Creating Custom Load Test Components

This topic explains how to extend Parasoft Load Test's functionality with custom load test components. In this section:

About Custom Load Test Components

Parasoft Load Test functionality can be extended by creating custom load test components. Parasoft Load Test acts as a container for load test components, or load test beans. A load test component is a functional core that defines the behavior of a virtual user during load testing. The load test container provides the following services to the component:

- Creates, invokes, and disposes virtual users (and therefore load test components) according to the load test scenario.
- Collects components reports from local and remote machines and accumulates them into a combined load test report.
- Serializes components to remote machines before the start of a load test.
- Provides a means for saving component configuration in a load test project file.
- Provides a means for component configuration through the GUI.

A load test component can be created by extending the SimRunnable class of the Load Test component Java API located in the com.parasoft.simulator.api package. SimRunnable stands for Simulator runnable. Simulator is the name for the Load Test container, which simulates multiple virtual user activities.

Building the Component Examples

The component examples are located in the `examples\dev\components` folder of your SOAtest/Load Test installation. The simple-runnable-source.jar and the socket-runnable-source.jar archives contain the source, configuration, and build files for the components described in this tutorial. You will need to unjar these archives to the Load Test component projects directory, which must be created manually.

Creating a Load Test Component Projects Directory

To create a component projects directory outside the Load Test installation:

1. Copy the `simple-runnable-source.jar` into the component projects directory and unjar the component source archive.
2. Unjar the `socket-runnable-source.jar` into the `SocketRunnable` directory, which must be created manually. Each source directory contains a `component-build.xml` ANT build script.
3. Open this file and make sure the `load.test.install.root` property is pointing to your SOAtest/Load Test installation directory.
4. Change the `load.test.jar.path` property to point to the `loadtest.jar` file (in the `eclipse\plugins\com.parasoft.xtest.libs.web_[version_number]\root` of the SOAtest/Load Test installation directory.

Creating a SimpleRunnable Load Test Component Project

To create a SimpleRunnable load test component project in Eclipse:

1. Open the New Project wizard, select **Java Project**, then click **Next**.
2. Name the project `SimpleRunnable`.
3. Choose **Create project from existing source**, select the **SocketRunnable** directory, then click **Next**.
4. In the **Source** tab, make sure the default output folder is set to `SocketRunnable/target/classes`.
5. Open the **Libraries** tab, click **Add External Jar** and add **parasoft.jar** and **com.parasoft.api.jar** (in the `eclipse\plugins\com.parasoft.xtest.libs.web_[version]\root` directory of your SOAtest/Load Test installation directory).
6. In Eclipse, right-click **component-build.xml** and choose **Run As> Ant Build** to create the component .jar file.

Notes

The components should be built with Java 1.5 JRE—preferably with JRE 1.5.0_18.

The default target of the `component-build.xml` ANT script in each example project builds a component jar archive, which could be deployed to the Load Test as a custom or built-in component. If you use Eclipse to compile the component project as described here, remove the compile target dependency from the jar target of the `component-build.xml` script.

Creating a SocketRunnable Load Test Component Project

The SocketRunnable component extends the SimpleRunnable component. Compile `SimpleRunnable` first and put the `main.jar` from the root directory of the SimpleRunnable project into the `lib` directory of the SocketRunnable project.

To create a SocketRunnable load test component project in Eclipse:

1. Open the New Project wizard, select **Java Project**, then click **Next**.
2. Choose **Create project from existing source**, select the **SimpleRunnable** directory, then click **Next**.
3. In the **Source** tab, make sure the default output folder is set to `SimpleRunnable/target/classes`.
4. Open the **Libraries** tab, click **Add External Jar** and add **parasoft.jar** and **com.parasoft.api.jar** (in the `eclipse\plugins\com.parasoft.xtest.libs.web_[version]\root` directory of your SOAtest/Load Test installation directory).
5. In the **Libraries** tab, ensure that the `lib/jtidy-r820.jar` and `lib/main.jar` are on the project class path.
6. In Eclipse, right-click **component-build.xml** and choose **Run As> Ant Build** to create the component .jar file.

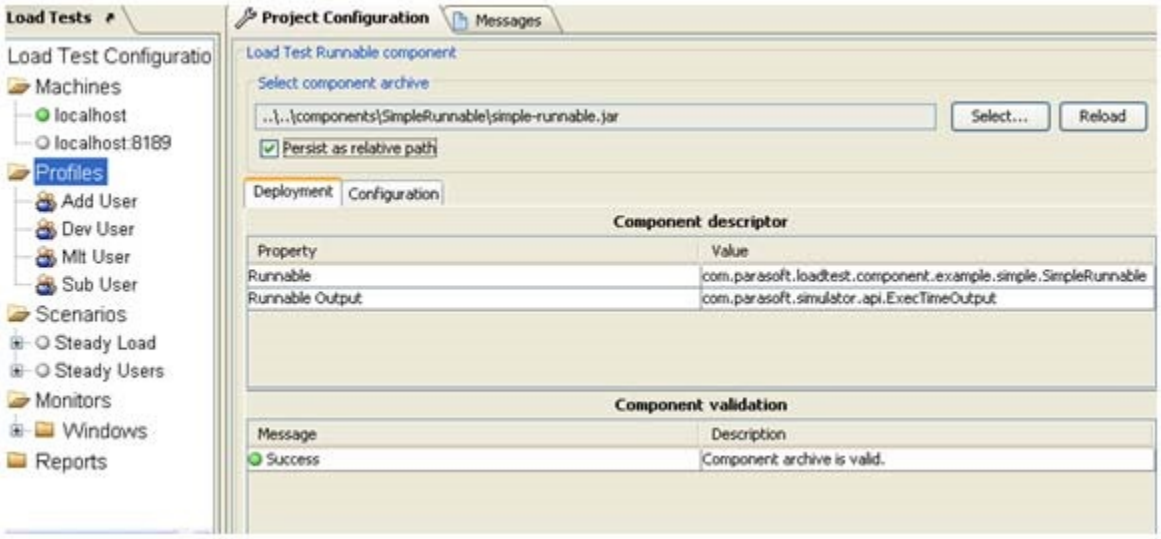
Loading/Deploying Example Components into Load Test

A component can be loaded into Load Test as an external archive or deployed as a built-in component.

Loading a Component from an External Archive

To load a component from an external archive:

1. In the Load Test GUI, select the **Profiles** node of the Load Test Configuration tree.
2. In the Project Configuration view, click **Select**.
3. In the Select a Component Archive dialog, choose **Local option**, click **Next**, then select your load test component archive.
4. Click **Finish**. A component deployment view will appear as shown below.



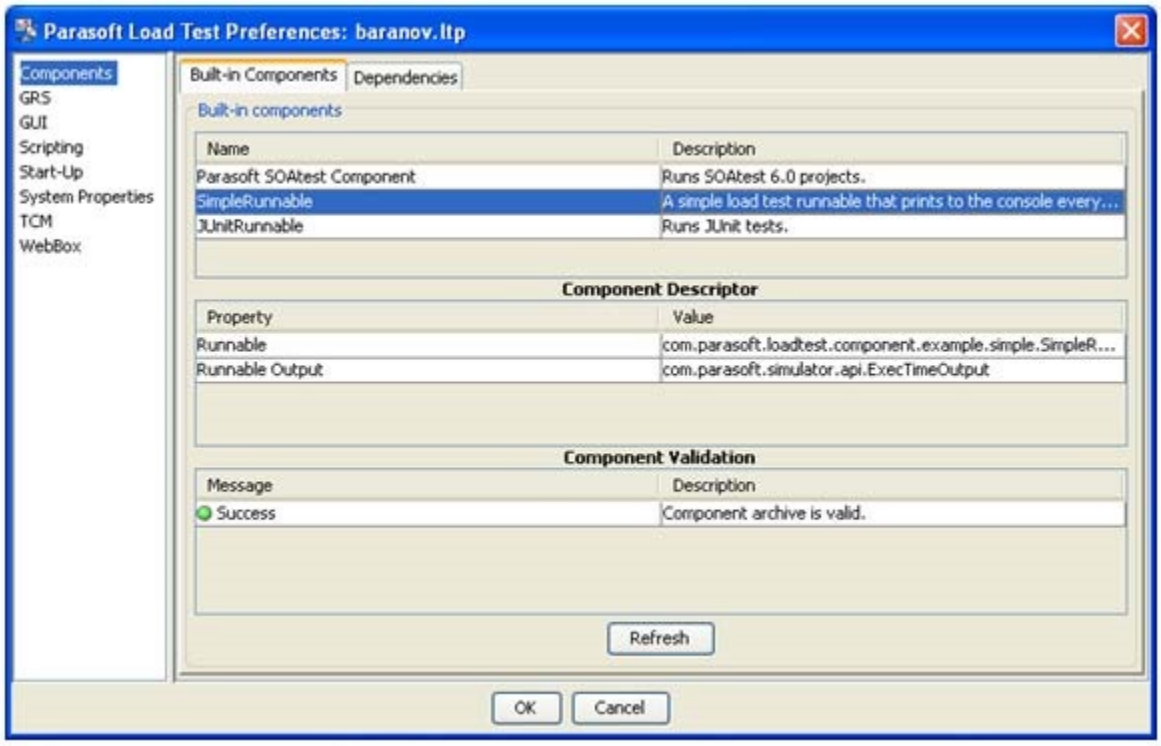
If component validation failed, the error messages will be shown in the lower table of the Deployment view.

Deploying a Component Archive as a Built-In Component

To deploy a component archive into the Load Test application as a built-in component:

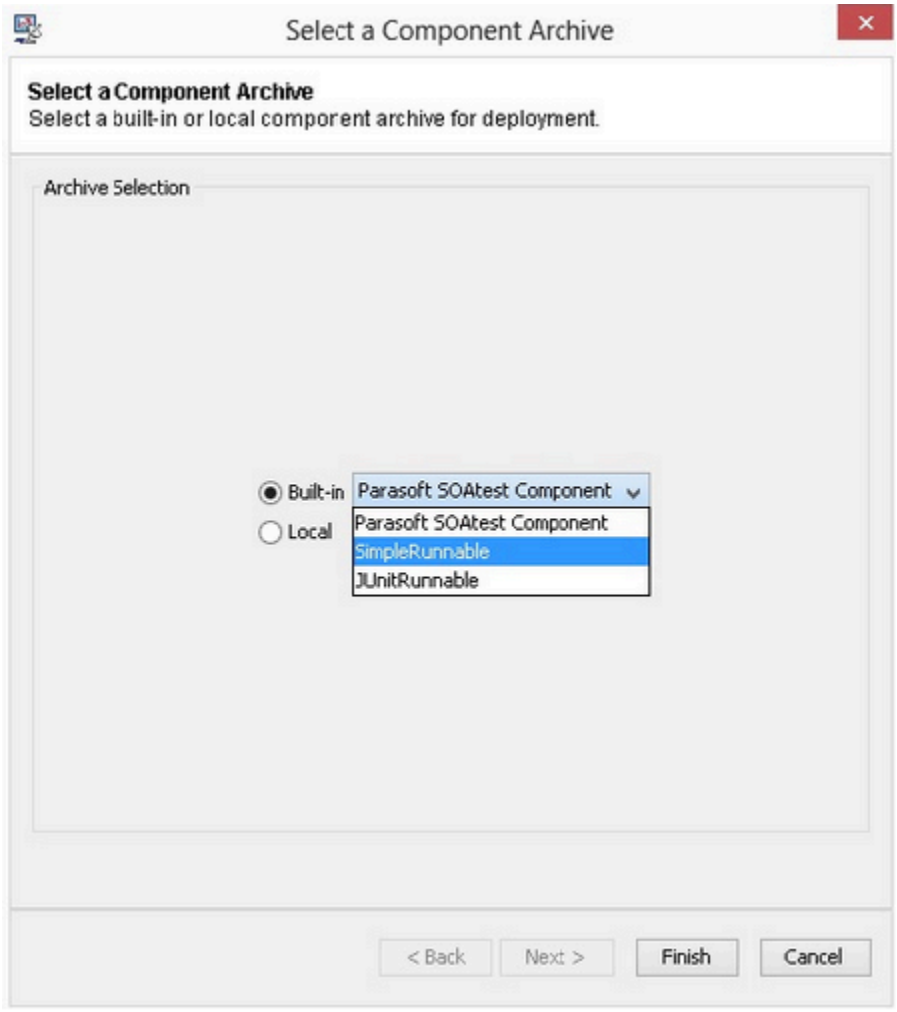
1. Place the component jar archive into the `eclipse\plugins\com.parasoft.xtest.libs.web_[version]\root\lt-components\built-in` folder of your SOAtest/Load Test installation.
2. Restart Load Test.
3.

3. From the **File** menu, select **Customize Preferences**. You will see your component deployment details in the built-in component deployment view as shown below.



If a built in component has been successfully deployed, it will be available in the Select Component Archive dialog in the Built-in drop

down list as shown below:



Using Load Test Component API

Javadoc documentation for the Load Test component API is available by selecting the **API> Component API** submenu from the **Help** menu of the Load Test application menu bar.

Analyzing the SimpleRunnable Component Example

This section examines the SimpleRunnable Load Test component.

SimpleRunnable is a basic component that emulates test execution of a configurable duration and produces errors with a configurable frequency. It's designed to get you acquainted with the Load Test Component API and the process of component archive creation and deployment.

The main method of a load test component is run(PrivateContext context). The component behavior that emulates user activities should be implemented inside this method. The component run method is invoked by a Virtual User object of a specific Profile. The Virtual Users in their turn are invoked by the Load Test application according to the load test Scenario.

The rest of the SimpleRunnable methods perform various services that are required by the container. Below, we will describe the most important of the component methods. For details on the rest of the methods, look at the source code of the SimpleRunnable class and the JavaDoc of SimRunnable.

Here are descriptions of the most important methods of the component:

- SimRunnableOutput run(PrivateContext context) – The core functionality method of the component. Various container-specific variables are available through the PrivateContext passed as an argument to this method. See PrivateContext in the API docs for the list of available context variables.

- String canRun() – This method is called by the container to ensure that the component is properly-configured and can be invoked. Check whether all component member variables are properly initialized. If not, return an error description. If the component is ready, return null. Returning a non-null error description will prevent Load Test from starting and the error string will display.
- boolean done() – This method is called by the container after invoking the run() method of this class to determine whether the component has completed its activities. A component can emulate a multi-step user activity with each step followed by the Virtual User think time as configured in the load test Profile. In such a case, the run() method will be called more than once until the done() returns true. Returning false will cause the framework to retain the Virtual User to which the instance of this component class belongs and call its run() method again. Returning true will cause the Virtual User to which the instance of this component class belongs to be removed from the Virtual User pool.

> **Important - Releasing Virtual Users**
>
> You eventually need to return true from the done() method in order to allow the Virtual Users to be removed from the pool. Once a Virtual User is removed from the pool, it will be replaced with a new Virtual User of the same profile if it is required by the load test Scenario. Releasing the Virtual Users is important because it allows the framework to follow the ups and downs of the scheduled virtual user or hit per second rate.

- SimRunnable prototype() – This is a factory method. Each time a Virtual User is created by the container, it is assigned a new instance of the component class. When a component is deployed into the container, the container creates an instance of the component class, which serves as a prototype for creating component class objects that are assigned to Virtual Users during the load test. The framework calls the prototype() method of the component class to obtain copies of the component prototype and it lets the component decide how to make that copy. When implementing this method, make sure that all the necessary component member variables are copied to the newly created component instance. You can copy component class members by value or by reference:
    - When copying component class members by reference, be aware that these class members should be adequately protected against concurrent access or modification. If this becomes a problem, consider copying by value (deep copy).
    - When copying component class members by value (deep copy), keep in mind, that hundreds—possibly thousands—of Virtual Users (and therefore instances of this class) can simultaneously exist in the Load Test application while it runs the load test. The number of Virtual Users inside the application depends on the Load Test mode and the values scheduled in the load test Scenario. A load test component has too large of a memory footprint, it can cause the application to run out of memory. In this case, consider copying member variables by reference.
- void onDeploy(DeploymentContext context) – This method is invoked by the container when a component archive gets deployed into the container. Various deployment-specific variables are available through the DeploymentContext passed as an argument to this method. The SimpleRunnable's implementation of this method shows how to get access to the configuration (and other) files included into the component deployment.
- void writeExternal() and void readExternal() – Methods that transfer component state to the remote machines.

Notice that the writeExternal() and readExternal() methods of the SimpleRunnable class have only skeleton implementation and do not write or read any data. These methods are used to transfer the component state to the remote Load Test machines. The SimpleRunnable component does not require such state transfer because the component external configuration file, which is included in the component archive, is used for the component configuration on local and remote machines. We will look at the implementation of writeExternal() and readExternal() in a different component example.

Next, let's look at the SimpleRunnable's external configuration file. The source folder of the SimpleRunnable component contains two component configuration files: LongExecTime.cfg and ZeroExecTime.cfg. One of them is included into the component archive (see component-build.xml ANT script).

When a component archive is deployed into the container, the archive is decompressed and the archive contents become available to the component via the DEPLOYMENT_BASE property of the DeploymentContext. The implementation of the onDeploy method of the SimpleRunnable class shows how the .cfg component configuration file is located and parsed to initialize an instance of SimpleRunnableConfiguration. If a load test configuration includes remote machines, the component archive is transferred to each remote machine before the beginning of the load test, after which each component is deployed.

After the deployment process is complete, the onDeploy method is called. The onDeploy method initializes the component from the .cfg file, which is included into the component archive. A component archive is serialized to a remote machine only if it is not present at the remote installation or if its checksums on the master and remote machines differ. This ensures that component archive transfer to the remote machines happens only when needed.

Component Archive Descriptor

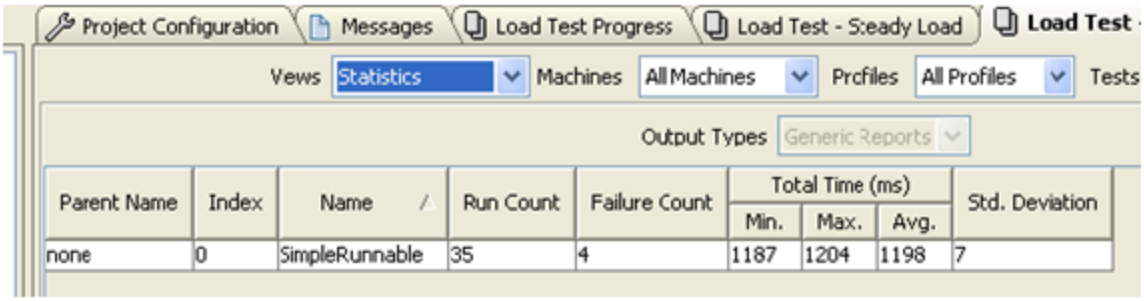
The lt-jar.xml component archive descriptor must be included into the root directory of each component archive. The descriptor must contain the class names of the main component class derived from the SimRunnable and the class name of the component output derived from SimRunnableOutput. See the lt-jar.xml files of the included components for the examples of the component archive descriptor file format.

Running Load Test with the SimpleRunnable Component

Once you have deployed the SimpleRunnable component into Load Test and familiarized yourself with the SimpleRunnable source code and component archive structure, you can make modifications to the component archive configuration, rerun the load test, and see the effects of the modified component configuration. For instance, you can follow these steps in order to change the test execution time emulated by this component:

1. Add the execTimeMs parameter in the .cfg file.
2. Run the component archive ANT build script.
3. In the Load Test component view, click the **Reload** button.

4. Rerun the load test. Note that the test execution time in the Statistics part of the load test report follows the configured execution time duration.



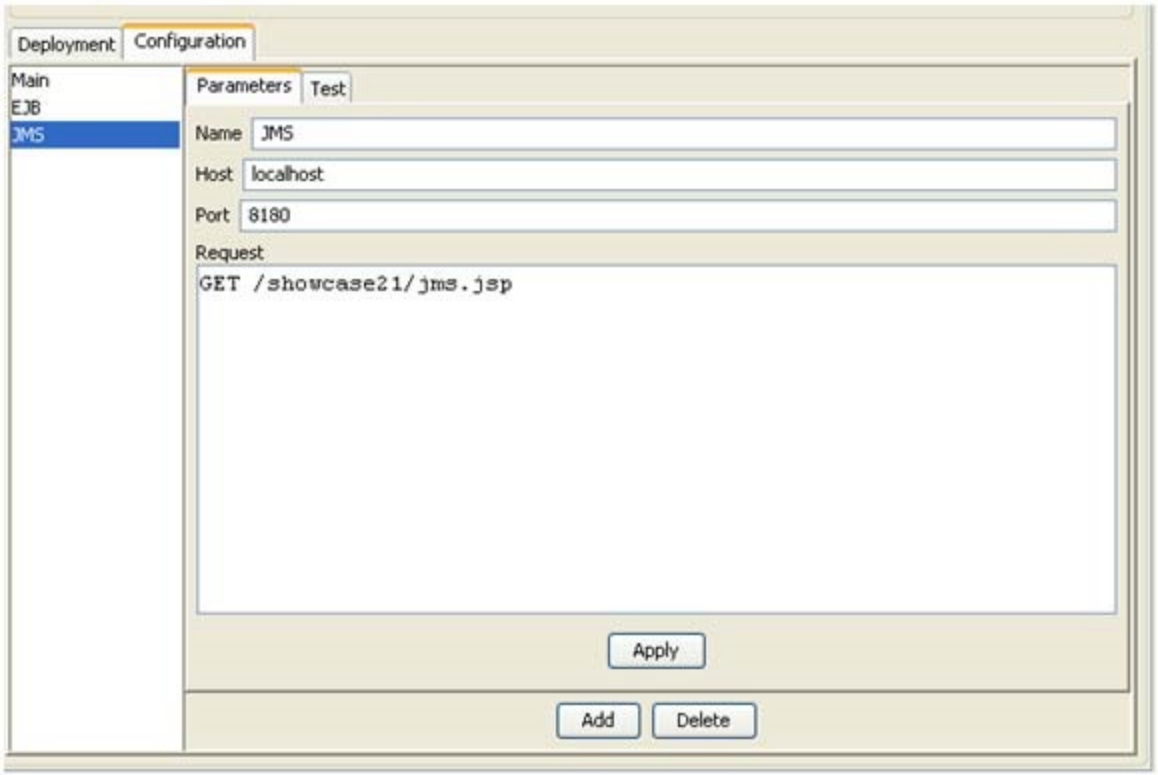
The execution statistics reflect the changes of the execTimeMs parameter of the component configuration file to 1200.

Analyzing the SocketRunnable Component Example

This section examines the SocketRunnable component. The SocketRunnable class extends SimpleRunnable and demonstrates the following:

- Logging internal errors in the application log.
- Creating component configuration GUI panels.
- Creating load test profile GUI panels which allow profile specific component configuration.
- Saving component configuration in the load test project.
- Transferring component configuration to remote machines.
- Using jar libraries to extend component functionality.
- Localizing component resources.

The SocketRunnable component uses SocketTest class instances to establish a connection to a specific host and port, send a customizable request, and receive a response using Java Sockets. The SocketTest instances that belong to the component are configurable in the SocketRunnable compnent view.



Logging Internal Errors in the Application Log

You may need the log immediately, so it is acquired from the Engine in the SocketRunnable constructor.

```
Engine engine = (Engine)context.get(PrivateContext.ENGINE);
ComponentLog log = engine.getComponentLog();
```

Creating Component Configuration GUI Panels

A component can provide its custom configuration view to the container by implementing the SimRunnableConfigViewProvider interface. The container will display this view in the Configuration tab of the Load Test Runnable component panel shown above. The container calls the getConfigView() method of the SimRunnableConfigViewProvider interface to get the configuration view of the component. In this example, the SocketRunnableView class implements the component configuration view. The SocketRunnableView class uses SocketTest*View classes located in com.parasoft.loadtest.component.example.socket.test.view package for individual SocketTest instance configuration.

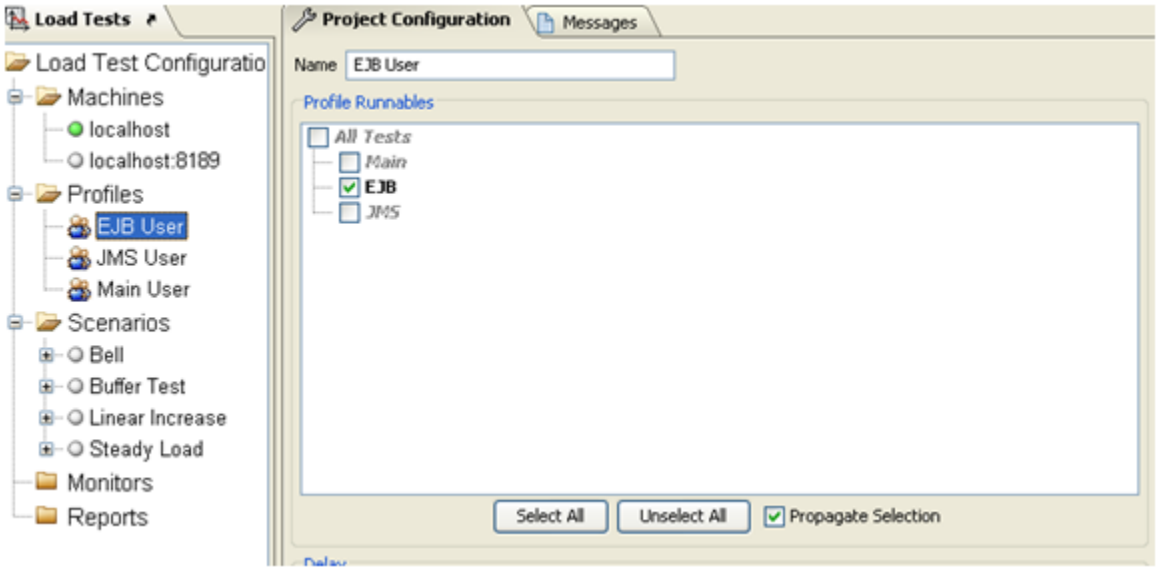
Creating Load Test Profile GUI Panels For Profile-Specific Component Configuration

A load test component can be designed to implement load test profile-specific behavior. Classes that implement ProfileRunnable or the SubProfileRunnable interfaces act as profile-specific subcomponents of the load test component. These classes implement the profile-specific behavior of the component.

There are two classes in the SocketRunnable example that implement ProfileRunnable interface: the SocketProfileRunnable and the SocketTestWrapper. The SocketProfileRunnable acts as a root profile runnable; it is a container for SocketTestWrappers. The SocketRunnable class keeps an instance of SocketProfileRunnable, where all configured SocketTestWrappers are stored. The SocketRunnable also keeps a Vector of SocketTestWrappers; this Vector represents a selection of ProfileRunnables for a specific Load Test profile. Each profile in the load test configuration keeps its copy of the component class. This guarantees that profile specific component configurations do not conflict and do not overwrite each other.

The populateSubProfileRunnables() method of the SocketRunnable fills the selectedProfileRunnables Vector based on the ProfileRunnableMap that is acquired from the profile specific PrivateContext. The ProfileRunnableMap contains load test profile-specific ProfileRunnable selections.

In order to let you select load test profile-specific ProfileRunnables, the SocketRunnable component implements the ProfileRunnableViewProvider interface as shown below.



SocketRunnable uses ProfileRunnableTreeView of the API.

Saving a Component Configuration in the Load Test Project

A load test component can save its configuration to a Load Test project by implementing the SaveablePropertiesProvider interface. The component configuration is represented by the SimRunnableProperties class, which stores component property name/value pairs.

The component properties must be saved as String values independent of the component classes. This ensures that a project can be read into the Load Test application even if the component is not deployed. At the same time, this allows a project to store different component type configurations. The container uses the tag field of the SimRunnableProperties class to return the appropriate properties instance to the component. The tag value should be set by the component developer. SoatestRunnable uses "SOCKET_RUNNABLE_SAVEABLE_PROPS_V.1" string as its saveable properties tag.

If saving your component configuration requires a more complex data structure than a set of name/value property pairs, you can use XML as an alternative. Serialize your component configuration to an XML representation and save the XML string in a single name/value pair of the SimRunnableProperties instance. See the SocketRunnableXMLSerializer for an example how to save and restore component configuration using XML.

Transferring a Component Configuration to Remote Machines

A load test component configuration, which can be either read from a load test project file or modified by you and your team, must be transferred to remote load test machines. The component state transfer should be implemented within the readExternal/writeExternal methods of the ParasoftExternalizable interface implemented by the component class. Since the configurable part of the SocketRunnable component is represented by the rootProfileRunnable member variable of type SocketProfileRunnable, it has to be transferred to the remote machines. The serialization of the SocketProfileRunnable class to the remote machines is demonstrated in the implementation of the readFrom/writeOn and readExternal/writeExternal methods this class. Alternatively, serialization to remote machines can reuse the code that saves component configuration to the load test project. For the SocketRunnable example, the SocketRunnableXMLSerializer class functionality can be reused to serialize/deserialize the component configuration to remote machines as an XML string.

Using jar Libraries to Extend Component Functionality

Your component code can use jar Java libraries of your choice. The SocketRunnable example code uses the JTidy library to analyze the HTML code that it can receive in response to a Web page request (see checkTidy method of the SocketTest class). The component dependency jar files must be included into the component jar archive lib folder (see the socket.jar component archive built by the component-build.xml ant script).

Localizing Component Resources

You can use the LocalizationUtil API class to localize component strings. The Localization class of the SocketRunnable example loads the appropriate resource bundle and wraps the LocalizationUtil methods for easier access.