Application Coverage

In this section:

Introduction

You can monitor and collect coverage data during manual or automated functional tests performed on a running web application server. You can also send coverage data and test results to DTP, which merges and correlates the data. The application coverage information can be displayed in the DTP Coverage Explorer (see the "Coverage Explorer" chapter in the DTP user manual), which provides insights about how well the application is tested, as well as the quality of your tests.

Prerequisites

The following components are required for collecting coverage:

- Java JDK 1.5
- Apache Maven, Gradle or Ant build system

Java JDK 8 line break calculations in bytecode are inconsistent with previous Java versions. This may lead to inconsistencies in coverage results.

If you use a source control system, ensure that your source control settings are properly configured; see Source Control Settings.

Process Overview

Jtest DTP Engine ships with a component called the coverage agent. The coverage agent is attached to the application under test (AUT) and monitors the code being executed as the AUT runs. When the coverage agent is attached to the AUT, a REST API is exposed that enables you to mark the beginning and end of each test and test session.

Metadata about the lines of code that can be covered (static coverage data) is collected by running a dedicated test configuration as part of the application build process. During test execution, interactions with the coverage agent are written to a dynamic coverage map, which contains markers that specify which lines of code were touched.

Jtest processes the dynamic coverage map and static coverage data. A coverage.xml file, which contains the coverage information, is produced and sent to DTP. When DTP receives the coverage data, it's loaded into a coverage image, which is a special tag that enables you to aggregate coverage data from runs with the same build ID. The coverage image enables you to associate coverage information with specific tests.

Test results are also sent to DTP from the tool executing the tests (i.e., SOAtest, tests executed by the DTP Engine, manual tests, etc.) in a report.xml file. If the build IDs for the coverage data file and the report match, DTP is able to correlate the data and display the coverage information.

Configuring the Application Under Test for Coverage

The following steps are required to prepare the application under test (AUT):

1. The static coverage file must be generated. The static coverage file contains metadata about user classes, methods, and lines; see Generating the Static Coverage File.
2. The coverage agent must be attached to the AUT; see Attaching the Coverage Agent to the AUT.
3. The coverage agent includes default settings for outputting files, determining scope, etc., but you can set properties in a configuration file to meet your application coverage goals; see Configuring the Coverage Agent.

Generating the Static Coverage File

The package that contains the static coverage file is created during the build process by the Jtest Maven, Gradle or Ant plugin. It must be generated on the build machine that contains the source code. The static coverage file can be used until the code changes.

In order to generate the package that contains the static coverage file, you need to execute an appropriate command in the AUT's main directory.

Maven

```
mvn package jtest:monitor
```

Gradle

```
gradle assemble jtest-monitor -I [INSTALL]/integration/gradle/init.gradle
```

Ant

```
ant -lib [INSTALL]/integration/ant/jtest-ant-plugin.jar -listener com.parasoft.Listener jtest-monitor
```

Ant requires all classes to be compiled before the monitor task is executed. Modify your project prior to the build and configure the task to ensure the correct sequence. The following example shows how the target can be configured:

```
<target name="jtest-monitor" depends="compile">
            <jtest:monitor/>
</target>
```

The monitor.zip package will be generated and placed into the build output directory. The path to the location will be printed on the console.

The package contains the following:

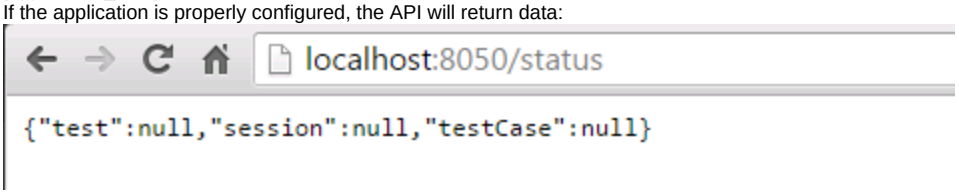
- static_coverage.xml - the file that contains static coverage information,
- agent.jar - the Jtest Java coverage agent jar archive,
- agent.properties - the agent settings file that contains scope parameters generated during the build process and other attributes,
- agent.sh/agent.bat - the script that generates the Jtest Java agent VM arguments necessary for attaching the agent to the AUT process.

Attaching the Coverage Agent to the AUT

1. Extract the contents of the monitor.zip package to the server machine.
2. Run the agent.sh/agent.bat script from the monitor package to generate the Jtest Java agent VM arguments. The scripts output the `-javaagent` VM argument to the console. You will need this argument to connect the agent to the AUT, which will result in the runtime_coverage subdirectory:

   ```
   Jtest Agent VM argument:
   -javaagent:"[path to agent dir]\agent.jar"=settings="[path to agent properties file]\agent.properties",
   runtimeData="[path to monitor dir]\monitor\ runtime_coverage"
   ```

3. Add the `-javaagent` flag to the application server's startup script and restart the server (see step 5c in Web Application Coverage Tutorial). Attac hing the coverage agent to the AUT exposes a REST API for controlling the agent.
4. Enter the following URL into the browser bar to verify that the coverage agent is set:
   [application_host]:8050/status/
   If the application is properly configured, the API will return data:



Configuring the Coverage Agent

Application servers usually contain more than one application. Additionally, common server classes or application libraries do not need to be instrumented. The Jtest DTP Engine only needs to collect coverage for application source code. Instrumenting all classes would be too time-consuming.

The application on the server is already built, so we cannot gather information about which classes come from the source code. For this reason, properly setting the scope of the coverage agent is very important.

You can configure the coverage agent with:

- the agent.properties file

- arguments to `-javaagent`

The agent.properties file is generated in the monitor.zip package (see Generating the Static Coverage File). It contains properties that can be modified to properly configure the coverage agent. The following example shows how the coverage agent can be configured with the agent.properties.file:

```
jtest.agent.runtimeData=[path to runtime_coverage directory]
jtest.agent.includes=com/myapp/data,com/myapp/common/**
jtest.agent.excludes=com/myapp/transport/*,com/myapp/autogen/**
jtest.agent.autostart=false
```

Adding properties to `-javaagent` requires modifying their names by removing the "jtest.agent" prefix, for example:

```
-javaagent:"[path to agent dir]\agent.jar"=settings="[path to agent properties file]\agent.properties",
autostart=true
```

Important

Properties provided as arguments to `-javaagent` override properties configured in the agent.properties file. In the above examples, the `autostart=true` will override `jtest.agent.autostart=false` from the agent.properties file.

The following table describes all properties that can be used to configure the coverage agent:

Property	Descripton
`jtest.agent.runtimeData`	Specifies where the runtime data will be stored. The following example will create files in the in 'C:/tmp/myapp/' directory with `runtime_`' as the name prefix: `'C:/tmp/myapp/runtime'`
`jtest.agent.includes`	A coma-separated list of patterns that specify classes to be instrumented. The following wildcards are supported: * matches zero or more characters ** matches multiple directory levels In the following example, all classes from the `com.myapp.data` package and all classes from package and subpackages that start with `com.myapp.common` will be instrumented: `com/myapp/data/*,com/myapp/common/**`
`jtest.agent.excludes`	Coma separated list of patterns that specify classes to be excluded from instrumentation. The following wildcards are supported: * matches zero or more characters ** matches multiple directory levels In the following example, all classes from the `com.myapp.transport` package and all classes from package and subpackages that start with `com.myapp.autogen` will be excluded from instrumentation: `com/myapp/transport/*,com/myapp/autogen/**`
`jtest.agent.autostart`	Enables/disables automatic runtime data collection; the default is `true`.
`jtest.agent.port`	Sets up agent communication port; the default is `8050`.
`jtest.agent.debug`	Enables/disables verbose output to console; the default is `false`.
`jtest.agent.collectTestCoverage`	Enables/disables collecting coverage information for test cases. The default value is `false`.
`jtest.agent.enableMultiuserCoverage`	Enables/disables collecting web application coverage for multiple users; the default value is `false`. Setting this property to true allows you to collect multi-user coverage with Coverage Agent Manager. See the "Coverage Agent Manager (CAM) section of the DTP documentation for details.

Test Configuration and Execution

You can use SOAtest to run functional tests (refer the Application Coverage chapter of the SOAtest documentation to set up the test configuration), as well as execute manual tests. At the end of the test session, coverage will be saved in `runtime_coverage_[timestamp].data` files in the directory specified in SOAtest. This information will eventually be merged with the static coverage data to create a `coverage.xml` file and uploaded to DTP.

Uploading Test Results to DTP

If you use Jtest with CAM

1. Go to **Report Center** in the DTP interface.
2. Click the gear icon and choose **Report Center Settings> Additional Settings> Report Center Administration> Tools> Data Collector Upload Form** (requires admin permissions).
3. Click **Choose File** and browse for the report.xml file you downloaded from CAM.
4. Click the **Upload** button to upload the file to DTP.

If you use SOAtest

For tests executed by SOAtest, the SOAtest XML report will need to be uploaded to DTP. See the "Uploading Rest Results to DTP" section in the Application Coverage topic in the SOAtest documentation for details.

Generating a Dynamic Coverage Data File and Uploading It to DTP

1. Ensure that Jtest DTP Engine is properly configured, including DTP, scope and authorship settings. See Connecting to DTP, Sending Results and Publishing Source Code to DTP, Configuration.
2. Configure the following settings in the jtestcli.properties file in order to properly merge coverage data:
   - `report.coverage.images` - Specifies a set of tags that are used to create coverage images in DTP. A coverage image is a unique identifier for aggregating coverage data from runs with the same build ID. DTP supports up to three coverage images per report.
   - `session.tag` - Specifies a unique identifier for the test run and is used to distinguish different runs on the same build.
   - `build.id` - Specifies a build identifier used to label results. It may be unique for each build, but it may also label several test sessions executed during a specified build.
3. Run the `Calculate Application Coverage` test configuration using the following arguments:

   ```
   jtestcli -staticcoverage [path to static_coverage.xml file] -runtimecoverage [path/dir] -config
   "builtin://Calculate Application Coverage" -publish
   ```

   This ensures that Jtest has access to the runtime coverage data generated during test execution, as well as the static coverage data, which is required to fill the coverage.xml file with runtime coverage data,

Reviewing Coverage in DTP

You can use the Coverage Explorer in DTP to review the application coverage achieved during test execution. See the DTP documentation for details on viewing coverage information.

Known Limitations

- If multiple users are simultaneously accessing the same web application, the coverage data they collect may be mixed. To ensure that coverage is properly associated with individual users, the multiuser mode must be enabled (see Configuring the Coverage Agent).
- The HTTP or HTTPS protocols are required to enable the multiuser mode, as the user-specific information must be provided with the HTTP header.
- In the multiuser mode, the "default" user (the user who has not specified their ID) may collect extra coverage information from other users who are accessing the same web application.
- In the multiuser mode, assigning coverage collected for multithreaded application to individual users is limited. Coverage data for child threads is not assigned to the user who is actually accessing the application, but to the "default" user.
- Coverage data collected for web application initialization is not assigned to a specific user, but to the "default" user.