConfig, a mesma que utilizamos para injetar a Calculadora por meio de um método. Note que ao fazer isso também informamos ao Spring que requisições pela página de login ou pelo método de login devem passar direto, caso contrário, estaríamos dizendo que um usuário não pode fazer login por não ter feito login. Sempre que o usuário tentasse realizar login, estaríamos direcionando sua requisição indefinidamente para a página de login, sem nunca deixar passar. Veja a Listagem 9.3. Note que é necessário alterar sua assinatura, fazendo com que implemente a interface **WebMvcConfigurer**.

Listagem 9.3

```
@Configuration
public class AppConfig implements WebMvcConfigurer {
	@Bean
	public Calculadora getCalcudora() {
		return new Calculadora();
	}
	@Override
	public void addInterceptors(InterceptorRegistry registry) {
	    registry.addInterceptor(
	            new LoginInterceptor()).
	    addPathPatterns("/**").
	    excludePathPatterns("/login", "/", "/fazerLogin");
	}
}
```

Bibliografia

[1] Evans, E. **Domain Driven Design**. 1st ed. Addison-Wesley Professional, 2003.