Testes de Integração

Site:	Moodle institucional da UTFPR	Impresso por:	MARLENE VASCONCELOS MORAES DE OLIVEIRA
Curso:	CETEJ158A - Teste de Software - J29 (2023_03) J30A (2024_01)	Data:	domingo, 25 ago. 2024, 09:17
Livro:	Testes de Integração		

Descrição

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 de 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?

Índice

1. Devemos mockar um DAO?

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 com 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 retorne 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 análoga 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 anteriormente, vamos mockar a Session do Hibernate. No caso, mockaremos a classe Session e a 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 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!

2. Testando DAOs

A resposta da primeira pergunta, portanto, é NÃO. Se o único objetivo do DAO é falar com o banco de dados, não faz sentido simular justamente o serviço externo com 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:

1. Precisamos montar um cenário;
2. Executar uma ação; e
3. Validar o resultado esperado.

Vamos testar 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 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:

```
@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 a seguir, 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 usuário 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 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 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.

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 (no caso, o banco de dados).

Muitas vezes ficamos na dúvida se devemos mockar ou fazer um teste de integração real. No último exemplo, mostrei que se mockássemos a sessão com o banco de dados teríamos um teste inútil, que não nos daria o feedback ideal. Lembre-se: se a tarefa da classe é comunicar com um outro serviço, a única maneira de garantir que ela funciona é fazendo-a comunicar-se de verdade com ele. E você precisa achar uma maneira de fazer isso acontecer. Os desafios são vários e totalmente contextuais: montar o cenário, fazer a validação etc. Mas quem disse que testar era tarefa fácil?

3. Testando cenários mais complexos

Vamos agora começar a testar nosso LeilaoDao. Um dos métodos desse DAO retorna a quantidade de leilões que ainda não foram encerrados. Veja:

```
    public Long total() {
        return (Long) session.createQuery("select count(l) from "
                + "Leilao l where l.encerrado = false")
    }
```

Ele faz um simples SELECT COUNT. Para testar essa consulta, adicionaremos dois leilões: um encerrado e outro não encerrado. Dado esse cenário, esperamos que o método total() nos retorne 1. Vamos ao teste.

Repare que, para criar um Leilão, precisaremos criar um Usuário e persisti-lo no banco também, afinal o Leilão referencia um Usuário; para fazer isso, utilizaremos o UsuarioDao, que sabe persistir um Usuario!

Essa é uma das dificuldades de se escrever um teste de integração: montar cenário é mais difícil. Dê uma olhada no código a seguir, ele é extenso, mas está comentado:

```
public class LeilaoDaoTests {

    private Session session;
    private LeilaoDao leilaoDao;
    private UsuarioDao usuarioDao;

    @Before
    public void antes() {
        session = new CriadorDeSessao().getSession();
        leilaoDao = new LeilaoDao(session);
        usuarioDao = new UsuarioDao(session);
    }

    @After
    public void depois() {
        session.close();
    }

    @Test
    public void deveContarLeiloesNaoEncerrados() {
// criamos um usuário
        Usuario mauricio
                = new Usuario("Mauricio Aniche", "mauricio@aniche.com.br");
// criamos os dois leilões
        Leilao ativo
                = new Leilao("Geladeira", 1500.0, mauricio, false);
        Leilao encerrado
                = new Leilao("XBox", 700.0, mauricio, false);
        encerrado.encerra();
// persistimos todos no banco
        usuarioDao.salvar(mauricio);
        leilaoDao.salvar(ativo);
        leilaoDao.salvar(encerrado);

// invocamos a ação que queremos testar
// pedimos o total para o DAO
        long total = leilaoDao.total();
        assertEquals(1L, total);
    }
}
```

Se rodarmos o teste, ele passa!

Mas esse teste ainda não está legal. Se estivéssemos rodando em um MySQL, por exemplo, esse teste poderia falhar. Cada vez que rodamos o teste, ele insere 2 linhas no banco de dados. Se rodarmos o teste 2 vezes, por exemplo, teremos 2 leilões não encerrados, o que faz com que o teste falhe.

A melhor maneira de garantir que, independente do banco em que você esteja rodando o teste, o cenário esteja sempre limpo para aquele teste. É ter a base de dados limpa. Um jeito simples de fazer isso é executar cada um dos testes dentro de um contexto de transação e, ao final do teste, fazer um “rollback”. Com isso, o banco rejeitará tudo o que aconteceu no teste e continuará limpo.

Isso é fácil de ser implementado. Basta mexermos nos métodos @Before e @After:

```
    @Before
    public void antes() {

        session = new CriadorDeSessao().getSession();
        leilaoDao = new LeilaoDao(session);
        usuarioDao = new UsuarioDao(session);
// inicia transação
        session.beginTransaction();
    }

    @After
    public void depois() {
// faz o rollback
        session.getTransaction().rollback();
        session.close();
    }
```

Pronto. Agora, mesmo no MySQL, esse teste passaria. Iniciar e dar rollback na transação durante testes de integração é prática comum. Faça uso do @Before e @After para isso, e dessa forma, seus testes ficam independentes e fáceis de manter.

4. Praticando com consultas mais complicadas

Algumas consultas são mais difíceis de serem testadas, simplesmente porque seus cenários são mais complicados. Nesses casos, precisamos facilitar a criação de cenários.

Veja, por exemplo, o método porPeriodo(Calendar inicio, Calendar fim), do nosso LeilaoDao. Ele devolve todos os leilões que foram criados dentro de um período e que ainda não foram encerrados:

```
    public List<Leilao> porPeriodo(Calendar inicio, Calendar fim) {

        return session.createQuery("from Leilao l where l.dataAbertura "
                + "between :inicio and :fim and l.encerrado = false")
                .setParameter("inicio", inicio)
                .setParameter("fim", fim)
                .list();
    }
```

Para testarmos esse método, precisamos pensar em alguns cenários, como:

- • Leilões não encerrados com data dentro do intervalo devem aparecer;
- • Leilões encerrados com data dentro do intervalo não devem aparecer;
- • Leilões encerrados com data fora do intervalo não devem aparecer;
- • Leilões não encerrados com data fora do intervalo não devem aparecer.

Vamos começar pelo primeiro cenário. Criaremos 2 leilões não encerrados, um com data dentro do intervalo, outro com data fora do intervalo, e vamos garantir que só o primeiro estará lá dentro. O método é grande, mas está comentado:

```
    @Test
    public void deveTrazerLeiloesNaoEncerradosNoPeriodo() {
// criando as datas
        Calendar comecoDoIntervalo = Calendar.getInstance();
        comecoDoIntervalo.add(Calendar.DAY_OF_MONTH, -10);
        Calendar fimDoIntervalo = Calendar.getInstance();
        Calendar dataDoLeilao1 = Calendar.getInstance();
        dataDoLeilao1.add(Calendar.DAY_OF_MONTH, -2);
        Calendar dataDoLeilao2 = Calendar.getInstance();
        dataDoLeilao2.add(Calendar.DAY_OF_MONTH, -20);
        Usuario mauricio = new Usuario("Mauricio Aniche",
                "mauricio@aniche.com.br");
// criando os leilões, cada um com uma data
        Leilao leilao1
                = new Leilao("XBox", 700.0, mauricio, false);
        leilao1.setDataAbertura(dataDoLeilao1);
        Leilao leilao2
                = new Leilao("Geladeira", 1700.0, mauricio, false);
        leilao2.setDataAbertura(dataDoLeilao2);
// persistindo os objetos no banco
        usuarioDao.salvar(mauricio);
        leilaoDao.salvar(leilao1);
        leilaoDao.salvar(leilao2);
// invocando o método para testar
        List<Leilao> leiloes
                = leilaoDao.porPeriodo(comecoDoIntervalo, fimDoIntervalo);
// garantindo que a query funcionou
        assertEquals(1, leiloes.size());
        assertEquals("XBox", leiloes.get(0).getNome());
    }
```

Ele passa. Vamos ao próximo cenário: leilões encerrados devem ser ignorados pela consulta. Nesse caso, criaremos apenas um leilão encerrado, dentro do intervalo. Esperaremos que a query não devolva nada:

```
    @Test
    public void naoDeveTrazerLeiloesEncerradosNoPeriodo() {
// criando as datas
        Calendar comecoDoIntervalo = Calendar.getInstance();
        comecoDoIntervalo.add(Calendar.DAY_OF_MONTH, -10);
        Calendar fimDoIntervalo = Calendar.getInstance();
        Calendar dataDoLeilao1 = Calendar.getInstance();
        dataDoLeilao1.add(Calendar.DAY_OF_MONTH, -2);
        Usuario mauricio = new Usuario("Mauricio Aniche",
                "mauricio@aniche.com.br");
// criando os leilões, cada um com uma data
        Leilao leilao1
                = new Leilao("XBox", 700.0, mauricio, false);
        leilao1.setDataAbertura(dataDoLeilao1);
        leilao1.encerra();
// persistindo os objetos no banco
        usuarioDao.salvar(mauricio);
        leilaoDao.salvar(leilao1);
// invocando o método para testar
        List<Leilao> leiloes
                = leilaoDao.porPeriodo(comecoDoIntervalo, fimDoIntervalo);
// garantindo que a query funcionou
        assertEquals(0, leiloes.size());
    }
```

Montar cenários é o grande segredo de um teste de integração. Queries mais complexas exigirão cenários de teste mais complexos. Nos capítulos anteriores, estudamos sobre como melhorar a escrita dos testes, como Test Data Builders etc. Você pode (e deve) fazer uso deles para facilitar a escrita dos cenários!

Geralmente, o grande desafio é justamente montar o cenário e validar o resultado esperado no sistema externo. No caso de banco de dados, precisamos fazer INSERTs, DELETEs, e SELECTs para validar. Além disso, ainda precisamos manter o estado do sistema consistente, ou seja, devemos limpar o banco de dados constantemente para que um teste não atrapalhe o outro.

Veja que usamos o JUnit da mesma forma. A diferença é que justamente precisamos nos comunicar com o outro sistema.

5. Testando alteração e deleção

Até agora testamos operações de inserção e seleção. Chegou a hora de testar remoção e atualização. Veja o método deleta() no UsuarioDao:

```
public void deletar(Usuario usuario) {
    session.delete(usuario);
}
```

Ele deleta o usuário que é passado. Para testar uma deleção, precisamos primeiro inserir um elemento, deletá-lo, e depois fazer uma consulta para garantir que ele não está lá. Vamos aos testes:

```
    @Test
    public void deveDeletarUmUsuario() {
        Usuario usuario
                = new Usuario("Mauricio Aniche", "mauricio@aniche.com.br");

        usuarioDao.salvar(usuario);
        usuarioDao.deletar(usuario);
        Usuario usuarioNoBanco
                = usuarioDao.porNomeEEmail("Mauricio Aniche", "mauricio@aniche.com.br");
        assertNull(usuarioNoBanco);
    }
```

Excelente, o teste está da maneira que pretendíamos. Mas se rodarmos, ele falha! A pergunta é: por quê?

Muitas vezes nossa camada de acesso a dados não envia as consultas SQL para o banco até que você finalize a transação. É o que está acontecendo aqui. Precisamos forçar o envio do INSERT e do DELETE para o banco, para que depois o SELECT não traga o objeto! Essa operação é chamada de flush:

```
    @Test
    public void deveDeletarUmUsuario() {
        Usuario usuario
                = new Usuario(
                        "Mauricio Aniche",
                        "mauricio@aniche.com.br");
        usuarioDao.salvar(usuario);
        usuarioDao.deletar(usuario);
// envia tudo para o banco de dados
        session.flush();
        Usuario usuarioNoBanco
                = usuarioDao.porNomeEEmail(
                        "Mauricio Aniche",
                        "mauricio@aniche.com.br");
        assertNull(usuarioNoBanco);
    }
```

Agora sim nosso teste passa! É muito comum fazer uso de flush sempre que fazemos testes com banco de dados. Dessa forma, garantimos que a consulta realmente chegou ao banco de dados, e que as futuras consultas levarão mesmo em consideração as consultas anteriores.

6. Organizando testes de integração

Nossa bateria de testes de integração crescerá, mas já conseguimos reparar em alguns padrões nela: em todo começo de teste, abrimos uma Session, e no fim a fechamos.Aprendemos que sempre que algo ocorre no começo e fim de todo o teste, a boa prática é não repetir código, mas sim fazer uso de métodos anotados com @Before e @After. Você se lembra? Esses métodos são executados respectivamente antes e depois de todos os testes.

Vamos lá. Criaremos dois atributos na classe, uma Session e um UsuarioDao e faremos o método com @Before instanciar esses objetos. No método com @After, fecharemos a sessão. Com isso, os métodos de testes ficarão mais enxutos:

```
public class UsuarioDaoTests {

    private Session session;
    private UsuarioDao usuarioDao;

    @Before
    public void antes() {
// criamos a sessão e a passamos para o dao
        session = new CriadorDeSessao().getSession();
        usuarioDao = new UsuarioDao(session);
    }

    @After
    public void depois() {
// fechamos a sessão
        session.close();
    }

    @Test
    public void deveEncontrarPeloNomeEEmail() {
        Usuario novoUsuario = new Usuario("João da Silva", "joao@dasilva.com.br");
        usuarioDao.salvar(novoUsuario);
        Usuario usuarioDoBanco = usuarioDao
                .porNomeEEmail("João da Silva", "joao@dasilva.com.br");
        assertEquals("João da Silva",
                usuarioDoBanco.getNome());
        assertEquals("joao@dasilva.com.br",
                usuarioDoBanco.getEmail());
    }

    @Test
    public void deveRetornarNuloSeNaoEncontrarUsuario() {
        Usuario usuarioDoBanco = usuarioDao
                .porNomeEEmail("João Joaquim", "joao@joaquim.com.br");
        assertNull(usuarioDoBanco);
    }
}
```

Excelente. Muito melhor, e os testes continuam passando. Lembre-se da discussão que permeia o curso inteiro sobre qualidade do código de testes.

7. Referência Bibliográfica

Extraído de:
Aniche, M. **Testes Automatizados de Software**. São Paulo: Casa do Código, 2015.