# Testing Framework version 1.0 Component Specification

## 1. Design

Cactus (http://jakarta.apache.org/cactus/) simplifies the process of testing web applications by enabling the application to be built, deployed, tested, and shut down entirely within an Ant build script. This component extends this concept to server-driven applications in general. It provides analogous functionality for database servers, web servers (with a wrapper for Cactus tests), and the extensibility to provide the same support for other types of servers.

For a web application that uses a Tomcat JSP server and PostgreSQL database, the test framework provides a simple way to specify how to configure and initialize both of these servers. Testing the application is then simply a matter of calling the framework, which initializes the servers, runs a test suite, and cleans up after itself.

This component provides the support for a Tomcat server using Cactus framework as a back-end. It also tries to support other types of servers, Cactus supports. The support for database servers should be provided by the other component.

*JUnit extension*
The test framework is a JUnit extension. But current implementation just wraps around Cactus test cases, because of using Cactus as a back-end for web application servers. The test cases also provide a description of interface to be used by the other implementations.

*Ant integration*
The component provides Ant tasks to enable tests defined in the test framework to be driven by an Ant build script. These tasks contain all of the configuration information necessary to perform their functions. There is no external configuration. Ant tasks provided by the component are **serverapptest, preparetestwar** and **preparetestear.** The documentation for these tasks is provided with the corresponding classes: *ServerAppTestTask*, *PrepareTestWar* and *PrepareTestEar*.

*Server Initialization / Clean-up*
The test framework provides a mechanism for starting a server if necessary, and initializing it with the proper data. For a SQL database server, this means starting the database server, creating the necessary tables and possibly filling them with test data. For a Tomcat (JSP) server, this means deploying the web application and starting the Tomcat server. It is possible to initialize several servers for one Ant server application testing task.
If the particular server was started and (or) initialized by the test framework, it will be stopped and (or) cleaned-up after running all the tests.
An element representing server in the Ant build file has two boolean attributes for controlling server initialization. These attributes are “startserver” and “initserver”. The “startserver” attribute specifies if the server should be started by the framework. The “initserver” attribute specifies if the server should be initialized with the data by the framework.
It's a pity, but Cactus framework doesn't provide such level of control and so, this implementation using Cactus as a back-end, has to ignore these attributes.

*Pluggable testing back-ends*
The new type of server can be added or the existing type of server may be implemented, using different testing back-end. These actions do not require any changes to the existing framework. The names of pluggable implementations are specified in the Ant build file.

### 1.1 Design Patterns

**The Factory Method Pattern.** The ServerAppTestTask and ServerElement classes use this pattern to handle nested elements in the Ant build file.

**The Adapter Pattern** is used in the DefaultWebApplicationServer to make Cactus adapt to the testing framework.

## 1.2 Industry Standards

None

## 1.3 Required Algorithms

None were required, but some complicated pieces of code are provided here.

### *1.3.1 Executing RunServerAppTest Ant task*

```
            //initialize all servers
            for (Iterator iterator = servers.iterator(); iterator.hasNext();) {
                final Server server = ((ServerElement) iterator.next()).getServer();
                server.startUp();
                server.initData();
            }

            //run the tests
            final Enumeration individualTests = getIndividualTests();
            while (individualTests.hasMoreElements()) {
                final JUnitTest jUnitTest = (JUnitTest) individualTests.nextElement();
                if (jUnitTest.shouldRun(getProject())) {
                    //set output dir
                    if (getReportsDir() != null) {
                        jUnitTest.setTodir(getReportsDir());
                    }
                    //execute test
                    execute(jUnitTest);
                }
            }

            for (Iterator iterator = servers.iterator(); iterator.hasNext();) {
                final Server server = ((ServerElement) iterator.next()).getServer();
                server.cleanUpData();
                server.shutDown();
            }
```

### *1.3.2 Start the server using a Cactus back-end*

```
// Get DeployableFile instance
 DeployableFile deployableFile = null;
 if (configuredWarFile != null) {
     deployableFile = WarParser.parse(configuredWarFile);
 } else if (configuredEarFile != null) {
     deployableFile = EarParser.parse(configuredEarFile);
 }

 // Retrieve Ant task information
 final AntTaskInfo taskInfo = getAntTaskInfo();
 // Create and set ant task factory
 container.setAntTaskFactory(new DefaultAntTaskFactory(taskInfo.getProject(),
          taskInfo.getTaskName(),
     taskInfo.getLocation(), taskInfo.getTarget()));

 // Set deployable file
 container.setDeployableFile(deployableFile);

 callSetterByReflection(container, "setServer", String.class, getServer());
 if (getPort() > 0) {
     callSetterByReflection(container, "setPort", int.class, new Integer(getPort()));
 }
 callSetterByReflection(container, "setDir", File.class, getDir());

 // Init the container
```

```
container.init();
// Set URL to test connection with
if (getTestURL() != null) {
    runner.setURL(getTestURL());
    // according to https://software.topcoder.com/forum/c_forum_message.jsp?f=20015788&r=21551343
    // we set the context url to be the testURL
    System.setProperty("cactus.contextURL", getTestURL().toExternalForm());
} else if (deployableFile != null) {
    final String spec = container.getBaseURL() + "/"
        + deployableFile.getTestContext()
        + deployableFile.getServletRedirectorMapping()
        + "?Cactus_Service=RUN_TEST";
    runner.setURL(new URL(spec));
    // sets the URL to be used by the test case client when
    // connecting to the container to call the container-side test cases
    // see https://software.topcoder.com/forum/c_forum_message.jsp?f=20015788&r=21553769
    System.setProperty("cactus.contextURL", container.getBaseURL() + "/" +
         deployableFile.getTestContext());
}
// Set timeout
runner.setTimeout(getTimeout());

// Start the container (starts server if server wasn't already running)
runner.startUpContainer();
```

## 1.4 Component Class Overview

### Class ServerAppTestTask

This class represents an Ant task that extends the optional JUnit task
to provide support for testing using this Component. This is the main Ant task of this component.

### Class ServerElement

This class represents a "server" XML element to be used in the Ant build file.
It acts as container for an element representing the actual server to be used for running tests.

### Class AntTaskInfo

This class simply stores some data about the Ant project. It is passed with current data to *ServerElement* class instance and then to *Server* derived classes' instances. It is currently used to provide information to the Cactus testing back-end.

### Class Server

This class is a base class for all classes representing an actual server to be used while running tests.
Derived classes should have corresponding XML elements to be used in the Ant build file.

### Class CredentialsElement

This class represents a "credentials" XML element to be used in the Ant build file. The credentials can include user name, password and other information needed to authenticate to the server. A particular *Server* implementation should require the specific credentials.
This element represents some kind of a property map, each property is described by one nested element. The name of nested element represents the property key,
the "value" attribute - property value.

### Class CredentialProperty

Represents nested element of the "credentials" XML element. This nested element represents single property from the property map. The name of nested element represents the property key, the "value" attribute - property value.

### Class ApplicationServer

This class is derived from *Server* class and is the base class for all classes representing an application server to be used while running tests.
Those classes will have corresponding elements in the Ant build file.

### Class DataServer

This class is derived from *Server* class and is the base class for all classes representing data server to be used while running tests.
Those classes will have corresponding elements in the Ant build file.
This class is a base for another component which will provide testing ability for database servers.
TestCase classes should also be provided by that component.

### Class DefaultWebApplicationServer

This class is derived from *ApplicationServer* class.
It represents web application server to be used while running tests. It uses Cactus as a back-end. It tries to support all the servers supported by Cactus.
Name of corresponding XML element in the Ant build file should be equal to the Cactus container type representing the desired server.

**Class PrepareTestWar**

This class represents an Ant task that injects elements necessary to run the tests into an existing WAR file. It can be used to prepare WARs for DefaultWebApplicationServer class.
For now it's just a wrapper for CactifyWarTask, but future implementations may be different.
This class provides an interface that is required for the future implementations.

**Class PrepareTestEar**

This class represents an Ant task that injects elements necessary to run the tests into an existing EAR file. It can be used to prepare EARs for DefaultWebApplicationServer class.
For now it's just a wrapper for CactifyEarTask, but future implementations may be different.
This class provides an interface that is required for the future implementations.

**Class TCServletTestCase**

This class represents JUnit test case to unit test code that needs an access to valid Servlet implicit objects (such as the HTTP request, the HTTP response, the servlet config, ...)
This implementation of the class is derived from  Cactus ServletTestCase, and just adds more high level access to Servlet objects.
Servlet objects should be accessed by the derived classes using the getter methods provided by this class. Derived classes should not  directly use methods and variables defined by the Cactus framework.

**Class TCJspTestCase**

This class represents JUnit test case to unit test code that needs an access to valid JSP implicit objects (such as the page context, the output jsp writer, the HTTP request, ...)
This implementation of the class is derived from Cactus JspTestCase, and just adds more high level access to Jsp objects.
Jsp objects should be accessed by the derived classes using the getter methods provided by this class. Derived classes should not directly use methods and variables defined by the Cactus framework.

**Class TCFilterTestCase**

This class represents JUnit test case to unit test code that needs an access to valid Filter implicit objects (such as the FilterConfig and FilterChain objects).
This implementation of the class is derived from Cactus FilterTestCase, and just adds more high level access to Filter objects.
Filter objects should be accessed by the derived classes using the getter methods provided by this class. Derived classes should not directly use methods and variables defined by the Cactus framework.

## 1.5 Component Exception Definitions

The design has decided not to throw *NullPointerException* from the methods to be used directly by Ant. But the *IllegalArgumentException* is thrown. The methods to be used internally by the Component throw both exceptions.

**Exception BuildException**

This exception is thrown by all the classes of the component, which are to be used from the Ant build. So, from all the classes of the Component, only JUnit test cases doesn't throw it.

**Exception Throwable**

This exception is thrown by JUnit test cases. It means that every exception is just passed by JUnit test cases directly to the JUnit test runner.

## 1.6 Thread Safety

This component is not thread-safe, because all of its classes are mutable. But there is no need for it to be thread-safe, because it will be used only in the Ant builds, and so it will be Ant framework's responsibility not to access class instances by the multiple threads, or to synchronize the access. They are most likely to be used by the single thread.

## 2. Environment Requirements

### 2.1 Environment

- Windows 2000/XP/2003
- Solaris 7
- RedHat Linux 7.1
- Java 1.4
- Tomcat 5.5

### 2.2 TopCoder Software Components

None used.

### 2.3 Third Party Components

- Ant 1.6.5: http://ant.apache.org/
- JUnit 3.8.1: http://www.junit.org/index.htm
- Cactus 1.7.1: http://jakarta.apache.org/cactus/

## 3. Installation and Configuration

### 3.1 Package Names

com.topcoder.testframework, com.topcoder.testframework.web

### 3.2 Configuration Parameters

This component does not have any external configuration. Instead it is used directly from an ant build file. Configuration properties are passed to the component by specifying nested child elements and tag attributes.

The following reference will describe the ant tasks and their configuration as provided by this component. This is a step-by-step documentation, for a comprehensive example see the demo section of this document.

#### *3.2.1 Declaring the component to be usable by ant*

To be usable by ant, the tasks provided by the component need to be registered with ant. This is done by declaring a `taskdef`:

```
<taskdef classpathref="toolclasspath" resource="testingFramework.tasks"/>
```

Now the three ant tasks defined by this component can be used in an ant build file.

#### 3.2.2 `preparetestwar`

The `preparetestwar` task is used to prepare an existent component war file (i.e. the actual productional component war) for usage as test web application. For this purpose the core libraries used for testing are added and the `web.xml` file of that war is modified for having externally callable entry points for testing by the test framework. For any actual tests to be performed, the test classes to be run and any libraries used by the test classes must be included in the war file in the following manner:

```
<preparetestwar srcfile="${build_dist}/component.war"
                destfile="${build_dist}/test.war">
     <lib file="${testing_framework.jar}"/>
     <classes dir="${build_testclassdir}"/>
</preparetestwar>
```

The following attributes are defined by the `preparetestwar` task:

| Name | description | example |
|---|---|---|
| srcfile | the path to an existent source war file to be prepared | build/dist/component.war |
| destfile | the path to write the prepared version of the war file to | build/dist/component-test.war |

The following nested elements are defined by the `preparetestwar` task:

| Name | description | example |
|---|---|---|
| fileset | a fileset specifying content to be put in the war file in the war file root | <fileset dir="src/resources/test"/> |
| lib | a fileset specifying content to be put in the war file under `WEB-INF/lib` | <lib file="lib/xmlunit/1.0/xmlunit.jar"/> |
| classes | a fileset specifying content to put in the war file under `WEB-INF/classes` | <classes dir="build/testclasses"/> |

### 3.2.3 `preparetestear`

The `preparetestear` task is used to prepare an existent component ear file (i.e. the actual productional component ear) for usage as test enterprise application. For this purpose the core libraries used for testing and the `web.xml` file for having externally callable entry points for testing by the test framework are added as additional war file to the ear. For any actual tests to be performed, the test classes to be run and any libraries used by the test classes must be included in the ear file in the following manner:

```
<preparetestear srcfile="${build_dist}/component.war"
                destfile="${build_dist}/test.war">
     <fileset file="${testing_framework.jar}"/>
     …
</preparetestear>
```

The following attributes are defined by the `preparetestear` task:

| Name | description | example |
|---|---|---|
| srcfile | the path to an existent source ear file to be prepared | build/dist/component.ear |
| destfile | the path to write the prepared version of the ear file to | build/dist/component-test.ear |

The following nested elements are defined by the `preparetestear` task:

| Name | description | example |
|---|---|---|
| fileset | a fileset specifying content to be put in the ear file in the ear file root | <fileset dir="lib/test"/> |

### 3.2.4 `serverapptest`

The `serverapptest` task is used to start up several containers used in the test scenario and then execute the in-container test cases. See the following example:

```
<serverapptest printsummary="yes" failureproperty="tests.failed" reportsDir="${builddir}">
            <!-- This element describes the application server using
                   Cactus back-end and Tomcat 5.5 server-->
            <com.topcoder.testframework.web.DefaultWebApplicationServer>
                <tomcat5x warfile="${build_dist}/test.war"
                          dir="D:\Tomcat5.0.28" port="8777">
                </tomcat5x>
            </com.topcoder.testframework.web.DefaultWebApplicationServer>

            <!--This element includes our custom server we have written in the test source path-->
            <demo.CustomServer>
                <!--The name of this child element will be given to the custom server
                class as constructor arg and the attribute will be given to the
                constructed instance in its setCustomAttribute(String) method-->
                <customDatabaseServerLauncher customAttribute="customAttributeValue"/>
            </demo.CustomServer>

            <!--All the parts below are functionality inherited from the Junit ant task:-->

            <!--This is the classpath for the client side run
        of the tests, every library that was used in test
        setup, teardown or in any of the beforeXX or
        afterXX must be declared here.-->
            <classpath>
                <path refid="testclasspath"/>
            </classpath>

            <!--This defines the output formatter-->
            <formatter type="xml"/>

            <!--This elment defines which tests to run-->
            <batchtest>
                <fileset dir="${javatests}">
                    <include name="**/*Test*.java"/>
                </fileset>
            </batchtest>
        </serverapptest>
```

The following attributes are defined by the `serverapptest` task:

| Name | description | example |
|---|---|---|
| reportsdir | the path to an existent directory in which to put the test run reports | build/testreports |
| all attributes defined by the JUnit ant task are inherited, see http://ant.apache.org/manual/OptionalTasks/junit.html | | |

The `serverapptest` task does allow all of the nested elements defined by the JUnit ant task (see http://ant.apache.org/manual/OptionalTasks/junit.html).
Additionally the servers and J2EE containers to be started are specified using nested elements. The tag name of the nested element is always the fully qualified class name of the Server subclass that is instantiated with the name of its child element as constructor argument (In the example above the class `com.topcoder.testframework.web.DefaultWebApplicationServer` is instantiated with the String “tomcat5x” as constructor arg.). Using this mechanism it is possible to provide any custom server implementation to the build file. The only currently available Server implementation intended for direct usage is the class `com.topcoder.testframework.web.DefaultWebApplicationServer`.
The nested element of the `DefautWebapplicationServer` element can have the following attributes:

| Name | description | example |
|---|---|---|
| dir | the path to the container installation home that contains the binaries needed to start the server | D:/tomcat5.0.28 |
| server | the hostname to connect the server at after it has started – optional -- | localhost |
| startServer | whether the server should be started by the framework – optional -- | true |
| initServer | whether the server should be initialized before test cases are run – optional -- | true |
| port | the port number on which the started server should listen –optional -- | 12345 |
| warFile | the war file to be deployed to the server – optional -- | build/dist/component-test.war |
| earFile | the ear file to be deployed to the server -- | build/dist/component-test.ear |

| | | |
|---|---|---|
| | optional -- | |
| testURL | an URL at which the framework can check whether the server has started successfully or not – optional -- | http://localhost:1234/test/alive.jsp |

As the implementation of the DefaultWebApplicationServer is currently backed by cactus and specifies the container type of the container to be initialized, the nested element can have one of the following names: `tomcat3x, tomcat4x, tomcat5x, jboss3x, orion1x, orion2x, resin2x, resin3x, weblogic7x`.

The DefaultWebApplicationServer element allows to specify an additional nested credentials element that can be used to specify any configuration parameters or credentials to the server:

```
…
<credentials>
      <user_name value="topcoder" />
      <password value="some_password" />
      <some_other_property="some_value"/>
</credentials>
```

The tag names of the elements nested inside the credentials element form the property name and their value attribute forms the property value.

### 3.3 Dependencies Configuration

None required

## 4. Usage Notes

### 4.1 Required steps to test the component

As this component does not have a configuration interface, the developer wanted to avoid to introduce any dependency to a configuration mechanism (such as TC `ConfigManager`) just for the configuration of the test cases.

Thus the necessary configuration for the unit test is specified to the test cases using system properties. The ant build script provides these system properties to the test by reading them from the ant properties in `build.xml:80+`.

The only property that needs to be changed is the value of '`testcases.tomcat.home`', which must be the path to a valid Tomcat 5.x. installation home. In case necessary, the value of the property '`testcases.tomcat.port`' can be used to define the port on which the test container should be started.

Afterwards the test cases are started by calling '`ant test`'

### 4.2 Required steps to use the component

See section 3.2 for an introduction on how to use the component from an ant build file.

### 4.3 Demo

This demo is included in a more comprehensive way inside the directory `test_files/demo`, which is meant to be the root of an example project that uses the Testing Framework component. The central parts of that demo are explained here but the minor configuration details can be read from that demo.

Consider this class to be the class to be tested using the Testing framework:

```
public class DemoServlet {
    /**
     * This method exists as placeholder for a method to be tested. The side effects produced by
     * this method are tested using the test cases of this demo.
     *
     * @param request  the current HttpServletRequest on which to operate
     * @param response the response for the request
     */
    public void saveToSession(final HttpServletRequest request, final HttpServletResponse response)
{
        final String testparam = request.getParameter("testparam");
        request.getSession().setAttribute("testAttribute", testparam);
        response.addCookie(new Cookie("initialized", "true"));
    }
}
```

Then there will be a test class for testing our component which could look like that:

```
public class DemoTestCase extends TCServletTestCase {
    /**
     * This method is called on the client side before {@link #testSaveToSessionOK()} is invoked on
     * the Server. The WebRequest instance supplied can be used to set request parameters to the
     * HTTP request.
     *
     * @param webRequest the request instance that will be sent to server and will be used as
     * request (i.e. as return value of {@link #getRequest()}) instance when {@link
     * #testSaveToSessionOK()} is called on the server
     */
    public void beginSaveToSessionOK(final WebRequest webRequest) {
        webRequest.addParameter("testparam", "it works!");
    }
    /**
     * This method performs the actual testing. It is called by the testing framework on server
     * after the context has been initialized (i.e. {@link #getRequest()}, {@link #getResponse()},
     * {@link #getSession()} etc. return valid values).
     */
    public void testSaveToSessionOK() {
        final DemoServlet servlet = new DemoServlet();
        servlet.saveToSession(getRequest(), getResponse());
        assertEquals("attribute not written to session", "it works!",
                   getSession().getAttribute("testAttribute"));
    }
     /**
     * This method is called on the client side after completion of {@link #testSaveToSessionOK()}
     * on server side, with the parsed version of the response sent on server during the test.
     *
     * @param response the parsed version of the response received from {@link
     *                 #testSaveToSessionOK()}
     */
    public void endSaveToSessionOK(final WebResponse response) {
        assertEquals("response cookie not set", "true",
                   response.getCookie("initialized").getValue());
    }
}
```

There could be some kind of custom server that needs to be initialized before test cases are run. This server can also be defined in the test code of the project under test, or it could be provided in some jar file as library. The version shown here does not do very much but should help to get the point:

```
public class CustomServer extends DataServer {
    public CustomServer(final String type) {
        super(type);
        System.out.println("CustomServer.CustomServer(" + type + ")");
    }
   public void setCustomAttribute(final String attribute) {
        System.out.println(
                   "Ant gave CustomServer instance the custom attribute value: " + attribute);
    }
    public void startUp() {
        System.out.println("CustomServer.startUp");
    }
    public void shutDown() {
        System.out.println("CustomServer.shutDown");
    }
    public void initData() {
        System.out.println("CustomServer.initData");
    }
    public void cleanUpData() {
        System.out.println("CustomServer.cleanUpData");
    }
}
```

Now comes the central part, the ant build file fragment that puts all these things together and runs the test for the Servlet in the container after starting the server and deploying the test case and starting our custom server:

```
<target name="test" depends="buildWar">
```

```
        <!--This is the normal compile of the test classes.
            As the custom server implementation is in that source dir,
            this compilation has to be the first step for using
            the testing framework-->
        <mkdir dir="${build_testclassdir}"/>
        <javac classpathref="testclasspath" srcdir="${javatests}" destdir="${build_testclassdir}"/>

        <!-- This does define the three ant tasks provided by the testing framework.
            The internals about what tasks are provided and what are the
            implementation classes for that tasks are hidden in the file
            testingFarmework.tasks, which is included in the testing
            framework jar file.-->
        <taskdef classpathref="toolclasspath" resource="testingFramework.tasks"/>

        <!-- Prepare existing web-application archive for testing, include the
             compiled test classes and all libraries used in the test cases
             (in this case testing framework is needed as we subclass classes
             defined in testing framework)-->
        <preparetestwar srcfile="${build_dist}/component.war"
                        destfile="${build_dist}/test.war">
            <lib file="${testing_framework.jar}"/>
            <classes dir="${build_testclassdir}"/>
        </preparetestwar>


        <!-- This Ant task starts the servers, runs the tests, and then stops the servers -->
        <serverapptest printsummary="yes" failureproperty="tests.failed" reportsDir="${builddir}">
            <!-- This element describes the application server
                   using Cactus back-end and Tomcat 5.5 server-->
            <com.topcoder.testframework.web.DefaultWebApplicationServer>
                <tomcat5x warfile="${build_dist}/test.war"
                          dir="D:\Tomcat5.0.28" port="8777">
                </tomcat5x>
            </com.topcoder.testframework.web.DefaultWebApplicationServer>

            <!--This element includes our custom server we have written in the test source path-->
            <demo.CustomServer>
                <!--The name of this child element will be given to the custom server
                class as constructor arg and the attribute will be given to the
                constructed instance in its setCustomAttribute(String) method-->
                <customDatabaseServerLauncher customAttribute="customAttributeValue"/>
            </demo.CustomServer>

            <!--All the parts below are functionality inherited from the Junit ant task:-->

            <!--This is the classpath for the client side run
        of the tests, every library that was used in test
        setup, teardown or in any of the beforeXX or
        afterXX must be declared here.-->
            <classpath>
                <path refid="testclasspath"/>
            </classpath>

            <!--This defines the output formatter-->
            <formatter type="xml"/>

            <!--This element defines which tests to run-->
            <batchtest>
                <fileset dir="${javatests}">
                    <include name="**/*Test*.java"/>
                </fileset>
            </batchtest>
        </serverapptest>
    </target>
```

## 5. Future Enhancements

- Implement support for other types of servers, using different back-ends.