View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Lodz University of Technology Repository

JOURNAL OF APPLIED
COMPUTER SCIENCE
Vol. 21 No. 1 (2013), pp. 53-69

Running and Testing Java EE Applications in Embedded Mode with JupEEter Framework

Marcin Kwapisz[1]

[1]*Technical University of Lodz*
Faculty of Technical Physics, Information Technology and Applied Mathematics
Institute of Information Technology
ul. Wolczanska 215, 90-924 Lodz
marcin.kwapisz@p.lodz.pl

Abstract. *This paper presents a design and usage of the author's innovative framework, called JupEEter. This framework helps running and testing Java Enterprise Edition (Java EE) applications [1] and to use Java EE components in Java SE [2] applications. The framework defines the application server and application life-cycle and exploits annotation based programming technique for its configuration.*
Keywords: *integration testing, unit testing, Java EE, embedded.*

1. Introduction

Java EE [1] applications require an application server to run. This is the main problem during unit or integration testing of an application. An application server and its containers deliver many services, e.g.:

- dependency injection,
- transaction management,

- security management,
- timers, etc,

that are not available in standard Java SE environment. Testing components that depend on mentioned services is very difficult. A developer must deliver these services to tested components, what can be sometimes impossible. Mocking them is pointless because tested components may behave differently in a testing environment than a production one.

The main goal of presented JupEEter framework is to help developers to test and run Java EE applications in a Java SE environment and to allow to use Java EE components in Java SE applications.

2. Service layers of the framework

The design of the framework is based on multi-tier architecture and is depicted on Figure 1.

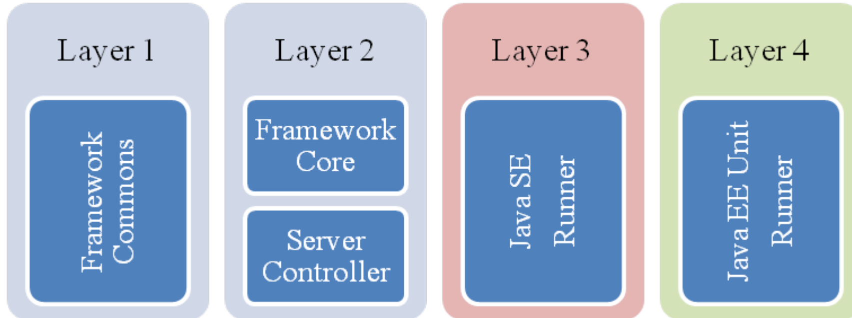

Figure 1. Framework design

One of the main advantage of multi-tier architecture is a possibility to replace, add or modify one layer instead of entire application or a framework. Developers can even insert a new layer between existing ones. This feature is widely used in the framework. For example, the *Java EE Unit Runner* is the integration layer of the framework with JUnit and makes possible to test Java EE applications with it. To test applications with TestNG, *Java EE Unit Runner* components have to be replaced with other ones, which integrate the framework with TestNG. Both, JUnit

and TestNG integration components from layer 4 must use Java Se Runner to work properly.

On the other hand, to run or test Java EE applications on a different application server than Glassfish Server [3], existing implementation of the *Server Controller* component from the layer 2 must be replaced with another implementation that is able to control that application server. The design, configuration and usage of all layers and their components from figure 1 are described in the following subsections and sections.

2.1. Layer 1 - Framework Commons components

This is the basic data definition layer layer that contains only common data types and interfaces used by the whole framework:

- ***ServerController*** - is an interface that must be provided by all components used to control different Java EE application servers. The framework provides DefaultServerController (see section 2.2) for a fail-over mechanism and a specialized controllers for Glassfish v3 and v3.x Application Servers (see section 2.3),

- ***ApplicationAssembler*** - is an interface that must be provided by all components used to prepare and assemble a Java EE application for deployment. These components have to be provided by developers of the framework along with server controllers because deployment of a Java EE application depends on a type of an application server and a type of an application: web (WAR), enterprise(EAR) or ejb (JAR).

The last worth mentioning class of this layer is *ApplicationServerContext*. Instances of this class store a configuration of an application server and of an Java EE applications. They are filled with data collected during processing of the framework annotations (see section 2.2), for example:

- Java EE Server identifier,
- network host name and port configuration,
- list of resources to install,
- list of application to deploy,
- security configuration and etc.

2.2. Layer 2 - Framework Core and Server Controller

This is the configuration and service layer of the framework. It provides basic services for the third layer *Java Se Runner* that can run applications in Java SE environment (see section 3.1)

The configuration of the framework is based on annotations. This is the one of the features that distinct it form another similar tool that is based on XML configuration[4]. Framework annotations can be used on a: class, method or a field level. Each annotation is paired with corresponding annotation processor, responsible for processing it and filling up *ApplicationServerContex* object. A name of an annotation processor is an annotation name with suffix *Processor*:

- ***@ApplicationServer*** (processed by *ApplicationServerProcessor*) - the class-level annotation that MUST be used. This annotation identifies and configures application server instance. All its attributes have default values, so it can be placed in a source code without any explicitly set ones. Probably in the next release of the framework this annotation will be optional and default configuration will be provided. An example usage of this annotation shows listing 1. The main attributes:
  - **security** - security realm configuration. A security realm contains user credentials for authentication and authorization,
  - **resources** - set of external files that contain configurations of application server resources required for an application to run properly, like for example: JDBC resources and connection pools,
  - **instancePorts** - network configuration of an application server. This configuration is required if an application have to be accessed remotely through a computer network.
- ***@Applications*** (processed by *ApplicationsProcesor*) - the class-level annotation that MUST be used. This annotation contains an array of *@Application* annotations that provide basic configuration for applications to be assembled and deployed on a target application server. Attributes:
  - **applicationPath** - path to the Java EE application in a form of a scattered archive (directory path) or WAR/JAR/EAR archive (file path),
  - **applicationType** - type of an application to be deployed. This attribute value can be set to WAR - web application, JAR - EJB module, EAR - enterprise application,

- **modules** - array of Java EE modules, that are parts of a web or an enterprise application. @*Module* annotation configuration is similar to *Application* one.

Listing 1. Sample usage of the @*ApplicationServer* and @*Applications* annotations

```
1  @ApplicationServer(instanceName = "domain2",
2  instanceParentPath = "target/glassfish",
3  security = @Security(securityRealmName = "test"),
4  resources = @AppResource({"resources.xml"}),
5  instancePorts = @Ports(
6    http=8090, https=8091, iiop=3800, admin=4949,
         iiops=3900)
7  )
8  @Applications(
9    applications = {
10     @Application(target="domain2",
11       name="classes",
12       applicationPath="target",
13       applicationType=ApplicationType.EAR,
14   modules = {
15     @Module(name="app",
16       modulePath="target/classes",
17       moduleType=ApplicationType.EJB),
18     @Module(name="tests",
19       modulePath="target/test-classes",
20       moduleType=ApplicationType.JAR)
21   })}
22 )
23 public class GlassfishServerTestSuite
```

An example usage of the @*Applications* annotation shows listing 1 starting from line 9. It is a configuration of an enterprise application that contains two modules:

- scattered EJB module with EJB components and its classes are located in *target/classes* directory,

- scattered JAR module (a standard Java library) and its classes are located in *target/test-classes* directory.

Each annotation processor explicitly extends *AbstractAnnotationProcessor* and thus implicitly implements *AnnotationProcessor* interface (see Figure 2).

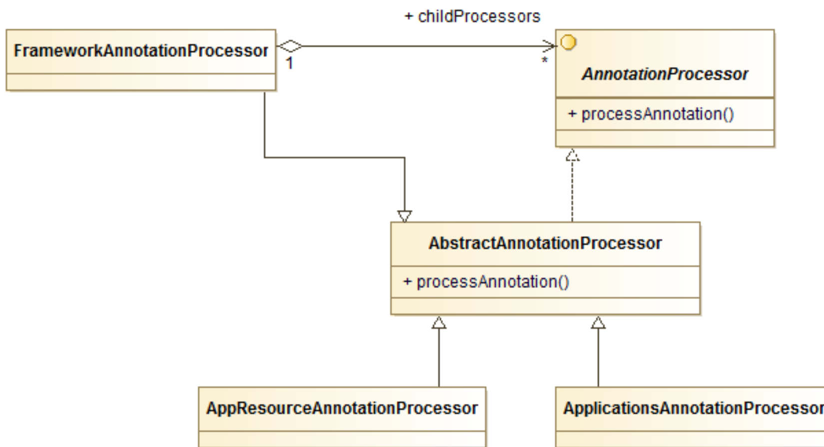

Figure 2. Annotation processors design

AbstractAnnotationProcessor forms a chain of child processors (it describes a node in a tree-like structure). As shown on listing 1, attributes of @Application-Server annotation are also annotations. These internal, second-level, annotations like: *@Security*, *@AppResource* and *@Ports* have corresponding annotation processors that are invoked by the first-level *ApplicationServerAnnotationProcessor* (see Figure 3).

All first-level annotation processors are put into root, top-level, annotation processor chain called *FrameworkAnnotationProcessor*. This class is a singleton and defines a chain (sequence) of first-level processors invocation. The basic chain of processors consist of:

- *ApplicationServerAnnotationProcessor*,
- *AppResourceAnnotationProcessor* (not shown on Figure 2),
- *ApplicationsAnnotationProcessor*.

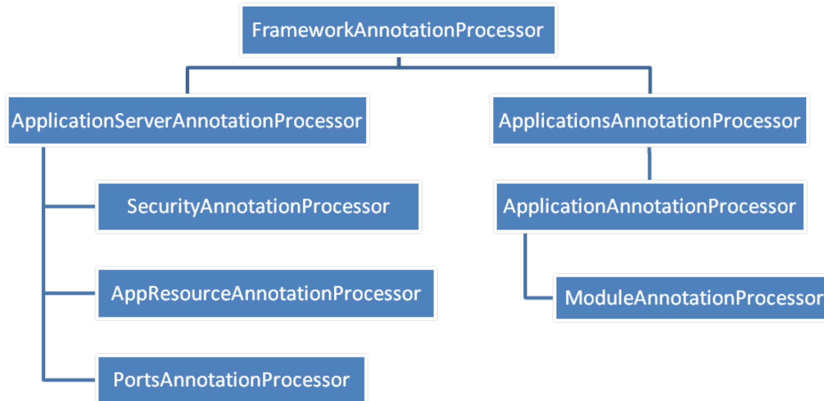

Figure 3. Annotation processors tree

The design, depicted on Figures 2 and 3 is very flexible. Modifications of existing annotations are closed only to its annotation processor classes. An addition of a new processor requires only inserting it into proper *FrameworkAnnotationProcessor* chain.

FrameworkAnnotationProcessor converts annotation-based configuration into single instance of *ApplicationServerContext* class. This instance is internally used during an application and an application server life-cycle. The layer 2 does not define any particular life-cycle, it provides only:

- commands (behavioral design pattern), classes implementing *FrameworkCommand* interface, that can be issued to the framework to call corresponding server controller method or methods
- and a factory that dynamically loads server controller implementation class for a given application server. In the case when the Java class loader cannot find and load a server controller class there is failover mechanism that loads default implementation called *DefaultServerController*. *DefaultServerController* does nothing more than just logging messages concerning the class loading problem.

Implementation elements described in section 2.1 and 2.2 are packed into single Java library file (JAR). The pluggable part, described in the next section (2.3) is

additional JAR library and is required to complete the functionality of the service layer 2.

2.3. Layer 2 - The pluggable server controller component - *Glassfish-Controller*

There are two server controller implementations currently in the framework. Both are able to control Glassfish Application Servers but one is designed for version 3 and the second one for version 3.x. That was necessary due to complete redesign of Glassfish Embedded API [5] in Glassfish 3.1. The *Glassfish Controller* library contains, except of implementation classes, additional resources, like the default Glassfish domain configuration file and XML templates required by the application server to validate deployment descriptors of Java EE applications.

This component is replaceable. A developer can implement a server controller that can control different Java EE application server, for example JBoss AS (JBoss Aplication Server), Tomcat (TomEE) and build additional server controller Java library. The design depicted on Figure 4 allows to use the framework with different application servers without any changes to the rest of the framework. A developer can even configure a project for the Glassfish at the beginning and then change an application server to JBoss AS, just by replacing the server controller library with the new one.

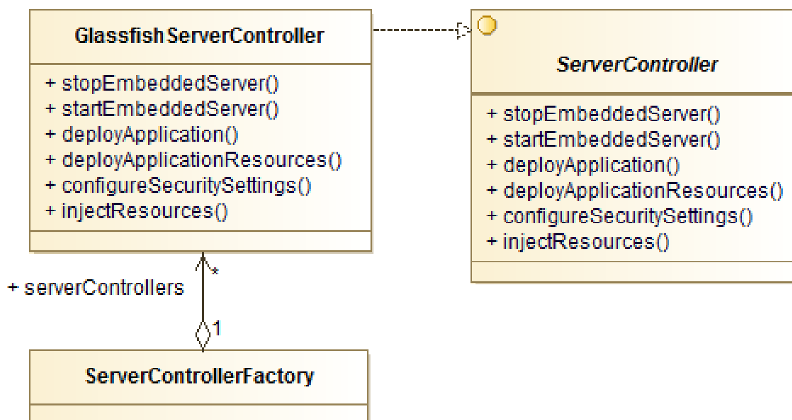

Figure 4. Integration of the *ServerController* implementation with the framework

The *Server Controller* interface and its implementation classes (see Fig.4) are responsible for:

- starting and stopping a Java EE application server in an embedded mode,
- deploying additional resources used by an application, for example: data sources and database connection pools,
- managing of security settings and user authentication (logging in and out),
- assembling and deploying of applications on a running embedded Java EE application server,
- accessing of Java EE resources and components deployed on an embedded Java EE application server.

3. Integration layer of the framework

3.1. Layer 3 - Java SE Runner

Java SE Runner is the main functional and the most complex layer of the framework. It provides features that make this framework completely distinct from others - allows to use Java EE components in a Java SE applications. It is possible by converting a Java SE application into a Java EE CDI application [6] and deploying it to an embedded Java EE 6 compliant application server. To control that server and deploy applications a *Server Controller* component described in the subsection 2.3 is required.

The layer 3 *Java Se Runner* defines:

- a Java EE application server life-cycle that spans on the life-cycle of a Java SE application (see Fig. 6),
- a new entry point to a Java SE application,
- an integration of a Java SE application with Java EE environment.

3.2. An application entry point and a server life-cycle

JVM specification [7] defines the entry point to a Java SE application. It is a method with the following signature (1):

static public void main(String[] args) (1)

A class containing that method has to be passed as a parameter to the JVM. From a technical point of view a Java SE application build with this framework is not strictly a Java SE application. It is assembled and deployed on an embedded Java EE application server by the *Server Controller* component. So it is a Java EE CDI [6] application and the *main* function will not work as the entry point any more.

Java SE Runner provides its own *main* method that must by passed to the JVM. This method starts an application server life-cycle (Fig. 6) and an application by firing the custom *RunJavaAppEvent* CDI event (Fig. 5). A Java SE application must register a method (2) that observes that event. The name of the method must only conform to Java language rules, but for convenience it was called *main*.

public void main(@Observes @JavaApp RunJavaAppEvent event) (2)

This method is called automatically by the CDI container when *RunJavaAppEvent* is fired. If a developer registers in a Java SE application more methods that observe *RunJavaAppEvent* all of them will be invoked.

Because Java SE application is a CDI one, all Java EE dependent components and resources can by automatically injected by annotating fields, setters or constructors of the application classes with the *@Inject* annotation [1].

As depicted on Fig. 6 after *DeployApplication* phase another application can be deployed along with its resources and security configuration. When all applications are deployed *RunJavaAppEvent* is fired to start them. When all applications processes finish *StopServer* phase is invoked to close an embedded Java EE application server and complete the applications life-cycle.

3.3. Security and transaction support

Java EE components and their methods can be invoked in a security and transactional context. Both functions are implemented in a similar way. The framework provides a CDI portable extension [6] with two methods that observe the same standard CDI event - *ProcessAnnotatedType*. This event is fired for each Java class

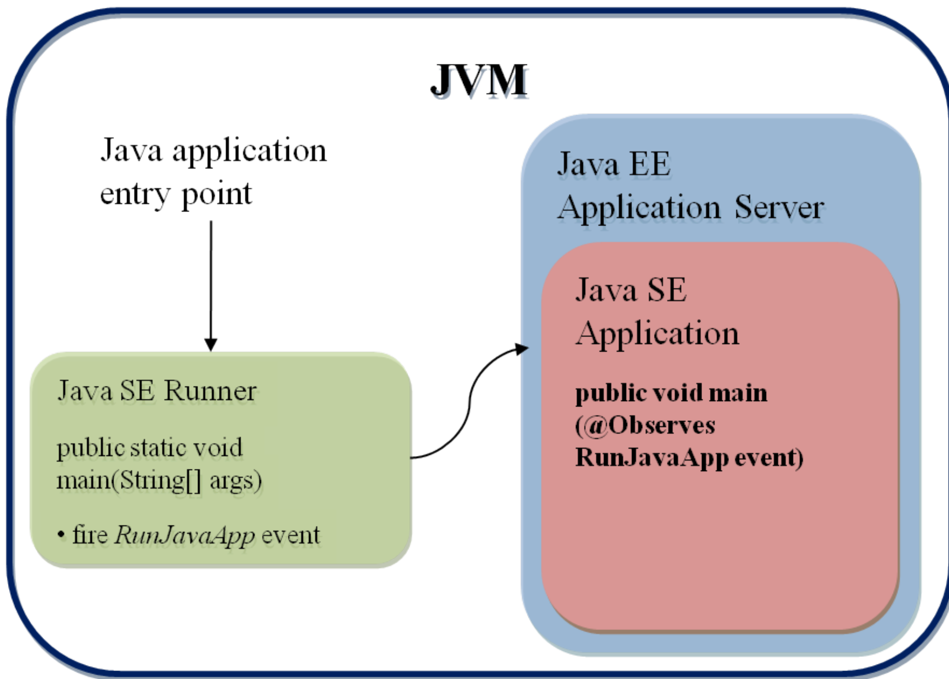

Figure 5. Java SE Runner application entry point

or interface that is discovered in an application. Developers can use this event to wrap or completely replace annotations of beans classes (*AnnotatedType*) before the CDI container builds their metamodel.

The first function scans beans classes for the @*RunAs* framework annotation. When an CDI *AnnotatedMethod* with this annotation is found it is wrapped to add *RunAsBinding* interceptor binding. Then the CDI container adds an *ArroudInvoke* interceptor called *RunAsInterceptor* to surround an invocation of the method. This interceptor simply authenticates and creates security context before an invocation of the target method and destroys it when the method finishes.

The second function works similarly, but scans beans classes for the @*Transactional* annotation and wraps *AnnotatedMethod* to add *TransactionalBinding* interceptor binding to it. *TransactionalBinding* annotation binds *TransactionalInterceptor* to the method. This interceptor creates a new transaction before an invocation of the target method and commits it when the method finishes.

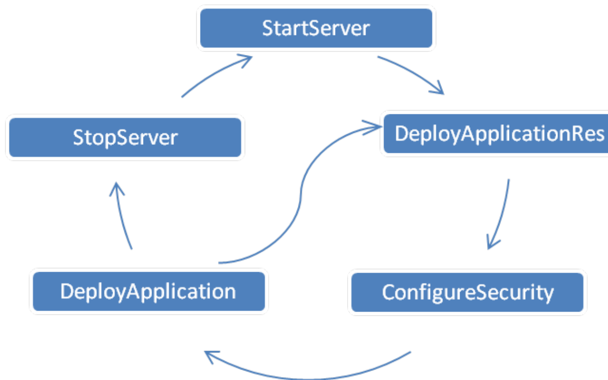

Figure 6. JavaSeRunner Java EE application server life-cycle

Both annotations, *RunAs* and *Transactional*, can be used together, making a method invocation to be performed in security and transactional contexts. Their usage is very important during application unit testing. Methods of Java EE components, like session EJBs, may require transaction (*TransactionAttributeType. Mandatory*) and security contexts propagation (*@RolesAllowed*)[1], see the next section 4.

4. Layer 4 - *Java EE Unit Runner*

The layer 4 is a typical integration layer that makes possible to use the framework *Java SE Runner* by the *JUnit* components. Developers can create custom test classes and test suites by extending *JUnit*:

- *BlockJUnit4ClassRunner*,
- *Suite*

classes respectively. This layer provides such classes to bind the *Java Se Runner* application serer life-cycle to *JUnit* operations. Detail binding contains Table 1.

JeeSuite extends JUnit *Suite* class and must be used in *@RunWith* annotation of the JUnit framework as showed on listing 2. The rest of the configuration is

Table 1. JUnit and application server life-cycle

JUnit method	Java SE Runner phase	Description
@BeforeClass	StartServer DeployApplicationRes ConfigureSecurity DeployApplication	Starts an application server, prepares all required resources and security configuration and finally deploys an application to the embedded application server.
@AfterClass	StopServer	Stops the application server after all test methods of a Test class or a TestSuite class have been executed.
@Test	Login BeginTransaction InjectResources TestMethod (JUnit) Logout EndTransaction	Before test method execution: authenticate user, begin transaction and inject components to the instance of a test class. After the test method is executed logout user and commit (or rollback) a transaction. All operations here are conditional and depends on annotations configuration.

practically the same as a sample configuration of *Java Se Runner* presented in listing 1. All rules and configuration parameters described in subsections 2.2 and 3.3 applies here also.

Listing 2. JUnit @RunWith annotation with provided JeeSuite class

```
1 @RunWith(JeeSuite.class)
2 @Suite.SuiteClasses({TestClass.class})
3 @ApplicationServer()
4 @Applications(applications = {
5     @Application(name = "classes",
6     applicationPath = "target/classes",
7     applicationType = ApplicationType.JAR)})
8 public class GlassfishServerTestSuite {
```

A JUnit suite groups and controls the execution of several JUnit test classes. A single test class and its test method execution is controlled by the custom *ClassRunner* called *JeeBlockClassRunner*. This is very important because *JeeBlockClassRunner* creates test objects as CDI managed beans instances and makes them a part of a tested application. There are two consequences of this approach:

- an automatic injection of resources and components is available,
- CDI portable extensions do work.

To get an instance of an application component for a testing purpose, a simple field of that class (or interface it implements) has to be placed in a test class and has to be marked with CDI *@Inject* annotation. When a test class instance is created, all dependent components are injected into that instance according to CDI specification [6]. A part of a sample test class (without test methods) is presented on listing 3, where different types of components are injected, like fo example:

- lines 3-4 injection of another test class instance,
- lines 5-7 injection of the transaction manager provided by embedded Java EE application server,
- lines 9-13 injection of EJB components with @EJB and @Inject annotation,
- lines 15-18 injection of an entity manager (*@TestDBPU* is a custom annotation describing injection point for CDI producer),
- lines 20-21 injection of a CDI component.

5. Conclusions

Current implementation of the JupEEter framework allows to:

- develop and run Java SE applications with Java EE components,
- perform unit and integration tests of components of Java EE applications with JUnit,
- perform system tests of Java EE applications with tools that can cooperate with JUnit.

Listing 3. Injection of components and resources into test class instance

```
1  class TestClass{
2
3    @Inject
4    AnotherTestClass testClass;
5
6    @Inject
7    UserTransaction tx;
8
9    @EJB
10   private ComponentEJB componentEJB;
11
12   @Inject
13   private ComponentEJB componentEJBInjected;
14
15   @Inject
16   @TestDBPU
17   @PersistenceContext
18   EntityManager em;
19
20   @Inject
21   private ComponentCDI componentCDI;
22 }
```

The framework uses Glassfish application server as an embedded Java EE application container, but different application servers can be used also. Only a new server controller component is required. Java SE applications build with the framework are able to exploit all capabilities of Java EE technologies. Developers can even create Java applications which can be access remotely through a WEB interface. This use case is quite common for "small" home devices, like network routers, set-top boxes and ect.

Like Java EE application server, JUnit can be also replaced with another testing framework - TestNG. A type of a testing framework is insignificant to the fact, that the JupEEter framework makes a process of development and continuous integration of Java EE applications very lean. It can be easily integrated with any continuous integration infrastructure through Apache Maven configuration. Developers can run unit test in a few or a dozen seconds. The whole application and

application server life-cycle takes about 15 seconds on a Intel Core 2 Duo E8200 based computer system. It is not much because application server has to be started only once. Additionally, there is no need to build Java EE development environment for a team because JupEEter delivers one. Developers must only add the framework libraries to the application:

- class-path to use Java EE technology or
- to test class-path to use it for testing purposes.

Methods of JUnit test classes can be annotated with @*RunAs* and @*Transactional* annotations to run them in a security and transaction context. It is especially important for an application unit testing, where there are no other components that could start transaction or perform user authentication.

The framework has been tested in author's real life Java EE projects and during Distributed Business Application lectures. Most of these projects are based on common Java EE technologies like for example: JAX-WS and JAX-RS web services (WS). With the JupEEter framework they can be tested just like standard components. WS server and WS client are assembled as a single application then. Normally, developers have to deploy WS server components to a remote application server prior running tests.

References

[1] Java Community Process, *JSR 316 - Java Platform, Enterprise Edition (Java EE) Specification, v6*, 2009, http://http://jcp.org/en/jsr/detail?id=316.

[2] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A., *The Java Language Specification - Java SE 7 Edition*, Oracle, 2011, http://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

[3] *Glassfish Server*, http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html.

[4] *JBoss Arquillian project site*, http://www.jboss.org/arquillian.html.

[5] *Glassfish Embedded API project site*, http://embedded-glassfish.java.net/.

[6] Java Community Process, *JSR 299 - Contexts and Dependency Injection for the Java EE platform*, 2009, http://jcp.org/en/jsr/detail?id=299.

[7] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A., *The Java Virtual Machine Specification - Java SE 7 Edition*, Oracle Corporation, 2012, http://docs.oracle.com/javase/specs/jvms/se7/html/index.html.