03

# ContentNegotiation

**Transcrição**

###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:

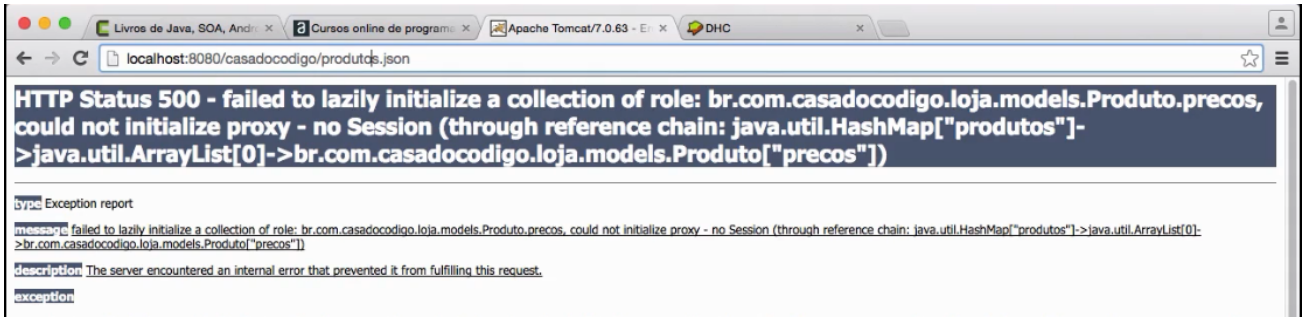
```
public class JsonViewResolver implements ViewResolver {

    @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.