WHITE PAPER

COMPOSITE SOFTWARE

Composite Data Virtualization

# *Composite PS Promotion and Deployment Tool*

# *Regression Module User Guide*

Composite Professional Services

November 2014

## TABLE OF CONTENTS

## DOCUMENT CONTROL

### Version History

| Version | Date | Author | Description |
|---|---|---|---|
| 1.0 | 6/27/2011 | Sergei Sternin | Initial revision for Regression Module User Guide |
| 1.0.1 | 8/1/2011 | Mike Tinius | Revision due to Architecture changes |
| 1.0.2 | 5/10/2012 | Mike Tinius | Added new feature – save results to file & file comparison |
| 1.0.3 | 6/8/2012 | Mike Tinius | Added new features for Migration Testing and Performance Testing and enhanced logging. |
| 1.0.4 | 8/10/2012 | Mike Tinius | Added separate log file delimiters vs. data file delimiters. Delimiters include: [COMMA\,\PIPE\|\TAB\TILDE\~] |
| 1.1 | 10/08/2012 | Mike Tinius | Added notes about New Web Service BARE and WRAPPED parameter style. Implemented support for New Composite Web Service. |
| 1.2 | 02/15/2013 | Mike Tinius | Added documentation regarding log output. Added ability to use variables within the RegressionModule.xml definition. |
| 1.3 | 05/08/2013 | Mike Tinius | Added a comment in the documentation about characters allowed in web service operation names. |
| 1.4 | 08/19/2013 | Mike Tinius | Fixed documentation to show output log file location and names. |
| 3.0 | 8/21/2013 | Mike Tinius | Updated docs with Cisco format. |
| 3.1 | 2/3/2014 | Mike Tinius | Added resources filter to createRegressionInputFile. |
| 3.2 | 2/14/2014 | Mike Tinius | Added new methods generateRegressionSecurityXML and executeSecurityTest. Prepare docs for open source. |
| 3.3 | 3/24/2014 | Mike Tinius | Changed references of XML namespace to www.dvbu.cisco.com |
| 3.4 | 6/30/2014 | Mike Tinius | Added outputFilename to regression SQL input file. |
| 3.5 | 11/17/2014 | Mike Tinius | Update license. |

### Related Documents

| Document | File Name | Author |
|---|---|---|
| *Composite PS Promotion and Deployment Tool User's Guide v1.0* | *Composite PS Promotion and Deployment Tool User's Guide v1.0.pdf* | Mike Tinius |
| | | |

### Composite Products Referenced

| Composite Product Name | Version |
|---|---|
| Composite Information Server | 5.1, 5.2, 6.0, 6.1, 6.2 |

# INTRODUCTION

## License



This software is released under the Eclipse Public License. The details can be found in the file LICENSE. Any dependent libraries supplied by third parties are provided under their own open source licenses as described in their own LICENSE files, generally named .LICENSE.txt. The libraries supplied by Cisco as part of the Composite Information Server/Cisco Data Virtualization Server, particularly csadmin-XXXX.jar, csarchive-XXXX.jar, csbase-XXXX.jar, csclient-XXXX.jar, cscommon-XXXX.jar, csext-XXXX.jar, csjdbc-XXXX.jar, csserverutil-XXXX.jar, csserver-XXXX.jar, cswebapi-XXXX.jar, and customproc-XXXX.jar (where -XXXX is an optional version number) are provided as a convenience, but are covered under the licensing for the Composite Information Server/Cisco Data Virtualization Server. They cannot be used in any way except through a valid license for that product.

This software is released AS-IS!. Support for this software is not covered by standard maintenance agreements with Cisco. Any support for this software by Cisco would be covered by paid consulting agreements, and would be billable work.

## Purpose

The purpose of the Regression Module User Guide is to describe its functionality and demonstrate how to use it.

## Audience

This document is intended to provide guidance for the following users:

- Architects
- Developers
- Administrators.
- Operations personnel.

## Module overview

The Regression Module assimilates two disparate tools (Pubtest and Perftest) that ship with each Composite server into a unified testing framework. While both of these two tools are quite simple in their implementation, the PDTool Regression Module takes these basic concepts and significantly enhances the overall features.

- Pubtest is a JDBC client that connects to an instance of CIS and executes requests against published resources on that instance based on the definitions of calls in pubtest input file.
- Perftest executes a number of threads for each query and prints out the performance statistics.

The Regression Module addresses the following requirements:

- The ability to execute an automated test against Composite published resources.

- The ability to define a test and the parameters required for executing a test.
- The ability to capture query execution results from queries and compare them.
- The ability to report/log the results of queries such as “PASS” or “FAIL”.

The Regression Module provides the following major capabilities:

1. Generate Input File

   Generate the input file by introspecting a CIS instance for view, procedures and web services. The input file is what drives the three types of tests (Functional, Migration, Performance and Security).

2. Functional Test

   Test whether the published view, procedure or web service is functionally operational.

3. Migration Test

   When moving from one release of Composite to another, test whether the data sets from two independent migration tests are equivalent.

4. Regression Test

   When moving from one application release to another, test whether the data sets from two independent migration tests are equivalent.

5. Performance Tests

   Test the performance by executing several threads over a period of time and reporting the results.

6. Security Tests

   The security test correlates users with functional test queries and provides a way to test the expected outcome for a given user/query combination to determine if the test results in a PASS or FAIL.

7. Log Comparison

   Compare the logs from two migration or regression tests to automate the comparison checking.

The Regression Module greatly enhances the experience by logging the results to specific Regression Module log files in addition to displaying on the command line. It also integrates with the PD Tool log file. Another key aspect of combining pubtest and perftest was to unify the input file so that it is used with all types of test whereas pubtest and perftest had their own input file formats.

**Here are the highlights of the Regression Module.**

- Like pubtest, Regression Module supports **published** Views, Procedures and Web Services.

- The input file creation part of the module is configurable to provide the same SQL for requests against all Views and same SQL for all Procedures, i.e. it doesn't contain filters or parameter values. For example: **SELECT count(*) cnt FROM.** After the View or Procedure name is appended to the end, such request can be executed. Or, if file creation and execution are two separate steps, the SQL can be manually changed in the input file after it is generated and before the test is executed.
- Regression Module adds on to the above theme by providing a section in the regression XML "`<regressionQueries>`" for defining custom queries. During the input file generation, the user may configure to use the regression queries instead of the default query mentioned above.
- While pubtest only provided a method of executing a test, Regression Module provides four methods of functionality which include: generate input file, execute regression test, compare result files, and compare log files.
- Like in pubtest, in Regression Module one request is executed for one call definition in the input file.
- Pubtest execution is driven solely the content of the input file, while Regression Module has configuration setting in the Module's configuration file **RegressionModule.xml** to further control which parts of the input file will be used during the test. For example, we can configure to execute only WS requests or only requests against Views.
- In Pubtest CIS connection information is provided in the command line, while in the Regression Module it is provided in the **servers.xml** that is common for all modules of the tool.
- In Pubtest if there is no data returned for a specific request, an exception is thrown, while in Regression Module such request is considered successful.
- Criteria of successful execution of a request in the Regression Module is that it doesn't throw an exception. Depending on the specific implementations simple **SELECT * FROM** requests may throw exceptions for some procedures, those may require manual editing of the queries in the input file before test execution. If an exception is thrown for a request, it is printed to the log file.
- The Regression Module input file differs slightly from the pubtest input file in that it has been enhanced for web services by providing support for soap12 in addition to original soap11. The existing pubtest input file will need to be migrated for web services only before it can be used.
- The Regression Module supports the "New Composite Web Service" and the concept of BARE and WRAPPED parameters. However, BARE parameters are only supported when there is only a single INPUT or OUTPUT parameter.

## USE CASES

### *Introduction*

The following use cases are supported by the Regression Module.

1. Functional Test

   This represents the current functionality which tests whether the published view, procedure or web service is functional or not. All that is necessary for this test is to execute a SELECT COUNT(1) cnt FROM <view> or SELECT COUNT(*) cnt FROM <procedure> to prove that the view or procedure is functional.

   This type of test is also known as a "smoke" test.

2. Migration or Regression Test (data set comparison)

   The PD Tool Regression Module will help Composite PS during Migrations from one version of Composite to another by performing the regression testing on the new version of Composite or by performing a Regression test against two releases of a project. These tests are functionally the same. The caveat to this test is that the result sets used for comparison must be derived from the same data sources. Any changes in data values may result in differences in result set and thus comparisons will not be equal.

3. Performance Test (log comparison)

   The PD Tool Regression Module can be used by Composite PS and customers to perform performance tests against views, procedures and web services. The two logs generated during the query execution are compared to determine success or failure based on a duration comparison.

4. Security Test (users/privileges)

   The PD Tool Regression Module can be used by Composite PS and customers to perform security tests. The security test correlates users with functional test queries. It provides a way to test the expected outcome for a given user and query to determine if the test results in a PASS or FAIL. All tests are executed and an overall result of PASS or FAIL is awarded. The security test can be configured to throw an exception upon an overall status of FAIL.

### *1. Functional Test Process:*

Scenario: The QA team wants to have a set of automated regression tests performed against CIS published resources each time a new release is rolled into test. They simply want to make sure the views are functional and do not throw any errors upon execution. No matter what query is contained in the regression queries section of the Regression XML, it will always issue a query based on the default published view query and the default published procedure query. Here is a high-level summary of the steps for the "Functional Test Process":

- Step 1 – Create Regression Module XML for "Generating the input file"

- Step 2 – Modify Regression Module XML for “Executing the regression test”
- Step 3 – Create deployment plan for “Generating the input file”
- Step 4 – Create deployment plan for “Executing the regression test”
- Step 5 - Generate the Regression input file
- Step 6 – Execute the Regression Test plan.
- Step 7 – Review the results of the output file.

Functional Test Details

Step 1 – Create Regression Module XML for “Generating the input file”

- Make a copy of RegressionModule.xml and call it something like RegressionFunctTest.xml
- Determine the location to generate the input file.

```
<inputFilePath>C:\\tmp\\functest.inp</inputFilePath>
```

- Focus on the following section of the XML:

```
<newFileParams>
```

- Generate the default query for Views, Procedures or Web Services with the following parameters:
  - useDefaultViewQuery is set to yes
  - useDefaultProcQuery is set to yes
  - useDefaultWSQuery is set to yes

  Example Regression Module XML (.xml) entry:

```
<useDefaultViewQuery>yes</useDefaultViewQuery>
<useDefaultProcQuery>yes</useDefaultProcQuery>
<useDefaultWSQuery>yes</useDefaultWSQuery>

<!--  Default queries for Views and Procedures. -->
<publishedViewQry>SELECT count(1) cnt FROM</publishedViewQry>
<publishedProcQry>SELECT count(*) cnt FROM</publishedProcQry>
```

- Select the list of data sources that you want to generate tests for:
  - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when generating the input file. If set to “yes” then it will use all datasources it finds in CIS to generate the input file.
  - datasources – list contains all databases and web services to be generated.

Example Regression Module XML (.xml) entry:

```
<useAllDatasources>no</useAllDatasources>
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
</datasources>
```

- Select the optional list of resources that you want to generate tests for. These can be tables or web services and they can use wildcards.
    - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when generating the input file. If set to “yes” then it will use all datasources it finds in CIS to generate the input file.
    - datasources – list contains all databases and web services to be generated.

Example Regression Module XML (.xml) entry:

```
<resources>
      <resource>TEST00</resource>
      <resource>testWebService00</resource>
</resources>
```

Resource filter: This is a list of resources for which to generate the input file.

A wild card ".*" may be used at the catalog or schema level to denote that all resources under that level should be compared.

A fully qualified resource may be identified as well. This is a filter that can be fine-tuned for one or more resources.

If left empty then all resources are generated. Overlapping resource wild cards will use the highest level specified in this list.

Examples of filters for web services:

- this is a legacy web service

  <resource>services.webservices.testWebService00</resource>
- filter all new composite soap 11 web services

  <resource>soap11.*</resource>
- filter all new composite soap 12 web services

  <resource>soap12.*</resource>
- new composite soap 11 web service with specific path

  <resource>soap11.testWebService00_NoParams_bare</resource>

- new composite soap 11 web service located in the folder1 web service folder

  <resource>soap11.folder1.*</resource> -

- new composite soap 11 web service located in folder2 and matching a wild card starting with testWebService00

  <resource>soap11.folder2.testWebService00*</resource>

Step 2 – Modify Regression Module XML for "Executing the regression test"

- Use RegressionFunctTest.xml
- Focus on the following section of the XML:

```
<testRunParams>
      <testType>functional</testType>
```

- Configure execution behavior attributes
  - logFilePath – set this to the location of the summary execution log file.
  - logDelimiter – set as PIPE. Delimiter for summary log file.
  - baseDir – leave this blank for functional tests. There is no need to write out results for each query to a file since the only purpose of this test is to do a "select count" and test whether the resource is functional or not.

Example Regression Module XML (.xml) entry:

```
<logFilePath>C:/tmp/cis51/functestrun.log</logFilePath>
<logDelimiter>PIPE</logDelimiter>
<logAppend>no</logAppend>
<baseDir></baseDir>
<delimiter>PIPE</delimiter>
<printOutput>summary</printOutput>
```

- Select the list of data sources that you want to execute tests for:
  - useAllDatasources – set this to "no" when you want to explicitly define your own list of datasources when executing the input file. If set to "yes" then it will use all datasources it finds in the generated input file to execute queries for.
  - datasources – list contains all databases and web services to be tested. This list can be different than the list you used to generate the input file. This is the list of datasources found in the input file to execute tests for.

Example Regression Module XML (.xml) entry:

```
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
```

```
</datasources>
```

Step 3 – Create deployment plan for “Generating the input file”

- Edit the deploy plan (.dp) file used for creating the input file:

  Example deployment plan file entry for RegressionFuncGen.dp:

```
PASS   TRUE   ExecuteAction      createRegressionInputFile   $SERVERID
       Test1"$MODULE_HOME/RegressionFuncTest.xml"
             "$MODULE_HOME/servers.xml"
```

Step 4 – Create deployment plan for “Executing the regression test”

- Edit the deploy plan (.dp) file used for creating the input file:

- Example deployment plan file entry for RegressionFunctional.dp:

```
PASS   TRUE   ExecuteAction      executeRegressionTest       $SERVERID
       Test1  "$MODULE_HOME/RegressionFuncTest.xml"
                    "$MODULE_HOME/servers.xml"
```

Step 5 – Generate the Regression input file.

- Generate the regression input file using the default queries and not the pre-defined regression queries. The following shows an example of how to execute a deployment plan that was previously created.

- Execution is performed from the <your-root-path>/PDTool/bin directory:

  Example Execute Line:

```
Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionFuncGen.dp

Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionFuncGen.dp
```

Step 6 – Execute the Regression Test plan.

- Executing the regression test is accomplished by running the ExecutePDTool.bat and pointing it to the deployment plan you have created.

- This regression test will use the queries generated in the regression input file and execute them against CIS. It will log the results into a summary execution file in the file system. It will use several filters that you previously set up in the regression XML file. For example, you can turn on and off execution of the three categories, queries, procedures and web services. You can also filter on which datasources you execute tests for. The input file may contain a broad list of datasources and queries but you don’t have to run all the tests.

- Execution is performed from the <your-root-path>/PDTool/bin directory:

  Example Execute Line:

```
Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionFunctional.dp

Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionFunctional.dp
```

Step 7 – Review the results of the output file.

The results of the regression test are saved to a summary log file of your choosing. Review the section later in this document title "Query Execution Log File" for specifics on file layout and example content.

## 2. *Migration or Regression Test Process:*

Scenario: The Composite PS team has been called in to a customer to help with migration from CIS 5.1/5.2 to CIS 6.1 (or higher). The customer wants to insure that result sets are consistently computed when moving from one CIS version to the next. The job of the PS team is to guide the customer through the migration process and come up with a set of automated regression tests. The PS team will execute the same scripts against the current system to establish a baseline and the new migrated system. PD Tool Regression Module will then be used to compare the result files and determine if there are any differences and address the issues.

Caveat: For the migration test, the data sources must be the same between the two versions of composite in order to get an accurate test. This test will not work as expected if the underlying data is different.

The Migration Test Process is broken down into three major steps each containing a sub-process of steps to perform.

### Current System Migration Test

- Step 1 – Create Regression Module XML for "Generating the input file"
- Step 2 – Create Regression Module XML for "Executing the regression test"
- Step 3 – Create deployment plan for "Generating the input file"
- Step 4 – Create deployment plan for "Executing the regression test"
- Step 5 – Generate the Migration input file.
- Step 6 – Execute the Migration Test plan.
- Step 7 – Review the results of the output file.

### Migrated System Migration Test

- Step 1 – Create Regression Module XML for "Executing the regression migration test"
- Step 2 – Create deployment plan for "Executing the regression migration"
- Step 3 – Execute the Regression Migration plan
- Step 4 – Review the results of the output file.

### Comparison Test

- Step 1 – Create Regression Module XML for “Executing the comparison test”
- Step 2 – Create deployment plan for “Executing the comparison test”.
- Step 3 – Execute the Regression Comparison Test plan.
- Step 4 – Review the results of the output file.

Migration Test Details

**Current System Migration Test**

Step 1 – Create Regression Module XML for “Generating the input file”

- Make a copy of RegressionModule.xml and call it something like RegressionMigrationTest.xml
- Modify a regression test identifier

  ```
  <id>CIS51</id>
  ```
- Determine the location to generate the input file.

  ```
  <inputFilePath>C:\\tmp\\migrationtest.inp</inputFilePath>
  ```
- Focus on the following section of the XML:

  ```
  <newFileParams>
  ```
- Generate the exact query for Views, Procedures or Web Services with the following parameters:
  - useDefaultViewQuery is set to no
  - useDefaultProcQuery is set to no
  - useDefaultWSQuery is set to no

  Example Regression Module XML (.xml) entry:

  ```
  <useDefaultViewQuery>no</useDefaultViewQuery>
  <useDefaultProcQuery>no</useDefaultProcQuery>
  <useDefaultWSQuery>no</useDefaultWSQuery>
  ```
- Provide a list of exact queries/input for Views, Procedures and Web Services using the “regressionQuery” section of the RegressionMigrationTest.xml. An optional durationDelta may be provided for each query if desired. If not the default durationDelta is used when comparing result files.

  Example “regressionQuery” section of the Regression Module XML (.xml) entry:

  ```
  <regressionQueries>
    <regressionQuery>
        <datasource>MYTEST</datasource>
        <query>SELECT * FROM ViewSales WHERE CategoryID=1</query>
        <durationDelta>000 00:00:01.0000</durationDelta>
    </regressionQuery>
    ...
  ```

```
</regressionQueries>
```

- Select the list of data sources that you want to generate tests for:
    - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when generating the input file. If set to “yes” then it will use all datasources it finds in CIS to generate the input file.
    - datasources – list contains all databases and web services to be generated.

Example Regression Module XML (.xml) entry:

```
<useAllDatasources>no</useAllDatasources>
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
</datasources>
```

Step 2 – Create Regression Module XML for “Executing the regression test”

- Use RegressionMigrationTest.xml
- Focus on the following section of the XML:

```
<testRunParams>
      <testType>migration</testType>
```

- Configure execution behavior attributes
    - logFilePath – set this to the location of the summary execution log file.
    - logDelimiter – set as PIPE. Delimiter for summary log file.
    - baseDir – provide the directory where result files for each query will be written.

Example Regression Module XML (.xml) entry:

```
<logFilePath>C:/tmp/cis51/migrationtestrun.log</logFilePath>
<logDelimiter>PIPE</logDelimiter>
<logAppend>no</logAppend>
<baseDir> C:/tmp/cis51</baseDir>
<delimiter>PIPE</delimiter>
<printOutput>summary</printOutput>
```

- Select the list of data sources that you want to execute tests for:
    - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when executing the input file. If set to “yes” then it will use all datasources it finds in the generated input file to execute queries for.

- datasources – list contains all databases and web services to be tested. This list can be different than the list you used to generate the input file. This is the list of datasources found in the input file to execute tests for.

Example Regression Module XML (.xml) entry:

```
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
</datasources>
```

Step 3 – Create deployment plan for "Generating the input file"

- Edit the deploy plan (.dp) file used for creating the input file:

Example deployment plan file entry for RegressionMigrationGen.dp:

```
PASS   TRUE   ExecuteAction      createRegressionInputFile   $SERVERID
       CIS51 "$MODULE_HOME/RegressionFuncTest.xml"
             "$MODULE_HOME/servers.xml"
```

Step 4 – Create deployment plan for "Executing the regression test"

- Edit the deploy plan (.dp) file used for creating the input file:
- Example deployment plan file entry for RegressionMigration.dp:

```
PASS   TRUE   ExecuteAction      executeRegressionTest       $SERVERID
       CIS51  "$MODULE_HOME/RegressionMigrationTest.xml"
                           "$MODULE_HOME/servers.xml"
```

Step 5 – Generate the Migration input file.

- Generate the regression input file using the default queries and not the pre-defined regression queries. The following shows an example of how to execute a deployment plan that was previously created.
- Execution is performed from the <your-root-path>/PDTool/bin directory:

Example Execute Line:

```
Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionMigrationGen.dp

Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionMigrationGen.dp
```

Step 6 – Execute the Migration plan.

- Executing the regression test is accomplished by running the ExecutePDTool.bat and pointing it to the deployment plan you have created.
- This regression test will use the queries generated in the regression input file and execute them against CIS. It will log the results into a summary execution file in the file system. It will use several filters that you previously set up in the regression XML file. For example, you can turn on and off execution of the three categories, queries, procedures and web services. You can also filter on which datasources you execute tests for. The input file may contain a broad list of datasources and queries but you don't have to run all the tests.

- Execution is performed from the <your-root-path>/PDTool/bin directory:

  Example Execute Line:

  Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionMigration.dp

  Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionMigration.dp

Step 7 – Review the results of the output file.

This execution is now considered a point in time execution. This is merely the first part of a migration where a baseline of queries is established against the current system.

The results of the regression test are saved to a summary log file of your choosing. Review the section later in this document title "Query Execution Log File" for specifics on file layout and example content.

Do not edit the individual result files as the checksum value may be affected if you inadvertently save the file. This will throw off the comparison.

**Migrated System Migration Test**

Step 1 – Create Regression Module XML for "Executing the regression migration test"

- Use RegressionMigrationTest.xml that you previously created.

  Copy the entire entry for `<id>CIS51</id>` an create a new entry `<id>CIS61</id>`
- Modify the regression test identifier

  ```
  <id>CIS61</id>
  ```
- Focus on the following section of the XML:

  ```
  <testRunParams>
        <testType>migration</testType>
  ```
- Configure execution behavior attributes

  - logFilePath – set this to the location of the summary execution log file. Notice how the path is pointing to a different directory than the "current test" was configured for.
  - logDelimiter – set as PIPE. Delimiter for summary log file.
  - baseDir – provide the directory where result files for each query will be written. Notice how the path is pointing to a different directory than the "current test" was configured for.

  Example Regression Module XML (.xml) entry:

  ```
  <logFilePath>C:/tmp/cis61/migrationtestrun.log</logFilePath>
  <logDelimiter>PIPE</logDelimiter>
  <logAppend>no</logAppend>
  <baseDir>C:/tmp/cis61</baseDir>
  <delimiter>PIPE</delimiter>
  <printOutput>summary</printOutput>
  ```

- Select the list of data sources that you want to execute tests for:
    - Use exactly the same configuration as you did for the “current test”. There must be an apples to apples comparison.
    - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when executing the input file. If set to “yes” then it will use all datasources it finds in the generated input file to execute queries for.
    - datasources – list contains all databases and web services to be tested. This list can be different than the list you used to generate the input file. This is the list of datasources found in the input file to execute tests for.

  Example Regression Module XML (.xml) entry:

```
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
</datasources>
```

Step 2 – Create deployment plan for “Executing the regression migration test”

- Edit the deploy plan (.dp) file used for creating the input file:
- Example deployment plan file entry for RegressionMigration.dp:

```
PASS   TRUE   ExecuteAction      executeRegressionTest       $SERVERID
       CIS61  "$MODULE_HOME/RegressionMigrationTest.xml"
                          "$MODULE_HOME/servers.xml"
```

Step 3 – Execute the Regression Migration plan.

- Executing the regression test is accomplished by running the ExecutePDTool.bat and pointing it to the deployment plan you have created.
- Execution is performed from the <your-root-path>/PDTool/bin directory:

  Example Execute Line:

```
Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionMigration.dp

Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionMigration.dp
```

Step 4 – Review the results of the output file.

This execution is now considered a second point in time execution. This is merely the second part of a migration where the migration queries are established against the migrated system.

The results of the regression test are saved to a summary log file of your choosing. Review the section later in this document title “Query Execution Log File” for specifics on file layout and example content.

Do not edit the individual result files as the checksum value may be affected if you inadvertently save the file. This will throw off the comparison.

**Comparison Test**

Note: In terms of execution, this major step may be combined with "Migration Test" step when executing the plan file. It does not have to be a separate execution step as inferred by this documentation. The documentation only separates the step for the sake of clarity of documenting the process.

Step 1 – Create Regression Module XML for "Executing the comparison test"

- Use RegressionMigrationTest.xml that you previously created.
- Modify the regression test identifier

  ```
  <id>CIS61</id>
  ```
- Focus on the following section of the XML:

  ```
  <compareFiles>
  ```
- Configure execution behavior attributes
  - logFilePath – set this to the location of the summary execution log file. Notice how the path is pointing to a different directory than the "current test" was configured for.
  - logDelimiter – set as PIPE. Delimiter for summary log file.
  - baseDir1 – provide the directory where the results from the "current test" were saved.
  - baseDir2 – provide the directory where the results from the "migration test" were saved.

  Example Regression Module XML (.xml) entry:

  ```
  <logFilePath>C:/tmp/cis61/migrationcompare.log</logFilePath>
  <logDelimiter>PIPE</logDelimiter>
  <logAppend>no</logAppend>
  <baseDir1>C:/tmp/cis51</baseDir>
  <baseDir2>C:/tmp/cis61</baseDir>
  <delimiter>PIPE</delimiter>
  <printOutput>summary</printOutput>

  <compareQueries>yes</compareQueries>
  <compareProcedures>yes</compareProcedures>
  <compareWS>yes</compareWS>
  ```
- Select the list of data sources that you want to execute tests for:
  - Use exactly the same configuration as you did for the "current test" and "migration test". There must be an apples to apples comparison.

- useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when executing the input file. If set to “yes” then it will use all datasources it finds in the generated input file to execute queries for.

- datasources – list contains all databases and web services to be tested. This list can be different than the list you used to generate the input file. This is the list of datasources found in the input file to execute tests for.

Example Regression Module XML (.xml) entry:

```
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
</datasources>
```

- Finally there is an optional filter capability that will be mentioned here. It is not recommended to use this for migration tests as you want to get a complete system comparison when doing a migration. The additional filter allows you to define which resources to perform the comparison on.

    - This is a list of resources for which to perform comparisons. A wild card ".*" may be used at the catalog or schema level to denote that all resources under that level should be compared. A fully qualified resource may be identified as well. This is a filter that can be fine-tuned for one or more resources. If left empty then all resources are compared. Overlapping resource wild cards will use the highest level specified in this list.

Example Regression Module XML (.xml) entry:

```
      <resources>
            <resource>CAT1.SCH1.LookupProduct</resource>
            <resource>CAT1.SCH2.*</resource>
            <resource>SCH1.*</resource>
            <resource>TEST1.SCH.VIEW1</resource>
            <resource>ViewSales</resource>
            <resource>LookupProduct</resource>
            <resource>services.testWebService00.*</resource>
      </resources>
```

Step 2 – Create deployment plan for “Executing the comparison test”.

- Edit the deploy plan (.dp) file used for creating the input file:

- Example deployment plan file entry for RegressionMigration.dp:

```
PASS   TRUE   ExecuteAction      compareRegressionFiles       $SERVERID
       CIS61  "$MODULE_HOME/RegressionMigrationTest.xml"
                          "$MODULE_HOME/servers.xml"
```

Step 3 – Execute the Regression Migration plan.

- Executing the regression test is accomplished by running the ExecutePDTool.bat and pointing it to the deployment plan you have created.

- Execution is performed from the <your-root-path>/PDTool/bin directory:

  Example Execute Line:

  ```
  Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionMigration.dp

  Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionMigration.dp
  ```

Step 4 – Review the results of the output file.

This is the final part of a migration where the result files from the current system test and the migrated test are compared to determine if they match and thus a successful test has occurred.

The results of the comparison test are saved to a summary log file of your choosing. Review the section later in this document title "Query Execution File Comparison Log" for specifics on file layout and example content.

## 3. *Performance Test Process:*

Scenario: The QA team wants to have a set of automated regression tests performed against CIS published resources each time a new release is rolled into test to insure that performance has not changed with some level of tolerance. This will require having more than functional queries execute against the target system. The QA team will want to have specific queries with where clauses and log the timings of each of the queries. Additionally, the QA team will have had to establish a baseline with these set of queries on a previously release so that they can compare the timings and insure that each query is within the allowed tolerance level.

Caveat: For the performance test, the data sources may be different between the two versions of composite. However, if the data source content is substantially different between execution runs, this may adversely affect the timings. Plan ahead when developing the performance tests.

The Performance Test Process is broken down into three major steps each containing a sub-process of steps to perform.

**Current System Performance Test**

- Step 1 – Create Regression Module XML for "Generating the input file"
- Step 2 – Create Regression Module XML for "Executing the regression test"
- Step 3 – Create deployment plan for "Generating the input file"
- Step 4 – Generate the Performance Test input file.
- Step 5 – Create deployment plan for "Executing the regression performance test"
- Step 6 – Execute the Performance deployment plan.

- Step 7 – Review the results of the output file.

**New Release System Performance Test**

- Step 1 – Create Regression Module XML for “Executing the regression performance test”
- Step 2 – Create deployment plan for “Executing the regression performance test”
- Step 3 – Execute the Regression Performance plan.
- Step 4 – Review the results of the output file.

**Compare Logs**

- Step 1 – Create Regression Module XML for “Executing the compare logs”
- Step 2 – Create deployment plan for “Executing the compare logs”.
- Step 3 – Execute the Regression Performance plan.
- Step 4 – Review the results of the output file.

Performance Test Details

**Current System Performance Test**

Step 1 – Create Regression Module XML for “Generating the input file”

- Make a copy of RegressionModule.xml and call it something like RegressionPerformanceTest.xml
- Modify a regression test identifier

```
<id>rel1.1</id>
```

- Determine the location to generate the input file.

```
<inputFilePath>C:\\tmp\\perftest.inp</inputFilePath>
```

- Focus on the following section of the XML:

```
<newFileParams>
```

- Generate the exact query for Views, Procedures or Web Services with the following parameters:
  - useDefaultViewQuery is set to no
  - useDefaultProcQuery is set to no
  - useDefaultWSQuery is set to no

Example Regression Module XML (.xml) entry:

```
<useDefaultViewQuery>no</useDefaultViewQuery>
<useDefaultProcQuery>no</useDefaultProcQuery>
<useDefaultWSQuery>no</useDefaultWSQuery>
```

- Provide a list of exact queries/input for Views, Procedures and Web Services using the “regressionQuery” section of the RegressionPerformanceTest.xml:

  Example “regressionQuery” section of the Regression Module XML (.xml) entry:

```
<regressionQueries>
  <regressionQuery>
      <datasource>MYTEST</datasource>
      <query>SELECT * FROM ViewSales WHERE CategoryID >
1</query>
  </regressionQuery>
  ...
</regressionQueries>
```

- Select the list of data sources that you want to generate tests for:

  - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when generating the input file. If set to “yes” then it will use all datasources it finds in CIS to generate the input file.

  - datasources – list contains all databases and web services to be generated.

  Example Regression Module XML (.xml) entry:

```
<useAllDatasources>no</useAllDatasources>
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
</datasources>
```

Step 2 – Create Regression Module XML for “Executing the regression test”

- Use RegressionMigrationTest.xml

- Focus on the following section of the XML:

```
<testRunParams>
      <testType>performance</testType>
```

- Configure execution behavior attributes

  - logFilePath – set this to the location of the summary execution log file.

  - logDelimiter – set as PIPE. Delimiter for summary log file.

  - baseDir – provide the directory where result files for each query will be written.

  Example Regression Module XML (.xml) entry:

```
<logFilePath>C:/tmp/rel11/perftestrun.log</logFilePath>
<logDelimiter>PIPE</logDelimiter>
<logAppend>no</logAppend>
```

```
<baseDir>C:/tmp/rel11</baseDir>
<delimiter>PIPE</delimiter>
<printOutput>summary</printOutput>
```

- Configure performance test execution behavior attributes
    - perfTestThreads – The number of threads to create when doing performance testing.
    - perfTestDuration – The duration in seconds to execute the performance test for.
    - perfTestSleepPrint – The number of seconds to sleep in between printing stats when executing the performance test.
    - perfTestSleepExec – The number of seconds to sleep in between query executions when executing the performance test.

Example Regression Module XML (.xml) entry:

```
<perfTestThreads>10</perfTestThreads>
<perfTestDuration>30</perfTestDuration>
<perfTestSleepPrint>5</perfTestSleepPrint>
<perfTestSleepExec>0</perfTestSleepExec>
```

- Select the list of data sources that you want to execute tests for:
    - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when executing the input file. If set to “yes” then it will use all datasources it finds in the generated input file to execute queries for.
    - datasources – list contains all databases and web services to be tested. This list can be different than the list you used to generate the input file. This is the list of datasources found in the input file to execute tests for.

Example Regression Module XML (.xml) entry:

```
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
</datasources>
```

Step 3 – Create deployment plan for “Generating the input file”

- Edit the deploy plan (.dp) file used for creating the input file:

Example deployment plan file entry for RegressionPerformanceGen.dp:

```
PASS   TRUE   ExecuteAction      createRegressionInputFile   $SERVERID
       rel1.1 "$MODULE_HOME/RegressionPerformanceTest.xml"
              "$MODULE_HOME/servers.xml"
```

Step 4 – Generate the Performance Test input file.

- Generate the regression input file using the default queries and not the pre-defined regression queries. The following shows an example of how to execute a deployment plan that was previously created.

- Execution is performed from the <your-root-path>/PDTool/bin directory:

  Example Execute Line:

  ```
  Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionPerformanceGen.dp

  Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionPerformanceGen.dp
  ```

Step 5 – Create deployment plan for "Executing the regression performance test"

- Edit the deploy plan (.dp) file used for creating the input file:

- Example deployment plan file entry for RegressionPerformance.dp:

```
PASS   TRUE   ExecuteAction      executePerformanceTest       $SERVERID
       rel1.1 "$MODULE_HOME/RegressionPerformanceTest.xml"
                              "$MODULE_HOME/servers.xml"
```

Step 6 – Execute the Performance deployment plan.

- Executing the regression test is accomplished by running the ExecutePDTool.bat and pointing it to the deployment plan you have created.

- This regression test will use the queries generated in the regression input file and execute them against CIS. It will log the results into a summary execution file in the file system. It will use several filters that you previously set up in the regression XML file. For example, you can turn on and off execution of the three categories, queries, procedures and web services. You can also filter on which datasources you execute tests for. The input file may contain a broad list of datasources and queries but you don't have to run all the tests.

- Execution is performed from the <your-root-path>/PDTool/bin directory:

  Example Execute Line:

  ```
  Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionPerformance.dp

  Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionPerformance.dp
  ```

Step 7 – Review the results of the output file.

This execution is now considered a point in time execution. This is merely the first part of a performance test where a baseline of queries is established against the current system.

The results of the regression test are saved to a summary log file of your choosing. Review the section later in this document title "Query Execution Performance Log File" for specifics on file layout and example content.

Do not edit the individual result files as the checksum value may be affected if you inadvertently save the file. This will throw off the comparison.

**New Release System Performance Test**

Step 1 – Create Regression Module XML for "Executing the regression test"

- Use RegressionPerformanceTest.xml that you previously created.

  Copy the entire entry for `<id>re1.1</id>` an create a new entry `<id>rel1.2</id>`
- Modify the regression test identifier

  ```
  <id>rel1.2</id>
  ```
- Focus on the following section of the XML:

  ```
  <testRunParams>
        <testType>performance</testType>
  ```
- Configure execution behavior attributes

  - logFilePath – set this to the location of the summary execution log file. Notice how the path is pointing to a different directory than the "current test" was configured for.
  - logDelimiter – set as PIPE. Delimiter for summary log file.
  - baseDir – provide the directory where result files for each query will be written. Notice how the path is pointing to a different directory than the "current test" was configured for.

  Example Regression Module XML (.xml) entry:

  ```
  <logFilePath>C:/tmp/re12/perftestrun.log</logFilePath>
  <logDelimiter>PIPE</logDelimiter>
  <logAppend>no</logAppend>
  <baseDir>C:/tmp/rel12</baseDir>
  <delimiter>PIPE</delimiter>
  <printOutput>summary</printOutput>
  ```

- Configure performance test execution behavior attributes

  - perfTestThreads – The number of threads to create when doing performance testing.
  - perfTestDuration – The duration in seconds to execute the performance test for.
  - perfTestSleepPrint – The number of seconds to sleep in between printing stats when executing the performance test.
  - perfTestSleepExec – The number of seconds to sleep in between query executions when executing the performance test.

Example Regression Module XML (.xml) entry:

```
<perfTestThreads>10</perfTestThreads>
<perfTestDuration>30</perfTestDuration>
<perfTestSleepPrint>5</perfTestSleepPrint>
<perfTestSleepExec>0</perfTestSleepExec>
```

- Select the list of data sources that you want to execute tests for:
    - Use exactly the same configuration as you did for the “current test”. There must be an apples to apples comparison.
    - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when executing the input file. If set to “yes” then it will use all datasources it finds in the generated input file to execute queries for.
    - datasources – list contains all databases and web services to be tested. This list can be different than the list you used to generate the input file. This is the list of datasources found in the input file to execute tests for.

Example Regression Module XML (.xml) entry:

```
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
</datasources>
```

Step 2 – Create deployment plan for “Executing the regression test”

- Edit the deploy plan (.dp) file used for creating the input file:
- Example deployment plan file entry for RegressionPerformance.dp:

```
PASS   TRUE   ExecuteAction      executePerformanceTest        $SERVERID
      rel1.2 "$MODULE_HOME/RegressionPerformanceTest.xml"
                          "$MODULE_HOME/servers.xml"
```

Step 3 – Execute the Regression Performance plan.

- Executing the regression test is accomplished by running the ExecutePDTool.bat and pointing it to the deployment plan you have created.
- Execution is performed from the <your-root-path>/PDTool/bin directory:

Example Execute Line:

```
Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionPerformance.dp

Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionPerformance.dp
```

Step 4 – Review the results of the output file.

This execution is now considered a second point in time execution. This is merely the second part of a performance test where the queries are established against the new release of the system.

The results of the regression test are saved to a summary log file of your choosing. Review the section later in this document title "Query Execution Performance Log File" for specifics on file layout and example content.

Do not edit the individual result files as the checksum value may be affected if you inadvertently save the file. This will throw off the comparison.

**Compare Logs**

Note: In terms of execution, this major step may be combined with "New Release System Test" step when executing the plan file. It does not have to be a separate execution step as inferred by this documentation. The documentation only separates the step for the sake of clarity of documenting the process.

Step 1 – Create Regression Module XML for "Executing the compare logs"

- Use RegressionMigrationTest.xml that you previously created.
- Modify the regression test identifier

  ```
  <id>CIS61</id>
  ```
- Focus on the following section of the XML:

  ```
  <compareLogs>
  ```
- Configure execution behavior attributes
  - logFilePath – set this to the location of the summary execution log file. Notice how the path is pointing to a different directory than the "current test" was configured for.
  - logDelimiter – set as PIPE. Delimiter for summary log file.
  - logAppend– set to no so it overwrites the log.
  - logFilePath1 – provide the location of the "Current System Test" log file. This is the query execution log file.
  - logFilePathr2 – provide the location of the "New Release System Test" log file. This is the query execution log file..
  - logDelimiter1 – set as PIPE for log file 1.
  - logDelimiter2 – set as PIPE for log file 2.
  - durationDelta – The delta/change in time +- that is allowed when comparing duration2 (New Release Test) with duration1 (baseline

Current System Test). The test fails when duration2 > duration1+durationDelta otherwise it is successful.

Example Regression Module XML (.xml) entry:

```
<compareLogs>

      <logFilePath>C:\\tmp\\comparelogs.log</logFilePath>
      <logDelimiter>PIPE</logDelimiter>
      <logAppend>no</logAppend>

      <logFilePath1>C:/tmp/re11/perftestrun.log</logFilePath1>
      <logFilePath2>C:/tmp/re12/perftestrun.log</logFilePath2>

      <logDelimiter1>PIPE</logDelimiter1>
      <logDelimiter2>PIPE</logDelimiter2>

      <durationDelta>000 00:00:01.0000</durationDelta>

</compareLogs>
```

Step 2 – Create deployment plan for "Executing the compare logs".

- Edit the deploy plan (.dp) file used for creating the input file:
- Example deployment plan file entry for RegressionMigration.dp:

```
PASS   TRUE   ExecuteAction      compareRegressionLogs      $SERVERID
       CIS61  "$MODULE_HOME/RegressionMigrationTest.xml"
                          "$MODULE_HOME/servers.xml"
```

Step 3 – Execute the Regression Performance plan.

- Executing the regression test is accomplished by running the ExecutePDTool.bat and pointing it to the deployment plan you have created.
- Execution is performed from the <your-root-path>/PDTool/bin directory:

  Example Execute Line:

```
Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionPerformance.dp

Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionPerformance.dp
```

Step 4 – Review the results of the compare log files.

This is the final part of a performance test where the two query execution log files are compared by checking to see how duration2 compares with duration1. The failure case is when the duration2 > duration1+durationDelta. The durationDelta allows for a tolerance level of change within the queries from one release to the next.

The results of the log comparison are saved to a summary log file of your choosing. Review the section later in this document title "Compare Regression Logs" for specifics on file layout and example content.

## 4. *Security Test Process:*

Scenario: The development and/or QA team wants to validate that the privileges are set correctly on the various folder structures. When it comes to security, the following are guidelines or best practices:

- There are representative test resources in each of the folders to be tested.
- The development team has setup groups and a corresponding test user in each of the groups.
- Privileges are assigned to groups which are applied to folders recursively.

Here is a high-level summary of the steps for the "Security Test Process":

- Step 1 – Prepare the Regression Module XML for generating the regression security XML file.
- Step 2 – Create deployment plan for generating the regression security XML file.
- Step 3 – Execute the Regression Security Generation plan which will execute the method "generateRegressionSecurityXML" to generate users, queries and security plan tests
- Step 4 – Validate the expected outcome for each plan test to insure that it is the correct outcome [PASS or FAIL]. This is a manual step performed by the developer, tester or administrator.
- Step 5 – Create Regression Module XML for executing a Security test
- Step 6 – Create deployment plan for "Executing the security test"
- Step 7 – Execute the Security Test plan.
- Step 8 – Review the results of the output file.

Security Test Details

Step 1 – Prepare the Regression Module XML for generating the regression security XML file.

- Make a copy of RegressionModule.xml and call it something like "RegressionSecurityGen.xml"

```
<newFileParams>
```

- Create queries, procedures and web services in the target regression security XML:

```
<createQueries>yes</createQueries>
<createProcedures>yes</createProcedures>
<createWS>yes</createWS>
```

- The soap type specifies whether to generate web services using soap11 (default), soap12 or all. soap12 is only applicable for CIS 6.1 and higher and only if a CIS New Composite Web Service has been created.

```
<createSoapType>soap11</createSoapType>
```

- Generate the default query for Views, Procedures or Web Services with the following parameters:
    - useDefaultViewQuery is set to yes
    - useDefaultProcQuery is set to yes
    - useDefaultWSQuery is set to yes

Example Regression Module XML (.xml) entry:

```
<useDefaultViewQuery>yes</useDefaultViewQuery>
<useDefaultProcQuery>yes</useDefaultProcQuery>
<useDefaultWSQuery>yes</useDefaultWSQuery>

<!--  Default queries for Views and Procedures. -->
<publishedViewQry>SELECT count(1) cnt FROM</publishedViewQry>
<publishedProcQry>SELECT count(1) cnt FROM</publishedProcQry>
```

- Select the list of data sources that you want to generate tests for:
    - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when generating the input file. If set to “yes” then it will use all datasources it finds in CIS to generate the input file.
    - datasources – list contains all databases and web services to be generated.

Example Regression Module XML (.xml) entry:

```
<useAllDatasources>no</useAllDatasources>
<datasources>
      <dsName>TEST00</dsName>
      <dsName>CustomerWS</dsName>
</datasources>
```

- Select the optional list of resources that you want to generate tests for. These can be tables or web services and they can use wildcards.
    - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when generating the input file. If set to “yes” then it will use all datasources it finds in CIS to generate the input file.

- datasources – list contains all databases and web services to be generated.

Example Regression Module XML (.xml) entry:

```
<resources>
      <resource>*.customers</resource>
      <resource>*.getCustomerById</resource>
      <resource>soap11.TEST00.*</resource>
</resources>
```

Resource filter: This is a list of resources for which to generate the input file.

A wild card ".*" may be used at the catalog or schema level to denote that all resources under that level should be compared.

A fully qualified resource may be identified as well. This is a filter that can be fine-tuned for one or more resources.

If left empty then all resources are generated. Overlapping resource wild cards will use the highest level specified in this list.

Examples of filters for web services:

- this is a legacy web service

  <resource>services.webservices.testWebService00</resource>

- filter all new composite soap 11 web services

  <resource>soap11.*</resource>

- filter all new composite soap 12 web services

  <resource>soap12.*</resource>

- new composite soap 11 web service with specific path

  <resource>soap11.testWebService00_NoParams_bare</resource>

- new composite soap 11 web service located in the folder1 web service folder

  <resource>soap11.folder1.*</resource> -

- new composite soap 11 web service located in folder2 and matching a wild card starting with testWebService00

  <resource>soap11.folder2.testWebService00*</resource>

- Configure the security generation options.

  Options for generating the Regression Security XML section.

```
<securityGenerationOptions>
```

```
<pathToRegressionXML>c:\\tmp\\reg.xml</pathToRegressionXML>
```

Path to Target Regression Module XML - a required path to the target configuration file for the regression module. Provides a way of writing to a different file than the source or original RegressionModule.xml.

```
<encryptedPassword>Encrypted:B0873483C56F7498</encryptedPassword>
```

[optional] A security user default encrypted password. [node="regressionSecurityUser"] It will be encrypted when the ExecutePDTool.bat -encrypt ..\resources\modules\RegressionModule.xml is executed.

```
<userFilter>user1,user2</userFilter>
```

[optional] Determines what CIS users to generate. Wildcards (*) may be used. [node="regressionSecurityUser"]

```
<domainFilter>composite</domainFilter>
```

[optional] Provides a way of specifying what domain the userFilter should be applied to. [node="regressionSecurityUser"]

```
<userMode>OVERWRITE</userMode>
```

[NOEXEC|OVERWRITE|APPEND] - NOEXEC (default)=do nothing, don't execute. OVERWRITE=overwrite existing security user XML, APPEND=add to existing security user XML if the user does not exist. [node="regressionSecurityUser"]

```
<queryMode>OVERWRITE</queryMode>
```

[NOEXEC|OVERWRITE|APPEND] - NOEXEC (default)=do nothing, don't execute. OVERWRITE=overwrite existing security query XML, APPEND=add to existing security query XML if the query does not exist. [node="regressionSecurityQuery"

```
<planMode>OVERWRITE</planMode>
```

[NOEXEC|OVERWRITE|APPEND] - NOEXEC (default)=do nothing, don't execute. OVERWRITE=overwrite existing security plan XML, APPEND=add to existing security plan XML if the plan does not exist. [node="regressionSecurityPlanTest"]

```
<planModeType>MULTIPLAN</planModeType>
```

[SINGLEPLAN|MULTIPLAN] - SINGLEPLAN=Generate the Cartesian plan as a single plan. MULTIPLAN=Generate the Cartesian plan as multiple plans for each user who has the same set of queries. [node="regressionSecurityPlanTest"]

- e.g. when planMode=OVERWRITE and planModeType=MULTIPLAN - will produce a new list where each user is a security plan with the full set of queries.
- e.g. when planMode=APPEND and planModeType=SINGLEPLAN - will produce a new plan appended to the existing set of plans where this plan will contain a Cartesian product of users and queries.

```
<planIdPrefix>sp</planIdPrefix>
```

The plan id prefix provides a way of overriding the default [sp]. For example a plan id=sp1,sp2, etc. This gives the user the ability to identify certain plans with different prefixes. [node="regressionSecurityPlanTest"]

```
<planGenerateExpectedOutcome>true</planGenerateExpectedOutcome>
```

The plan generate expected outcome determines whether to (true) generate the outcome based on retrieving privileges for a given query resource and cross-referencing with the user and their groups to determine what the expected outcome should be upon execution.

Upside: it provides a ready-made plan with expected outcome. [CAUTION: The developer needs to double-check that the expected outcome is what they want.

The generation is based on how the privileges are actually set in the environment that PDTool connects to. It may be the same environment that you are intending to execute a test. Generating a test and executing the test defeats the purpose of testing. A validation of expected outcome is a required step between generation of the test plan and executing the test plan. Downside: it is a very slow process as it requires retrieving groups for each user and privileges for each resource query from CIS and then calculating the expected outcome. (false) do not generate the outcome and leave blank.

Flatten the security XML sections into a table structure for easier viewing.

```
<flattenSecurityUsersXML>true</flattenSecurityUsersXML>
```

The flatten security users XML [node="regressionSecurityUser"] output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.

```
<flattenSecurityQueryQueriesXML>true</flattenSecurityQueryQueriesXML>
```

The flatten security query (SQL Queries) [node="regressionSecurityQuery" and "queryType"=QUERY] XML output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.

```
<flattenSecurityQueryProceduresXML>true</flattenSecurityQueryProceduresXML>
```

The flatten security query (SQL Procedures) [node="regressionSecurityQuery" and "queryType"=PROCEDURE] XML output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.

```
<flattenSecurityQueryWebServicesXML>true</flattenSecurityQueryWebServicesXML>
```

The flatten security query (Web Services) [node="regressionSecurityQuery" and "queryType"=WEB_SERVICES] XML output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.

```
<flattenSecurityPlansXML>true</flattenSecurityPlansXML>
```

The flatten security plans [node="regressionSecurityPlanTest"] XML output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.

```
</securityGenerationOptions>
```

Step 2 – Create deployment plan for “Generating the security XML”

- Edit the deploy plan (.dp) file used for creating the input file:

Example deployment plan file entry for RegressionSecurityGen.dp:

```
PASS   TRUE   ExecuteAction  generateRegressionSecurityXML $SERVERID
       Test4.1 "$MODULE_HOME/RegressionSecurityGen.xml"
              "$MODULE_HOME/servers.xml"
```

Step 3 – Generate the Regression Security XML.

- Execute the Regression Security Generation plan which will execute the method “generateRegressionSecurityXML” to generate users, queries and security plan tests.

- Execution is performed from the <your-root-path>/PDTool/bin directory:

Example Execute Line:

```
Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionSecurityGen.dp
```

Step 4 – Validate Regression Security Plan tests.

- Validate the expected outcome for each plan test to insure that it is the correct outcome [PASS or FAIL]. This is a manual step performed by the developer, tester or administrator.

Step 5 – Create Regression Module XML for “Executing the security test”

- Use RegressionSecurityTest.xml

- Focus on the following section of the XML:

```
<testRunParams>
      <testType>security</testType>
```

- Configure execution behavior attributes
    - logFilePath – set this to the location of the summary execution log file.
    - logDelimiter – set as PIPE. Delimiter for summary log file.
    - baseDir – leave this blank for functional tests. There is no need to write out results for each query to a file since the only purpose of this test is to do a “select count” and test whether the resource is functional or not.

Example Regression Module XML (.xml) entry:

```
<logFilePath>C:/tmp/cis51/functestrun.log</logFilePath>
<logDelimiter>PIPE</logDelimiter>
<logAppend>no</logAppend>
```

```
<baseDir></baseDir>
<delimiter>PIPE</delimiter>
<printOutput>summary</printOutput>
```

- Set the Security Execution options

```
Optional regression security testing section

<securityExecution>

      Identifies one or more security plan ids to execute security
      tests with.

      <securityPlanIds>sp1,sp2</securityPlanIds>

      true=throw an exception when the overall security rating=FAIL,
      false=don't throw an exception when the overall security
      rating=FAIL

      <securityOverallRatingException>false</securityOverallRatingExce
      ption>

      true=throw an exception when there were errors, result=ERROR,
      false=don't throw an exception when there were errors,
      result=ERROR

      <securityExecutionErrorException>true</securityExecutionErrorExc
      eption>

</securityExecution>
```

- Determine what category filters to use:

  Category filter:  Turn of/off the execution of an entire category.

```
<runQueries>yes</runQueries>

<runProcedures>yes</runProcedures>

<runWS>yes</runWS>
```

- Select the list of data sources that you want to execute tests for:
  - useAllDatasources – set this to “no” when you want to explicitly define your own list of datasources when executing the input file.  If set to “yes” then it will use all datasources it finds in the generated input file to execute queries for.
  - datasources – list contains all databases and web services to be tested. This list can be different than the list you used to generate the input file. This is the list of datasources found in the input file to execute tests for.

  Example Regression Module XML (.xml) entry:

If “yes”, run tests for all data sources from the input file, datasources below are ignored.

```
<useAllDatasources>no</useAllDatasources>
<datasources>
      <dsName>TEST00</dsName>
      <dsName>testWebService00</dsName>
</datasources>
```

Step 6 – Create deployment plan for “Executing the regression security test”

- Edit the deploy plan (.dp) file used for executing the security test:
- Example deployment plan file entry for RegressionSecurityTest.dp:

```
PASS   TRUE   ExecuteAction      executeSecurityTest       $SERVERID
      Test4.1 "$MODULE_HOME/RegressionSecurityTest.xml"
                          "$MODULE_HOME/servers.xml"
```

Step 7 – Execute the Regression Security Test plan.

- Executing the regression test is accomplished by running the ExecutePDTool.bat and pointing it to the deployment plan you have created.
- This regression security test will use the queries generated in the regression security XML and execute them against CIS. It will log the results into a summary execution file in the file system. It will use several filters that you previously set up in the regression XML file. For example, you can turn on and off execution of the three categories, queries, procedures and web services. You can also filter on which datasources you execute tests for. The input file may contain a broad list of datasources and queries but you don’t have to run all the tests.
- Execution is performed from the <your-root-path>/PDTool/bin directory:

  Example Execute Line:

```
Windows: ExecutePDTool.bat -exec ../resources/plans/RegressionSecurityTest.dp

Unix: ./ExecutePDTool.sh -exec ../resources/plans/RegressionSecurityTest.dp
```

Step 8 – Review the results of the output file.

The results of the security test are saved to a summary log file of your choosing. Review the section later in this document title “Security Test Execution Log File” for specifics on file layout and example content.

## REGRESSION TEST MODULE DEFINITION

### *Method Definitions and Signatures*

1. **createRegressionInputFile**

   Generates input file for regression tests for one published datasource on a given CIS server for a given user. The server connection information comes from servers.xml by serverId parameter, the other parameter specifies the name of the published data source on that server.

```
@param serverId - server Id from servers.xml
@param regressionIds - comma-separated list of the published regression identifiers to run test against
@param pathToRegressionXML - path to the config file of this module
@param pathToServersXML - path to servers.xml
@return void
@throws CompositeException


public void createRegressionInputFile(String serverId, String regressionIds, String pathToRegressionXML, String pathToServersXML) throws CompositeException;
```

2. **executeRegressionTest**

   Executes all queries, procedures and web services in the input file but only for published datasources (virtual databases). Uses regressionConfig to control different aspects of execution, for example can skip some items from the input file if they are not from a predefined list of datasources. In that, this method is different from the original Pubtest utility which executes everything in the input file without restriction.

```
@param serverId - server Id from servers.xml
@param regressionIds - comma-separated list of the published regression identifiers to run test against
@param pathToRegressionXML - path to the config file of this module
@param pathToServersXML - path to servers.xml
@return void
@throws CompositeException


public void executeRegressionTest(String serverId, String regressionIds, String pathToRegressionXML, String pathToServersXML) throws CompositeException;
```

3. **compareRegressionFiles**

Compares the contents of two files for a given CIS server. The objective of the comparison is to compare the before and after results. This method should be invoked in the context of a higher-level process where PD Tool is invoked to executeRegressionTest() and generate a file. PD Tool is invoked again at a different point in time or against a different CIS server and produces another file. Then PD Tool is executed to compare the results of the two files and determine if they are a match or not. The results of this are output to a result file. The result of each regression comparison is either SUCCESS when the files match or FAILURE when the files do not match.

```
@param serverId - server Id from servers.xml

@param regressionIds - comma-separated list of the published regression
identifiers to execute the file comparison

@param pathToRegressionXML - path to the config file of this module

@param pathToServersXML - path to servers.xml

@return void

@throws CompositeException


public void compareRegressionFiles(String serverId, String regressionIds,
String pathToRegressionXML, String pathToServersXML) throws
CompositeException;
```

4. **compareRegressionLogs**

Compare the Query Execution log files for two separate execution runs. Determine if queries executed in for two similar but separate tests are within the acceptable delta level. Compare each similar result duration and apply a +- delta level to see if it falls within the acceptable range. Write out a log file with the results of the comparison.

```
Compare the Query Execution log files for two separate execution runs.
Determine if queries executed in for two similar but separate tests are
within the acceptable delta level.

Compare each similar result duration and apply a +- delta level to see if
it falls within the acceptable range.

@param serverId - server Id from servers.xml

@param regressionIds - comma-separated list of the regression identifiers
to execute the log comparison

@param pathToRegressionXML - path to the config file of this module

@param pathToServersXML - path to servers.xml

@throws CompositeException


public void compareRegressionLogs(String serverId, String regressionIds,
String pathToRegressionXML, String pathToServersXML) throws
CompositeException;
```

5. **executePerformanceTest**

Execute a performance regression test. Execute the same query, procedure or web service for a specified duration of time and with a specified number of threads. Print out the results after a specified sleep interval.

```
@param serverId - server Id from servers.xml

@param regressionIds - comma-separated list of the published regression identifiers to run test against

@param pathToRegressionXML - path to the config file of this module

@param pathToServersXML - path to servers.xml

@return void

@throws CompositeException


public void executePerformanceTest(String serverId, String regressionIds, String pathToRegressionXML, String pathToServersXML) throws CompositeException;
```

6. **executeSecurityTest**

Executes a security test based on queries found in the RegressionModule.xml. The security test correlates users and queries with an expected outcome. The regression security XML contains a list of users, a list of published queries that include (SQL views, SQL procedures and Web Services) and a list of security plan tests that tie a user to a query. The objective of security testing is to test for holes in the privilege scheme as well as access issues that should be granted but are not properly granted. Each plan test contains an expected outcome of either PASS or FAIL. This leads to the concepts of positive tests and negative tests. A positive test, tests for the proper access to be granted and is expecting a “PASS” status. If the test passes, the user has the proper privileges. If the test fails wit “insufficient privileges”, then the admin needs to address the privileges. A negative test, tests for holes in the privilege scheme and is expecting a “FAIL” with insufficient privileges. If the test fails as it should, then the overall status is a PASS which is good. If the test is successful (the user is able to execute the query), then the overall status is a fail and the test has uncovered a security hole. The administrator needs to address the privileges.

For example, user1 is allowed to execute CAT1.SCH1.VIEW1 but is not allowed to execute CAT2.SCH2.VIEW2. Therefore the expected outcome of the query for VIEW1 is a PASS and the expected outcome for VIEW2 is FAIL. When user1 connects to CIS and executes VIEW1 and the query is successful, the overall status is a PASS. However, if user1 receives “insufficient privileges” then the overall status is a FAIL and the security test fails. Likewise if user2 executes VIEW1 and it passes, the overall status is a FAIL and also points to a security hole in the privilege setup. Therefore the security test fails and the administrator who runs the test may be notified with an exception being thrown by PDTool Regression Module.

For output summary information the tester/administrator should review the "Security Test Execution Log File" section.

*Debugging* – For different levels of debug output the tester/administrator can set the following parameter in the RegressionModule.xml:

<!-- Print output to command line: [verbose,summary,silent] verbose=print summary and results, summary=print query context, silent=nothing is printed to the command line. -->

<printOutput>summary</printOutput>

```
@param serverId - server Id from servers.xml
@param regressionIds - comma-separated list of the regression identifiers to run test against.
@param pathToRegressionXML - path to the config file of this module
@param pathToServersXML - path to servers.xml
@return void
@throws CompositeException


public void executeSecurityTest(String serverId, String regressionIds, String pathToRegressionXML, String pathToServersXML) throws CompositeException;
```

7. **generateRegressionSecurityXML**

This procedure generates the Regression Security XML section of the RegressionModule.xml. The filters for this are found in the "/newFileParams" section of the Regression Module XML. This section defines what virtual databases "/newFileParams/datasources" should be accessed and what resource filters "/newFileParams/resources" should be applied. Additionally, there are a number of other filters in "/newFileParams/securityGenerationOptions" section of the XML that provide further filters and default settings for the regression security XML generation.

The best practice is to have a consistent "TEST" resource in all folders so that when the queries are generated, you have a representative view, procedure and web service in all folders. Only a single functional test is required to test security. It is not necessary to test all queries as much as it is to test the "groups" and folder assignments.

This method generates to a different RegressionModule.xml file than the source file so that formatting can be maintained in the XML in the source. This is based on the xml schema "securityGenerationOptions/pathToTargetRegressionXML". The best practice is to not overwrite the original Regression Module XML.

This method generates Regression Security Users from the given filter applying the xml schema userMode=[NOEXEC|OVERWRITE|APPEND]. The "userFilter" and "domainFilter" are used as filters when generating users. The "encrpytedPassword"

element provides a default password for the users. PDTool is not allowed to access the passwords from CIS so the user may provide a default encrypted password or modify the user passwords after generation.

- userMode=NOEXEC – don't generate any users.
- userMode=OVERWRITE – overwrite the existing list of users.
- userMode=APPEND – Given the existing list of users…append any new users found in CIS. Note…this will only work with "composite" domain users as LDAP users do not physically exist in CIS.

This method generates Regression Security Queries from the given filter applying the xml schema queryMode=[NOEXEC|OVERWRITE|APPEND]. Queries consist of any published SQL view or procedures or published web services. These are the virtual data services in Composite. The "datasources" and "resources" elements are used as filters to find the set of resources.

- queryMode=NOEXEC – don't generate any queries.
- queryMode=OVERWRITE – overwrite the existing list of queries.
- queryMode=APPEND – Given the existing list of queries…append any new queries found in CIS.

This method generates a Cartesian product for the Regression Security Plans applying the xml schema planMode=[NOEXEC|OVERWRITE|APPEND] and planModeType=[SINGLEPLAN|MULTIPLAN].

- planMode=NOEXEC – don't generate any plans.
- planMode=OVERWRITE – overwrite the existing list of plans.
    - SINGLEPLAN – overwrite all plans found in the list.
    - MULTIPLAN – overwrite all plans in the list.
- planMode=APPEND – Given the existing list of users…append any new users found in CIS. Note…this will only work with "composite" domain users as LDAP users do not physically exist in CIS.
    - SINGLEPLAN – Append the last plan found in the list.
    - MULTIPLAN – Append each user section in the list as it correlates to the existing set of users. If a plan exists for a given user but that user no longer exists in the plan, that plan is left-alone however, as a cautionary note, this will produce execution errors. If a plan is not found for a given user in the list a new plan is created. If a multiple plans exist for the same user (not recommended), then it will only update the first plan it finds in the list for that user and skips the rest.
- planModeType=SINGLEPLAN – single plan means that there is a single plan that contains the entire list of all users and all queries.

- planModeType=MULTIPLAN – a multi-plan means that there is a plan for each user where the plan contain an iteration of all the queries for that user. The next plan contains the next user and an iteration of all the queries and so on and so forth until there is a plan for each user in the user list.

It is recommended that the users copy the results out of the target, generated file and paste into the source file as needed. A Cartesian product is where each user contains an execution for all of the queries. A security plan is as follows:

- A security plan consists of executing all queries for a single user.
- A Cartesian product involves creating a plan for each user with all queries. It is important for each user to execute all queries so that you can achieve both "positive" and "negative" tests. This was discussed in detail in the "executeSecurityTest" method above. The point of a security test is to verify access and expose any holes in the privilege scheme.

*Debugging* - The tester/administrator can glean more information with respect to the generation by turning on the debug parameters in the deploy.properties file. The "debug1" flag will print out high-level information. The "debug2" flag will print out intermediate level information and the "debug3" flag will track each user, query and plan that is being generated. For an even deeper level of debug, use the log4j.properties setting "log4j.logger.com.cisco.dvbu.ps.deploytool=DEBUG".

```
@param serverId - server Id from servers.xml

@param regressionIds - comma-separated list of the published regression
identifiers to run test against

@param pathToRegressionXML - path to the config file of this module

@param pathToServersXML - path to servers.xml

@return void

@throws CompositeException


public void generateRegressionSecurityXML(String serverId, String
regressionIds, String pathToSourceRegressionXML, String pathToServersXML)
throws CompositeException;
```

**General Notes:**

Currently execution of views and procedures should be performed in a separate test (i.e. with a different test ID) from execution of Web Services. This is a temporary limitation that should be removed soon.

The arguments pathToRegressionXML and pathToServersXML will be located in PDTool/resources/modules. The value passed into the methods will be the fully qualified path. The paths get resolved when executing the property file and evaluating the $MODULE_HOME variable

## REGRESSION TEST XML CONFIGURATION

A full description of the PDToolModule XML Schema can be found by reviewing /docs/PDToolModules.xsd.html.

### *Description of the Regression Module XML*

The Regression Test Module XML (RegressionModule.xml) provides a structure "regressionTest" identified by an **ID** to create a test input file and/or run a test. The Regression XML is broken into the parts shown below. Each section also contains a more detailed explanation provided in the "Attributes of Interest" section.

Here is an example of XML file broken down into the following areas:

1. Use of Variables – provides a description of how to use variables in the XML
2. Main section – general input file attributes
3. Create Regression Input File – attributes used for creating a new published test input file
4. Execute a Test Run – attributes used for performing a "regression", "migration" or "performance" test
5. Compare two regression test runs – attributes for the result files from a "regression" or "migration" test
6. Compare two query execution log files – attributes for comparing the log files from a "regression", "migration" or "performance" test
7. Regression Queries used to create the input file – attributes used for setting up custom queries provided by the user other than the ones based on the default queries.

### *Use of Variables:*

The use of variables can provide a flexible way of parameterizing a deployment for the Regression Module. Variables can be used in all of the XML nodes except <perfTestThreads>, <perfTestDuration>, <perfTestSleepPrint>, and <perfTestSleepExec> because they are defined as xs:integer.

A variable is defined with any $ (e.g. $VAR). Even though the other modules in PDTool allow you to define a variable with % (e.g. %VAR), the regression module restricts this because the <query> node allows the use of % for wildcards. Variables can be defined in the operating system, in a batch file that invokes ExecutePDTool.sh|.bat, using a java environment "-DVAR=value" command in the ExecutePDtool batch file or in the PDTool configuration property file (e.g. deploy.properties). The easiest and most common approach is the latter.

Escaping values with $$:

There are times when it is necessary to escape a value in the RegressionModule.xml when the value itself is not a variable but contains a $. For example, what if a SQL query contained a field with the name “$status”? PDTool would interpret that as a variable. However, by using two $$ to escape the value (e.g. “$$status”), PDTool will interpret this as a single $ and not translate it as a variable. Here is an example SQL:

```
SELECT "$$Status" FROM ViewSales2 WHERE "$$Status" = '$VIEWSALES_STATUS'
```

Finally, in the above example, the variable VIEWSALES_STATUS=open is set in the configuration property file. The above query is interpreted by PDTool and executed against the Composite JDBC driver as the following:

```
SELECT "$Status" FROM ViewSales2 WHERE "$Status" = 'open'
```

A platform independence use case:

In this use case there are environment/system considerations to take into account such as where the files will be created and accessed. If the deployment will take place across both Windows and UNIX, the path information will be different. However, by using a variable to represent the base path information, it is possible to achieve platform independence within a single RegressionModule.xml file. In this example we specify a variable in the configuration property file such as REGRESSION_HOME=D:/dev/regression if windows or REGRESSION_HOME=/opt/regression if UNIX. To reference the variable, the RegressionModule.xml would have an entry as shown below that identifies where the input file will be created or accessed:

```
<inputFilePath>$REGRESSION_HOME/regression1.inp</inputFilePath>
```

PDTool will interpret <inputFilePath> in windows as “D:/dev/regression/regression1.inp” and in UNIX as “/opt/regression/regression1.inp”.

A SQL wildcard use case:

As stated above, the % is a SQL wildcard. A user may simply specify this wildcard in the query node such as ‘Mega%’. It is not necessary to escape the %.

```
<query>SELECT * FROM CAT1.SCH2.ViewSales WHERE ProductName like 'Mega%'</query>
```

It is also possible to specify a wild card as content in a variable albeit in this case escaping of the wildcard will be necessary. In the query below the variable $PRODUCT_WILDCARD is being referenced.

```
<query>SELECT * FROM ViewSales2 WHERE ProductName like
'$PROD_WILDCARD'</query>
```

The variable is defined in the configuration property file as “PROD_WILDCARD=Widget%%”. Note the use of %% which is required to escape a

% sign when it is used as a value and not a variable. Remember outside of the RegressionModule.xml file, all other modules and property files treat $ and % as variable designators. The above query is interpreted by PDTool and executed against the Composite JDBC driver as the following:

```
SELECT * FROM ViewSales2 WHERE ProductName like 'Widget%'
```

In summary, the use of variables in the Regression Module can be very powerful when creating a platform independent module and providing a flexible solution for query parameters.

### *Main section XML:*

```
<?xml version="1.0"?>
<p1:RegressionModule xmlns:p1="http://www.dvbu.cisco.com/ps/deploytool/modules">

<regressionTest>

    <id>Test1</id>
    <inputFilePath>C:\\temp\\pdtool\\pubtestWS.inp</inputFilePath>
    <tempDirPath>C:\\temp\\pdtool</tempDirPath>
    <createNewFile>yes</createNewFile>
```

### Attributes of Interest:

***id*** – identifies one test execution, unique within the file.

***inputFilePath*** – full path to the input file on the file system. For windows environments, use double "\\" for paths such as "C:\\tmp\\regression\\file.inp" or use this style "C:/tmp/regression/file.inp". For UNIX it would be "/tmp/regression/file.inp".

***tempDirPath*** – directory location for temporary files to be created when generating the input file or generating the regression security XML.

***createNewFile*** – yes/no parameter that defines whether a new input file should be created.

### *Create regression input file XML or Generate regression security XML:*

```
    <newFileParams>
        <createQueries>yes</createQueries>
        <createProcedures>yes</createProcedures>
        <createWS>yes</createWS>
        <createSoapType>soap11</createSoapType>

        <useDefaultViewQuery>no</useDefaultViewQuery>
        <useDefaultProcQuery>no</useDefaultProcQuery>
        <useDefaultWSQuery>no</useDefaultWSQuery>

        <publishedViewQry>SELECT count(1) cnt FROM</publishedViewQry>
        <publishedProcQry>SELECT count(*) cnt FROM</publishedProcQry>

        <useAllDatasources>no</useAllDatasources>
```

```
        <datasources>
            <dsName>EXAMPLES</dsName>
            <dsName>ProductWebService</dsName>
        </datasources>

        <resources>
              <resource>TABLE1</resource>
              <resource>services.ProductWebService.method1</resource>
              <resource>soap11.ProductWebService.method2</resource>
              <resource>soap11.ProductWebService.somemethod*</resource>
        </resources>

         <!-- For generating regression security XML only -->
        <securityGenerationOptions>
              <pathToTargetRegressionXML>C:\\temp\\RegGen.xml</pathToTargetRegressionXML>
              <encryptedPassword>password</encryptedPassword>
              <userFilter>user_</userFilter>
              <domainFilter>composite</domainFilter>
              <userMode>OVERWRITE</userMode>
              <queryMode>OVERWRITE</queryMode>
              <planMode>OVERWRITE</planMode>
              <planModeType>MULTIPLAN</planModeType>
              <planIdPrefix>sp</planIdPrefix>
              <planGenerateExpectedOutcome>false</planGenerateExpectedOutcome>
              <flattenSecurityUsersXML>true</flattenSecurityUsersXML>
              <flattenSecurityQueryQueriesXML>true</flattenSecurityQueryQueriesXML>
              <flattenSecurityQueryProceduresXML>true</flattenSecurityQueryProceduresXML>
              <flattenSecurityQueryWebServicesXML>false</flattenSecurityQueryWebServicesXML>
              <flattenSecurityPlansXML>true</flattenSecurityPlansXML>
       </securityGenerationOptions>

        <defaultProcParamValues>
              <bit>1</bit>
              <varchar>'A'</varchar>
              <char>'a'</char>
              <clob>A</clob>
              <integer>1</integer>
              <int>1</int>
              <bigint>1</bigint>
              <smallint>1</smallint>
              <tinyint>1</tinyint>
              <decimal>1.0</decimal>
              <numeric>1.0</numeric>
              <real>1.0</real>
              <float>1.0</float>
              <double>1.0</double>
              <date>'2011-01-01'</date>
              <time>'00:00:00'</time>
              <timestamp>'2011-01-01 00:00:00'</timestamp>
              <binary>''</binary>
              <varbinary>''</varbinary>
              <blob>''</blob>
              <xml>''</xml>
        </defaultProcParamValues>

    </newFileParams>
```

## Attributes of Interest:

***newFileParams*** – parameters for the input file creation part of the module. Used by the method **createRegressionInputFile().**

***createQueries*** – yes/no parameter that defines whether query statement should be created in the published test input file for views of the configured Composite data source.

***createProcedures*** – yes/no parameter that defines whether procedure invocations should be created in the published test input file for procedures of the configured Composite data source.

***createWS*** – yes/no parameter that defines whether web service invocation should be created in the published test input file for web services of the configured Composite data source.

***createSoapType*** – The soap type specifies whether to generate web services using soap11 (default), soap12 or all. soap12 is only applicable for CIS 6.1 and higher and only if a CIS New Composite Web Service has been created.

***useDefaultViewQuery*** – yes/no parameter that defines when to use the default query when generating input file instead of using the list of pre-defined regression queries.

***useDefaultProcQuery*** – yes/no parameter that defines when to use the default procedure when generating input file instead of using the list of pre-defined regression queries.

***useDefaultWSQuery*** – yes/no parameter that defines when to use the default web service when generating input file instead of using the list of pre-defined regression queries.

***publishedViewQry*** – a template query that should be used when generating the queries for the published test input file. A standard template query for views is: "SELECT count(1) cnt FROM".

***publishedProcQry*** – a template query that should be used when generating the queries for the published test input file. A standard template query for procedures is: "SELECT count(*) cnt FROM".

***useAllDatasources*** – yes/no parameter that defines whether to use all the Composite data sources published by Composite or use the list provided in the input file. If yes, generate entries for all data sources that are published. If no, only generate entries for data sources listed in the <datasources> section for the <newFileParams>.

***datasources*** – a list of data sources to interrogate when generating the input file

***dsName*** – the name of the Composite published data source.

- For Databases they will be found at the path: “Composite Data Services/Databases/<MYDB>”. Simply list the data service name. The Regression Module will introspect all catalogs and schemas to find views and procedures when generating the input file.
- For Web Services they will be found at the path: “Composite Data Services/Web Services/<MYDWS>”. Simply list the web service container name and not the port or operations. The Regression Module will introspect and generate for each operation found.

***resources/resource –*** The list of relational structures or web services to include when generating the regression input file. All references use periods as separators and not forward slashes. A wild card before "*.resource" or after "resource.*" may be used at the catalog or schema level to denote that all resources under that level should be compared. A wild card in the middle is ignored "resource.*.resource". A fully qualified resource may be identified as well. This is a filter that can be fine-tuned for one or more resources. If left empty then all resources are compared. Overlapping resource wild cards will use the highest level specified in this list.

Relational Syntax:

- [*]CATALOG.SCHEMA.TABLE[*]

Legacy Web Service Syntax:

- [*]services.webservices.WEBSERVICE.METHOD[*]

Composite Web Service Syntax:

- [*]soap11.FOLDER(s).WEBSERVICE.METHOD[*]
- [*]soap12.FOLDER(s).WEBSERVICE.METHOD[*]

Examples of filters for web services:

- this is a legacy web service

  <resource>services.webservices.testWebService00</resource>
- filter all new composite soap 11 web services

  <resource>soap11.*</resource>
- filter all new composite soap 12 web services

  <resource>soap12.*</resource>
- new composite soap 11 web service with specific path

<resource>soap11.testWebService00_NoParams_bare</resource>

- new composite soap 11 web service located in the folder1 web service folder

  <resource>soap11.folder1.*</resource> -

- new composite soap 11 web service located in folder2 and matching a wild card starting with testWebService00

  <resource>soap11.folder2.testWebService00*</resource>

- Find all resources in all paths that end in .customers

  <resource>*.customers</resource>

- Find all resources in all paths that end in .getCustomerById. This may include published procedures or web service operations

  <resource>*.getCustomerById</resource>

***securityGenerationOptions*** – The security generation options is an optional structure that is only used with the method “generateRegressionSecurityXML”.

***pathToTargetRegressionXML*** – a required path to the target configuration file for the regression module. This provides a way of writing to a different file than the source or original RegressionModule.xml. When an XML file is written out, the original XML comments are not retained. It is a best practices to generate a separate file containing the regression security settings.

***encryptedPassword*** – this is a default password that is used when generating the regression security user XML. Because PDTool is not allowed to get the password from CIS for a user, the only recourse is to provide a default. The tester/administrator may need to edit the XML after generation to provide the actual passwords for a user. The PDTool tester/administrator may encrypt clear text passwords using the command ExecutePDTool.bat -encrypt <path-to-regression.xml>

***userFilter*** – This optional filter is a comma separated list of users or wild card users that can be used to filter the generation of the Regression Security User list.

***domainFilter*** – This optional filter provides a way of specifying what domain the userFilter should be applied to.

***userMode*** – [NOEXEC|OVERWRITE|APPEND] - NOEXEC (default)=do nothing, don't execute. OVERWRITE=overwrite existing security user XML, APPEND=add to existing security user XML if the user does not exist.

***queryMode*** – [NOEXEC|OVERWRITE|APPEND] - NOEXEC (default)=do nothing, don't execute. OVERWRITE=overwrite existing security query XML, APPEND=add to existing security query XML if the query does not exist.

***planMode*** – [NOEXEC|OVERWRITE|APPEND] - NOEXEC (default)=do nothing, don't execute. OVERWRITE=overwrite existing security plan XML, APPEND=add to existing security plan XML and create plans that do not exist.

***planModeType*** – [SINGLEPLAN|MULTIPLAN] - SINGLEPLAN=Generate the Cartesian plan as a single plan. MULTIPLAN=Generate the Cartesian plan as multiple plans for each user who has the same set of queries.

- e.g. when planMode=OVERWRITE and planModeType=MULTIPLAN - will produce a new list where each user is a security plan with the full set of queries.
- e.g. when planMode=APPEND and planModeType=SINGLEPLAN - will produce a plans appended to the existing plan where this plan will contain a Cartesian product of users and queries.

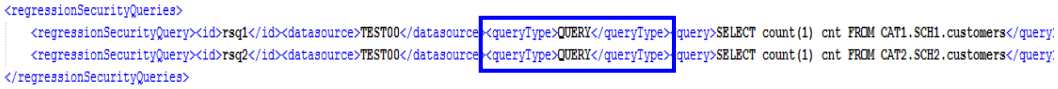
***planIdPrefix*** – The plan id prefix provides a way of overriding the default [sp]. For example a plan id=sp1,sp2, etc. This gives the user the ability to identify certain plans with different prefixes.

***planGenerateExpectedOutcome*** – The plan generate expected outcome determines whether to (true) generate the outcome based on retrieving privileges for a given query resource and cross-referencing with the user and their groups to determine what the expected outcome should be upon execution (false) do not generate the outcome and leave blank.

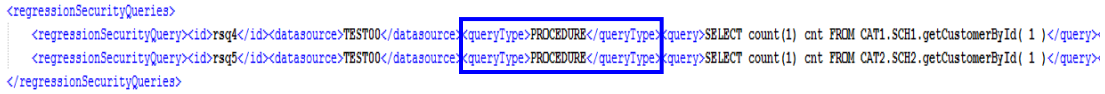
***flattenSecurityUsersXML*** – The flatten security users XML [node="regressionSecurityUser"] output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.

```
<regressionSecurityUsers>
    <regressionSecurityUser><id>rsu1</id><userName>user1</userName><encryptedPassword>Encrypted:B0873483C56F7498</encryptedPassword><domain>composite</domain></regressionSecurityUser>
    <regressionSecurityUser><id>rsu2</id><userName>user2</userName><encryptedPassword>Encrypted:B0873483C56F7498</encryptedPassword><domain>composite</domain></regressionSecurityUser>
</regressionSecurityUsers>
```

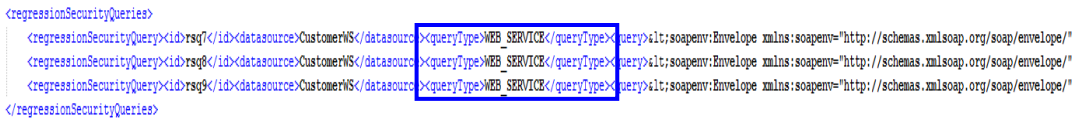
***flattenSecurityQueryQueriesXML*** – The flatten security query (SQL Queries) [node="regressionSecurityQuery" and "queryType"=QUERY] XML output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.



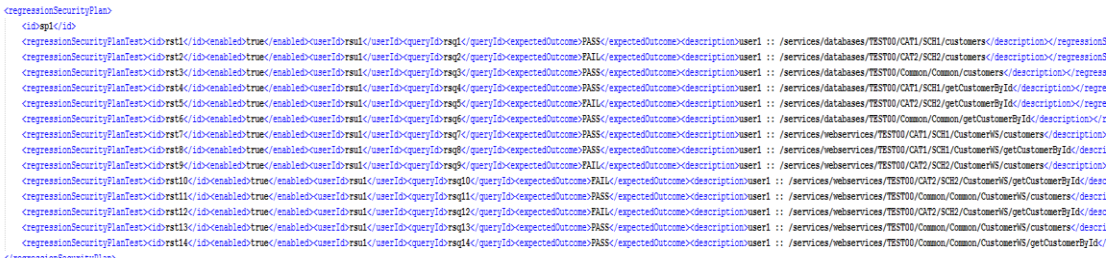

***flattenSecurityQueryProceduresXML*** – The flatten security query (SQL Procedures) [node="regressionSecurityQuery" and "queryType"=PROCEDURE] XML output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.




***flattenSecurityQueryWebServicesXML*** – The flatten security query (Web Services) [node="regressionSecurityQuery" and "queryType"=WEB_SERVICE] XML output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.




***flattenSecurityPlansXML*** – The flatten security plans [node="regressionSecurityPlanTest"] XML output determines whether to (true) flatten the results (false) print out with pretty XML output. It is recommended to print user XML as flattened so it can be read and edited like a table and is more compact.




***defaultProcParamValues*** – The listing of variable types provides the user with the ability to set default parameter values for procedures. For example, if a published procedure has an integer data type for a parameter and the default value for integer is configured as <integer>2</integer> then the published test input file will contain the following generated query:

[PROCEDURE]
database=EXAMPLES

SELECT count(*) cnt FROM LookupProduct( 2 )

### *Execute test run XML:*

```
<testRunParams>

     <testType>migration</testType>
     <logFilePath>C:/tmp/cis51/pubtestrun.log</logFilePath>
     <logDelimiter>PIPE</logDelimiter>
     <logAppend>no</logAppend>

     <baseDir>C:/tmp/cis51</baseDir>
     <delimiter>PIPE</delimiter>
     <printOutput>silent</printOutput>

     <!-- Performance Testing attributes →
     <perfTestThreads>10</perfTestThreads>
     <perfTestDuration>60</perfTestDuration>
     <perfTestSleepPrint>5</perfTestSleepPrint>
     <perfTestSleepExec>1</perfTestSleepExec>

     <!—Optional Filters for security execution only -->
     <securityExecution>
          <securityPlanIds>sp*</securityPlanIds>
          <securityOverallRatingException>true</securityOverallRatingException>
          <securityExecutionErrorException>true</securityExecutionErrorException>
     </securityExecution>

     <-- define the types of resources to execute on -->
    <runQueries>yes</runQueries>
    <runProcedures>yes</runProcedures>
    <runWS>no</runWS>

     <-- data source filters -->
      <useAllDatasources>no</useAllDatasources>
      <datasources>
         <dsName>EXAMPLES</dsName>
          <dsName>TestWebService</dsName>
          <dsName>ProductWebService</dsName>
      </datasources>

     <-- resource filters for above data sources -->
     <resources>
          <resource>TEST</resource>
     </resources>

</testRunParams>
```

### Attributes of Interest:

***testRunParams*** – parameters for the regression execution part of the module. Used by the method **executeRegressionTest().**

> ***testType*** – This parameter defines what type of test will be executed [functional|migration|performance|security].
>
> > ***Functional*** – Only execute the default query against a table or procedure. For a web service it is whatever input is provided. This test only cares about whether the

resources executes or not. Results may be output if the logFilePath contains a file path. If the input file contains a full query, it will be rewritten using the FROM clause to use the default queries described in the <newFileParams> section migration - A full query like "select * from <table> where <where_clause>" is executed so that a result set is returned and saved to a file.

***Migration*** – A full query like "select * from <table> where <where_clause>" is executed so that a result set is returned and saved to a file. A second instance of this test is executed and the result files are compared for equality.

***Performance*** – A full query like "select * from <table> where <where_clause>" is executed repeatedly on multiple threads for a specified duration of time to capture results. A performance test does not write results to an output file. It is only concerned with gathering statistics on executions.

***logFilePath*** – Full path to the test run results which are written to this log file.

***logDelimiter*** – This is the delimiter used when generating the summary log file. Options include [COMMA\,\PIPE\|\TAB\TILDE\~].

***logAppend*** – yes/no parameter to append log entries to the log file.

***baseDir*** – Base directory for output result files. These are the results/content files generated for each query which is executed. These are the files that will be used during the comparison phase. They are used for Migration Testing.

***delimiter*** – This is the delimiter used when generating the result data files. It is recommended that PIPE be used since data often contains commas. Options include [COMMA\,\PIPE\|\TAB\TILDE\~].

***printOutput*** – Print output: [verbose,summary,silent] verbose=print summary and results, summary=print query context, silent=nothing is printed to the command line.

***perfTestThreads*** – The number of threads to create when doing performance testing.

***perfTestDuration*** – The duration in seconds to execute the performance test for.

***perfTestSleepPrint*** – The number of seconds to sleep in between printing stats when executing the performance test.

***perfTestSleepExec*** – The number of seconds to sleep in between query executions when executing the performance test.

***securityExecution*** – an optional section for security testing only.

***securityPlanIds –*** The identifier or list of comma separated identifiers for the security test plan to execute.

***securityOverallRatingException*** – [yes|no|true|false] yes|true=Throw an exception when the overall security rating=FAIL, no|false=Don't throw an exception when the overall security rating=FAIL.

***securityExecutionErrorException*** – [yes|no|true|false] yes|true=throw an exception when their were errors, result=ERROR, no|false=don't throw an exception when their were errors, result=ERROR.

***runQueries*** – yes/no parameter that defines whether query statement should be executed from the published test input file for views of the configured Composite data source.

***runProcedures*** – yes/no parameter that defines whether procedure invocations should be executed from the published test input file for procedures of the configured Composite data source.

***runWS*** – yes/no parameter that defines whether web service invocation should be executed from the published test input file for web services of the configured Composite data source.

***useAllDatasources*** – yes/no parameter that defines whether to run queries from all the Composite data sources found in the published test input file or use the list provided in the input file. If yes, run tests for all data sources from the input file, data sources from the <datasources> list are ignored. If no, then only run queries for the data sources specified by the list of <datasources> found in the <testRunParams> section.

***datasources*** – a list of data sources to execute when running the input file

***dsName*** – the name of the Composite published data source.

- For Databases they will be found at the path: “Composite Data Services/Databases/<MYDB>”. Simply list the data service name. The Regression Module will introspect all catalogs and schemas to find views and procedures when generating the input file.
- For Web Services they will be found at the path: “Composite Data Services/Web Services/<MYDWS>”. Simply list the web service container name and not the port or operations. The Regression Module will introspect and generate for each operation found.

### *Compare two regression test runs XML:*

```
<compareFiles>

   <logFilePath>C:\\tmp\\regression\\pubtestcompare.log</logFilePath>
   <logDelimiter>PIPE</logDelimiter>
   <logAppend>no</logAppend>

   <baseDir1>C:/tmp/cis51</baseDir1>
   <baseDir2>C:/tmp/cis61</baseDir2>

   <compareQueries>yes</compareQueries>
```

```
        <compareProcedures>yes</compareProcedures>
        <compareWS>yes</compareWS>

        <useAllDatasources>no</useAllDatasources>
        <datasources>
                <dsName>MYTEST</dsName>
                <dsName></dsName>
                <dsName/>
                <dsName>ProductWebService</dsName>
                <dsName>testWebService00</dsName>
                <dsName>admin</dsName>
        </datasources>

        <resources>
                <resource>CAT1.SCH1.LookupProduct</resource>
                <resource>CAT1.SCH2.*</resource>
                <resource>SCH1.*</resource>
                <resource>TEST1.SCH.VIEW1</resource>
                <resource>ViewSales</resource>
                <resource>LookupProduct</resource>
                <resource>services.testWebService00.*</resource>
        </resources>

    </compareFiles>

</regressionTest>
```

### Attributes of Interest:

***compareFiles*** – parameters for the comparison. Used by the method **compareRegressionFiles().**

***logFilePath*** – Full path to the test run results which are written to this log file.

***logDelimiter*** – This is the delimiter used when generating the summary log file. Options include [COMMA\,\PIPE\|\TAB\TILDE\~].

***logAppend*** – yes/no parameter to append log entries to the log file.

***baseDir1*** – Base directory for output result files for instance 1 of the test runs. This would represent the baseline system from which you are migrating from or using as a baseline test. These files are the results/content files generated for each query which is executed.

***baseDir2*** – Base directory for output result files for instance 2 of the test runs. This would represent the target system from which you are migrating to or using as the incremental release. These files are the results/content files generated for each query which is executed.

***compareQueries*** – yes/no parameter that defines whether query results should be compared based on entries found in the regression input file.

***compareProcedures*** – yes/no parameter that defines whether procedure results should be compared based on entries found in the regression input file

***compareWS*** – yes/no parameter that defines whether web service results should be compared based on entries found in the regression input file.

***useAllDatasources*** – yes/no parameter that defines whether to compare queries from all the Composite data sources found in the published test input file or use the list provided in the input file. If yes, run tests for all data sources from the input file, data sources from the <datasources> list are ignored. If no, then only run queries for the data sources specified by the list of <datasources> found in the <compareFiles> section.

***datasources*** – a list of data sources to execute when running the input file

***dsName*** – the name of the Composite published data source.

- For Databases they will be found at the path: “Composite Data Services/Databases/<MYDB>”. Simply list the data service name. The Regression Module will introspect all catalogs and schemas to find views and procedures when generating the input file.
- For Web Services they will be found at the path: “Composite Data Services/Web Services/<MYDWS>”. Simply list the web service container name and not the port or operations. The Regression Module will introspect and generate for each operation found.

***resources*** – This is a list of resources for which to perform comparisons. A wild card ".*" may be used at the catalog or schema level to denote that all resources under that level should be compared. A fully qualified resource may be identified as well. This is a filter that can be fine-tuned for one or more resources. If left empty then all resources are compared. Overlapping resource wild cards will use the highest level specified in this list.

***resource*** – the resource filter .

### *Compare two query execution log files XML:*

```
<compareLogs>

      <logFilePath>C:\\tmp\\regression\\comparelogs.log</logFilePath>
      <logDelimiter>PIPE</logDelimiter>
      <logAppend>no</logAppend>

      <logFilePath1>C:/tmp/cis51/regression51.log</logFilePath1>
      <logFilePath2>C:/tmp/cis61/regression61.log</logFilePath2>

      <logDelimiter1>PIPE</logDelimiter1>
      <logDelimiter2>PIPE</logDelimiter2>
```

```
      <!-- Default duration delta for all queries.  When the difference between
duration in file2 and file1 is greater than the default duration then it is an error
(outside acceptable range). -->
      <durationDelta>000 00:00:01.0000</durationDelta>

</compareLogs>
```

### Attributes of Interest:

***compareLogs*** – parameters for the regression compare logs part of the module. Used by the method **compareRegressionLogs().**

***logFilePath*** – Full path to the test run results which are written to this log file.

***logDelimiter*** – This is the delimiter used when generating the summary log file. Options include [COMMA\,\PIPE\|\TAB\TILDE\~].

***logAppend*** – yes/no parameter to append log entries to the log file.

***logFilePath1*** – Path to the query execution log file representing the baseline system.

***logFilePath2*** – Path to the query execution log file representing the system to compare results with.

***logDelimiter1 (delimiter1)*** – This is the delimiter used within query execution log file 1. Options include [COMMA\,\PIPE\|\TAB\TILDE\~].\~]. Note: delimiter1 was deprecated but is still a valid XML element for the RegressionModule.xml.

***logDelimiter2 (delimiter2)*** – This is the delimiter used within query execution log file 2. Options include [COMMA\,\PIPE\|\TAB\TILDE\~]. Note: delimiter2 was deprecated but is still a valid XML element for the RegressionModule.xml.

***durationDelta*** – Default duration delta for all queries. When the difference between duration in file2 and file1 is greater than the default duration then it is an error (outside acceptable range). Format must be as follows within brackets: [000 00:00:00.0000]

### *Regression queries used to create the input file XML:*

```
    <!--Element regressionQueries is optional-->
    <regressionQueries>

        <regressionQuery>
            <datasource>MYTEST</datasource>
            <query>SELECT * FROM ViewSales WHERE ReorderLevel &lt;= 3</query>
        </regressionQuery>

        <regressionQuery>
            <datasource>MYTEST</datasource>
            <query>SELECT * FROM SCH1.ViewOrder</query>
            <!--Element durationDelta is optional-->
            <durationDelta>000 00:00:01.0000</durationDelta>
        </regressionQuery>
```

```
        <regressionQuery>
            <datasource>MYTEST</datasource>
            <query>SELECT * FROM SCH1.ViewSales WHERE CategoryID = 7</query>
        </regressionQuery>

        <regressionQuery>
            <datasource>MYTEST</datasource>
            <query>SELECT * FROM CAT1.SCH1.ViewSales where Discount > 0</query>
        </regressionQuery>

        <regressionQuery>
            <datasource>MYTEST</datasource>
            <query>SELECT * FROM CAT1.SCH2.ViewSales WHERE ProductName like
'Mega%'</query>
        </regressionQuery>

        <regressionQuery>
            <datasource>MYTEST</datasource>
            <query>SELECT * FROM CAT1.SCH2.ViewSupplier</query>
        </regressionQuery>

        <regressionQuery>
            <datasource>MYTEST</datasource>
            <query>SELECT * FROM getProductName(1)</query>
        </regressionQuery>

        <regressionQuery>
            <datasource>ProductWebService</datasource>
            <!-- soap11 namesspace must be:
xmlns:soapenv=&quot;http://schemas.xmlsoap.org/soap/envelope/&quot; -->
            <query>
&lt;soapenv:Envelope
xmlns:soapenv=&quot;http://schemas.xmlsoap.org/soap/envelope/&quot;
xmlns:tem=&quot;http://tempuri.org/&quot;&gt;
       &lt;soapenv:Header/&gt;
       &lt;soapenv:Body&gt;
              &lt;tem:LookupProduct&gt;
       &lt;tem:LookupProductDesiredproduct&gt;10&lt;/tem:LookupProductDesiredproduct&
gt;
             &lt;/tem:LookupProduct&gt;
       &lt;/soapenv:Body&gt;
&lt;/soapenv:Envelope&gt;
             </query>
            <!--Element durationDelta is optional-->
           <durationDelta>000 00:00:01.0000</durationDelta>

            <!--Element wsPath is optional-->
            <wsPath>/soap11/ProductWebService</wsPath>
            <!--Element wsAction is optional-->
            <wsAction>LookupProduct</wsAction>
            <!--Element wsEncrypt is optional-->
            <wsEncrypt>false</wsEncrypt>
            <!--Element wsContentType is optional-->
            <wsContentType>text/xml;charset=UTF-8</wsContentType>
        </regressionQuery>

        <regressionQuery>
            <datasource>ProductWebService</datasource>
            <!-- soap12 namesspace must be:
xmlns:soapenv=&quot;http://www.w3.org/2003/05/soap-envelope&quot; -->
            <query>
```

```
&lt;soapenv:Envelope xmlns:soapenv=&quot;http://www.w3.org/2003/05/soap-
envelope&quot; xmlns:tem=&quot;http://tempuri.org/&quot;&gt;
       &lt;soapenv:Header/&gt;
       &lt;soapenv:Body&gt;
              &lt;tem:LookupProduct&gt;
       &lt;tem:LookupProductDesiredproduct&gt;10&lt;/tem:LookupProductDesiredproduct&
gt;
              &lt;/tem:LookupProduct&gt;
              &lt;/soapenv:Body&gt;
&lt;/soapenv:Envelope&gt;
             </query>
            <!--Element wsPath is optional-->
            <wsPath>/soap12/ProductWebService</wsPath>
            <!--Element wsAction is optional-->
            <wsAction>LookupProduct</wsAction>
            <!--Element wsEncrypt is optional-->
            <wsEncrypt>false</wsEncrypt>
            <!--Element wsContentType is optional-->
            <wsContentType>application/soap+xml;charset=UTF-8</wsContentType>
        </regressionQuery>
    </regressionQueries>

</p1:RegressionModule>
```

### Attributes of Interest:

***regressionQueries*** – parameters for the input file creation part of the module. Used by the method **createRegressionInputFile().** This section is separated from the <newFileParams> because it is considered a common section for all regression tests.

***regressionQuery*** – An repression query instance.

***datasource*** – The datasource for which this query belongs to. Required for QUERY, PROCEDURE and WEB_SERVICE.

- For Databases they will be found at the path: “Composite Data Services/Databases/<MYDB>”.
- For Web Services they will be found at the path: “Composite Data Services/Web Services/<MYDWS>”.

***query*** – The actual query which will be used to replace the template query. Required for QUERY, PROCEDURE and WEB_SERVICE.

- QUERY – This is a full query with where clause conditions. It is recommended that where clauses be used to limit the result sets to a reasonable size.
  - Example: SELECT * FROM CAT1.SCHEMA1.TABLE1 WHERE ReorderLevel <= 3
  - Notice that the “<” sign uses XML escape sequences as you cannot use “<” directly in the XML

- PROCEDURE – This is a complete procedure invocation using a SELECT style of invocation. CALL statements are not supported at this time. For procedures returning SCALAR output, you must use SELECT * FROM proc(param1,…). Similarly, for procedures returning CURSOR output, you must use SELECT * FROM proc(param1,…). Same syntax.
    - Example: SELECT * FROM getProductName(1) – returns scalar
    - Example: SELECT * FROM LookupProduct(1) – returns cursor
- WEB_SERVICE – Web services may invoke Composite Legacy web services, Composite New Web Services with soap11 and Composite New Web Services with soap12. Namespaces for soap11 and soap12 must be as follows:
- **Note**: Web Service operations may not contain spaces or begin with special characters such as `,~,!,@,#,$,%,^,&,*,(,),-,+,=,',?,<,>,?,/,\,| or any other illegal character not accepted by the XML standard. Web Service operations may begin with alphanumeric and underscore characters.
    - soap11 namespace must be:
    - **xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"**
    - soap12 namespace must be:

      **xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"**
    - Query input strings for web services are XML. You cannot put pure XML tags within the RegressionModule.xml file. You have to escape the XML. If you have access to the Composite Utilities you can generate the result by executing "**/shared/Utilities/xml/escapeXML**". Paste in the XML input query string and the result is an escaped string. Paste the escaped string in the <query></query> node. Examples are shown below. But first, the following list shows what XML needs to be escaped:
        - & → &
        - < → <
        - > → >
        - ' → '
        - " → "

- Invoking New Composite Web Services in 6.1 or higher using a parameter style of BARE or WRAPPED.
  - BARE refers to a single parameter in the Body of the input message with no wrapping element. An example follows:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns1="http://tempuri.org/">
   <soapenv:Header/>
   <soapenv:Body>

<ns1:LookupProductDesiredproduct>5</ns1:LookupProductDesiredproduct>
    </soapenv:Body>
</soapenv:Envelope>
```

  - WRAPPED refers to multiple parameters in the input Body with an element wrapping the input parameters. An example follows:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns1="http://tempuri.org/">
   <soapenv:Header/>
   <soapenv:Body>
      <ns1:WrapperedLookupproduct2>

<ns1:LookupProduct2Desiredproduct>3</ns1:LookupProduct2Desiredproduct
>
        <ns1:LookupProduct2Debug>Y</ns1:LookupProduct2Debug>
     </ns1:WrapperedLookupproduct2>
   </soapenv:Body>
</soapenv:Envelope>
```

- soap11 query example – WRAPPED invocation parameter style:

```
&lt;soapenv:Envelope
xmlns:soapenv=&quot;http://schemas.xmlsoap.org/soap/envelope/&qu
ot; xmlns:tem=&quot;http://tempuri.org/&quot;&gt;
       &lt;soapenv:Header/&gt;
       &lt;soapenv:Body&gt;
              &lt;tem:LookupProduct&gt;

       &lt;tem:LookupProductDesiredproduct&gt;10&lt;/tem:LookupPr
oductDesiredproduct&gt;
              &lt;/tem:LookupProduct&gt;
       &lt;/soapenv:Body&gt;
&lt;/soapenv:Envelope&gt;
```

- soap11 query example – BARE invocation parameter style:

```
&lt;soapenv:Envelope
xmlns:soapenv=&quot;http://schemas.xmlsoap.org/soap/envelope/&qu
ot; xmlns:tem=&quot;http://tempuri.org/&quot;&gt;
       &lt;soapenv:Header/&gt;
       &lt;soapenv:Body&gt;
```

```
        &lt;tem:LookupProductDesiredproduct&gt;10&lt;/tem:LookupPr
oductDesiredproduct&gt;
        &lt;/soapenv:Body&gt;
&lt;/soapenv:Envelope&gt;
```

- soap12 query example – WRAPPED invocation parameter style:

```
&lt;soapenv:Envelope
xmlns:soapenv=&quot;http://www.w3.org/2003/05/soap-
envelope&quot; xmlns:tem=&quot;http://tempuri.org/&quot;&gt;
        &lt;soapenv:Header/&gt;
        &lt;soapenv:Body&gt;
                &lt;tem:LookupProduct&gt;

        &lt;tem:LookupProductDesiredproduct&gt;10&lt;/tem:LookupPr
oductDesiredproduct&gt;
                &lt;/tem:LookupProduct&gt;
                &lt;/soapenv:Body&gt;
&lt;/soapenv:Envelope&gt;
```

***durationDelta*** – ***(optional)*** Duration delta for this query which overrides the default duration delta set in the <compareLogs> section. When the difference between duration in file2 and file1 is greater than the default duration then it is an error (outside acceptable range). Format must be as follows within brackets: [000 00:00:00.0000]

***wsPath*** – ***(optional)*** The web service path is only used when the query is for WEB_SERVICE. This is known as the endpoint URL. The endpoint URL is how an application will reference this web service. There is a different way to determine the endpoint URL for Composite Legacy and Composite New web service.

The web service path is really the endpoint URL or the port path. For Composite Legacy Web Services open the port container in CIS Studio, click on the info tab and look at Endpoint URL: http://localhost:9400/services/testWebService00/testService/testPort.ws. The wsPath=/services/testWebService00/testService/testPort.ws

For Composite New Web Service (6.1 or higher) open the web service in Studio and click on the SOAP tab and Service panel and look at Endpoint and WSDL URLS: /testWebService. For Soap11 wsPath=/soap11/testWebService. For Soap12 wsPath=/soap12/testWebService

***wsAction*** – ***(optional)*** The web service action is only used when the query is for WEB_SERVICE. The action is the same thing as the operation.

***wsEncrypt*** – ***(optional)*** The web service encrypt is only used when the query is for WEB_SERVICE. The web service encrypt determines if http (false) or https (true) should be used.

***wsContentType*** – ***(optional)*** The web service content type is only used when the query is for WEB_SERVICE. Content Type for soap11 and soap12 must be as follows:

- soap11: text/xml;charset=UTF-8
- soap12: application/soap+xml;charset=UTF-8

## HOW TO EXECUTE

The following section describes how to setup a property file for command line and execute the script. This script will use the **RegressionModule.xml** that was described in the previous section.

### *Script Execution*

The full details on property file setup and script execution can be found in the document "*Composite PS Promotion and Deployment Tool User's Guide v1.0.pdf*". The abridged version is as follows:

```
Windows: ExecutePDTool.bat -exec ../resources/plans/UnitTest-Regression.dp

Unix: ./ExecutePDTool.sh -exec ../resources/plans/UnitTest-Regression.dp
```

***Properties File (UnitTest-Regression.dp):***

Property File Rules:

```
# ----------------------------
# UnitTest-Regression.dp
# ----------------------------
#   1. All parameters are space separated.  Commas are not used.
#         a. Any number of spaces may occur before or after any parameter and are
trimmed.
#
#   2. Parameters should always be enclosed in double quotes according to these
rules:
#         a. when the parameter value contains a comma separated list:
#                                     ANSWER: "ds1,ds2,ds3"
#
#         b. when the parameter value contain spaces or contains a dynamic variable
that will resolve to spaces
#            i.   There is no distinguishing between Windows and Unix variables.
Both UNIX style variables ($VAR) and
#                 and Windows style variables (%VAR%) are valid and will be parsed
accordingly.
#            ii.  All parameters that need to be grouped together that contain
spaces are enclosed in double quotes.
#            iii. All paths that contain or will resolve to a space must be enclosed
in double quotes.
#                 An environment variable (e.g. $MODULE_HOME) gets resolved on
invocation PDTool.
#                       Paths containing spaces must be enclosed in double quotes:
#                              ANSWER: "$MODULE_HOME/LabVCSModule.xml"
#                       Given that MODULE_HOME=C:/dev/Cis Deploy
Tool/resources/modules, PDTool automatically resolves the variable to
#                       "C:/dev/Cis Deploy Tool/resources/modules/LabVCSModule.xml".
#
#         c. when the parameter value is complex and the inner value contains spaces
```

```
#                    i. In this example $PROJECT_HOME will resolve to a path that
contains spaces such as C:/dev/Cis Deploy Tool
#                       For example take the parameter -pkgfile
$PROJECT_HOME$/bin/carfiles/testout.car.
#                       Since the entire command contains a space it must be
enclosed in double quotes:
#                                ANSWER: "-pkgfile
$PROJECT_HOME/bin/carfiles/testout.car"
#
#   3. A comment is designated by a # sign preceding any other text.
#         a. Comments may occur on any line and will not be processed.
#
#   4. Blank lines are not processed
#         a. Blank lines are counted as lines for display purposes
#         b. If the last line of the file is blank, it is not counted for display
purposes.
```

Property File Parameters:

```
# ----------------------------
# Parameter Specification:
# ----------------------------
# Param1=[PASS or FAIL]  :: Expected Regression Behavior.  Informs the script
whether you expect the action to pass or fail.  This can be used for regression
testing.
# Param2=[TRUE or FALSE] :: Exit Orchestration script on error
# Param3=Module Batch/Shell Script name to execute (no extension).  Extension is
added by script.
# Param4=Module Action to execute
# Param5-ParamN=Specific space separated parameters for the action.
#    Comma separated list parameters need to be enclosed in double quotes "".
```

Property File Example:

```
# ----------------------------------------
# Begin task definition list:
# ----------------------------------------
PASS   TRUE   ExecuteAction      createRegressionInputFile   $SERVERID
       Test1  "$MODULE_HOME/RegressionModule.xml"     "$MODULE_HOME/servers.xml"
PASS   TRUE   ExecuteAction      executeRegressionTest       $SERVERID
       Test2  "$MODULE_HOME/RegressionModule.xml" "$MODULE_HOME/servers.xml"
PASS   TRUE   ExecuteAction       compareRegressionFiles     $SERVERID
       Test2  "$MODULE_HOME/RegressionModule.xml" "$MODULE_HOME/servers.xml"
PASS   TRUE   ExecuteAction      compareRegressionLogs       $SERVERID
       Test2         "$MODULE_HOME/RegressionModule.xml" "$MODULE_HOME/servers.xml"
PASS   TRUE   ExecuteAction      executePerformanceTest       $SERVERID
       Test3         "$MODULE_HOME/RegressionModule.xml" "$MODULE_HOME/servers.xml"
PASS   TRUE   ExecuteAction      executeSecurityTest       $SERVERID
       Test4.1       "$MODULE_HOME/RegressionModule.xml" "$MODULE_HOME/servers.xml"
PASS   TRUE   ExecuteAction      generateRegressionSecurityXML $SERVERID
       Test4.1       "$MODULE_HOME/RegressionModule.xml" "$MODULE_HOME/servers.xml"
```

### *Ant Execution*

The full details on build file setup and ant execution can be found in the document "*Composite PS Promotion and Deployment Tool User's Guide v1.0.pdf*". The abridged version is as follows:

Windows: ExecutePDTool.bat -ant ../resources/ant/build-Regression.xml

Unix: ./ExecutePDTool.sh -ant ../resources/ant/build-Regression.xml

***Build File:***

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="PDTool" default="default" basedir=".">

  <description>description</description>

  <!-- Default properties -->
  <property name="SERVERID"                 value="localhost"/>
  <property name="noarguments"              value="&amp;quot;&amp;quot;"/>

  <!-- Default Path properties -->
  <property name="RESOURCE_HOME"            value="${PROJECT_HOME}/resources"/>
  <property name="MODULE_HOME"              value="${RESOURCE_HOME}/modules"/>
  <property name="pathToServersXML"         value="${MODULE_HOME}/servers.xml"/>
  <property name="pathToArchiveXML"         value="${MODULE_HOME}/ArchiveModule.xml"/>
  <property name="pathToDataSourcesXML"     value="${MODULE_HOME}/DataSourceModule.xml"/>
  <property name="pathToGroupsXML"          value="${MODULE_HOME}/GroupModule.xml"/>
  <property name="pathToPrivilegeXML"       value="${MODULE_HOME}/PrivilegeModule.xml"/>
  <property name="pathToRebindXML"          value="${MODULE_HOME}/RebindModule.xml"/>
  <property name="pathToRegressionXML"      value="${MODULE_HOME}/RegressionModule.xml"/>
  <property name="pathToResourceXML"        value="${MODULE_HOME}/ResourceModule.xml"/>
  <property name="pathToResourceCacheXML"   value="${MODULE_HOME}/ResourceCacheModule.xml"/>
  <property name="pathToServerAttributeXML" value="${MODULE_HOME}/ServerAttributeModule.xml"/>
  <property name="pathToTriggerXML"         value="${MODULE_HOME}/TriggerModule.xml"/>
  <property name="pathToUsersXML"           value="${MODULE_HOME}/UserModule.xml"/>
  <property name="pathToVCSModuleXML"       value="${MODULE_HOME}/VCSModule.xml"/>

  <!-- Custom properties -->
  <property name="regressionIds"            value="Test1,Test2"/>
  <property name="pathToGenRegressionXML"   value="${MODULE_HOME}/getRegressionModule.xml"/>

  <!-- Default Classpath [Do Not Change] -->
  <path id="project.class.path">
       <fileset dir="${PROJECT_HOME}/lib"><include name="**/*.jar"/></fileset>
       <fileset dir="${PROJECT_HOME}/dist"><include name="**/*.jar"/></fileset>
       <fileset dir="${PROJECT_HOME}/ext/ant/lib"><include name="**/*.jar"/></fileset>
  </path>

  <taskdef name="executeJavaAction" description="Execute Java Action"
classname="com.cisco.dvbu.ps.deploytool.ant.CompositeAntTask"
classpathref="project.class.path"/>
```

```
  <!-- ================================
       target: default
    ================================= -->
  <target name="default" description="Update CIS with environment specific parameters">

     <!-- Execute Line Here -->
        <executeJavaAction description="Generate"           action="createRegressionInputFile"

        arguments="${SERVERID}^${regressionIds}^${pathToGenRegressionXML}^${pathToServersXML}"
               endExecutionOnTaskFailure="TRUE"/>

    <!-- Windows or UNIX
        <executeJavaAction description="Generate"           action="createRegressionInputFile"

        arguments="${SERVERID}^${regressionIds}^${pathToGenRegressionXML}^${pathToServersXML}"
               endExecutionOnTaskFailure="TRUE"/>

        <executeJavaAction description="Execute"            action="executeRegressionTest"

        arguments="${SERVERID}^${regressionIds}^${pathToRegressionXML}^${pathToServersXML}"
               endExecutionOnTaskFailure="TRUE"/>

        <executeJavaAction description="CompareResults"     action="compareRegressionFiles"

        arguments="${SERVERID}^${regressionIds}^${pathToRegressionXML}^${pathToServersXML}"
               endExecutionOnTaskFailure="TRUE"/>

        <executeJavaAction description="CompareLogs"        action="compareRegressionLogs"

        arguments="${SERVERID}^${regressionIds}^${pathToRegressionXML}^${pathToServersXML}"
               endExecutionOnTaskFailure="TRUE"/>

        <executeJavaAction description="Execute Performance" action="executePerformanceTest"
        arguments="${SERVERID}^${regressionIds}^${pathToRegressionXML}^${pathToServersXML}"
               endExecutionOnTaskFailure="TRUE"/>

        <executeJavaAction description="Generate Security XML"
        action="generateRegressionSecurityXML"
        arguments="${SERVERID}^Test4.1^${pathToRegressionXML}^${pathToServersXML}"
               endExecutionOnTaskFailure="TRUE"/>

        <executeJavaAction description="Execute Security Test" action="executeSecurityTest"
               arguments="${SERVERID}^Test4.1^${pathToRegressionXML}^${pathToServersXML}"
               endExecutionOnTaskFailure="TRUE"/>
    -->
  </target>
</project>
```

## *Module ID Usage*

The following explanation provides a general pattern for module identifiers. The module identifier for this module is “regressionIds”.

- Possible values for the module identifier:
- 1. ***Inclusion List*** - CSV string like “id1,id2”

- PDTool will process only the passed in identifiers in the specified module XML file.

Example command-line property file

```
PASS   FALSE  ExecuteAction   executeRegressionTest      $SERVERID
"test1,test2"              "$MODULE_HOME/RegressionModule.xml"
"$MODULE_HOME/servers.xml"
```

Example Ant build file

```
<executeJavaAction description="Update"        action=" executeRegressionTest"
arguments="${SERVERID}^test1,test2^${pathToRegressionXML}^${pathToServersXML}"
```

- 2. ***Process All*** - '*' or whatever is configured to indicate all resources
  - PDTool will process all resources in the specified module XML file.

Example command-line property file

```
PASS   FALSE  ExecuteAction   executeRegressionTest $SERVERID "*"
       "$MODULE_HOME/RegressionModule.xml" "$MODULE_HOME/servers.xml"
```

Example Ant build file

```
<executeJavaAction description="Update"        action="executeRegressionTest"
arguments="${SERVERID}^*^${pathToRegressionXML}^${pathToServersXML}"
```

- 3. ***Exclusion List*** - CSV string with '-' or whatever is configured to indicate exclude resources as prefix like “-id1,id2”
  - PDTool will ignore passed in resources and process the rest of the identifiers in the module XML file.

Example command-line property file

```
PASS   FALSE  ExecuteAction   executeRegressionTest $SERVERID "-test3,test4"
              "$MODULE_HOME/RegressionModule.xml" "$MODULE_HOME/servers.xml"
```

Example Ant build file

```
<executeJavaAction description="Update"        action="executeRegressionTest"
arguments="${SERVERID}^-
test3,test4^${pathToRegressionXML}^${pathToServersXML}"
```

## EXECUTION RESULTS

The following section describes the output produced by the execution runs. The following is a summary of what is described in this section:

(1) General Log File Execution – log file of the results of PD Tool execution (app.log)

(2) Generated Published Test Input File – drives regression or performance tests

(3) Query Execution Log File – regression test log

(4) Query Execution File Comparison Log – log of regression result file comparisons

(5) Compare Regression Logs – summary log of comparing two test logs

(6) Query Execution Performance Log File – performance test log

(7) Security Execution Test – security test log

### *General Log File of Execution*

The full details on property file setup and script execution can be found in the document "*Composite PS Promotion and Deployment Tool User's Guide v1.0.pdf*". The abridged version is as follows:

Output: Log file results are output to /log/app.log.

### *Generated Published Test Input File*

Result from **createRegressionInputFile()**.

Output: Regression input file is output to the location identified by:

<inputFilePath>C:\\tmp\\pdtool\\regression1.inp</inputFilePath>

The following provides the key summary information for what will be output to app.log:

```
<Calling Action createRegressionInputFile>
<------------------------------------------------------------------------>
<Processing action "generate input file" for regression id: Test1>
<------------------------------------------------------------------------>
<-------------------------------------------------------->
<                                                        >
<            Total Queries Generated: 8                  >
<         Total Procedures Generated: 5                  >
<       Total Web Services Generated: 5                  >
<                                     ---------          >
<Total Combined ---------> Generated: 18                 >
<                                                        >
<     Input file generation duration:  000 00:00:01.0138 >
<                                                        >
```

```
<Review input file:
D:/dev/PDTool_Test/regression/RegressionModule/regression1.inp>
<-------------------------------------------------------->
<CisDeployTool::Successfully completed createRegressionInputFile.>
```

The published test input file can be constructed in three ways. It can be generated by the Regression Module using the **createRegressionInputFile()** method. It can be constructed manually by a developer or it can be generated and then modified by a developer.

The "outputFilename" parameter used by the Regression or Migration tests to output data to a text file. It is not used for either Performance or Security tests. The data files are generated in a folder based on the XML tag <baseDir> concatenated with the virtual data source name. The "outputFilename" is constructed as follows:

- QUERY – CATALOG.SCHEMA.VIEW_NAME.txt
- PROCEDURE – CATALOG.SCHEMA.PROCEDURE_NAME.txt
- WEB_SERVICE – path.action.txt
    - Note: All slashes of any type found in the path are converted to periods.

The example below shows the format of a published test input file. It provides an example of how the Regression Module can invoke published views, procedures and web services:

```
[QUERY]
database=EXAMPLES
outputFilename=ViewSales.txt
SELECT * FROM ViewSales

 [PROCEDURE]
database=EXAMPLES
outputFilename=CAT1.SCH1.LookupProduct.txt
SELECT count(*) cnt FROM CAT1.SCH1.LookupProduct( 1 )

[WEB_SERVICE]
database=demoPDTool
path=/services/demoPDTool/demoPDToolService/demoPDToolPort.ws
action=getOrdersById
encrypt=false
contentType=text/xml;charset=UTF-8
outputFilename=services.demoPDTool.demoPDToolService.demoPDToolPort.ws.getOrdersById .txt
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
   <soap-env:Body>
      <ns1:getOrdersById
xmlns:ns1="http://www.dvbu.cisco.com/services/webservices/demoPDTool/demoPDToolService">
         <ns1:OrderId>2</ns1:OrderId>
      </ns1:getOrdersById>
   </soap-env:Body>
</soap-env:Envelope>
```

### *Input File Query Governor (TOP)*

The regression query input file can be modified after it is generated to add the TOP (n) command to QUERY and PROCEDURE sections of the input file. The TOP command will return n number of rows. An example is shown below:

```
[QUERY]
database=MYTEST
outputFilename=ViewSales2.txt
SELECT TOP 2 ProductID, ProductName, ReorderLevel, LeadTime FROM ViewSales2
```

Results – without the top command there would be 50 rows returned. With the TOP command there are only 2 rows returned:

```
ProductID|ProductName|ReorderLevel|LeadTime
1|Maxtific 40GB ATA133 7200|20|14 Days
2|Mega Zip 750MB USB 2.0|5|7 Days
```

```
[PROCEDURE]
database=MYTEST
outputFilename=LookupProduct2.txt
SELECT TOP 1 * FROM LookupProduct2( 1 , 'M' )
```

Results – without the TOP command there would be 6 rows returned. With the TOP command there is only 1 row returned.

```
ProductID|ProductName|ProductDescription|CategoryID|SerialNumber|UnitPrice|ReorderLevel|LeadTime
1|Maxtific 40GB ATA133 7200|Maxtific Storage  40 GB|1|221-887-3458|89.99|20|14 Days
```

### *Query Execution Log File*

Result from **executeRegressionTest()**.

Output:

- Execution log file output to the location identified by:

  <logFilePath>C:/tmp/pdtool/FunctionalTest.log</logFilePath>

- Base directory location identified by:

  <baseDir>c:/tmp/pdtool/functest</baseDir>

  Each individual query result file is saved to sub-directories based on the published data service DB name. The catalog+schema+view name are used for the file name. For example, given a published data service of MYDB.MyCatalog.MySchema.View1 the folder and filename created would be as follows:

  c:/tmp/pdtool/functest/MYDB/MyCatalog.MySchema.View1.txt

  The contents of the file depend on the query being executed. A simple functional test performs a count(*) on the view. For example, the contents for View1 might be:

  cnt

35

The following provides the key summary information for what will be output to app.log:

```
<Calling Action executeRegressionTest>
<------------------------------------------------------------------------>
<Processing action "execute test" for regression id: Test1>
<------------------------------------------------------------------------>
<-------------------------------------------------------->
<---------- Regression Published Test Summary ----------->
<-------------------------------------------------------->
<Test Type: functional                                    >
<  Execute a default query: SELECT COUNT(1) FROM...       >
<                                                         >
<Total Successful        Queries: 8                       >
<Total Successful     Procedures: 5                       >
<Total Successful   Web Services: 5                       >
<                                 ---------               >
<Total Successful -------> Tests: 18                      >
<                                                         >
<Total Skipped           Queries: 0                       >
<Total Skipped        Procedures: 0                       >
<Total Skipped      Web Services: 0                       >
<                                 ---------               >
<Total Skipped ----------> Tests: 0                       >
<                                                         >
<Total Failed            Queries: 0                       >
<Total Failed         Procedures: 0                       >
<Total Failed       Web Services: 0                       >
<                                 ---------               >
<Total Failed -----------> Tests: 0                       >
<                                                         >
<Total Combined ---------> Tests: 18                      >
<                                                         >
<        Published Test duration:  000 00:00:00.0640      >
<                                                         >
<Review "pubtest" Summary:
D:/dev/PDTool_Test/regression/RegressionModule/FunctionalTest.log>
<-------------------------------------------------------->
<CisDeployTool::Successfully completed executeRegressionTest.>
```

This is the output log for the method that executes the regression test. Each line in the log file is driven by a query contained in the “regression test input file”. Result status can be SKIPPED, SUCCESS, or ERROR.

- SKIPPED – regression test skipped due to wrong type, run test turned off for this type, no database match or no resource match.
- SUCCESS – the regression test was successfully executed.
- ERROR – there was an error during processing. The Message will contain error.

An example log file is provided below which demonstrate what the query execution log looks like:

Result| ExecutionStartTime| Duration| Rows| Database| Query| Type| OutputFile| Message

SUCCESS| 2012-06-01 17:45:10.0798| 000 00:00:00.0000| 9| MYTEST| SELECT * FROM ViewSales WHERE ReorderLevel <= 3| QUERY| C:/tmp/cis51/MYTEST/ViewSales.txt|

SUCCESS| 2012-06-01 17:45:10.0798| 000 00:00:00.0031| 50| MYTEST| SELECT * FROM SCH1.ViewOrder| QUERY| C:/tmp/cis51/MYTEST/SCH1.ViewOrder.txt|

SUCCESS| 2012-06-01 17:45:10.0845| 000 00:00:00.0015| 23| MYTEST| SELECT * FROM CAT1.SCH1.ViewSales where Discount > 0| QUERY| C:/tmp/cis51/MYTEST/CAT1.SCH1.ViewSales.txt|

SUCCESS| 2012-06-01 17:45:10.0876| 000 00:00:00.0015| 35| MYTEST| SELECT * FROM CAT1.SCH2.ViewSupplier| QUERY| C:/tmp/cis51/MYTEST/CAT1.SCH2.ViewSupplier.txt|

SUCCESS| 2012-06-01 17:45:10.0891| 000 00:00:00.0016| 1| MYTEST| SELECT * FROM getProductName(1)| PROCEDURE| C:/tmp/cis51/MYTEST/getProductName.txt|

SUCCESS| 2012-06-01 17:45:10.0923| 000 00:00:00.0000| 1| MYTEST| SELECT count(*) cnt FROM LookupProduct( 1 )| PROCEDURE| C:/tmp/cis51/MYTEST/LookupProduct.txt|

SUCCESS| 2012-06-01 17:45:10.0923| 000 00:00:00.0015| 1| ProductWebService| /soap11/ProductWebService/LookupProduct| WS| C:/tmp/cis51/ProductWebService/soap11.ProductWebService.LookupProduct.txt|

SUCCESS| 2012-06-01 17:45:10.0938| 000 00:00:00.0000| 1| ProductWebService| /soap12/ProductWebService/LookupProduct| WS| C:/tmp/cis51/ProductWebService/soap12.ProductWebService.LookupProduct.txt|

ERROR| 2012-06-01 17:45:10.0954| 000 00:00:00.0015| 0| testWebService00| /services/testWebService00/testService/testPort.ws/testprocsimple| WS| C:/tmp/cis51/testWebService00/services.testWebService00.testService.testPort.ws.testprocsimple.txt|executeWs(): Server returned HTTP response code: 500 for URL: http://localhost:9420/services/testWebService00/testService/testPort.ws DETAILED_MESSAGE=[executeWs(): <?xml version="1.0" encoding="utf-8"?> <soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"> <soap-env:Body> <soap-env:Fault> <faultcode>soap-env:Server</faultcode> <faultstring>/shared/test00/executeProc/testprocsimple.CISEXCEPTION: dummyStr at /shared/test00/executeProc/testprocsimple (line 25) …[the rest of the message has been trimmed.]

### *Query Execution File Comparison Log*

Result from **compareRegressionFiles()**.

Output:

- Comparison log file output to the location identified by:

  <logFilePath>C:\\tmp\\pdtool\\MigrationCompareFiles2.log</logFilePath>

- Base directory location identified by:

  <baseDir1>C:/tmp/pdtool/cis51</baseDir1>

  Compare: MYDB/MyCatalog.MySchema.View1.txt

  <baseDir2>C:/tmp/pdtool/cis61</baseDir2>

  Compare with: MYDB/MyCatalog.MySchema.View1.txt

The database and views must be the same between two releases. For example, the database, catalog, schema and views are stored in folder and files such as MYDB/MyCatalog.MySchema.View1.txt.

The following provides the key summary information for what will be output to app.log:

```
<Calling Action compareRegressionFiles>
<------------------------------------------------------------------------>
<processing action "compare result files" for regression id: Test2.2>
<------------------------------------------------------------------------>
<-------------------------------------------------------->
<----- Regression Query Content Comparison Summary ------>
<-------------------------------------------------------->
<                                                        >
<Compares the contents of two regression files to        >
<determine if they are the same or not.                  >
<                                                        >
<      Success=checksums are the same.                   >
<      Skipped=did not match database or resource filter.>
<      Failure=file checksums are different.             >
<      Error=Unknown issue encountered.                  >
<                                                        >
<      Total Successful Comparisons: 14                  >
<         Total Skipped Comparisons: 4                   >
<         Total Failure Comparisons: 0                   >
<           Total Error Comparisons: 0                   >
<                                    ---------           >
<                 Total Comparisons: 18                  >
<                                                        >
<    Regression comparsion duration:  000 00:00:00.0047  >
<                                                        >
<Review "content comparison" Summary:
D:/dev/PDTool_Test/regression/RegressionModule/MigrationCompareFiles2.log>
<-------------------------------------------------------->
<CisDeployTool::Successfully completed compareRegressionFiles.>
```

This is the output log for the method that compares the individual query execution result files from two separate test runs to determine if the file checksums match or not. For each query it consumes the corresponding text result files performs the comparison and prints out a log comparison message shown below. Result status can be SKIPPED, SUCCESS, FAILURE, or ERROR.

- SKIPPED – file comparison skipped due to wrong type, compare turned off for this type, no database match or no resource match.
- SUCCESS – the file checksum is equal.
- FAILURE – file checksums do not match.
- ERROR – there was an error during processing. The Message will contain error.

An example log file is provided below which demonstrate what the query execution file comparison looks like:

Result| CompareStartTime| Database| ResourceURL| Type| File1| File2| Message

SUCCESS| 2012-06-01 17:45:12.0015| MYTEST| ViewSales| QUERY| C:/tmp/cis51/MYTEST/ViewSales.txt| C:/tmp/cis61/MYTEST/ViewSales.txt|

SUCCESS| 2012-06-01 17:45:12.0030| MYTEST| SCH1.ViewOrder| QUERY| C:/tmp/cis51/MYTEST/SCH1.ViewOrder.txt| C:/tmp/cis61/MYTEST/SCH1.ViewOrder.txt|

SUCCESS| 2012-06-01 17:45:12.0030| MYTEST| SCH1.ViewSales| QUERY| C:/tmp/cis51/MYTEST/SCH1.ViewSales.txt| C:/tmp/cis61/MYTEST/SCH1.ViewSales.txt|

SKIPPED| 2012-06-01 17:45:12.0030| MYTEST| CAT1.SCH1.ViewSales| QUERY| C:/tmp/cis51/MYTEST/CAT1.SCH1.ViewSales.txt| C:/tmp/cis61/MYTEST/CAT1.SCH1.ViewSales.txt|

SUCCESS| 2012-06-01 17:45:12.0030| MYTEST| CAT1.SCH2.ViewSales| QUERY| C:/tmp/cis51/MYTEST/CAT1.SCH2.ViewSales.txt| C:/tmp/cis61/MYTEST/CAT1.SCH2.ViewSales.txt|

SUCCESS| 2012-06-01 17:45:12.0030| MYTEST| CAT1.SCH2.ViewSupplier| QUERY| C:/tmp/cis51/MYTEST/CAT1.SCH2.ViewSupplier.txt| C:/tmp/cis61/MYTEST/CAT1.SCH2.ViewSupplier.txt|

SKIPPED| 2012-06-01 17:45:12.0030| MYTEST| getProductName| PROCEDURE| C:/tmp/cis51/MYTEST/getProductName.txt| C:/tmp/cis61/MYTEST/getProductName.txt|

SUCCESS| 2012-06-01 17:45:12.0030| MYTEST| SCH1.LookupProduct| PROCEDURE| C:/tmp/cis51/MYTEST/SCH1.LookupProduct.txt| C:/tmp/cis61/MYTEST/SCH1.LookupProduct.txt|

SUCCESS| 2012-06-01 17:45:12.0030| MYTEST| LookupProduct| PROCEDURE| C:/tmp/cis51/MYTEST/LookupProduct.txt| C:/tmp/cis61/MYTEST/LookupProduct.txt|

SKIPPED| 2012-06-01 17:45:12.0030| ProductWebService| soap11.ProductWebService.LookupProduct| WS| C:/tmp/cis51/ProductWebService/soap11.ProductWebService.LookupProduct.txt|C:/tmp/cis61/ProductWebService/soap11.ProductWebService.LookupProduct.txt|

SKIPPED| 2012-06-01 17:45:12.0030| ProductWebService| soap12.ProductWebService.LookupProduct| WS| C:/tmp/cis51/ProductWebService/soap12.ProductWebService.LookupProduct.txt|C:/tmp/cis61/ProductWebService/soap12.ProductWebService.LookupProduct.txt|

### *Compare Regression Logs*

Result from **compareRegressionLogs()**.

Output:

- Comparison log file output to the location identified by:

  <logFilePath>C:\\tmp\\pdtool\\MigrationCompareFiles2.log</logFilePath>

- Base directory location identified by:

  <logFilePath1>C:/tmp/pdtool/MigrationTest51.log</logFilePath1>

  <logFilePath2>C:/tmp/pdtool/MigrationTest61.log</logFilePath2

The following provides the key summary information for what will be output to app.log:

```
<Calling Action compareRegressionLogs>
<------------------------------------------------------------------------>
<Processing action "compare log files" for regression id: Test2.2>
<------------------------------------------------------------------------>
<---------------------------------------------------------->
```

```
<-------- Regression Log File Comparison Summary -------->
<-------------------------------------------------------->
<                                                        >
<Compare the query execution log files for two separate  >
<"Published Test" execution runs.  Compare each query    >
<result duration and apply a +- delta level to see if it >
<falls within the acceptable range.  Log File 1 is the   >
<baseline. Log File 2 is used to compare with log file 1.>
<                                                        >
<  Success = query duration met the following criteria:  >
<    performance match:     duration2 = duration1        >
<    performance improved:  duration2 < duration1        >
<    performance in acceptable range:                    >
<                   duration2 > duration1 and            >
<                   duration2-duration1 <= deltaDuration>
<                                                        >
<  Failure = query duration met the following criteria:  >
<    performance worsened: duration2 > duration1         >
<    performance out of acceptable range:                >
<                   duration2 > duration1 and            >
<                   duration2-duration1 > deltaDuration  >
<                                                        >
<  No Match = query could not be matched in either file. >
<                                                        >
<     Total Successful Comparisons: 14                   >
<        Total Failure Comparisons: 0                    >
<                                  ---------             >
<                Total Comparisons: 14                   >
<                                                        >
<           Total No Match Queries: 0                    >
<                                                        >
<   Regression comparsion duration:  000 00:00:00.0020   >
<                                                        >
<Review "log comparison" Summary:
D:/dev/PDTool_Test/regression/RegressionModule/MigrationCompareLogs2.log>
<-------------------------------------------------------->
<CisDeployTool::Successfully completed compareRegressionLogs.>
```

This is the output log for the method that compares two log files. The log file may be the log file from the “regression test” or the “performance test”. It compares the “duration” times within the two files and applies the duration delta which allows for an acceptable range. Result status can be SUCCESS, FAILURE, or NO MATCH.

- SUCCESS – the durations match or are within the accepted range based on duration delta.
- FAILURE – the durations are out of the accepted range.
- NO MATCH – could not find a matching query or the durations are not set properly.

An example log file is provided below which demonstrate what the compare regression log looks like:

Result| Message| BaselineDuration| CompareDuration| DiffDuration| DurationDelta| Database| ResourceURL| Type| Query1|Query2

SUCCESS| performance improved| 000 00:00:00.0093| 000 00:00:00.0016| -000 00:00:00.0077| 000 00:00:01.0000| MYTEST| MYTEST.ViewSales| QUERY| SELECT * FROM ViewSales WHERE ReorderLevel <= 3|Same as query1

SUCCESS| performance improved| 000 00:00:00.0032| 000 00:00:00.0031| -000 00:00:00.0001| 000 00:00:00.0005| MYTEST| MYTEST.SCH1.ViewOrder| QUERY| SELECT * FROM SCH1.ViewOrder|Same as query1

SUCCESS| performance improved| 000 00:00:00.0047| 000 00:00:00.0000| -000 00:00:00.0047| 000 00:00:01.0000| MYTEST| MYTEST.CAT1.SCH2.ViewSales| QUERY| SELECT * FROM CAT1.SCH2.ViewSales WHERE ProductName like 'Mega%'|Same as query1

SUCCESS| duration match| 000 00:00:00.0031| 000 00:00:00.0031| 000 00:00:00.0000| 000 00:00:01.0000| MYTEST| MYTEST.CAT1.SCH2.ViewSupplier| QUERY| SELECT * FROM CAT1.SCH2.ViewSupplier|Same as query1

SUCCESS| performance improved| 000 00:00:00.0047| 000 00:00:00.0000| -000 00:00:00.0047| 000 00:00:01.0000| MYTEST| MYTEST.getProductName| PROCEDURE| SELECT * FROM getProductName(1)|Same as query1

NO MATCH| query 2 entry not found| | | | 000 00:00:01.0000| MYTEST| MYTEST.CAT1.SCH1.LookupProduct| PROCEDURE| SELECT count(*) cnt FROM CAT1.SCH1.LookupProduct( 4 )|SELECT count(*) cnt FROM CAT1.SCH1.LookupProduct( 1 )

NO MATCH| query 2 entry not found| | | | 000 00:00:01.0000| MYTEST| MYTEST.CAT1.SCH2.LookupProduct| PROCEDURE| SELECT count(*) cnt FROM CAT1.SCH2.LookupProduct( 4 )|SELECT count(*) cnt FROM CAT1.SCH2.LookupProduct( 1 )

SUCCESS| performance improved| 000 00:00:00.0016| 000 00:00:00.0015| -000 00:00:00.0001| 000 00:00:01.0000| MYTEST| MYTEST.SCH1.LookupProduct| PROCEDURE| SELECT count(*) cnt FROM SCH1.LookupProduct( 1 )|Same as query1

SUCCESS| performance improved| 000 00:00:00.0031| 000 00:00:00.0000| -000 00:00:00.0031| 000 00:00:01.0000| MYTEST| MYTEST.LookupProduct| PROCEDURE| SELECT count(*) cnt FROM LookupProduct( 1 )|Same as query1

SUCCESS| within accepted range| 000 00:00:00.0015| 000 00:00:00.0016| 000 00:00:00.0001| 000 00:00:01.0000| testWebService00|testWebService00.services.testWebService00.testService.testPort.ws.testprocecho| WS| /services/testWebService00/testService/testPort.ws/testprocecho|Same as query1

### *Query Execution Performance Log File*

Result from **executePerformanceTest()**.

Output:

- Comparison log file output to the location identified by:

  <logFilePath>C:/tmp/pdtool/PerfTest1.log</logFilePath>

- Base directory location identified by:

  <baseDir>C:/tmp/pdtool/cis51</baseDir>

The following provides the key summary information for what will be output to app.log:

```
<Calling Action executePerformanceTest>
<-------------------------------------------------------------------->
<Processing action "execute performance test" for regression id: Test3.2>
<-------------------------------------------------------------------->
<------------------------------------------------------>
<--------- Regression Performance Test Summary ---------->
```

```
<-------------------------------------------------------->
<-------------------------------------------------------->
<Test Type: performance                                  >
<  Execute a full query from the query list.             >
<                                                        >
<Total Successful        Queries: 8                      >
<Total Successful     Procedures: 4                      >
<Total Successful   Web Services: 2                      >
<                                 ---------              >
<Total Successful -------> Tests: 14                     >
<                                                        >
<Total Skipped           Queries: 0                      >
<Total Skipped        Procedures: 1                      >
<Total Skipped      Web Services: 3                      >
<                                 ---------              >
<Total Skipped ----------> Tests: 4                      >
<                                                        >
<Total Failed            Queries: 0                      >
<Total Failed         Procedures: 0                      >
<Total Failed       Web Services: 0                      >
<                                 ---------              >
<Total Failed -----------> Tests: 0                      >
<                                                        >
<Total Combined ---------> Tests: 18                     >
<                                                        >
<      Performance Test duration:  000 00:02:20.0236     >
<                                                        >
<Review "perftest" Summary:
D:/dev/PDTool_Test/regression/RegressionModule/PerfTest2.log>
<-------------------------------------------------------->
<CisDeployTool::Successfully completed executePerformanceTest.>
```

This is the output log for the method that executes a performance test. Each line in the log file is driven by a query contained in the "performance test input file". Result status can be SKIPPED, SUCCESS, ERROR, HEADER, DETAIL, or TOTALS.

- SKIPPED – performance test skipped due to wrong type, run test turned off for this type, no database match or no resource match.
- SUCCESS – the performance test was successfully executed.
- ERROR – there was an error during processing. The Message will contain error.
- HEADER – there are two header rows.
    - Header1: this is the performance test settings header which provides
        - "Threads" – the number of threads to execute.
        - "Duration (s)" – the duration in seconds of each test.
        - "Print Sleep (s)" – how long to sleep in seconds for calculating printing statistics.
        - "Exec Sleep (s)" – how long to sleep in seconds between each query execution run.
    - Header2: this is the performance test execution header which provides
        - "Execs" – the number of executions.

- "Execs/sec" – the executions per second.
- "Rows/exec" – the rows per execution.
- "Latency (ms)" – the latency in milliseconds.
- "1st row (ms)" – the first row execution time.
- "Duration (ms)" – the duration of this detail line.

- DETAIL – these are the statistics that correspond to the execution header.
- TOTALS – these are the average detail statistics for this execution run.

An example log file is provided below which demonstrate what the query execution log looks like:

```
Result |ExecutionStartTime       |Duration          |Rows              |Database                |Query
|Type      |OutputFile                          |Message

SUCCESS |2012-06-08 10:28:43.0404  |000 00:00:10.0025   |9               |MYTEST                  |SELECT * FROM
ViewSales WHERE ReorderLevel <= 3                  |QUERY     |                                  |

 HEADER|Threads=1   |Duration (s)=10  |Print Sleep (s)=5  |Exec Sleep (s)=0  ||||

 HEADER|Execs      |Execs/sec   |Rows/exec   |Latency (ms) |1st row (ms) |Duration (ms) ||

 DETAIL|1537       |305.99      |9.00        |3.25         |3.15         |5023.00           ||

 DETAIL|1800       |359.85      |9.00        |2.77         |2.74         |5002.00           ||

 TOTALS|1668.50     |332.92      |9.00        |3.01         |2.94         |5012.50           ||

SKIPPED |2012-06-08 10:29:53.0529  |000 00:00:00.0000   |0               |MYTEST                  |SELECT * FROM
getProductName(1)                  |PROCEDURE  |                                  | ::Reason: type=PROCEDURE
runProcedures=true  databaseMatch=true  resourceMatch=false

SUCCESS |2012-06-08 10:29:53.0529  |000 00:00:10.0021   |1               |MYTEST                  |SELECT count(*) cnt
FROM CAT1.SCH1.LookupProduct( 3 )          |PROCEDURE  |                                  |

 HEADER|Threads=1   |Duration (s)=10  |Print Sleep (s)=5  |Exec Sleep (s)=0  ||||

 HEADER|Execs      |Execs/sec   |Rows/exec   |Latency (ms) |1st row (ms) |Duration (ms) ||

 DETAIL|5210       |1040.33     |1.00        |0.95         |0.95         |5008.00           ||

 DETAIL|5152       |1027.72     |1.00        |0.97         |0.97         |5013.00           ||

 TOTALS|5181.00     |1034.02     |1.00        |0.96         |0.96         |5010.50           ||

SUCCESS |2012-06-08 10:30:23.0591  |000 00:00:10.0034   |1               |MYTEST                  |SELECT count(*) cnt
FROM LookupProduct( 1 )                |PROCEDURE  |                                  |

 HEADER|Threads=1   |Duration (s)=10  |Print Sleep (s)=5  |Exec Sleep (s)=0  ||||

 HEADER|Execs      |Execs/sec   |Rows/exec   |Latency (ms) |1st row (ms) |Duration (ms) ||

 DETAIL|5220       |1046.51     |1.00        |0.95         |0.94         |4988.00           ||

 DETAIL|5387       |1076.10     |1.00        |0.92         |0.92         |5006.00           ||
```

TOTALS|5303.50 |1061.30 |1.00 |0.93 |0.93 |4997.00 ||

**SKIPPED |2012-06-08 10:30:33.0625 |000 00:00:00.0000 |0 |ProductWebService |/soap11/ProductWebService/LookupProduct |WS | | ::Reason: type=WS runWs=false databaseMatch=true resourceMatch=true**

### *Security Test Execution Log File*

Result from **executeSecurityTest()**.

Output:

- Execution log file output to the location identified by:

  <logFilePath>C:/tmp/pdtool/FunctionalTest.log</logFilePath>

- Base directory location identified by:

  <baseDir>c:/tmp/pdtool/functest</baseDir>

  Each individual query result file is saved to sub-directories based on the published data service DB name. The catalog+schema+view name are used for the file name. For example, given a published data service of MYDB.MyCatalog.MySchema.View1 the folder and filename created would be as follows:

  c:/tmp/pdtool/functest/MYDB/MyCatalog.MySchema.View1.txt

  The contents of the file depend on the query being executed. A simple functional test performs a count(*) on the view. For example, the contents for View1 might be:

  cnt

  35

The following provides the key summary information for what will be output to app.log:

```
<-------------------------------------------------------->
<---------- Regression Security Test Summary ------------>
<-------------------------------------------------------->
<Test Type: security  Security Plan Id=sp1                >
<  Execute a full query from the query list.              >
<                                                         >
<Total Successful        Queries: 3                       >
<Total Successful     Procedures: 3                       >
<Total Successful   Web Services: 6                       >
<                                 ---------               >
<Total Successful -------> Tests: 12                      >
<                                                         >
<                                                         >
<Total Failed            Queries: 0                       >
<Total Failed         Procedures: 0                       >
<Total Failed       Web Services: 0                       >
<                                 ---------               >
<Total Failed -----------> Tests: 0                       >
<                                                         >
```

```
<                                                          >
<Total Skipped           Queries: 0                        >
<Total Skipped        Procedures: 0                        >
<Total Skipped      Web Services: 0                        >
<                                 ---------                >
<Total Skipped ----------> Tests: 0                        >
<                                                          >
<                                                          >
<Total Error             Queries: 0                        >
<Total Error          Procedures: 0                        >
<Total Error        Web Services: 0                        >
<                                 ---------                >
<Total Error ------------> Tests: 0                        >
<                                                          >
<Total Combined ---------> Tests: 12                       >
<                                                          >
<         Security Test duration:  000 00:01:03.0635       >
<                                                          >
<Review "Security Test" Summary: C:/tmp/pdtool/SecurityTest.log>
<---------------------------------------------------------->
<-                                                        ->
<- Overall Security Test Rating: PASS                     ->
<-                                                        ->
<---------------------------------------------------------->
```

This is the output log for the method that executes the regression test. Each line in the log file is driven by a query contained in the “regression security XML”. Result status can be SKIPPED, PASS, FAIL, or ERROR.

- SKIPPED – regression test skipped due to wrong type, run test turned off for this type, no database match or no resource match.
- PASS – the security test execution status is PASS.
- FAIL – the security test status is FAIL based on an expected outcome of PASS.
- ERROR – there was an error during processing. The Message will contain error.

An example log file is provided below which demonstrate what the query execution log looks like:

# Processing action "executeSecurityTest" for security plan id: sp1

Result |Actual |Expected|Username |ExecutionStartTime |Duration |Rows |Database |Query |Type |OutputFile |Message

**PASS |PASS |PASS |user1** |2014-02-25 00:21:28.0223 |000 00:00:00.0391 |1 |TEST00 |SELECT count(1) cnt FROM CAT1.SCH1.customers |QUERY |C:/tmp/pdtool/cis61/TEST00/CAT1.SCH1.customers.txt|

**PASS |FAIL |FAIL |user1** |2014-02-25 00:21:28.0630 |000 00:00:00.0000 |0 |TEST00 |SELECT count(1) cnt FROM CAT2.SCH2.customers |QUERY |C:/tmp/pdtool/cis61/TEST00/CAT2.SCH2.customers.txt|executeQuery(): An exception occurred when executing the following query: "SELECT count(1) cnt FROM CAT2.SCH2.customers". Cause: User "user1/composite" has **insufficient privileges** to access "/services/databases/TEST00/CAT2"…

**PASS |PASS |PASS |user1** |2014-02-25 00:21:28.0645 |000 00:00:00.0000 |1 |TEST00 |SELECT count(1) cnt FROM Common.Common.customers |QUERY |C:/tmp/pdtool/cis61/TEST00/Common.Common.customers.txt|

**PASS |PASS |PASS |user1** |2014-02-25 00:21:28.0661 |000 00:00:00.0015 |1 |TEST00 |SELECT count(1) cnt FROM CAT1.SCH1.getCustomerById( 1 ) |PROCEDURE |C:/tmp/pdtool/cis61/TEST00/CAT1.SCH1.getCustomerById.txt|

**PASS |FAIL |FAIL |user1** |2014-02-25 00:21:28.0676 |000 00:00:00.0016 |0 |TEST00 |SELECT count(1) cnt FROM CAT2.SCH2.getCustomerById( 1 ) |PROCEDURE |C:/tmp/pdtool/cis61/TEST00/CAT2.SCH2.getCustomerById.txt|executeProcedure(): An exception occurred when executing the following query: "SELECT count(1) cnt FROM CAT2.SCH2.getCustomerById( 1 )". Cause: User "user1/composite" has **insufficient privileges** to access "/services/databases/TEST00/CAT2",

**PASS |PASS |PASS |user1** |2014-02-25 00:21:28.0692 |000 00:00:00.0016 |1 |TEST00 |SELECT count(1) cnt FROM Common.Common.getCustomerById( 1 ) |PROCEDURE |C:/tmp/pdtool/cis61/TEST00/Common.Common.getCustomerById.txt|

**PASS |PASS |PASS |user1** |2014-02-25 00:21:28.0723 |000 00:00:00.0031 |1 |CustomerWS |/soap11/TEST00/CAT1/SCH1/CustomerWS/customers |WS |C:/tmp/pdtool/cis61/CustomerWS/soap11.TEST00.CAT1.SCH1.CustomerWS.customers.txt|

**PASS |PASS |PASS |user1** |2014-02-25 00:21:28.0754 |000 00:00:00.0016 |1 |CustomerWS |/soap11/TEST00/CAT1/SCH1/CustomerWS/getCustomerById |WS |C:/tmp/pdtool/cis61/CustomerWS/soap11.TEST00.CAT1.SCH1.CustomerWS.getCustomerById.txt|

**PASS |FAIL |FAIL |user1** |2014-02-25 00:21:28.0770 |000 00:00:00.0016 |0 |CustomerWS |/soap11/TEST00/CAT2/SCH2/CustomerWS/customers |WS |C:/tmp/pdtool/cis61/CustomerWS/soap11.TEST00.CAT2.SCH2.CustomerWS.customers.txt|executeWs(): Server returned HTTP response code: 500 for URL: http://localhost:9420/soap11/TEST00/CAT2/SCH2/CustomerWS DETAILED_MESSAGE=[executeWs(): <?xml version='1.0' encoding='UTF-8'?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope"><faultcode>S:Server</faultcode><faultstring>User "user1/composite" has **insufficient privileges** to access "/services/webservices/TEST00/CAT2".

**PASS |FAIL |FAIL |user1** |2014-02-25 00:21:28.0801 |000 00:00:00.0016 |0 |CustomerWS |/soap11/TEST00/CAT2/SCH2/CustomerWS/getCustomerById |WS |C:/tmp/pdtool/cis61/CustomerWS/soap11.TEST00.CAT2.SCH2.CustomerWS.getCustomerById.txt|executeWs(): Server returned HTTP response code: 500 for URL: http://localhost:9420/soap11/TEST00/CAT2/SCH2/CustomerWS DETAILED_MESSAGE=[executeWs(): <?xml version='1.0' encoding='UTF-8'?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope"><faultcode>S:Server</faultcode><faultstring>User "user1/composite" has **insufficient privileges** to access "/services/webservices/TEST00/CAT2".

**PASS |PASS |PASS |user1** |2014-02-25 00:21:28.0817 |000 00:00:00.0031 |1 |CustomerWS |/soap11/TEST00/Common/Common/CustomerWS/customers |WS |C:/tmp/pdtool/cis61/CustomerWS/soap11.TEST00.Common.Common.CustomerWS.customers.txt|

**PASS |PASS |PASS |user1** |2014-02-25 00:21:28.0848 |000 00:00:00.0016 |1 |CustomerWS |/soap11/TEST00/Common/Common/CustomerWS/getCustomerById |WS |C:/tmp/pdtool/cis61/CustomerWS/soap11.TEST00.Common.Common.CustomerWS.getCustomerById.txt|

#

# Overall Security Rating=PASS Security Plan Id=sp1

# Total Error Tests=0

# Summary: Total Success=12 Total Skipped=0 Total Failed=0 Total Error=0

#

## EXCEPTIONS AND MESSAGES

An exception that is thrown by a procedure being called during execution is propagated to this calling client framework and printed in the logs.

Message 1: `The delimiter [,] was not found in the log file content.`

This message occurs when "compareLogs" is executed and the either one of the two log files does not contain the configured delimiter as described by <logDelimiter1> or <logDelimiter2>.

Resolution: Fix the RegressionModule.xml configuration for the regression id being executed to use the correct delimiter for the log file 1 and log file 2 being compared. It is possible for the two log files to have different delimiters.

Message 2: `The delimiter [,] resulted in an incorrect number of fields [1] parsed from the log file`

This message occurs when "compareLogs" is executed and the either one of the two log files does not contain the configured delimiter as described by <logDelimiter1> or <logDelimiter2>. In this case, the log file content contained the delimiter but the wrong number of fields were parsed.

Resolution: Fix the RegressionModule.xml configuration for the regression id being executed to use the correct delimiter for the log file 1 and log file 2 being compared. It is possible for the two log files to have different delimiters. It may be necessary to choose a different delimiter if the content also contains the delimiter as data.

Message 3: `The following required fields were not populated with log file data [result executionStartTime duration]`

This message occurs when "compareLogs" is executed and the either one of the two log files does not contain the configured delimiter as described by <logDelimiter1> or <logDelimiter2>. In this case, the parsing did not yield data for the required fields shown in the list.

Resolution: Fix the RegressionModule.xml configuration for the regression id being executed to use the correct delimiter for the log file 1 and log file 2 being compared. It is possible for the two log files to have different delimiters. This issue points to a potential delimiter issue.

Message 4: `Generate Regression Input File Error: For New Composite Web Services, the parameter style must be defined as WRAPPED when there are multiple input parameters.`

This message occurs when "`generateInputFile`" is executed the input parameter style is configured as BARE for the New Composite Web Service and it has multiple

input parameters. The setting is incorrect and the input parameter style must be set as WRAPPED.

<faultstring>User "user1/composite" has insufficient privileges to access "/services/webservices/TEST00/CAT2". User has no privileges for that resource. The required access is READ. [repository-1900300]</faultstring>

<faultcode>S:Server</faultcode><faultstring>There is no WSDL operation in SOAP message for the port binding "CustomerWS". [SOA-1900228]</faultstring>

## CONCLUSION

### *Concluding Remarks*

The PS Promotion and Deployment Tool is a set of pre-built modules intended to provide a turn-key experience for promoting CIS resources from one CIS instance to another. The user only requires system administration skills to operate and support. The code is transparent to operations engineers resulting in better supportability. It is easy for users to swap in different implementations of a module using the Spring framework and configuration files.

### How you can help!

Build a module and donate the code back to Composite Professional Services for the advancement of the "***PS Promotion and Deployment Tool***".

## ABOUT COMPOSITE SOFTWARE

Composite Software, Inc. ® is the only company that focuses solely on data virtualization.

Global organizations faced with disparate, complex data environments, including ten of the top 20 banks, six of the top ten pharmaceutical companies, four of the top five energy firms, major media and technology organizations as well as government agencies, have chosen Composite's proven data virtualization platform to fulfill critical information needs, faster with fewer resources.

Scaling from project to enterprise, Composite's middleware enables data federation, data warehouse extension, enterprise data sharing, real-time and cloud computing data integration.

Founded in 2002, Composite Software is a privately held, venture-funded corporation based in Silicon Valley. For more information, please visit www.composite.com.


Americas Headquarters
Cisco Systems, Inc.
San Jose, CA

Asia Pacific Headquarters
Cisco Systems (USA) Pte. Ltd.
Singapore

Europe Headquarters
Cisco Systems International BV Amsterdam,
The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at **www.cisco.com/go/offices.**





Printed in USA CXX-XXXXXX-XX 10/11