ApplicationComposer

the swiss knife of TomEE

ApplicationComposer API is mainly contained in org.apache.openejb.testing package (historically, today we would have called the package org.apache.tomee.applicationcomposer).

Dependencies

To start using ApplicationComposer you need to add some dependencies.

The minimum required one is openejb-core:

```
<dependency>
  <groupId>org.apache.tomee</groupId>
  <artifactId>openejb-core</artifactId>
  <version>${openejb.version}</version>
</dependency>
```

If you need JAXRS services you'll add (or replace thanks to transitivity of maven) openejb-cxf-rs:

```
<dependency>
  <groupId>org.apache.tomee</groupId>
  <artifactId>openejb-cxf-rs</artifactId>
  <version>${openejb.version}</version>
</dependency>
```

If you need JAXWS services you'll add (or replace thanks to transitivity of maven) openejb-cxf:

```
<dependency>
  <groupId>org.apache.tomee</groupId>
  <artifactId>openejb-cxf</artifactId>
  <version>${openejb.version}</version>
</dependency>
```

ApplicationComposer Components

@Module

An ApplicationComposer needs at minimum a module (the application you need to deploy).

To do so you have two cases:

before TomEE 7.x: you can only write method(s) decorated with @Module since TomEE 7.x: you can skip it and use @Classes directly on the ApplicationComposer class as a shortcut for:

```
@Module public WebApp app() { return new WebApp(); }
```

The expected returned type of these methods are in org.apache.openejb.jee package:

- Application: entry point to create an ear
- WebApp: a web application
- EjbJar: an ejb module
- EnterpriseBean children: a simple EJB
- Persistence: a persistence module with multiple units
- PersistenceUnit: a simple unit (automatically wrapped in a Persistence)
- Connector: a JCA connector module
- Beans: a CDI module,
- Class[] or Class: a set of classes scanned to discover annotations

Note that for easiness @Classes was added to be able to describe a module and some scanned classes. For instance the following snippet will create a web application with classes C1, C2 as CDI beans and E1 as an EJB automatically:

```
@Module
@Classes(cdi = true, value = { C1.class, C2.class, E1.class })
public WebApp app() {
    return new WebApp();
}
```

@Configuration

Often you need to customize a bit the container or at least create some resources like test databases. To do so you can create a method returning Properties which will be the container properties.

Note: to simplify writing properties you can use PropertiesBuilder util class which is just a fluent API to write properties.

In these properties you can reuse OpenEJB/TomEE property syntax for resources.

Here is a sample:

```
@Configuration
public Properties configuration() {
    return new PropertiesBuilder()
        .p("db", "new://Resource?type=DataSource")
        .p("db.JdbcUrld", "jdbc:hsqldb:mem:test")
        .build();
}
```

Since TomEE 7.x you can also put properties on ApplicationComposer class using @ContainerProperties API:

```
@ContainerProperties({
  @ContainerProperties.Property(name = "db", value = "new://Resource?type=DataSource"
),
  @ContainerProperties.Property(name = "db.JdbcUrl", value = "jdbc:hsqldb:mem:test")
})
public class MyAppComposer() {
  // ...
}
```

@Component

Sometimes you need to customize a container component. The most common use case is the security service to mock a little bit authorization if you don't care in your test.

To do so just write a method decorated with @Component returning the instance you desire.

Components in TomEE are stored in a container Map and the key needs to be a Class. This one is deduced from the returned type of the @Component method:

```
@Component
public SecurityService mockSecurity() {
    return new MySecurityService();
}
```

@Descriptors

You can reuse existing file descriptors using @Descriptors. The name is the file name and the path either a classpath path or a file path:

```
// runner if needed etc...
@Descriptors(@Descriptor(name = "persistence.xml", path = "META-INF/persistence.xml"))
public class MyTest {
   //...
}
```

Note: this can be put in a @Module method as well.

Services

If you want to test a JAXRS or JAXWS service you need to activate these services.

To do so just add the needed dependency and use @EnableServices:

```
// runner if needed etc...
@EnableService("jaxrs") // jaxws supported as well
public class MyTest {
   //...
}
```

Random port

Services like JAXRS and JAXWS relies on HTTP. Often it is nice to have a random port to be able to deploy multiple tests/projects on the same CI platform at the same time.

To shortcut all the needed logic you can use @RandomPort. It is simply an injection giving you either the port (int) or the root context (URL):

```
// runner, services if needed etc...
public class MyTest {
   @RandomPort("http")
   private int port;
}
```

Note: you can generate this way multiple ports. The value is the name of the service it will apply on (being said http is an alias for httpejbd which is our embedded http layer).

Nice logs

@SimpleLog annotation allows you to have one liner logs

@JaxrsProvider

@JaxrsProvider allows you to specify on a @Module method the list of JAXRS provider you want to use.

Dependencies without hacky code

@Jars allows you to add dependencies (scanned) to your application automatically (like CDI libraries):

```
@Module
@Classes(cdi = true, value = { C1.class, C2.class, E1.class })
@Jars("deltaspike-")
public WebApp app() {
    return new WebApp();
}
```

@Default

@Default (openejb one not CDI one) automatically adds in the application target/classes as binaries and src/main/webapp as resources for maven projects.

@CdiExtensions

This annotation allows you to control which extensions are activated during the test.

@AppResource

This annotation allows injection of few particular test resources like:

the test AppModule (application meta) the test Context (JNDI) the test ApplicationComposers (underlying runner) ContextProvider: allow to mock JAXRS contexts

@MockInjector

Allows to mock EJB injections. It decorates a dedicated method returning an instance (or Class) implementing FallbackPropertyInjector.

@WebResource

Allow for web application to add folders containing web resources.

How to run it?

JUnit

If you use JUnit you have mainly 2 solutions to run you "model" using the ApplicationComposer:

using ApplicationComposer runner:

```
@RunWith(ApplicationComposer.class) public class MyTest { // ... }
```

using ApplicationComposerRule rule: public class MyTest { @Rule // or @ClassRule if you want the container/application lifecycle be bound to the class and not test methods public final ApplicationComposerRule rule = new ApplicationComposerRule(this); }

Tip: since TomEE 7.x ApplicationComposerRule is decomposed in 2 rules if you need: ContainerRule and DeployApplication. Using JUnit RuleChain you can chain them to get the samebehavior as ApplicationComposerRule or better deploy multiple ApplicationComposer models and controlling their deployment ordering (to mock a remote service for instance).

Finally just write `@Test` method using test class injections as if the test class was a managed bean!

TestNG

TestNG integration is quite simple today and mainly ApplicationComposerListener class you can configure as a listener to get ApplicationComposer features.

Finally just write TestNG @Test method using test class injections as if the test class was a managed bean!

Standalone

Since TomEE 7.x you can also use ApplicationComposers to directly run you ApplicationComposer model as a standalone application:

```
public class MyApp {
    public static void main(String[] args) {
        ApplicationComposers.run(MyApp.class, args);
    }

    // @Module, @Configuration etc...
}
```

Tip: if MyApp has `@PostConstruct` methods they will be respected and if MyApp has a constructor taking an array of String it will be instantiated getting the second parameter as argument (ie you can propagate your main parameter to your model to modify your application depending it!)

JUnit Sample

```
@Classes(cdi = true, value = { MyService.class, MyOtherService.class })
@ContainerProperties(@ContainerProperties.Property(name = "myDb", value =
"new://Resource?type=DataSource"))
@RunWith(ApplicationComposer.class)
public class MyTest {
    @Resource(name = "myDb")
    private DataSource ds;

    @Inject
    private MyService service;

    @Test
    public void myTest() {
        // do test using injections
    }
}
```

Start and Deploy once

When having a huge suite of test it can be long to start/deploy/undeploy/shutdown he

container/application for each method.

That's why `SingleApplicationComposerRunner` allows to just reuse the same instance accross several test.

The first test will start and deploy the application and then other tests will reuse this instance until the JVM is destroyed where the server/application will be undeployed/shutdown.

Here a simple usage:

```
import org.apache.openejb.testing.SingleApplicationComposerRunner;
// other imports

@RunWith(SingleApplicationComposerRunner.class)
public class MyTest {
    @Inject
    private ACdiBean bean;

    @Application
    private TheModel model;

    @Test
    public void aTest() {
        // ...
    }
}
```

TIP if you need a real TomEE container then you can have a look to `TomEEEmbeddedSingleRunner` which does deploys the classpath using tomee-embedded.

Configure what to deploy

As all tests will reuse the same application the model (the class declaring the application with `@Classes`, `@Module` etc...) needs to be extracted from the test class itself.

The application lookup uses this strategy (ordered):

- the fully qualified name is read from the system property `tomee.application-composer.application`
- a **single** class decorated with `@Application` is looked in the jar/folder containing the test class

If you have several "groups" you can use JUnit `@Category` to differentiate them and write one application class by category. Then in `surefire` plugin you declare two `executions` enforcing the system property `tomee.application-composer.application` for each of them and the associated `@Category`.

Available injections

- If the application model class uses `@RandomPort` then the test classes can get it as well

- CDI injections are supported
- `@Application` on a field allows to get the application model to get injected

Compared to a standalone usage it misses all other EE injections (`@PersistenceContext`, `@Resource` etc... but you can inject them in the application model and just expose them or wrap them in your tests thanks to the `@Application` field.

Going further

If you want to learn more about ApplicationComposer see ApplicationComposer Advanced page.