02

Escrevendo o primeiro teste de integração

Até esse ponto, todas as classes de negócio foram testadas isoladamente, com testes de unidade. Algumas delas inclusive, eram mais complicadas, dependiam de outras classes, e nesses casos fizemos uso de Mock Objects. Mocks são muito importantes quando queremos testar a classe isolada do "resto", ou seja, das outras classes que ela depende e faz uso. Mas a pergunta que fica é: será que vale a pena mockar as dependências de uma classe no momento de testá-la?

Veja um DAO, por exemplo. Um DAO é uma classe que esconde toda a complexidade de se comunicar com o banco de dados. É ela que contém os comandos SQLs que explicarão ao banco o que fazer com o conjunto de dados que está lá. Um DAO depende de um sistema externo: o banco de dados.

Veja um exemplo de DAO abaixo, que salva e busca por usuários do sistema. Repare que ele usa Hibernate para acessar o banco de dados. O Hibernate é uma ferramenta que facilita o acesso a banco de dados. Se você não o conhece, não tem problema; não precisará entender os detalhes dele para escrever o teste (mas se mesmo assim quiser entender melhor a ferramenta, recomendamos o nosso curso de jpa e Hibernate (https://www.alura.com.br/curso-online-persistencia-de-objetos-com-jpa-hibernate))!

```
public class UsuarioDao {

    private final Session session;

    public UsuarioDao(Session session) {
        this.session = session;
    }

    public Usuario porId(int id) {
        return (Usuario) session.load(Usuario.class, id);
    }

    public Usuario porNomeEEmail(String nome, String email) {
        return (Usuario) session.createQuery(
                "from Usuario u where u.nome = :nome and x.email = :email")
                .setParameter("nome", nome)
                .setParameter("email", email)
                .uniqueResult();
    }

    public void salvar(Usuario usuario) {
        session.save(usuario);
    }
}
```

Assim como nos outros cursos, nosso projeto é um sistema de leilões, onde usuários dão lances em leilões. O projeto, pronto para esse curso, pode ser baixado aqui (http://s3.amazonaws.com/caelum-online-public/PM-73/pm73-dao.zip).

Será que faz sentido testar nosso DAO e "mockar o banco de dados"? Vamos tentar testar o método `porNomeEEmail()`, que busca um usuário pelo nome e e-mail. Usaremos o JUnit, framework que já estamos acostumados.

Como todo teste, ele tem cenário, ação e validação. O cenário será mockado; faremos com que a Session retorne um usuário. A ação será invocar o método `porNomeEEmail()`. A validação será garantir que o método retorna um `Usuario`

com os dados corretos.

Para isso, precisamos instanciar um `UsuarioDao`. Repare que essa classe depende de uma "Session" do Hibernate. A `Session` é análogo ao `Connection`, ou seja, é a forma de falar com o banco de dados. Todo sistema geralmente tem sua forma de conseguir uma conexão com o banco de dados; o nosso não é diferente.

Conforme visto no curso anterior, vamos mockar a "Session" do Hibernate. No caso, mockaremos a classe `Session` e a classe `Query`:

```
@Test
public void deveEncontrarPeloNomeEEmailMockado() {
    Session session = Mockito.mock(Session.class);
    Query query = Mockito.mock(Query.class);
    UsuarioDao usuarioDao = new UsuarioDao(session);
}
```

Em seguida, vamos setar o comportamento desses mocks para que funcionem de acordo. Precisaremos simular os métodos `createQuery()`, `setParameter()` e `thenReturn` (que são os métodos usados pelo DAO):

```
@Test
public void deveEncontrarPeloNomeEEmailMockado() {
    Session session = Mockito.mock(Session.class);
    Query query = Mockito.mock(Query.class);
    UsuarioDao usuarioDao = new UsuarioDao(session);

    Usuario usuario = new Usuario
            ("João da Silva", "joao@dasilva.com.br");
    String sql = "from Usuario u where u.nome = :nome and x.email = :email";

    Mockito.when(session.createQuery(sql)).thenReturn(query);
    Mockito.when(query.uniqueResult()).thenReturn(usuario);
    Mockito.when(query.setParameter("nome", "João da Silva")).thenReturn(query);
    Mockito.when(query.setParameter("email", "joao@dasilva.com.br")).thenReturn(query);
}
```

Por fim, vamos invocar o método que queremos testar e validar a saída:

```
@Test
public void deveEncontrarPeloNomeEEmailMockado() {
    Session session = Mockito.mock(Session.class);
    Query query = Mockito.mock(Query.class);
    UsuarioDao usuarioDao = new UsuarioDao(session);

    Usuario usuario = new Usuario
            ("João da Silva", "joao@dasilva.com.br");
    String sql = "from Usuario u where u.nome = :nome and x.email = :email";

    Mockito.when(session.createQuery(sql)).thenReturn(query);
    Mockito.when(query.uniqueResult()).thenReturn(usuario);
    Mockito.when(query.setParameter("nome", "João da Silva")).thenReturn(query);
    Mockito.when(query.setParameter("email", "joao@dasilva.com.br")).thenReturn(query);

    Usuario usuarioDoBanco = usuarioDao
```

```
                .porNomeEEmail("João da Silva", "joao@dasilva.com.br");

        assertEquals(usuario.getNome(), usuarioDoBanco.getNome());
        assertEquals(usuario.getEmail(), usuarioDoBanco.getEmail());

    }
```

Excelente. Se rodarmos o teste, ele passa! Isso quer dizer que conseguimos então simular o banco de dados e facilitar a escrita do teste, certo? Errado!

Olhe a consulta SQL com mais atenção: `from Usuario u where u.nome = :nome and x.email = :email`. Veja que o "x.email" está errado! Deveria ser "u.email". Isso seria facilmente descoberto se não estivéssemos simulando o banco de dados, mas sim usando um banco de dados real! A SQL seria imediatamente recusada!

A resposta da primeira pergunta então é **NÃO**. Se o único objetivo do DAO é falar com o banco de dados, não faz sentido simular justamente o serviço externo que ele se comunica. Nesse caso, precisamos testar a comunicação do nosso DAO com um banco de dados de verdade; queremos garantir que nossos INSERTs, SELECTs e UPDATEs estão corretos e funcionam da maneira esperada. Se simulássemos um banco de dados, não saberíamos ao certo se, na prática, ele funcionaria com nossas SQLs!

Escrever um teste para um DAO é parecido com escrever qualquer outro teste: (i) precisamos montar um cenário, (ii) executar uma ação e (iii) validar o resultado esperado.

Vamos testar então novamente o método `porNomeEEmail()`, mas dessa vez batendo em um banco de dados real. Como exemplo, podemos usar o usuário "João da Silva", com o e-mail "joao@dasilva.com.br". Vamos já corrigir o método do DAO e fazer "u.email" que é o certo:

```
public Usuario porNomeEEmail(String nome, String email) {
    return (Usuario) session.createQuery(
            "from Usuario u where u.nome = :nome and u.email = :email")
            .setParameter("nome", nome)
            .setParameter("email", email)
            .uniqueResult();
}
```

Vamos ao teste. Começaremos por invocar esse método do DAO:

```
@Test
public void deveEncontrarPeloNomeEEmail() {
    Usuario usuario = usuarioDao
            .porNomeEEmail("João da Silva", "joao@dasilva.com.br");
}
```

Mas para criar o DAO, precisamos passar uma Session do Hibernate; e dessa vez não vamos mockar. A classe `CriadorDeSessao` cria a `Session`. Devemos então passá-la para o DAO. No teste:

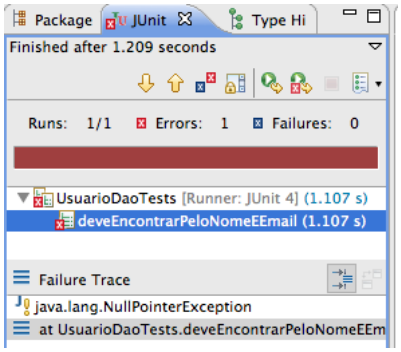
```
@Test
public void deveEncontrarPeloNomeEEmail() {
    Session session = new CriadorDeSessao().getSession();
    UsuarioDao usuarioDao = new UsuarioDao(session);
```

```
        Usuario usuario = usuarioDao.porNomeEEmail(
            "João da Silva", "joao@dasilva.com.br");
    }
```

Ótimo. Se tudo deu certo, espera-se que a instância `usuario` contenha o nome e e-mail passados. Vamos escrever os asserts então:

```
    @Test
    public void deveEncontrarPeloNomeEEmail() {
        Session session = new CriadorDeSessao().getSession();
        UsuarioDao usuarioDao = new UsuarioDao(session);

        Usuario usuario = usuarioDao
                .porNomeEEmail("João da Silva", "joao@dasilva.com.br");

        assertEquals("João da Silva", usuario.getNome());
        assertEquals("joao@dasilva.com.br", usuario.getEmail());
    }
```

O teste está pronto, mas se o rodarmos, ele falhará.



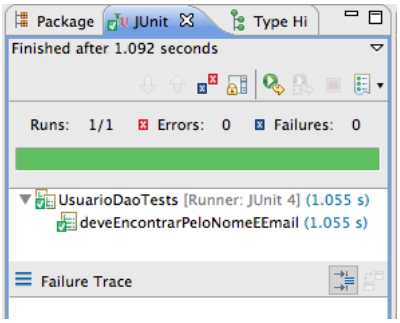
O motivo é simples: para que o teste passe, o usuário "João da Silva" deve existir no banco de dados! Precisamos salvá-lo no banco antes de invocar o método `porNomeEEmail` . Essa é a principal diferença entre testes de unidade e testes de integração: precisamos montar o cenário, executar a ação e validar o resultado esperado no software externo.

Para salvar o usuário, basta invocarmos o método `salvar()` do próprio DAO. Veja o código abaixo, onde criamos um usuário e o salvamos. Não podemos também esquecer de fechar a sessão com o banco de dados (afinal, sempre que consumimos um recurso externo, precisamos fechá-lo!):

```
    @Test
    public void deveEncontrarPeloNomeEEmail() {
        Session session = new CriadorDeSessao().getSession();
        UsuarioDao usuarioDao = new UsuarioDao(session);

        // criando um usuario e salvando antes
        // de invocar o porNomeEEmail
        Usuario novoUsuario = new Usuario
                ("João da Silva", "joao@dasilva.com.br");
        usuarioDao.salvar(novoUsuario);

        // agora buscamos no banco
        Usuario usuarioDoBanco = usuarioDao
                .porNomeEEmail("João da Silva", "joao@dasilva.com.br");
```

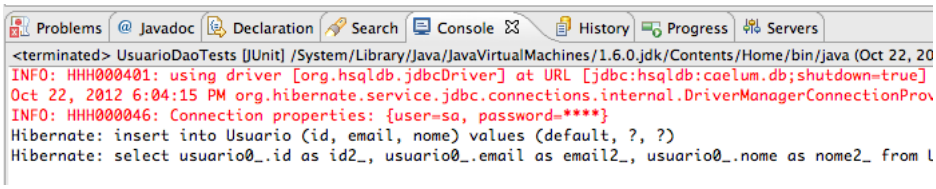
```
        assertEquals("João da Silva", usuarioDoBanco.getNome());
        assertEquals("joao@dasilva.com.br", usuarioDoBanco.getEmail());

        session.close();
    }
```

Agora sim, o teste passa!



Veja então que escrever um teste para um DAO não é tão diferente; é só mais trabalhoso, afinal precisamos nos comunicar com o software externo o tempo todo, para montar cenário, para validar se a operação foi efetuada com sucesso e etc. Em nosso caso, criamos uma "Session" (uma conexão com o banco), inserimos um usuário no banco (um INSERT, da SQL), e depois uma busca (um SELECT).

Isso pode inclusive ser visto pelo log do Hibernate, no console do Eclipse:



Chamamos esses testes de **testes de integração**, afinal estamos testando o comportamento da nossa classe integrada com um serviço externo real. Testes como esse são úteis para classes como nossos DAOs, cuja tarefa é justamente se comunicar com outro serviço.