**SPRING MVC**

**CACHE**

**Para habilitar o cache no método do controler a exibir a view colocar a anotação:**

**@Cacheable(value= “nomePagina”)**

**E na classe AppWebConfiguration em cima do nome da classe adicionar a anotação:**

**@EnableCaching**

**E criar o método CacheManager conforme abaixo:**

**@Bean**

**public CacheManager cacheManager() {**

**CacheBuilder<Object, Object> builder =**

**CacheBuilder.newBuilder()**

**.maximumSize(100)**

**.expireAfterAccess(50, TimeUnit.MINUTES);**

**GuavaCacheManager manager =**

**new GuavaCacheManager();**

**manager.setCacheBuilder(builder);**

**return manager;**

**}**

**Quando o cache é habilitado ele não vai mais ao banco fazer a consulta e usa a lista que esta em memória, para não haver defasagem com o banco de dados devemos ao gravar um novo elemento nesta classe onde esta sendo habilitado o cache em cima do método gravar() adicionar:**

**@CacheEvict(value= “nomePagina”, allEntries = true)**

**Assim quando e salvo um novo elemento o cache e destruído e quando for feita uma nova requisição ele fara a consulta no banco novamente mas somente uma vez ate ser salvo outro elemento no banco.**

**CRIANDO SERVICO REST**

Um recurso interessante nas atuais aplicações é a possibilidade de integração entre as mesmas. Em muitas aplicações encontramos o recurso de permitir a leitura de dados dos produtos ou serviços em formatos diferentes do HTML, por exemplo, o nome e o preço de um item.

Geralmente, o formato usado é o **JSON**, mas encontraremos o **XML**, além de outros. Usaremos este recurso em nossa aplicação. O usuário ao acessar a página de produtos, passando um determinado id no caminho /produtos/{id}, deverá receber os dados do produto no formato JSON.

Como se trata de uma representação dos detalhes de um determinado produto, duplicaremos o método detalhe da classe ProdutosController.

@RequestMapping("/detalhe/{id}")

public ModelAndView detalhe(@PathVariable("id") Integer id){

ModelAndView modelAndView = new ModelAndView("/produtos/detalhe");

Produto produto = produtoDao.find(id);

modelAndView.addObject("produto", produto);

return modelAndView;

}

Não precisaremos mais do ModelAndView, então, removeremos todas as linhas referentes a este objeto. Iremos retornar o resultado da chamada do método produtoDao.find(id). O caminho que o método atende ficará apenas /{id} e chamaremos este método de detalheJSON. Com estas modificações, teremos:

@RequestMapping("/{id}")

public Produto detalheJSON(@PathVariable("id") Integer id){

return produtoDao.find(id);

}

Note que o retorno do método em sua assinatura mudou, não estamos mais retornando um objeto ModelAndView, mas sim um Produto. Podemos testar e ver que algo estranho acontece. Ao reiniciar o servidor e tentar acessar localhost:8080/casadocodigo/produtos/5 um erro 404 é exibido.



O *Spring* deveria retornar o JSON do livro, no entanto, ele está buscando uma view com o mesmo id do livro. Não é o que queremos. Para que o *Spring* consiga responder a requisição da forma que queremos neste caso, usaremos a anotação ResponseBody no método detalheJSON da classe ProdutosController.

@RequestMapping("/{id}")

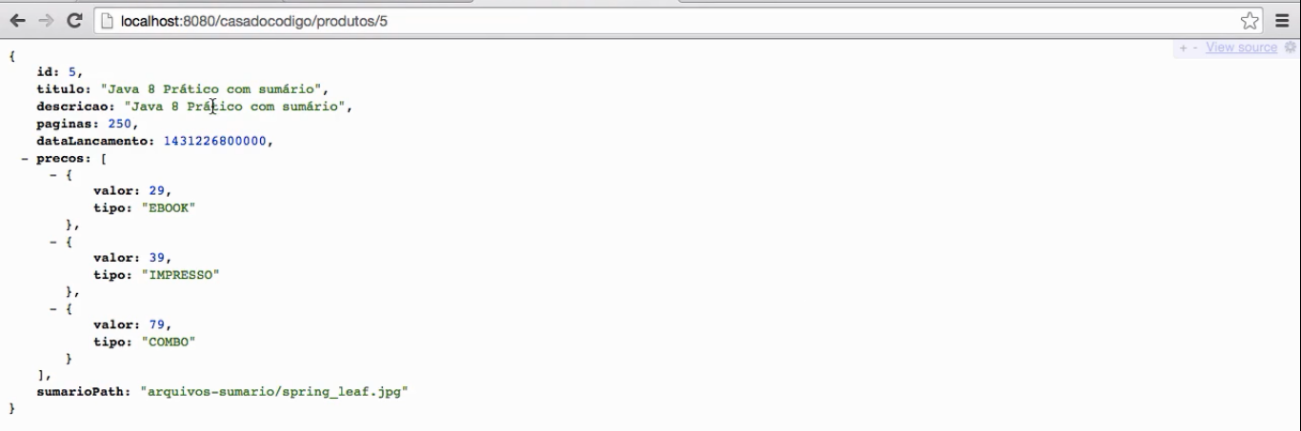
@ResponseBody

public Produto detalheJSON(@PathVariable("id") Integer id){

return produtoDao.find(id);

}

Dessa forma, estamos dizendo para o *Spring* que o retorno daquele método será o corpo da resposta da requisição. Se acessarmos novamente localhost:8080/casadocodigo/produtos/5, teremos:



Funcionou! Temos um livro no formato JSON como resposta da requisição!

É importante notar que só funcionou, porque o Jackson está configurado no projeto. Se não tivéssemos configurado a biblioteca anteriormente no primeiro módulo do curso, o recurso não teria funcionado. Teríamos problema também, caso mais de uma biblioteca estivesse configurada para transformar objetos em JSON, no projeto. Haveria conflito e teríamos que realizar novas configurações para obter o mesmo resultado.

###Content Negotiation

Prover integrações da maneira que fizemos é interessante, mas existe um grave problema. Imagine que todos os métodos da classe ProdutosController terão um equivalente que possibilita a comunicação de informações via JSON, teríamos códigos duplicados por toda a aplicação. Duplicação de código não é bom.

Pensando nisto, desenvolveu-se no mercado um padrão que provê uma negociação do conteúdo retornado pela aplicação. Através da técnica chamada de **Content Negotiation** é possível que uma mesma URL retorne as informações em formatos diferentes. Exemplo: acessar a URL localhost:8080/casadocodigo/produtos/5 traria como resposta o HTML da página de detalhes daquele produto, enquanto acessar localhost:8080/casadocodigo/produtos/5.json retornaria o JSON que representa aquele produto.

Perceba que a URL não muda, mas sua **extensão**, sim. Faremos uso dessa técnica em nossa aplicação e já começaremos as atualizações em nosso código removendo o método detalheJSON.

Abra a classe AppWebConfiguration e crie um novo método chamado contentNegotiationViewResolver que irá retornar um objeto do tipo ViewResolver. Este método precisa receber como parâmetro um objeto do tipo ContentNegotiationManager, que iremos nomear como manager.

O manager que estamos recebendo por parâmetro será o responsável pela decisão de qual view será utilizada. Mas que opções o manager terá? Nenhuma. Então, criaremos uma lista que chamaremos de viewResolvers e nela adicionaremos as duas opções que precisaremos. A primeira é em HTML promovida pelo método internalResourceViewResolver da mesma classe. A outra será de uma classe que ainda não criamos, chamada JsonViewResolver. Até aqui, teremos:

@Bean

public ViewResolver contentNegotiationViewResolver(ContentNegotiationManager manager){

List<ViewResolver> viewResolvers = new ArrayList<>();

viewResolvers.add(internalResourceViewResolver());

viewResolvers.add(new JsonViewResolver());

ContentNegotiatingViewResolver resolver = new ContentNegotiatingViewResolver();

return resolver;

}

Por último, precisamos definir que o objeto resolver tenha a lista de Resolvers e o manager que escolherá entre os viewsResolvers disponíveis. Fazemos isso desta forma:

resolver.setViewResolvers(viewResolvers);

resolver.setContentNegotiationManager(manager);

Então, o método completo ficará da seguinte forma:

@Bean

public ViewResolver contentNegotiationViewResolver(ContentNegotiationManager manager){

List<ViewResolver> viewResolvers = new ArrayList<>();

viewResolvers.add(internalResourceViewResolver());

viewResolvers.add(new JsonViewResolver());

ContentNegotiatingViewResolver resolver = new ContentNegotiatingViewResolver();

resolver.setViewResolvers(viewResolvers);

resolver.setContentNegotiationManager(manager);

return resolver;

}

Ao pedirmos para o **Eclipse** criar a classe JsonViewResolver - ou quando criamos manualmente - e fazemos esta implementar a interface ViewResolver, teremos o seguinte código em mãos:

public class JsonViewResolver implements ViewResolver {

@Override

public View resolveViewName(String arg0, Locale arg1) throws Exception {

// TODO Auto-generated method stub

return null;

}

}

Iremos renomear os argumentos do método: de arg0 para viewName, e em seguida, arg1 para locale. Assim, o código ficará mais legível.

Agora precisamos fazer com que este método consiga retornar como view o JSON que representa nossos produtos. O próprio Spring tem uma classe que é proveniente da integração com a biblioteca **Jackson** e se chama MappingJackson2JsonView e tudo que precisamos fazer é retornar um objeto desta classe, da seguinte forma:

public class JsonViewResolver implements ViewResolver {

@Override

public View resolveViewName(String arg0, Locale arg1) throws Exception {

MappingJackson2JsonView jsonView = new MappingJackson2JsonView();

jsonView.setPrettyPrint(true);

return jsonView;

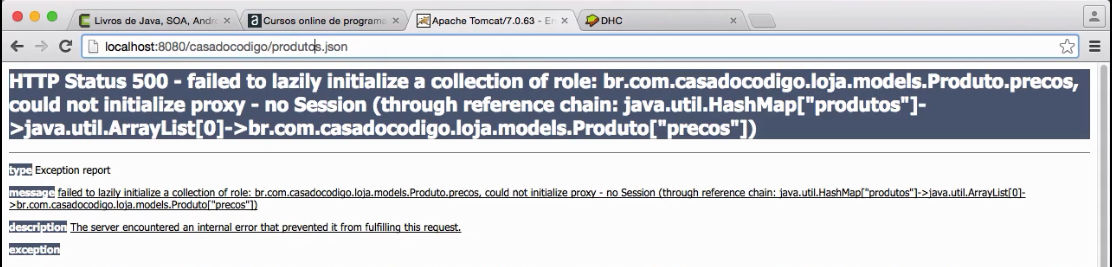
}

}

O trecho jsonView.setPrettyPrint(true) foi adicionado para que o Jackson mantenha uma formatação amigável ao retornar o JSON dos nossos produtos. Pronto! Já funciona!

Façamos alguns testes agora. Reinicie o servidor e tente acessar a página de detalhe de algum produto como por exemplo em: localhost:8080/casadocodigo/produtos/5 e verá a página em HTML normal. E se adicionarmos .json na mesma URL teremos o produto representado no formato JSON.

Teste também fazer requisições com outras ferramentas além da barra de endereço do navegador, além de usar a extensão do Chrome DHC que vimos no primeiro módulo deste curso ou o HTTPRequester também apresentado junto ao DHC.

Apesar de não aparentar, nossa aplicação inteira já pode ser lida em dois formatos, JSON e HTML. Experimente fazer o mesmo com a listagem dos produtos em localhost:8080/casadocodigo/produtos e localhost:8080/casadocodigo/produtos.json. Ao tentarmos o .json teremos um error 500:

O problema acontece do **Lazy Load** utilizado pelo Spring para carregar os dados do banco de dados. Resolveremos isso mais adiante. Por enquanto, pratique com os exercícios.

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1) Para exportar o produto como JSON, XML, etc, crie a classe **JsonViewResolver**, no pacote br.com.casadocodigo.loja.conf, que implemente a interface ViewResolver para atender os pedidos de dados em JSON:

public class JsonViewResolver implements ViewResolver {

@Override

public View resolveViewName(String viewName,

Locale locale) throws Exception {

MappingJackson2JsonView jsonView = new MappingJackson2JsonView();

jsonView.setPrettyPrint(true);

return jsonView;

}

}

2) Para agrupar todos os viewResolver, crie o método contentNegotiationViewResolver na sua classe AppWebConfiguration, conforme abaixo:

@Bean

public ViewResolver contentNegotiationViewResolver(ContentNegotiationManager manager) {

List<ViewResolver> viewResolvers = new ArrayList<>();

viewResolvers.add(internalResourceViewResolver());

viewResolvers.add(new JsonViewResolver());

ContentNegotiatingViewResolver resolver =

new ContentNegotiatingViewResolver();

resolver.setViewResolvers(viewResolvers);

resolver.setContentNegotiationManager(manager);

return resolver;

}

Agora você exibir um produto em JSON apenas adicionando a extensão .json no final da URL.