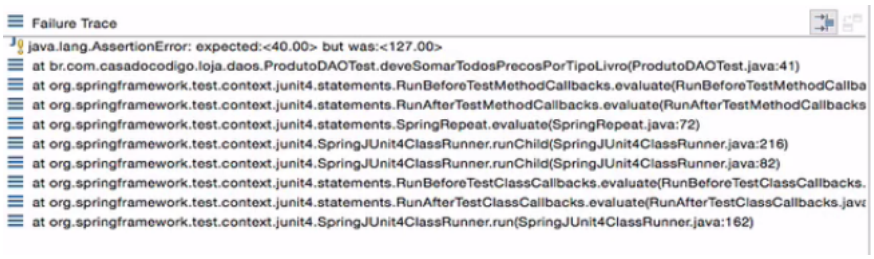
04

Trabalhando com profiles

Transcrição

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 LocalContainerEntityManagerFactoryB
    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 LocalContainerEntityManagerFactoryB
    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 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;
    }
```

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("dev")
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:

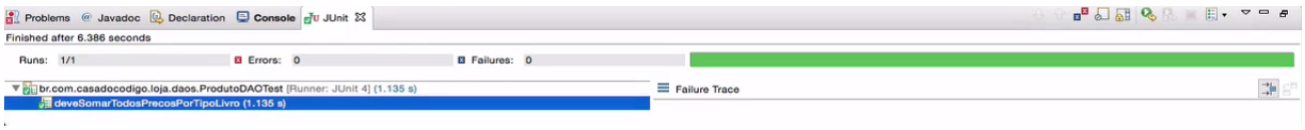
```
@EnableTransactionManagement
public class JPAConfiguration {

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {
        LocalContainerEntityManagerFactoryBean factoryBean = new LocalContainerEntityManagerFact
        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("dev")
    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, DataSourceConfiguratio
@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.