07

Testando a aplicação

Testando a aplicação

Imagine que agora o nosso gestor quer que ao entrar na parte administrativa do nosso sistema, este consiga de alguma forma ver algo como um relatório que exibem o valor total de todos os produtos separados por tipo de preço (*Ebook*, *Impresso* e *Combo*).

O primeiro passo para conseguirmos fazer com que isso seja possível é pedir ao banco de dados a soma do valor de todos os produtos por tipo de preço. Na classe `ProdutoDAO` criaremos um novo método chamado `somaPrecosPorTipo` que retornará um objeto do tipo `BigDecimal` .

Este método deve receber o tipo de preço por parametro, realizar a consulta ao banco de dados e retornar o resultado, isso através do objeto `manager` , como já feito anteriormente.

```
public BigDecimal somaPrecosPorTipo(TipoPreco tipoPreco){
    TypedQuery<BigDecimal> query = manager.createQuery("select sum(preco.valor) from Produto p join
    query.setParameter("tipoPreco", tipoPreco);
    return query.getSingleResult();
}
```

Mas quem garante que este código funciona e que o resultado do mesmo esteja correto? No mínimo, teríamos que criar uma nova página de relatórios e verificar manualmente o resultado e assim validar se o código funciona como esperado ou não.

Para validar o funcionamento de determinado código, precisamos de testes. Existem duas formas básicas de se testar código, uma delas foi comentarada anteriormente e é conhecida como teste manual, onde precisamos de alguém para verificar manualmente os resultados gerados pelo código em algum lugar.

A segunda forma, mais prática é que sempre funcionará por meio de testes automatizados. Falando de forma mais objetiva, escreveremos um código que testa se o nosso código funciona corretamente usando a comparação de resultados esperado.

Observação: Na Alura, há diversos cursos de testes em diversas tecnologias, basta usar a busca e encontrará vários destes cursos ou basta clicar aqui para ver a lista de cursos de testes (https://cursos.alura.com.br/search?q=testes)

Projetos **Maven** sempre contam com um **Source Folder** especifico para testes, localizado em: `src/tests/java` . Para criação dos testes para nossa aplicação, dentro deste *Source Folder* criaremos as classes de testes usando os mesmos pacotes das classes de *produção*.

Criaremos então a classe `ProdutoDAOTest` que conterá os testes relacionados a classe `ProdutoDAO` . Esta classe estará no *source folder* de testes e terá o mesmo pacote da `ProdutoDAO` : `br.com.casadocodigo.loja.daos` .

Esta classe terá um método que testa se a soma dos preços dos produtos encontrados no banco é igual a um valor esperado por nós. O método responsável por esta verificação se chamará `deveSomarTodosOsPrecosPorTipoLivro` e não terá retorno.

Este método usará a classe `ProdutoDAO` para salvar uma determinada quantidade de produtos que criaremos para cada tipo de preço. A criação dos produtos será feita com a ajuda de uma classe auxiliadora chamada `ProdutoBuilder` , o código desta classe se encontra abaixo. Copiaremos esta classe para o pacote `br.com.casadocodigo.loja.builders` dentro do *source folder tests*.

```
package br.com.casadocodigo.loja.builders;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;

import br.com.casadocodigo.loja.models.Preco;
import br.com.casadocodigo.loja.models.Produto;
import br.com.casadocodigo.loja.models.TipoPreco;

public class ProdutoBuilder {

    private List<Produto> produtos = new ArrayList<>();

    private ProdutoBuilder(Produto produto) {
        produtos.add(produto);

    }

    public static ProdutoBuilder newProduto(TipoPreco tipoPreco, BigDecimal valor) {
        Produto livro = create("livro 1", tipoPreco, valor);
        return new ProdutoBuilder(livro);
    }

    public static ProdutoBuilder newProduto() {
        Produto livro = create("livro 1", TipoPreco.COMBO, BigDecimal.TEN);
        return new ProdutoBuilder(livro);
    }

    private static Produto create(String nomeLivro, TipoPreco tipoPreco, BigDecimal valor) {
        Produto livro = new Produto();
        livro.setTitulo(nomeLivro);
        livro.setDataLancamento(Calendar.getInstance());
        livro.setPaginas(150);
        livro.setDescricao("Livro top sobre testes");
        Preco preco = new Preco();
        preco.setTipo(tipoPreco);
        preco.setValor(valor);
        livro.getPrecos().add(preco);
        return livro;
    }

    public ProdutoBuilder more(int number) {
        Produto base = produtos.get(0);
        Preco preco = base.getPrecos().get(0);
        for (int i = 0; i < number; i++) {
            produtos.add(create("Book " + i, preco.getTipo(), preco.getValor()));
        }
        return this;
    }

    public Produto buildOne() {
        return produtos.get(0);
    }

    public List<Produto> buildAll() {
```

```
        return produtos;
    }
}
```

Note que o há um método que se chama `newProduto` que recebe um objeto `TipoPreco` e o valor do produto em si. Outro método a ser utilizado é o método `more` que recebe um número que indica quandos produtos queremos criar e o método `buildAll` que nos retorna a lista de produtos criados.

Dessa forma podemos fazer algo como:

```
List<Produto> livrosImpressos = ProdutoBuilder.newProduto(TipoPreco.IMPRESSO, BigDecimal.TEN).more(
```

Assim teremos uma lista com três produtos do tipo impresso, com o preço 10 em mãos. Podemos fazer o mesmo processo para a criação dos produtos do tipo `EBOOK` da seguinte forma:

```
List<Produto> livrosEbook = ProdutoBuilder.newProduto(TipoPreco.EBOOK, BigDecimal.TEN).more(3).build
```

Agora, podemos usar um laço para percorrer cada uma das listas e salvar cada um dos produtos no banco de dados com o objeto da classe `ProdutoDAO`. Usando `streams` do Java 8, teremos o seguinte resultado.

```
public void deveSomarTodosOsPrecosPorTipoLivro() {
    ProdutoDAO produtoDAO = new ProdutoDAO();

    List<Produto> livrosImpressos = ProdutoBuilder.newProduto(TipoPreco.IMPRESSO, BigDecimal.TEN).mo
    List<Produto> livrosEbook = ProdutoBuilder.newProduto(TipoPreco.EBOOK, BigDecimal.TEN).more(3).b

    livrosImpressos.stream().forEach(produtoDAO::gravar);
    livrosEbook.stream().forEach(produtoDAO::gravar);
}
```

Observação: Caso não tenha entendido bem o `forEach` feito através do `stream`, saiba que estes são recursos do Java 8 e que você pode aprende-los no curso de Java 8: Tire proveito dos novos recursos da linguagem (https://cursos.alura.com.br/course/java8-lambdas) disponível aqui na Alura.

Agora, para que possamos realmente testar a aplicação, precisamos adicionar uma nova dependência no arquivo `pom.xml`. Esta se trata do *framework* de testes **JUnit**.

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

E ao final do métoto usar a classe `Assert` do *JUnit* para verificar se o valor retornado do banco de dados é igual ao valor da soma dos produtos que temos no código. Primeiro precisamos recuperar este valor, claro!

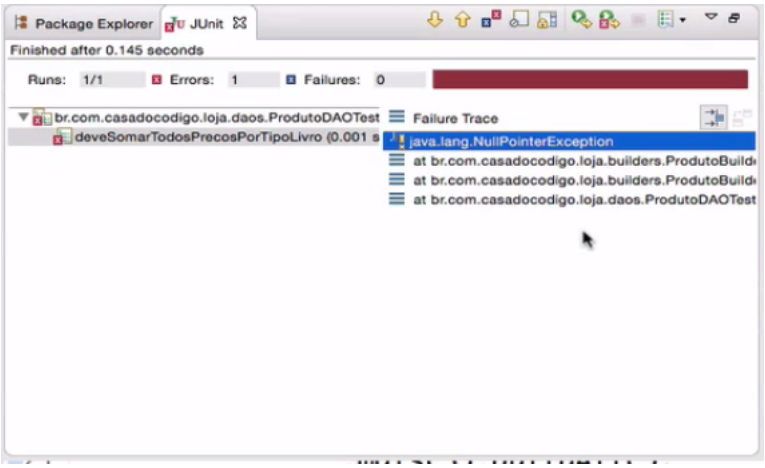
```
BigDecimal valor = produtoDAO.somaPrecosPorTipo(TipoPreco.EBOOK);
Assert.assertEquals(new BigDecimal(40).setScale(2), valor);
```

Note que estamos usando o método `somaPrecosPorTipo` da classe `ProdutoDAO` para recuperar o valor da soma dos preços dos produtos do tipo `EBOOK`. Comparando com o valor que esperamos de acordo com a lista de produtos que foi criada a partir do `ProdutoBuilder`. O `setScale(2)` simplesmente adiciona duas casas decimais ao valor `40`. Assim teremos:

```
@Test
public void deveSomarTodosOsPrecosPorTipoLivro() {
    ProdutoDAO produtoDAO = new ProdutoDAO();

    List<Produto> livrosImpressos = ProdutoBuilder.newProduto(TipoPreco.IMPRESSO, BigDecimal.TEN).mo
    List<Produto> livrosEbook = ProdutoBuilder.newProduto(TipoPreco.EBOOK, BigDecimal.TEN).more(3).b

    livrosImpressos.stream().forEach(produtoDAO::gravar);
    livrosEbook.stream().forEach(produtoDAO::gravar);

    BigDecimal valor = produtoDAO.somaPrecosPorTipo(TipoPreco.EBOOK);
    Assert.assertEquals(new BigDecimal(40).setScale(2), valor);
}
```

Observação: Lembre-se de marcar o método com a anotação `@Test` para que o *JUnit* saiba que este método é um teste a ser executado.

Ao executarmos o códico com o *JUnit Test*, receberemos um erro mostrando que o teste falhou por causa de um `NullPointerException`.



Este erro acontece por que a classe `ProdutoDAO` usa o objeto `manager` para fazer consultas no banco de dados. Mas o objeto `manager` só é criado pelo *Spring* e dentro do contexto do *Spring*. Para solucionar o problema, teriamos que capturar o contexto do *Spring* de alguma forma, criar o objeto `manager` para só depois podermos realizar o teste. Complicado, certo? Vamos ver como faremos.

Para que o *JUnit* possa reconhecer o texto dos objetos do *Spring*, precisamos adicionar uma nova biblioteca em nosso projeto, sendo esta a *Spring Test* por meio da declaração da mesma no arquivo `pom.xml`

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>4.1.0.RELEASE</version>
```

```
    <scope>test</scope>
</dependency>
```

Mas apenas isso não é o suficiente. Precisamos dizer agora que o *JUnit* deverá carregar configurações do *Spring Test* para poder executar os testes e fazemos isso realizando duas configurações por anotação, são elas:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JPAConfiguration.class,ProdutoDAO.class})
```

A anotação `@RunWith` do próprio *JUnit* nos permite dizer que classe irá executar os testes encontrados na nossa suite de testes. Já a anotação `@ContextConfiguration` nos permite configurar quais são as classes de configurações para execução dos testes. Como estamos usando conexão com o banco de dados, precisamos da classe que configura a *JPA* e o `ProdutoDAO` neste caso.

Após este passo, podemos transformar o objeto `produtoDAO` em um atributo da classe `ProdutoDAOTest` , anota-lo com `@Autowired` e renomea-lo para `produtoDao` para que fique mais claro a mudança. Por último, precisamos que o método seja anotado com `@Transactional` porque o mesmo precisa de uma transação com o banco de dados para que tudo funcione corretamente. Assim teremos:

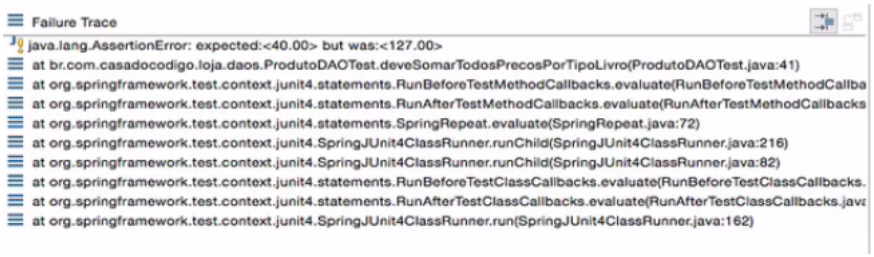
```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JPAConfiguration.class,ProdutoDAO.class})
public class ProdutoDAOTest {

    @Autowired
    private ProdutoDAO produtoDao;

    @Test
    @Transactional
    public void deveSomarTodosOsPrecosPorTipoLivro() {

        List<Produto> livrosImpressos = ProdutoBuilder.newProduto(TipoPreco.IMPRESSO, BigDecimal.TEN
        List<Produto> livrosEbook = ProdutoBuilder.newProduto(TipoPreco.EBOOK, BigDecimal.TEN).more

        livrosImpressos.stream().forEach(produtoDao::gravar);
        livrosEbook.stream().forEach(produtoDao::gravar);

        BigDecimal valor = produtoDao.somaPrecosPorTipo(TipoPreco.EBOOK);
        Assert.assertEquals(new BigDecimal(40).setScale(2), valor);
    }

}
```

Outros dois problemas que podem acontecer é o *Spring Test* não conseguir instanciar o objeto `precos` na classe `Produto` . Para garantir que não tenhamos tal problema, alteraremos a declaração deste objeto instanciando um objeto do tipo `ArrayList` , assim teremos na classe produto o seguinte:

```
@Entity
public class Produto {
    // [...]
```

```
    @ElementCollection
    private List<Preco> precos = new ArrayList<>();

    // [...]
}
```

Ao tentarmos executar o teste novamente, vemos que funciona, embora o mesmo não tenha passado. Nosso teste espera que o valor retornado seja 40, mas 127 foi o valor retornado do banco de dados.



Isto acontece porque já tinhamos os produtos cadastrados no banco de dados. Na soma dos valores, ele considera os produtos cadastrados no momento do teste e os que já estavam lá.

Isso pode se tornar ainda mais problemático quando o projeto for compartilhado para ser desenvolvido em equipe, um teste pode passar para um desenvolvedor e falhar para outro por causa das diferenças entre os bancos de dados de cada um. Como podemos resolver isso?

Podemos criar um banco de dados diferente. Este será usado somente para testes. Criaremos um novo banco de dados chamado `casadocodigo_test`. Usando os comandos: `mysql -uroot` para entrar no gerenciador de banco de dados e `create database casadocodigo_test` para criar o banco de dados.

Temos um novo banco de dados e um novo problema surge. Como faremos para que os testes usem o banco de dados destes e a aplicação naturalmente use o que não é para ser testado - também conhecido como banco em produção?

Podiamos trocar o banco na configuração da `JPA`, mas esta não é uma boa idéia já que precisaríamos ficar trocando a todo momento a configuração.

O que podemos fazer é dividir as configurações da aplicação por meio de **Profiles**, que é um recurso no qual podemos agrupar configurações para determinadas partes da aplicação. A anotação `@ActiveProfiles` nos ajuda com esta tarefa. Marcaremos então a classe `ProdutoDAOTest` com esta anotação passando o valor `test`.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JPAConfiguration.class,ProdutoDAO.class})
@ActiveProfiles("test")
public class ProdutoDAOTest {
    // restante do codigo
}
```

Prisaremos também criar uma nova classe de configuração para definir qual o banco de dados será usado para testes em nossa aplicação. A classe receberá o nome de `DataSourceConfigurationTest` e terá apenas um método chamado `dataSource`, que retornará um objeto `DataSource` que descreve os dados de acesso ao banco.

Este método precisa ser anotado com `@Bean` para que o *Spring* consiga manipulá-lo e com `@Profile("test")` para que o mesmo consiga relacionar os *Profiles* de testes da aplicação.

Esta classe deve estar no *Source Folder* de testes e no pacote `br.com.casadocodigo.loja.confs`

```
public class DataSourceConfigurationTest {

    @Bean
    @Profile("test")
    public DataSource dataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUrl("jdbc:mysql://localhost:3306/casadocodigo_test");
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUsername("root");
        dataSource.setPassword('');
        return dataSource;
    }

}
```

Esta configuração é a mesma da classe `JPAConfiguration`, muda apenas o nome do banco de dados. Para que os *Profiles* fiquem bem divididos, vamos fazer algumas melhorias na classe `JPAConfiguration`. Os passos serão descritos abaixo.

Veja como está o método `LocalContainerEntityManagerFactoryBean` da classe `JPAConfiguration`:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean factoryBean = new LocalContainerEntityManagerFactoryBean(
    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

    factoryBean.setJpaVendorAdapter(vendorAdapter);

    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setUsername("root");
    dataSource.setPassword("root");
    dataSource.setUrl("jdbc:mysql://localhost:3306/casadocodigo");
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");

    factoryBean.setDataSource(dataSource);


    Properties props = new Properties();
    props.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL5Dialect");
    props.setProperty("hibernate.show_sql", "true");
    props.setProperty("hibernate.hbm2ddl.auto", "update");

    factoryBean.setJpaProperties(props);

    factoryBean.setPackagesToScan("br.com.casadocodigo.loja.models");

    return factoryBean;
}
```

Este método está enorme, com muitas responsabilidades. Vamos refatorá-lo para separar algumas partes deste código em outros métodos. Primeiro vamos remover a criação do objeto `dataSource` para um outro método e recebê-lo por parâmetro no método `LocalContainerEntityManagerFactoryBean`.

O método `dataSource` terá:

```
@Bean
public DataSource dataSource(){
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setUsername("root");
    dataSource.setPassword("root");
    dataSource.setUrl("jdbc:mysql://localhost:3306/casadocodigo");
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    return dataSource;
}
```

E o método `LocalContainerEntityManagerFactoryBean` ficará mais simples:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {
    LocalContainerEntityManagerFactoryBean factoryBean = new LocalContainerEntityManagerFactoryBean(
    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

    factoryBean.setJpaVendorAdapter(vendorAdapter);


    factoryBean.setDataSource(dataSource);

    Properties props = new Properties();
    props.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL5Dialect");
    props.setProperty("hibernate.show_sql", "true");
    props.setProperty("hibernate.hbm2ddl.auto", "update");

    factoryBean.setJpaProperties(props);

    factoryBean.setPackagesToScan("br.com.casadocodigo.loja.models");

    return factoryBean;
}
```

Extrairemos também o trecho de código que configura as propriedades do **Hibernate** para um outro método chamado `AditionalProperties` .

```
private Properties aditionalProperties(){
    Properties props = new Properties();
    props.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL5Dialect");
    props.setProperty("hibernate.show_sql", "true");
    props.setProperty("hibernate.hbm2ddl.auto", "update");
    return props;
}
```

E refletir no método `LocalContainerEntityManagerFactoryBean` estas mudanças.

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {
    LocalContainerEntityManagerFactoryBean factoryBean = new LocalContainerEntityManagerFactoryBean
```

```
    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

    factoryBean.setJpaVendorAdapter(vendorAdapter);
    factoryBean.setDataSource(dataSource);

    Properties props = aditionalProperties();

    factoryBean.setJpaProperties(props);

    factoryBean.setPackagesToScan("br.com.casadocodigo.loja.models");

    return factoryBean;
}
```

Perceba que recebemos o `dataSouce` por parâmetro mas o `props` não. O `dataSource` precisa ser feito desta forma porque só assim o *Spring* conseguirá diferenciar este objeto do `dataSource` de testes. Agora podemos definir o `Profile` da classe `JPAConfiguration` com o valor `dev` .

```
@Bean
@Profile("test")
public DataSource dataSource(){
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setUsername("root");
    dataSource.setPassword("root");
    dataSource.setUrl("jdbc:mysql://localhost:3306/casadocodigo");
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    return dataSource;
}
```

A classe `JPAConfiguration` agora está mais organizada e mais simples. Veja o código desta classe por completo:

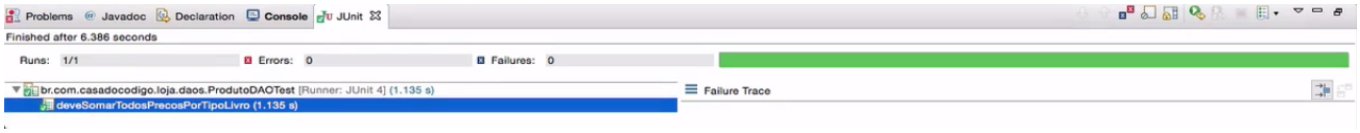
```
@EnableTransactionManagement
public class JPAConfiguration {

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {
        LocalContainerEntityManagerFactoryBean factoryBean = new LocalContainerEntityManagerFactoryB
        JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

        factoryBean.setJpaVendorAdapter(vendorAdapter);
        factoryBean.setDataSource(dataSource);

        Properties props = aditionalProperties();

        factoryBean.setJpaProperties(props);
        factoryBean.setPackagesToScan("br.com.casadocodigo.loja.models");

        return factoryBean;
    }

    @Bean
    @Profile("test")
    public DataSource dataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
```

```
            dataSource.setUsername("root");
            dataSource.setPassword("root");
            dataSource.setUrl("jdbc:mysql://localhost:3306/casadocodigo");
            dataSource.setDriverClassName("com.mysql.jdbc.Driver");
            return dataSource;
        }

        private Properties aditionalProperties(){
            Properties props = new Properties();
            props.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL5Dialect");
            props.setProperty("hibernate.show_sql", "true");
            props.setProperty("hibernate.hbm2ddl.auto", "update");
            return props;
        }

        @Bean
        public JpaTransactionManager transactionManager(EntityManagerFactory emf){
            return new JpaTransactionManager(emf);
        }

    }
```

E como último passo de configuração para que este teste funcione, precisamos apenas adicionar a classe `DataSourceConfigurationTest` a lista de classes de configuração na `ProdutoDAOTest`.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JPAConfiguration.class,ProdutoDAO.class, DataSourceConfigurationTes
@ActiveProfiles("test")
public class ProdutoDAOTest {
    // restante do codigo
}
```

Agora, ao executar novamente o teste com o *JUnit*, devemos ver o sinal verde, indicando que o teste foi executado e passou com sucesso.



Observação: Se verificarmos o banco de dados de testes após a execução de qualquer teste, não teremos dados neste, o banco estará em branco, somente com as tabelas criadas. Isso acontece por que o *Spring Test* limpa todos os dados do banco para que um teste não seja prejudicado com dados deixados por outros testes.

Agora o que acontece se tentarmos iniciar o servidor para ver a aplicação no navegador? O esperado é que a aplicação funcione sem nenhum problema, certo? Não! Ao tentarmos isso teremos um erro com a seguinte mensagem:

```
Error creating bean with name 'entityManagerFactory' defined
```

Como separamos as configurações de *Data Source* da aplicação com *Profiles*, o *Spring* não consegue saber qual configuração usar e assim, não usa nenhuma, causando o erro.

Para que possamos definir qual configuração de *Data Source* o *Spring* deve usar ao inicializar a aplicação, precisaremos de um ouvinte de contexto, que ao perceber a inicialização da aplicação, defina que o *profile* a ser utilizado será o de `dev`.

Para isso usaremos um novo método na classe `ServletSpringMVC` que faz a inicilização do *Spring*. Este método se chama `onStartup` e recebe um objeto do tipo `ServletContext` com o qual, através do método `addListeners` adicionaremos um ouvinte de contexto de requisição, objeto da classe `RequestContextListener` e por meio do método `setInitParameter` definiremos o parametro que define o *Profile* da aplicação com o valor `dev` da seguinte forma:

```
@Override
public void onStartup(ServletContext servletContext) throws ServletException {
    super.onStartup(servletContext);
    servletContext.addListener(new RequestContextListener());
    servletContext.setInitParameter("spring.profiles.active", "dev");
}
```

Agora sim, podemos iniciar a aplicação sem nenhum problema.

Testando controllers

Perceba que sempre que adicionamos um recurso na aplicação, sempre vamos até o navegador e verificamos se esta se comporta da forma que esperamos. Este é um processo natural no devenvolvimento web, mas e se quisessemos realizar este tesde dentro do próprio *Eclipse*, como faríamos?

Por exemplo, como podemos garantir que o `HomeController` em seu método `index` realmente esta retornando como resposta a *view* `home.jsp`? E como podemos garantir que os produtos estão sendo carregados para esta página? A resposta é simples! Por meio dos testes automatizados!

Atenção: Caso não tenha conhecimentos sobre testes de software, não se preocupe. Neste curso, veremos apenas alguns pontos sobre isso, apesar de não ser o foco deste curso. Por isso, recomendamos que faça os cursos de testes em Java (https://cursos.alura.com.br/category/programacao) disponíveis aqui na Alura.

Da mesma forma que fizemos os testes do `ProdutoDAO`, criando a classe `ProdutoDAOTest`. Criaremos testes para a classe `ProdutosController` com a classe `ProdutosControllerTest` no *Source Folder* de testes e no pacote de *controllers*.

O primeiro teste será fazer uma requisição para a página inicial da nossa aplicação e verificar se a *view* `home.jsp` está realmente sendo retornada. O primeiro passo ao criar a classe é fazer as devidas configurações, bem parecidas com as da classe `ProdutoDAOTest`.

```
@WebAppConfiguration
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JPAConfiguration.class, WebAppConfiguration.class, DataSourceConfig
@ActiveProfiles("test")
public class ProdutosControllerTest {

}
```

As únicas diferenças são que na anotação `@ContextConfiguration` em vez de carregarmos a classe `ProdutoDAO`, carregamos a classe que tem todas as configurações de *MVC* da aplicação, `AppWebConfiguration`. A outra diferença é a presença da anotação `@WebAppConfiguration` que faz o carregamento das demais configurações de *MVC* do *Spring*.

Para que possamos fazer requisições sem o uso de um navegador, precisamos de um objeto capaz de simular este procedimento. Estes objetos que simulam comportamentos são conhecidos como **Mock** s. E para a criação de um *mock* no *Spring*, precisaremos definir um contexto. Por isso, criaremos mais dois atributos na classe `ProdutosControllerTest` .

```
@Autowired
private WebApplicationContext wac;

private MockMvc mockMvc;
```

O *Spring* já conhece o contexto da aplicação, por isso o atributo `WebApplicationContext` é anotado com `@Autowired` . O Objeto `MockMvc` será o objeto que fará as requisições para os *controllers* da nossa aplicação.

Apesar do *Spring* criar o objeto de contexto da aplicação de forma automática. Este não cria o objeto *Mock*, mas facilita essa tarefa através de classes auxiliadoras. Para a criação do objeto `MockMvc` , definiremos um método que será executado antes dos testes e instanciaremos o objeto usando a classe `MockMvcBuilders` . Iremos fornecer para o método, que se chamará `setup` , o contexto `webAppContextSetup` .

```
@Before
public void setup(){
    mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
}
```

A anotação `@Before` é quem faz com que o método seja chamado antes que qualquer teste seja executado. Métodos de *setup* são muito comuns para carregamento de recursos e definição de configurações que devem ser executadas antes de qualquer teste. A classe `ProdutosControllerTest` até então se encontra com o seguinte código:

```
@WebAppConfiguration
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JPAConfiguration.class, AppWebConfiguration.class, DataSourceConfig
@ActiveProfiles("test")
public class ProdutosControllerTest {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup(){
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

}
```

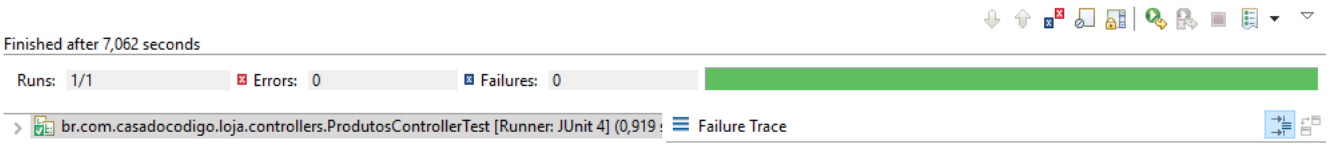
Com tudo configurado, faremos o primeiro teste! Para esta tarefa, definiremos um novo método que se chamará `deveRetornarParaHomeComOsLivros` que usará o método `perform` do objeto `mockMvc` para simular uma requisição. Mas para tal, precisaremos de algum outro objeto que construa um *Request*, objeto de requisição. Este objeto será fornecido pela classe `MockMvcRequestBuilders` por meio do método `get` para o qual será passado o caminho da requisição.

```
@Test
public void deveRetornarParaHomeComOsLivros() {
    mockMvc.perform(MockMvcRequestBuilders.get("/"))
}
```

Com a requisição feita, nos resta verificar o resultado da mesma por meio do método `andExpect` do objeto `mockMvc` que receberá o resultado do método `forwardedUrl` da classe `MockMvcResultMatchers` no qual verificará se foi feito um redirecionamento no servidor para a *view* localizada em `WEB-INF/views/home.jsp` . Assim teremos:

```
@Test
public void deveRetornarParaHomeComOsLivros() throws Exception{
    mockMvc.perform(MockMvcRequestBuilders.get("/"))
           .andExpect(MockMvcResultMatchers.forwardedUrl("/WEB-INF/views/home.jsp"));
}
```

Agora, ao executarmos o teste, veremos que o mesmo passa!



E para verificarmos a presença dos produtos na resposta da requisição? Faremos praticamente a mesma coisa que já fizemos: usaremos o `andExpect` novamente e também a classe `MockMvcResultMatchers` , mas com a pequena diferença que, desta vez, usaremos o método `model` para que consigamos obter informações sobre o objeto retornado pela requisição e neste objeto, verificaremos a existencia do atributo `produtos` utilizando o método `attributeExists` .

```
@Test
public void deveRetornarParaHomeComOsLivros() throws Exception{
    mockMvc.perform(MockMvcRequestBuilders.get("/"))
           .andExpect(MockMvcResultMatchers.model().attributeExists("produtos"))
           .andExpect(MockMvcResultMatchers.forwardedUrl("/WEB-INF/views/home.jsp"));
}
```

Outro teste interessante de ser feito é verificar se a página de cadastro de produto em `/produtos/form` é realmente acessada apenas pelos administradores. Para tal tipo, é necessário configurar uma nova dependência no arquivo `pom.xml` . Esta dependência torna possível realizar testes no contexto do *Spring Security*. Adicinaremos a seguinte dependência em nosso projeto.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <version>4.0.0.RELEASE</version>
    <scope>test</scope>
</dependency>
```

Após isso, podemos voltar para a classe `ProdutosControllerTest` e criar um novo método chamado `somenteAdminDeveAcessarProdutosForm` no qual realizaremos um novo *request* para o caminho `/produtos/form` .

```
@Test
public void somenteAdminDeveAcessarProdutosForm(){
    mockMvc.perform(MockMvcRequestBuilders.get("/produtos/form"));
}
```

Mas o teste não se resume a fazer uma requisição. Neste caso é necessário fazer uma tentativa de autenticação, assim, poderemos verificar se usuários diferentes dos administradores são redirecionados para outra página, que é a de login, neste caso.

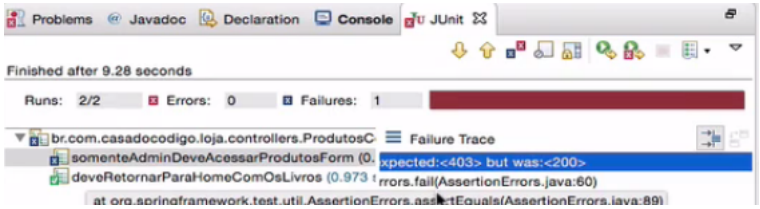
Para testar uma tentativa de autenticação, precisamos que o *Mock MVC* faça a requisição um *Post Processor*. Ou seja, um processo de *POST* antes de executar o *GET* da página, passando neste *Post Processor* os dados de autenticação do usuário. Observe o código abaixo:

```
@Test
public void somenteAdminDeveAcessarProdutosForm() throws Exception{
    mockMvc.perform(MockMvcRequestBuilders.get("/produtos/form")
            .with(SecurityMockMvcRequestPostProcessors
                .user("user@casadocodigo.com.br").password("123456")
                .roles("USUARIO")))
            .andExpect(MockMvcResultMatchers.status().is(403));
}
```

Note que para podermos usar o *Post Processor*, precisaremos fazer uso do método `with` que adiciona dados adicionais à requisição. A classe `SecurityMockMvcRequestPostProcessors` do *Spring Security Test* nos permite simular os dados de um usuário, com senha e *Role* configurados para a requisição. Por último, verificamos na resposta da requisição se o *status* da mesma foi um código **403** que significa um redirecionamento.

Observação: Caso o `roles` recebesse o valor `ADMIN` o teste falharia, pois usuários com este valor de *role* podem acessar o formulário de produtos. Para resolver este caso, o código de *status* da resposta, deveria ser posto com o valor **200**.

Mas algo estranho acontece ao tentar executar os testes. O teste de autenticação de usuários falha.



O esperado que era o código **403** não foi coorespondido. O sistema retornou o código **200**. O que aconteceu? Parece estranho, mas na verdade, cometemos um pequeno deslize.

Acontece que nas configurações da classe de testes, não especificamos as classes de configurações do *Spring Security*, desta forma, sem que este seja carregado, fica impossível de realizar os testes de segurança. Vamos resolver isso fazendo os seguintes passos.

Na anotação `@ContextConfiguration` da classe `ProdutosControllerTest` adicionaremos a classe de configuração do *Spring Security* `SecurityConfiguration.class` . Criaremos um noto atributo do tipo `Filter` que chamaremos de `springSecurityFilterChain` anotado com `@Autowired` e por fim, adicionaremos este filtro ao `mockMvc` através do método `AddFilter` .

```
@ContextConfiguration(classes = {JPAConfiguration.class, AppWebConfiguration.class, DataSourceConfig
@ActiveProfiles("test")
public class ProdutosControllerTest {

    // codigo suprimido

    @Autowired
    private Filter springSecurityFilterChain;

    @Before
    public void setup(){
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).addFilter(springSecurityFilterChain).build
    }

  // codigo suprimido

}
```

Agora, ao executarmos o teste novamente, este passa. Significando que o *Spring Security* foi carregado e fez a verificação de autenticação do usuário.