Seção 07 - Resumo

Projeção de consultas com Spring Data

Quando realizamos uma consulta com JPA temos basicamente dois possíveis retornos, um é objeto que está sendo consultado, como exemplo, o objeto `Agendamento`:

```
@Query("select a from Agendamento a")
List<Agendamento> findTodos();
```

Outro tipo seria apenas um único valor pertencente ao objeto `Agendamento`, como a data da consulta:

```
@Query("select a.dataConsulta from Agendamento a where a.id = :id")
Agendamento findDataConsultaById(Long id);
```

Mas algumas vezes precisamos manipular o retorno da consulta, mesclando valores que não pertencem ou não são propriamente dito, pertencentes ao objeto da consulta, nesse caso, ao objeto `Agendamento`. Outras vezes você não precisa que todos os atributos do objeto sejam retornados e assim, seria interessante projetar o resultado com apenas, vamos supor, dois atributos:

```
@Query("select a.dataConsulta, a.horario.horaMinuto from Agendamento a where
a.id = :id")
Agendamento findDataConsultaById(Long id);
```

Analisando a consulta acima, seria impossível que ela fosse retornada, porque quando o Hibernate fosse realizar a conversão, a partir objeto obtido como resposta da tabela de `agendamentos`, para o objeto `Agendamento`, um exceção seria lançada. A exceção é lançada porque `Agendamento` não possui um atributo `String` para receber a `horaMinuto`. A classe `Agendamento` possui um objeto `Horario`.

Então, seria necessário somente alterar o retorno da consulta, trocando `a.horario.horaMinuto` para `a.horario`? Não, também teríamos uma exceção nesse caso. Por conta disso, precisamos de um objeto que receba exatamente o que está sendo retornado.

No Spring Data podemos criar esse objeto através de uma interface de projeção, que é algo bastante simples.

Vejamos a consulta apresentada no curso para retorna o histórico de agendamentos para pacientes:

```
@Query("select a.id as id,"
                    + "a.paciente as paciente,"
                    + "CONCAT(a.dataConsulta, ' ', a.horario.horaMinuto) as
dataConsulta,"
                    + "a.medico as medico,"
                    + "a.especialidade as especialidade "
             + "from Agendamento a "
             + "where a.paciente.usuario.email like :email")
Page<HistoricoPaciente> findHistoricoByPacienteEmail(String email, Pageable
pageable);
```

Esta consulta precisa ser projetada porque vamos retornar em um único atributo do tipo `String` a data da consulta concatenada ao horário da consulta, um tipo de atributo inexistente na classe `Agendamento`.

Agora, na interface de projeção representamos cada um dos atributos com os nomes dos alias usados na JPQL a partir de métodos get:

```
public interface HistoricoPaciente {
       Long getId();
       Paciente getPaciente();
       String getDataConsulta();
       Medico getMedico();
       Especialidade getEspecialidade();
}
```

Observe o retorno do método de consulta e veja que ele não é mais um `Agendamento` e sim um `HistoricoPaciente`, que é o tipo referente a interface de projeção.

Acesso a recursos do Thymeleaf no JavaScript

Quando usamos JavaScript nas páginas HTML, as vezes precisamos acessar alguns objetos que estão no contexto da aplicação.

O Thymeleaf fornece alguns recursos para este objetivo e entre eles, um recurso para reproduzir dentro do JavaScript certos componentes do

Thymeleaf, que são costumeiramente usados nas tags do HTML. Um desses recursos é o *Natural JavaScript* que pode ser acessado via JavaScript Inline. Observe o seguinte exemplo a seguir:

```
<script th:inline="javascript" th:fragment="inlinescript">
// buscar o médico pela especialidade
/*[# th:if="${agendamento.medico != null}" ]*/
	$( document ).ready(function() {
		var id = [[${agendamento.medico.id}]];
		var titulo =
[(${#strings.concat("'",agendamento.especialidade.titulo,"'")})];
		$.get( "/medicos/especialidade/titulo/" + titulo, function( result
) {
			$.each(result, function (k, v) {
				$("#medicos").append(
					'<div class="custom-control custom-radio">'
				+	'<input class="custom-control-input"
type="radio" id="customRadio'+ k +'" name="medico.id" value="'+ v.id +'"
required '+ (v.id == id ? "checked" : "")+' />'
				+	'<label class="custom-control-label"
for="customRadio'+ k +'">'+ v.nome +'</label>'
					+'</div>'
				);
			});
		});
	});
/*[/]*/
</script>
```

Veja que no topo do código javascript temos uma expressão representada por

```
/*[# ]*/
```

Essa é uma expressão do tipo *Natural JavaScript* e com ela é possível acessar o componente `th:if` para testar se o objeto `medico` é diferente de `null`:

```
/*[# th:if="${agendamento.medico != null}" ]*/
```

Outros componentes pode ser usados com esse recurso, como `th:each`, `th:text` e etc.

Criando exceções personalizadas

Algo bastante simples ao usar o Spring é lançar e capturar exceções para então tratá-las na classe do tipo `@ControllerAdvice`. Mas não são apenas

as exceções padrões das bibliotecas pode são possíveis de se trabalhar com tal recurso, mas também, exceções personalizadas, que são aquelas exceções criadas pelo programador.

Para criar uma exceção personalizada não há mistério algum, esse é um recurso que aprendemos quando damos os primeiros passos na linguagem Java. Então, basta reproduzi-lo da mesma forma, como no exemplo a seguir:

```
public class AcessoNegadoException extends RuntimeException {
	public AcessoNegadoException(String message) {
		super(message);
	}
}
```

Adicionamos uma nova classe ao projeto a qual estende `RuntimeException`. Assim, tornamos essa classe um exceção via conceito de herança. Vamos sobrescrever o método construtor de `RuntimeException` para termos acesso ao objeto de mensagem da exceção, assim, é possível alterar a mensagem padrão da exceção para aquela que você desejar.

O próximo passo seria lançar a exceção quando algo de errado acontecer. Para isso, vamos usar o método `orElseThrow()` da classe `Optional`. E como parâmetro deste método passamos uma expressão lambda com a instancia da exceção criada mais uma mensagem personalizada:

```
@Transactional(readOnly = true)
public Agendamento buscarPorIdEUsuario(Long id, String email) {
	return repository
			.findByIdAndPacienteOrMedicoEmail(id, email)
			.orElseThrow(() -> new AcessoNegadoException("Acesso negado
ao usuário: " + email));
}
```

Quando o método `findByIdAndPacienteOrMedicoEmail()` tiver um retorno nulo, por nenhum resultado ter sido localizado na base de dados, a exceção `AcessoNegadoException` será lançada. Para capturar e tratar essa exceção, vamos trabalhar com a classe `ExceptionController`, onde será adicionado o método de captura e tratamento:

```
@ExceptionHandler(AcessoNegadoException.class)
public ModelAndView acessoNegadoException(AcessoNegadoException ex) {
	ModelAndView model = new ModelAndView("error");
	model.addObject("status", 403);
```

```
        model.addObject("error", "Operação não pode ser realizada.");
        model.addObject("message", ex.getMessage());
        return model;
    }
```

Dessa forma, podemos tratar a exceção e envia-la para a página de erro contendo mensagens personalizadas sobre o erro.

Regras de Autorização via Anotações

Durante o curso todas as regras de autorizações de acessos baseadas em perfis foram cadastradas na classe `SecurityConfig` via métodos `hasAuthority()` e `hasAnyAuthority()`.

Agora, será apresentada outra forma de se trabalhar com esse recurso baseado em autorização via perfis. O Spring Security fornece uma anotação que pode ser incluída sobre a assinatura de métodos. Essa anotação é a `@PreAuthorize`. Como parâmetro devemos passar uma String com o nome do método de autorização `hasAuthority()` ou `hasAnyAuthority()` contendo no método o perfil desejado. Veja um exemplo no método `agendarConsulta()` que possui a URI `/agendamentos/agendar`:

```
@PreAuthorize("hasAnyAuthority('PACIENTE', 'MEDICO')")
@GetMapping("/agendar")
public String agendarConsulta(Agendamento agendamento) {
        return "agendamento/cadastro";
}
```

Mas para que esse recurso funcione é preciso habilita-lo na classe `SecurityConfig` via anotação `@EnableGlobalMethodSecurity`:

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

}
```

Caso tenha preferencia pelos métodos `hasRole()` e `hasAnyRole()` a anotação a ser adicionada sobre os métodos é a `@Secured`. E como parâmetro basta informar o tipo de perfil `@Secured("ROLE_MEDICO")`.

Já na classe de configuração a anotação `@EnableGlobalMethodSecurity` deve ser configurada com o atributo `securedEnabled = true`:

```
@EnableGlobalMethodSecurity(securedEnabled = true)
```

Referencias

- Spring Data JPA Projections - https://docs.spring.io/spring-data/jpa/docs/2.1.4.RELEASE/reference/html/#projections
- Thymeleaf Textual template modes - https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#textual-template-modes
- Thymeleaf Natural JavaScript and CSS templates - https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#natural-javascript-and-css-templates
- Spring Method Security - https://docs.spring.io/spring-security/site/docs/5.1.5.RELEASE/reference/htmlsingle/#jc-method

Código Fonte

Caso tenha tido algum tipo de dificuldade para acompanhar a desenvolvimento do código fonte até o final desta seção, ele está disponível na área de arquivo para download.