Testes de integração utilizando CDI, JPA e EJB e Arquillian

Neste artigo veremos uma introdução sobre o Arquillian e de como utilizá-lo. Veremos como criar e enriquecer seus testes de integração utilizando os principais recursos do Java EE, como a CDI, JPA e EJB, bem como executá-los no WildFly.

Antes de conhecermos melhor o Arquillian, é interessante considerar algumas terminologias que iremos abordar durante este artigo:

1. **Container**: Um ambiente de execução para uma implantação;
2. **Implantação**: O processo de envio de um artefato para um container para torná-lo operacional;
3. **Arquivo**: Um conjunto empacotado de código, configuração e recursos.

O Arquillian é uma plataforma de testes para o Java, que permite aos desenvolvedores escrever e executar testes de integração e funcionais de forma simples. Basicamente, o Arquillian aproveita JUnit ou TestNG para executar casos de teste contra um container Java (por exemplo, GlassFish).

Além disso, o Arquillian lida com todas as canalizações da gestão de containers, implantação e inicialização, deixando apenas que o desenvolvedor se concentre na lógica de negócio em que o teste está sendo escrito. Em vez de configurar um equipamento de teste potencialmente complexa, o Arquillian abstrai isso em tempo de execução, assim:

- Gerência o ciclo de vida do(s) container (s);
- Faz agregação do(s) caso de teste(s), classes e recursos dependentes em um arquivo ShrinkWrap (mostraremos mais a diante);
- Realiza a implantação do(s) arquivo(s) no(s) container(s);
- Enriquece o(s) caso(s) de teste(s), fornecendo injeção de dependência e outros serviços declarativos do Java EE;
- Executa os testes no interior do container;
- Captura os resultados dos testes, logo em seguida devolve os resultados para o corredor de teste (por exemplo, JUnit) para assim serem mostrados no IDE;
- O Arquillian integra-se perfeitamente com os frameworks de teste (por exemplo, JUnit 4, TestNG 5), permitindo que os testes possam ser lançados usando IDE, Ant e Maven, sem qualquer add-ons.

Por sua vez, o Arquillian fornece em sua arquitetura um corredor de testes adaptado para JUnit e TestNG, que permite ao framework de teste executar o ciclo do mesmo com o Arquillian. O ShrinkWarp é usado para definir de forma declarativa quais arquivos (por exemplo, classes) serão usadas, assim fazendo o empacotamento das classes e seus recursos dependentes, em seguida é implantado e executado no container destino, após a execução os resultados dos testes são capturados. Ao final, o Arquillian desimplanta o arquivo e mostra os resultados dos testes por meio do corredor de testes (por exemplo, TestNG). Veja na **Figura 1** como é composta a arquitetura do Arquillian.

Figura 1. Mostra a arquitetura do Arquillian

Por fim, outro ponto fundamental sobre o Arquillian é que ele possui três estilos de interação com os containers, são eles:

- **Container remoto:** O Arquillian faz uso de um container já em execução, ou seja, o servidor já deve estar rodando para que o deploy do pacote de testes seja realizado e executado;
- **Container gerenciado:** É similar ao container remoto, exceto seu ciclo de vida (inicialização / desligamento) que é gerenciado pelo Arquillian;
- **Container incorporado (ou embutido)**: O Arquillian simplesmente sobe uma instância do container (por exemplo, WildFly) dentro do próprio teste.

Veremos na sequência a como criar testes de integração utilizando CDI, JPA e EJB com o apoio do Arquillian através do projeto Maven, bem como executá-los no WildFly, mas antes disso, destacaremos na próxima seção as principais ferramentas que iremos utilizar.

Ferramentas utilizadas

Utilizaremos o **Arquillian** (vide seção **Links**) junto com a **IDE Eclipse** e a perspectiva Java EE. A versão utilizada neste artigo é o Eclipse Mars (4.5 – vide seção **Links**), mas podem utilizar outra versão do Eclipse com suporte a Java EE.

O **Apache Maven** é uma ferramenta de gerenciamento de projetos de software e compreensão baseado no conceito de um modelo de objeto de projeto (POM), que pode gerenciar um projeto de construção, elaboração de relatórios e documentação de uma peça central de informações. Com isso, a versão utilizada neste artigo é o Maven 3.3.3 (seção **Links**), mas podem utilizar outra versão.

O **JUnit** é um framework open-source com suporte à criação de testes automatizados na linguagem de programação Java, e a versão utilizada neste artigo é a 4.11 (na seção **Links** você tem o link para a versão).

Criando um novo o projeto

Vamos agora criar um novo projeto Maven para que possamos adicionar as dependências como JUnit, Arquillian e outras configurações dentro do arquivo **pom.xml**.

Primeiro inicie a IDE Eclipse e depois selecione "File > New> Other ..." e "Maven > Maven Project". Clique em "Next" e na seguinte tela marque o checkbox "Create a simple project" e clique em "Next" novamente. Na janela que aparece na **Figura 2** preencha os campos "Group Id " e "Artifact Id " e clique em "Finish".

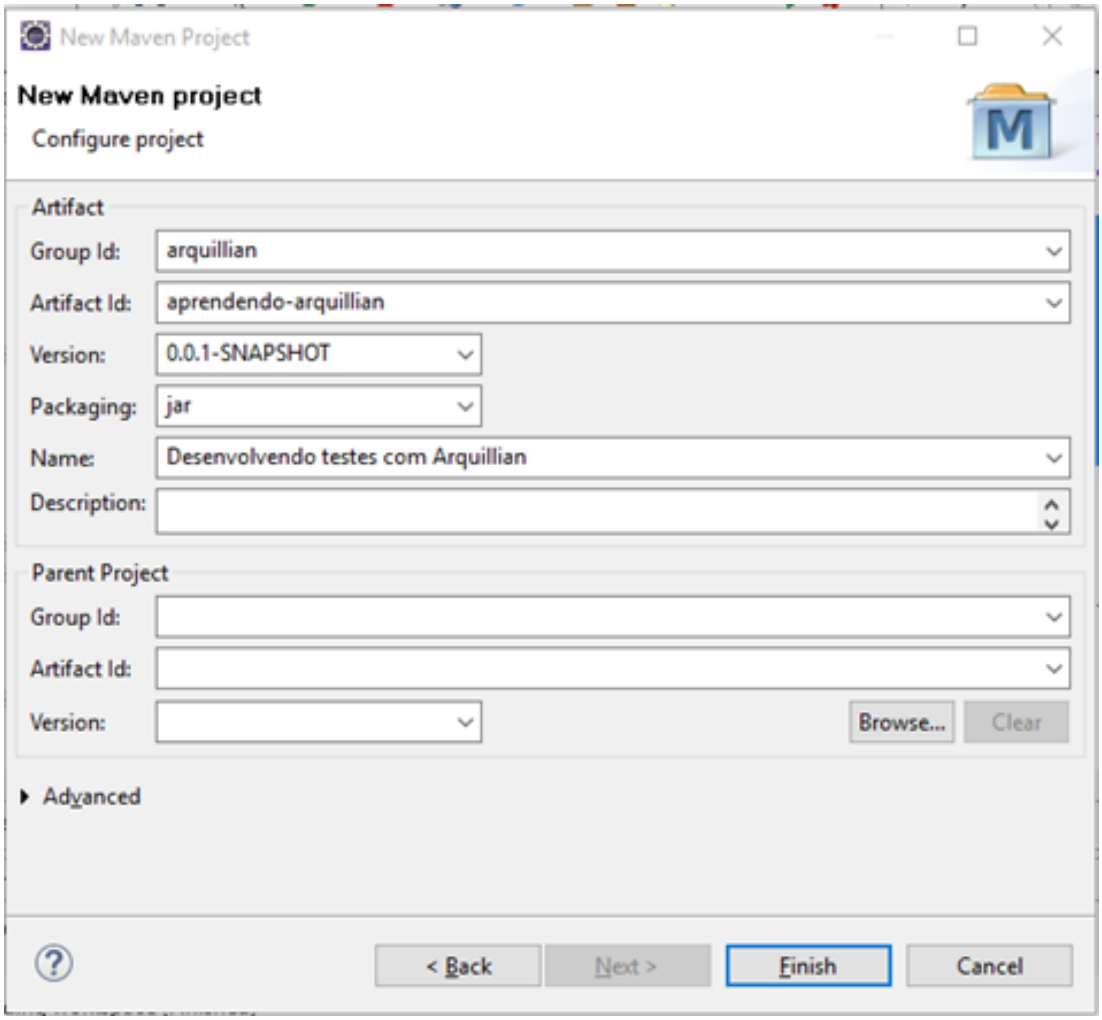

Figura 2. Mostra a janela do eclipse com as configurações do projeto Maven

Configuração do projeto

Após criado o projeto Maven, caso o seu projeto esteja utilizando uma versão do Java diferente da qual você utilize, podemos configurar isto. Isso acontecer porque se você não declarou no pom.xml explicitamente a versão do Java em que o projeto deverá ser compilado, o Maven assumirá por padrão o Java 1.5. Como exemplo, mostraremos como configurar do Java 1.5 para o 1.8. Veja na **Figura 3** como era o projeto com Java 1.5.

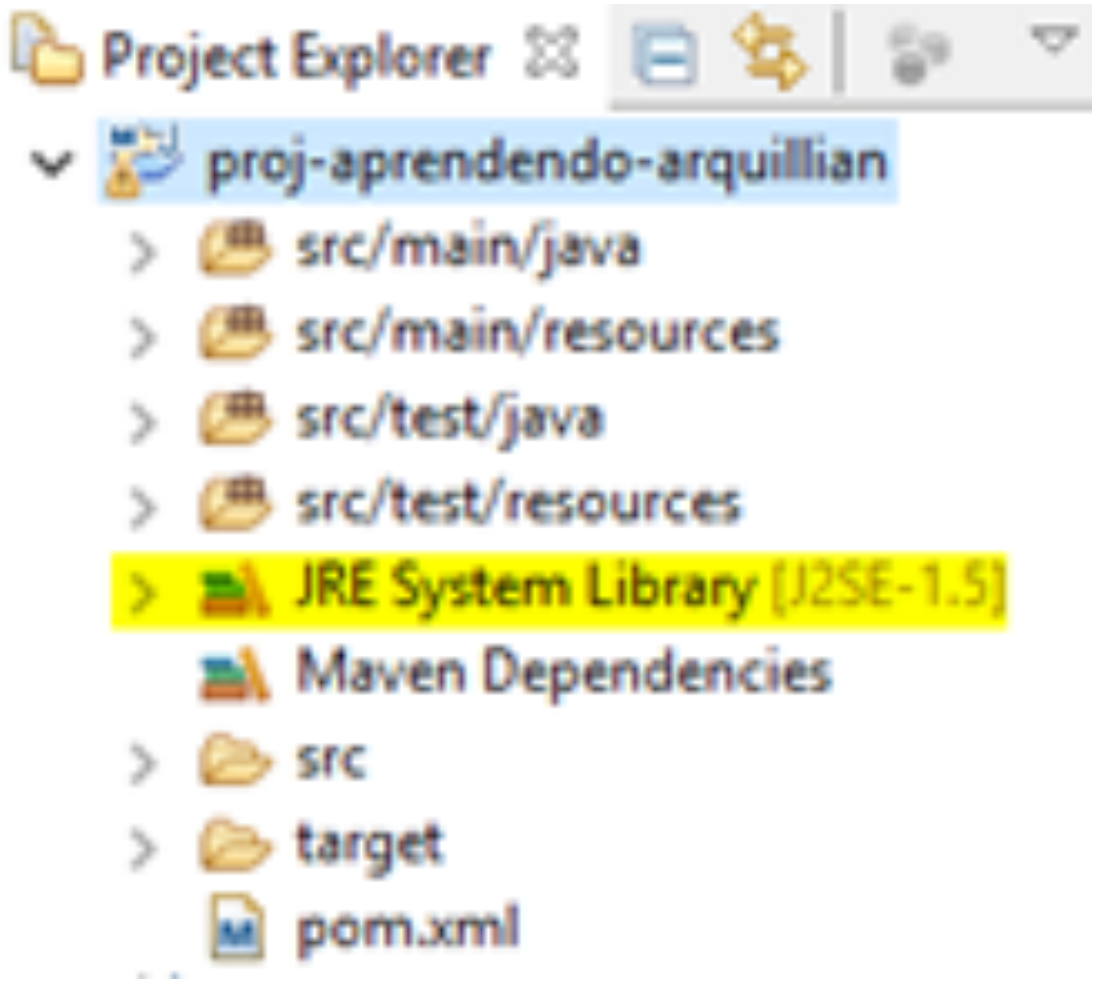

Figura 3. Projeto Maven utilizando o Java 1.5

Abra o arquivo **pom.xml** e acrescente o código da **Listagem 1** entre as tags para que possamos utilizar a versão do Java 1.8 no projeto Maven (criado anteriormente).

Listagem 1. Código do build com a configuração de compilar o projeto Maven com o Java 1.8

```
<build>
      <plugins>
            <plugin>
                  <artifactId>maven-compiler-plugin</artifactId>
                  <!-- Versão do plugin Maven -->
                  <version>3.1</version>
                  <configuration>
                  <!-- Versão do Java -->
                        <source>1.8</source>
                        <target>1.8</target>
                  </configuration>
            </plugin>
      </plugins>
</build>
```

Logo depois, selecione o projeto e execute o Maven Update Project utilizando o Alt+F5 ou com o botão direito do mouse escolhendo a opção "Maven > Update Project". Feito isso, o projeto Maven já está apto para utilizar o Java 1.8, como mostra a **Figura 4**.

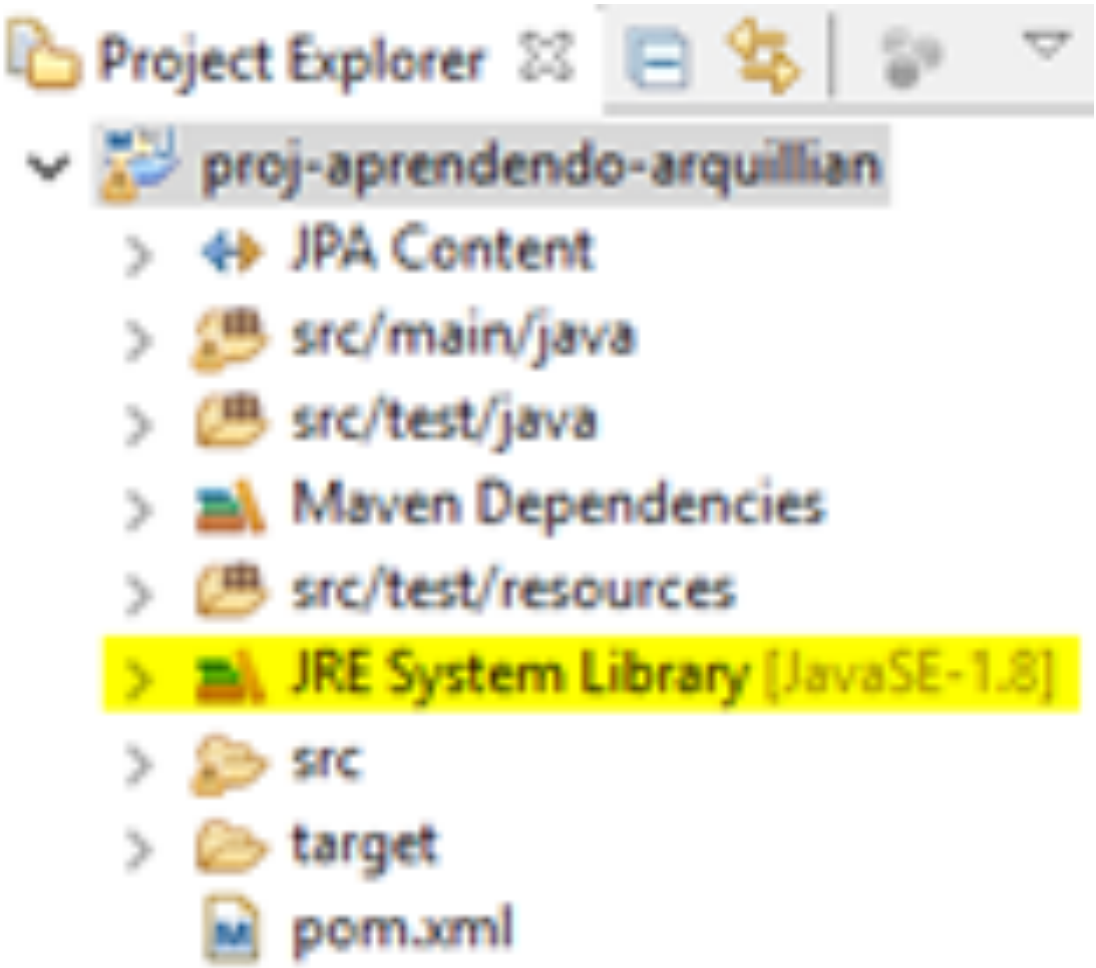

Figura 4. Projeto Maven com o Java 1.8

Ainda no arquivo **pom.xml** acrescente o código da **Listagem 2** para o projeto Maven realizar os testes com o Arquillian.

Listagem 2. Dependências necessárias

```
<dependencyManagement>
      <dependencies>
            <!-- Arquillian Jboss -->
            <dependency>
                  <groupId>org.jboss.arquillian</groupId>
                  <artifactId>arquillian-bom</artifactId>
                  <version>1.1.4.Final</version>
                  <type>pom</type>
                  <scope>import</scope>
            </dependency>
      </dependencies>
</dependencyManagement>

<dependencies>
      <!-- Arquillian JUnit Container -->
      <dependency>
            <groupId>org.jboss.arquillian.junit</groupId>
            <artifactId>arquillian-junit-container</artifactId>
            <scope>test</scope>
      </dependency>
            <!-- JUnit -->
      <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
<scope>test</scope>
      </dependency>
      <!-- Java EE 7 API -->
      <dependency>
            <groupId>javax</groupId>
```

```
            <artifactId>javaee-api</artifactId>
            <version>7.0</version>
      </dependency>
</dependencies>
```

Precisamos adicionar ainda no **pom.xml** o perfil que contém o link para o container que irá ser chamado pelo Arquillian. Assim, acrescente o código da **Listagem 3**.

Listagem 3. Perfil do WildFly 8.1.0 gerenciado no arquivo pom.xml

```
<profiles>
      <!-- Perfil do WildFly 8.1.0 do tipo gerenciado -->
            <profile>
                  <id>jboss</id>
                  <dependencies>
                        <dependency>
                              <groupId>org.wildfly</groupId>
                              <artifactId>wildfly-arquillian-container-managed</artifactId>
                              <version>8.1.0.Final</version>
                              <scope>test</scope>
                              <!-- Exclui a chamada desnecessária do Jconsole (não vamos utilizar) -->
                              <exclusions>
                                    <exclusion>
                                          <groupId>sun.jdk</groupId>
                                          <artifactId>jconsole</artifactId>
                                    </exclusion>
                              </exclusions>
                        </dependency>
                  </dependencies>
            </profile>
      </profiles>
```

Note que o código contém entre as tags o perfil (chamado jboss) que aponta para o container do WildFly 8.1.0 do tipo gerenciado. Caso deseje, podem ser adicionados outros perfis para chamar outros containers. Para usarmos esse perfil primeiro selecionamos o projeto Maven e depois com o botão direito do mouse escolha a opção "Maven > Select Maven Profiles" ou pressione Ctrl+Alt+P, obtendo a janela da **Figura 5.** Por fim marque o perfil "jboss" e clique em **ok**.

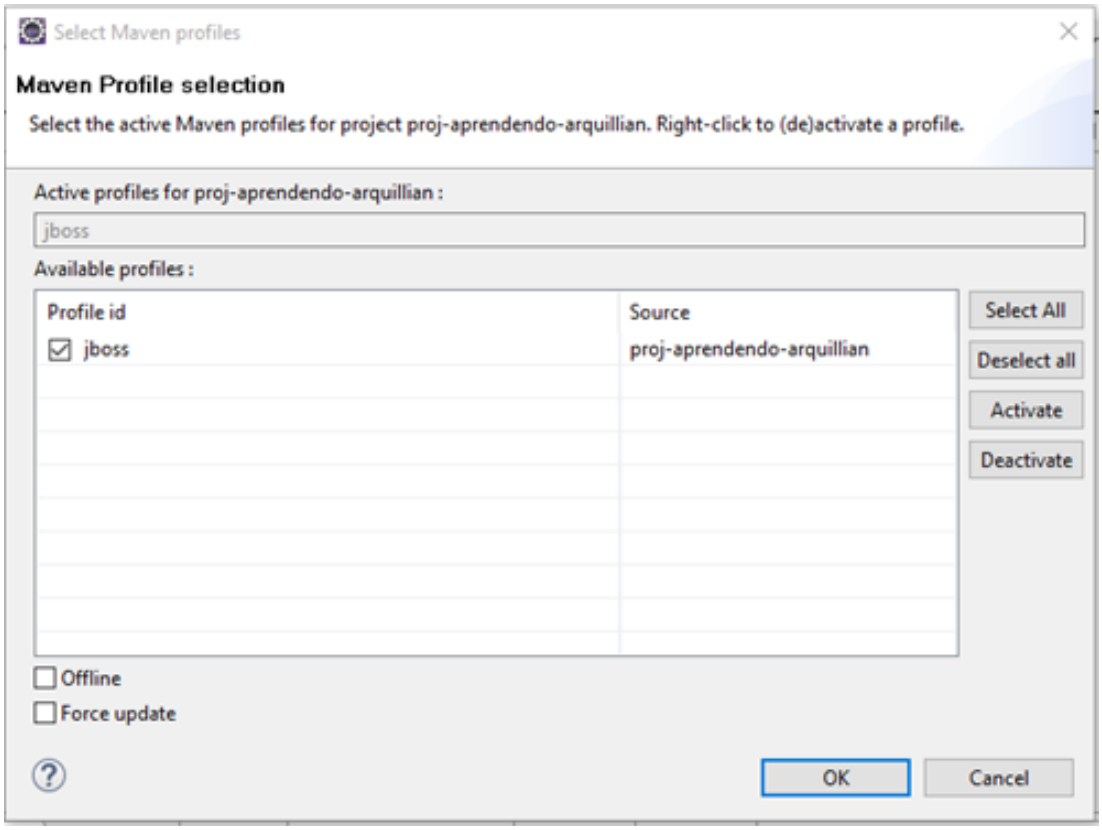

Figura 5. Janela do Eclipse para escolher o perfil (ou profile) do Maven

Caso você não tenha o WildFly no computador, este container que estamos utilizando é da versão 1.8.0.final. Para completar a configuração, crie um arquivo chamado "arquillian.xml" no diretório \src\test\resources**arquillian.xml** colocando o conteúdo da **Listagem 4**. Lembre-se de alterar o

caminho do container (WildFly) na tag "jbossHome".

Listagem 4. Configuração do arquivo arquillian.xml contendo a URLde localização do WildFly 1.8.0.final (localizado no computador).

```
<?xml version="1.0" encoding="UTF-8"?>
<arquillian xmlns="http://jboss.org/schema/arquillian"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://jboss.org/schema/arquillian
       http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

      <container qualifier="jboss" default="true">
            <configuration>
                  <property name="jbossHome"><coloque aqui o diretório absoluto ou relativo do WildFly >\wildfly-8.1.0.Final</property>
            </configuration>
      </container>
</arquillian>
```

Observe que no arquivo **pom.xml** que contém o perfil do WildFly tem o id "jboss", que por sua vez tem uma ligação com qualifier "jboss" do arquivo **arquillian.xml** da **Listagem 4**. Contudo, é importante manter o mesmo nome, pois ajuda o Arquillian na identificação do container a ser chamado para execução.

E para concluir a configuração do projeto precisamos criar um arquivo vazio chamado **beans.xml** no diretório /src/main/resources/META-INF/**beans.xml** para que possamos utilizar o CDI. Por fim, para utilizarmos o JPA crie o arquivo chamado **persistence.xml** no mesmo diretório do **beans.xml** com o conteúdo da **Listagem 5**.

Listagem 5. Arquivo persistence.xml com as configurações de conexão.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
      xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
      <persistence-unit name="teste">

            <!-- Conexão com o banco de dados H2 utilizando o JTA -->
            <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>

            <properties>
                  <!-- Atualiza o banco, gera as tabelas se for preciso -->
                  <property name="hibernate.hbm2ddl.auto" value="update" />
                  <!-- Mostrando o sql gerado pelo hibernate -->
                  <property name="hibernate.show_sql" value="true" />
                  <!-- Mostrando o sql gerado pelo hibernate -->
                  <property name="hibernate.show_sql" value="true" />
                  <!-- Indentação do código sql gerado -->
                  <property name="hibernate.format_sql" value="true" />

            </properties>
      </persistence-unit>
</persistence>
```

Note que no arquivo **persistence.xml** estamos utilizando a conexão com o banco de dados H2 já pré-configurada no WildFly, caso queria mudar para outro datasource (por exemplo MySQL, PostgreSQL e outros) é só alterar e colocar entre as tags. Veja como fazer em um artigo da DevMedia, na seção "Configurando um datasource".

Desenvolvendo os testes utilizando CDI, EJB e JPA com Arquillian

Após realizadas as configurações necessárias, por uma questão de organização, precisamos criar alguns pacotes. Com isso, selecione o diretório (/src/main/java) e crie um pacote chamado "br.com.aprendendo.arquillian.modelo", e em seguida crie uma classe (ou entidade) chamada "Pessoa.java" com o conteúdo da **Listagem 6**.

Listagem 6. Código da classe Pessoa.java

```
@Entity
@Table(name = "pessoa")
public class Pessoa implements Serializable {

      private static final long serialVersionUID = 1L;

      @Id
      @GeneratedValue
      @Column(name = "id")
      private int id;
      @Column(name = "nome", length = 60)
      private String nome;
      @Column(name = "idade")
      private Integer idade;

      public int getId() {
            return id;
```

```
        }

        public void setId(int id) {
              this.id = id;
        }

        public String getNome() {
              return nome;
        }

        public void setNome(String nome) {
              this.nome = nome;
        }

        public Integer getIdade() {
              return idade;
        }

        public void setIdade(Integer idade) {
              this.idade = idade;
        }
  }
```

Temos que adicionar a classe Pessoa.java no arquivo **persistence.xml** entre as tags (por exemplo, utilize assim br.com.aprendendo.arquillian.modelo.Pessoa) para que esta entidade seja gerenciada.

Na sequência, selecione o diretório (/src/main/java) e crie o pacote chamado "br.com.aprendendo.arquillian.dao" e nele crie uma classe chamada "PessoaDao.java" contendo o código da **Listagem 7.**

Listagem 7. Código da classe PessoaDao.java

```
  @Stateless
  public class PessoaDao {

        @PersistenceContext(unitName = "teste")
        EntityManager em;

        public void salvar(Pessoa p) {
              em.persist(p);
        }

        public void atualizar(Pessoa p) {
              em.merge(p);
        }

        public Pessoa buscar(int id) {
              return em.find(Pessoa.class, id);
        }

        public List<Pessoa> buscarTodasPessoas() {
              return em.createQuery("SELECT p FROM Pessoa p ORDER BY p.id", Pessoa.class).getResultList();
        }
  }
```

Agora criaremos outra classe para testar a classe PessoaDao para depois executar os testes utilizando o servidor gerenciado WildFly 8.1.0. Como o foco deste artigo é o Arquillian, vamos detalhar alguns pontos importantes sobre ele adiante.

No diretório /src/test/java crie um pacote chamado "br.com.aprendendo.arquillian.dao", e nele crie uma classe chamada "PessoaDaoTeste.java" acrescentando o código da **Listagem 8**.

Listagem 8. Código para o funcionamento do Arquillian

```
@RunWith(Arquillian.class)
public class PessoaDaoTeste{
  @Deployment
 public static Archive<?> criarArquivoTeste() {
  Archive<?> arquivoTeste = ShrinkWrap.create(WebArchive.class, "aplicacaoTeste.war")
  // Adicionando o pacote inteiro da classe PessoaDao, ou seja incluí todas as outras classes deste pacote
  .addPackage(PessoaDao.class.getPackage())
  // Adicionando apenas a classe Pessoa, e não o pacote inteiro como na linha anterior
  .addClass(Pessoa.class)
  // Adicionando o arquivo persistence.xml para conexão JPA
  .addAsResource("META-INF/persistence.xml")
  // Adicionando o beans.xml para ativação do CDI
  .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml");
  return arquivoTeste;
 }
}
```

Note que na classe PessoaDaoTeste a anotação "@RunWith" diz ao JUnit para usar o Arquillian como o controlador de teste. Outro ponto a destacar

é a anotação "@Deployment", que faz com o que o Arquillian execute o método estático "criarArquivoTeste()", que realiza a implantação incluindo todas as classes especificadas e recursos (em vez de implantar o aplicativo inteiro), considerando que essa parte de incluir esses arquivos é de responsabilidade da API ShrinkWrap.

Perceba que na **Listagem 8,** em vez de incluímos classe por classe (por exemplo, o método .addClass(Pessoa.class)), usamos o método AddPackage, que adiciona todas as classes que estão contidas no mesmo pacote de classe, como está na PessaoDaoTeste com inclusão do método .addPackage(PessoaDao.class.getPackage()), ou seja, este método irá incluir todas as outras classes que estão no mesmo pacote da classe PessoaDao. Por outro lado, pode ser feito também adicionando classe por classe por meio de um único método chamado .addClasses, como por exemplo:

```
.addClasses(Pessoa.class, PessoaDao.class)
```

E para podermos utilizar o CDI nos testes acrescentamos o método .addAsWebInfResource(EmptyAsset.*INSTANCE*, "beans.xml"), e do mesmo jeito acontece com o JPA incluindo o método .addAsResource("META-INF/persistence.xml"). Esses dos métodos dizem ao ShrinkWrap para incluir esses arquivos de configuração dentro da aplicação "aplicacaoTeste.war".

Agora podemos desfrutar do poder que o Arquillian nos proporciona. Lembrando que podemos escrever os testes utilizando os recursos do Java EE 7 normalmente, dentro de uma classe de teste. Vale destacar que anteriormente criamos as classes "PessoaDao" e "PessoaDaoTeste" só por uma questão de responsabilidades, mas isso não impede que em uma classe só utilize os recursos do Java EE, mas a exigência que se tem é que a classe esteja configurada com o Arquillian, como mostrou a **Listagem 8**.

Agora adicione a anotação @Inject acima da variável pessoaDao para podermos utilizar a injeção de dependência do CDI, como mostra o código a seguir:

```
  @Inject
  PessoaDao pessoaDao;
```

Pode acontecer que, ao invés de utilizarmos o @Inject, a mesma possa ser substituída pela a anotação @EJB, pois a classe PessoaDao está marcada com a anotação @Stateless. No entanto, utilizaremos o @Inject mesmo, porque o CDI, através do @Inject, permite injetar objetos sejam eles EJBs ou não. Por fim, acrescente os testes da **Listagem 9** na classe PessaoDaoTest.java.

Listagem 9. Código com os testes

```
@Test
@InSequence(1)
public void testeSalvarPessoa() {
   Pessoa p1 = new Pessoa();
   p1.setIdade(10);
   p1.setNome("Patrício Neto");
   pessoaDao.salvar(p1);

   Pessoa p2 = new Pessoa();
   p2.setIdade(21);
   p2.setNome("Brendo Felipe");
   pessoaDao.salvar(p2);

}

@Test
@InSequence(2)
public void testeAtualizarPessoaP1() {
   Pessoa p1 = pessoaDao.buscar(1);
   p1.setNome("Pedro");
   p1.setIdade(11);
   pessoaDao.atualizar(p1);

   assertEquals("Pedro", p1.getNome());
   assertEquals(11, p1.getIdade().intValue());

}

@Test
@InSequence(3)
public void testeBuscarPessoaP2() {
   Pessoa p2 = pessoaDao.buscar(2);

   assertEquals("Brendo Felipe", p2.getNome());
   assertEquals(21, p2.getIdade().intValue());
}

@Test
@InSequence(4)
public void testeBuscarTodasPessoas() {
    List<Pessoa> pessoas = pessoaDao.buscarTodasPessoas();
    assertEquals(2, pessoas.size());
}
```

Perceba que no código utilizamos a anotação @Test, que diz ao JUnit que o método anotado deve ser executado durante os testes. Além disso,

ditamos a ordem de execução deles por meio da anotação @InSequence().

Para executá-los selecione a classe de teste "PessoaDaoTeste" e depois de clicar com o botão direito do mouse selecione a opção "Run As > JUnit Test".

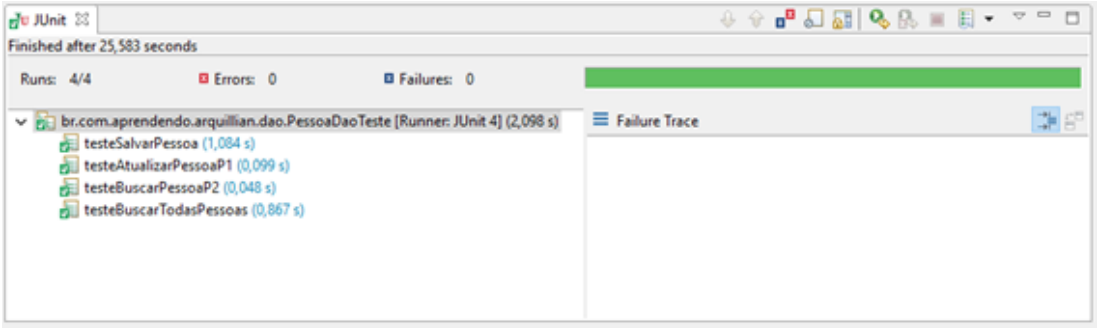

Figura 6. Resultado dos testes com Arquillian e JUnit

Como mostra a **Figura 6,** todos os testes estão marcados com verde, o que significa que todos passaram, e assim conseguimos o feito de executar os nossos testes como se estivessem em produção. Diante disso, percebemos a grande vantagem do Arquillian, que é a de proporcionar e dar a liberdade para desenvolvedor e utilizar os recursos do Java EE nos testes, e assim é considerado um excelente framework de apoio a teste.

Espero que este artigo tenha ajudado e contribuído para a construção e aplicação do conhecimento sobre o Arquillian.

Obrigado e até mais!

BIBLIOGRAFIA

ĆMIL, MICHAL; MATLOKA, Michał; MARCHIONI, Francesco. **Java EE 7 Development with WildFly.** 2. ed. Birmingham: Packt Publishing Ltd., 2014.

AMENT, John D.. **Arquillian Testing Guide.** 1. ed. Birmingham: Packt Publishing Ltd., 2013.

RUBINGER, Andrew Lee; KNUTSEN, Aslak. **Continuous Enterprise Development in Java.** 1. ed. Sebastopol: O'Reilly Media, 2014.

JUnit
http://junit.org/

Maven
https://maven.apache.org/

Arquillian
http://arquillian.org/

Documentação do Arquillian
https://docs.jboss.org/author/display/ARQ/Container+adapters?_sscc=t

Brendo Felipe Rodrigues Arcanjo