Telerik Test Studio- Best Practices

This document contains few best practices to make the tests run on different environments, make them maintainable and extensible and to make them more robust for frameworks.

**Himanshu Saraowgi**

**Ver 1.1**

**Himanshu.saraowgi@in.pega.com**

**10/22/2013**

Contents

[Version Control 2](#_Toc370313331)

[Scope and Objective of the document: 3](#_Toc370313332)

[Best Practices 3](#_Toc370313333)

[Folder Levels 3](#_Toc370313334)

[Folder Name 3](#_Toc370313335)

[Project Name 3](#_Toc370313336)

[Script Name 3](#_Toc370313337)

[Script Information on Top 4](#_Toc370313338)

[Script Organizing & Writing a good test 4](#_Toc370313339)

[Image Verification 4](#_Toc370313340)

[Coded Steps 5](#_Toc370313341)

[Testing Your Script 5](#_Toc370313342)

[Avoid element-recognition which includes indices, changeable text, or class (style) information. 5](#_Toc370313343)

["Wait for" Step 6](#_Toc370313344)

[CI Tool and Hudson 7](#_Toc370313345)

[Test List Structure 9](#_Toc370313346)

[Dynamic Test List 9](#_Toc370313347)

[Test Maintainability and Extensibility 10](#_Toc370313348)

[Test Modularization 10](#_Toc370313349)

[Data Driving (Optional) 11](#_Toc370313350)

[Excel Formulas 12](#_Toc370313351)

[Adding Comments and Annotations Generously 12](#_Toc370313352)

[Naming of Test, Objects and Steps 12](#_Toc370313353)

[URL Independence 12](#_Toc370313354)

[Pages 12](#_Toc370313355)

[Frames 13](#_Toc370313356)

[Popups 13](#_Toc370313357)

# Version Control

Version 1.0

Created By: Himanshu Saraowgi

Reviewed By: Maheedhar Srinivas

Dated:23 Oct, 2013

Version 1.1

Modified By: Himanshu Saraowgi

Modifications: Merging the document to best practices document available

Reviewer: Maheedhar Srinivas

Dated:24 Oct, 2013

# Scope and Objective of the document:

This document is intended to cover the below mentioned issues:

* Define best practices for creating tests using Telerik Test Studio.
* Structuring tests and test lists so as to run scheduled tests.
* Making tests URL independent to make them run on any environment.
* Strategies to modularize the tests to make them extensible and maintainable.
* Using the CI tool to configure Hudson, in order to run the Telerik tests on daily basis.

# Best Practices

## Folder Levels

Keep it to minimum (3 - 5 levels) and also be wary of 260 character limit on file paths set by the Windows API.

## Folder Name

Use lower case with no spaces.  Example:userinterface

## Project Name

Use Upper Camel Case with no spaces examples:  UIGallery, DesignerStudio

## Script Name

Script name should be meaningful and easy to understand the purpose of the script.  This should not contain spaces also.

 Examples:

 For scenario based testing \*\*These are just examples, more descriptive the better, but take some time to make it meaningful but not too long, in 25 characters, we should be able to describe what we are testing.  No need to have prefixes like verify or test.

* RunSimpleTaskProcess
* CreateSimpleFlow
* RefreshOnChange

Most of times we will need more specific name than in above example especially for functional level automation, it is preferred to use the naming convention as below, this way, if some one new can easily guess what those tests may be doing.

* Section\_DesignTab\_AddControls
* Section\_DesignTab\_IncludeSection
* Section\_DesignTab\_PropertyPanels

## Script Information on Top

Some basic information on top of each script should be provided.  This is helpful for both helper tests and the test cases.  Below screenshot can be used as examples and some templates will be available in SVN

* Make sure that PQD test case id is entered for all the tests, only exception is helper tests
* Purpose:  A brief description of what your test does, Example:  New Harness and Basic Edit is tested
* What does your test do:  This need not be step by step explanation but at least should give an idea what test does to serve the Purpose.  Example:  Creates a new Harness using context menu, saves, checkout, adds a new section, verifies taht is saved deletes checkout and deletes the Harness created

## Script Organizing & Writing a good test

* All the tests are independent of each other.  Meaning data created in test A will not be used in test B.  This allows test B to continue irrespective of test A failure
* Each script has a specific purpose and does only that.
* Be aware of under verification and over verification in a test script.
  + Under Verification example:  Purpose of the test is to delete a rule, If a test only conducts the delete action and stops verification at a confirmation message, this is under verification.  Test needs to confirm the deletion by trying to search or any other mechanism followed manually to confirm
  + Over Verification example:  Purpose of the test is to delete a rule,  If a test conducts the delete action, verifies confirmation message, confirm the deletion by trying to search or any other mechanism followed manually to confirm, do a db query to prove that rule is deleted, logout and log back in and verify that there is no rule. \*\*\*You may need to do these kind of things in special circumstances, not for a typical test.
* Use the test list feature to create a suite of tests.
* Reuse the code where possible \*

## Image Verification

 Even though Image Verification is available, use it only if absolutely needed and you are tolerant for a failure.  This is what Telerik support says about using Image Verification "Image verification is very brittle and should be used only as a last resort. If you feel it necessary use as small of an image as possible, like an icon or thumbnail. Trying to do an image verification of the entire page is pretty much guaranteed to fail every time, even on the same browser version but on different machines simply due to screen resolution differences"

## Coded Steps

 Use coded steps, when absolutely necessary only.  When used many coded steps, moving to next Telerik version will be slower because of this.  However, if coding is the best option for a particular test and you are comfortable in doing so, please write the test, **test thoroughly and get it reviewed by a peer. Also, avoid any coded step which uses object reference in it.**

## Testing Your Script

 This is very important.  As we all know, UI automation needs lot of stabilization before we call these as reliable.  Great amount of testing reduces the maintenance time.  Pairing could help in producing more reliable scripts.

* Run Multiple times
  + If there are any time stamp added to element attributes, this may identify
* Run against multiple PRPC installations
  + This will identify any hard coded attributes such as 'Src'
* Run in Multiple environments.  Example: on a test server, on individual machine, faster vs slower machine etc.
* Run in all the targeted browsers that the test is intended to run against.
* Run the testlist, by a scheduled Hudson job

## ****Avoid element-recognition which includes indices, changeable text, or class (style) information.****

 Recorded elements do not always have recognition characteristics which are generic, but unique.  When you're connected to a browser in Record mode, Telerik allows you to do something called Edit in Live (right-click on the element to get this).

This opens a modal dialog which shows you the element-recognition criteria and suggestions for other attributes of the element that could be used to identify it.  The dialog also provides a Validate button, which will check the chosen criteria and tell you if it identifies an element on screen.  I always try to use Edit in Live as soon as I've recorded an element, in order to tweak the criteria while the element I want is still visible on screen. I can examine the criteria, remove things which might change between runs, add generic attributes and other unique values which won’t' change and can be used to always find the right element.

 An example of how recording an element can get you into trouble is the PRPC Operator menu.  When you record it with Telerik, you get recognition criteria like "id = a:32" and "TextContent = My UserName".  This may work a second time, as long as you always log-in as the same user, and PRPC is always in the same state when you want to click this element.  If you Edit in Live this element, you can change out the "id = a:32" for "tagname = a", and replace the "TextContent = My UserName" with "title = Operator menu".  (After each change, use the Validate button to make sure you still get the same element.)  With these changes, you can log-in as any user, and the script will still be able to recognize the Operator menu link and use it.

## ****"Wait for" Step****

WaitFor is the preferred method of waiting for an element appear.  Always Wait For the element which is rendered in the last, that way you will be able to execute all the actions on the current page and will be able to navigate to the next page successfully.  Sometimes you may need to use 'Wait', one example is that Telerik not having the visibility of all the fields on a Modal dialog that you have connected to.

 When Telerik acts on an element, it usually assumes that that element is present in the browser DOM.  Actions like Click and Verify Visible try to act on the element immediately.  The "Wait For X" action has two steps:  1) Make sure the element is present in the DOM before 2) doing the actual action.  This is a critical difference:  If the element doesn't exist when the action takes place, Telerik throws an exception and the script ends.  For applications like PRPC, where the DOM changes often, it's important to make sure your element is present before acting on it.

You can change a script line's behavior by right-clicking it and selecting "Set As..." with the options Verify, Wait, or Extract.  Changing Verifies to Waits will slow down the script a little, but it makes the step more reliable.

**Use "wait for <test>"**

 The "Wait for <something>" Telerik step is more reliable than the "verify <test>" step, because "wait for" checks for the target element's existence before applying the test.  This is very important for products like PRPC, where the product DOM changes frequently and elements may be added or removed at any time.  Using "wait for" with a reasonable timeout (global 30 seconds, or step-specific of longer) can allow PRPC to update in time for Telerik to find the object first, then apply the test.  (Note that applying the test may also take some time; see else where in this doc about how the "is visible" test will check every parent node in an element's hierarchy for visibility.)  When you use "verify," Telerik assumes that the element already exists in the DOM and immediately tries to use it.  If it's not there, Telerik throws an exception and the test ends.

**After every action, use a "wait for" to let the UI catch up.**

 After performing an action which might change the DOM or redraw the browser, I always put a "wait for visible" for an element I expect to be there, so that I give PRPC some time to get itself organized while Telerik waits for the element to first exist, then be visible.  Sometimes, if different parts of the browser redraw more slowly than others, I will put in several "wait for visible" steps, in the order that the elements should appear, to give it even more time.

**Usage of “wait for element” with large timeout is needed whenever a new application page gets loaded. This is especially useful when we are using the application for first time.**

# CI Tool and Hudson

The CI tool is a front end to create a configuration file which would be used to configure Hudson. The configured Hudson server would then be used to run tasks like installation of PRPC, importing application bundle or running Telerik scripts. What this document deals, is configuring the tool to run the Telerik scripts and then running it from Hudson.

CI Configuration:

Under the Testing Tab, Select the Unit testing Tab under which the Telerik tab is to be selected. The advance view of the CI tool will provide you with two options:

* Run the tests from SCM.

For this option, check the checkbox “Checkout/Update tests from SCM” and give the SCM path.

* Run Test from machine.

For this option, you need to give the project path.

In the input box for “Project Containing Unit Tests”, enter the name of the project.

The other inputs are as follows:

Included Tests: Enter the name of the test lists that you want to run. \*.aiilist can be used in case all test lists are to be run.

Excluded tests: If the number of test to be run are more than the number of test list to exclude, we can select all test list in included tests and mention the excluded one’s in this option.

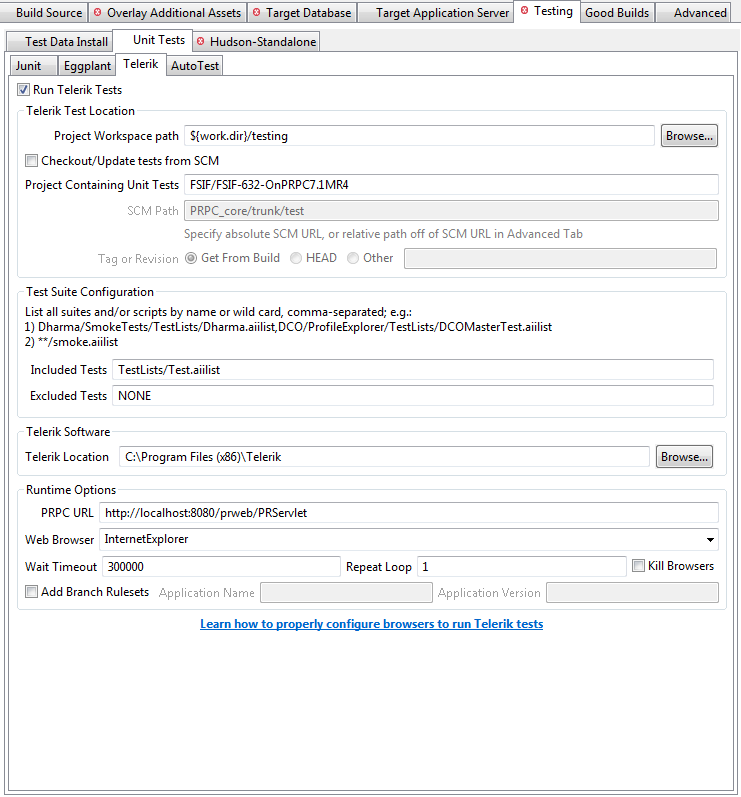
Telerik Location: Enter the path of the Telerik.exe file in the system. Usually it would be “C:\Program Files (x86)\Telerik”. CI tool works independent of Telerik version. Only issue being with the reporting of test for Telerik 2013 version. This can be solved with an additional dll file.

PRPC URL: Mention the URL of the application or if the application is on the local system, mention http://localhost:8080/prweb/PRServlet.

Web Browser: Select the web browser you are planning to run the tests on. These properties will over-ride the Telerik’s properties.

Wait Timeout: This is the time you want Telerik to wait for an object. There is not much harm in keeping this high if you are expecting the application to be slow.

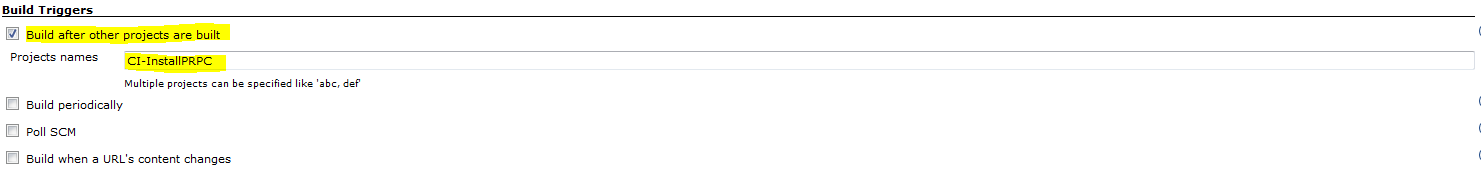
Below is a screenshot of Configurations for CI tool:

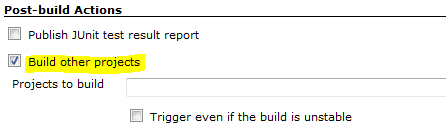


Running Hudson:

Once the CI tool has been configured, the saved file will be taken up by Hudson server and jobs would be created accordingly. If we want to run only the Telerik test without doing a fresh installation, we are required to remove the Upstream and Downstream projects for the Telerik project in Hudson.

This can be done from the configuration of the project when you open Hudson.





Once these settings are done, you can safely schedule or run the test from Hudson directly by clicking on  button next to the Telerik project on dashboard.

You can access the console by selecting the running project from Build Executor Status menu.

Once the tests are run, you will get the proper run summary. Also, you will have an option to import the results in Telerik format to your system.

# Test List Structure

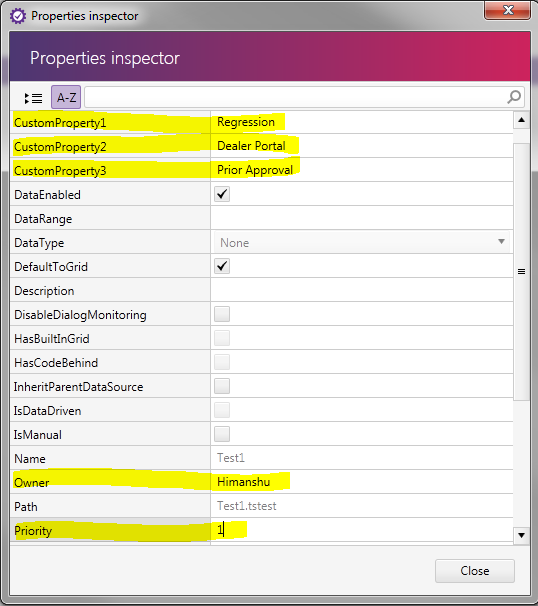
## Dynamic Test List

A dynamic test list reduces the effort of creating a new test list every time a test is added to the suite. Also, it would help the tester to create test list with similar kind of test in very short time as compared to normal test list. It works on the filtration procedure where you can filter your tests by different properties. These properties could be Name, Priority, Owner, Path or Custom Property.

This will help separate out the regression test cases from smoke test cases while executing.

Custom properties for Dynamic List:

Telerik gives an option of adding 3 custom properties to any test. These properties can then be used for 3 level filtration while creating dynamic test list. You can add a custom property to a test by right clicking on the test and selecting properties.



Using the above properties, I can configure a test list to run all ‘REGRESSION’ test on ‘DEALER PORTAL’ which uses ‘PRIOR APPROVAL’, are created by’ HIMANSHU’ and have priority ‘1’.

# Test Maintainability and Extensibility

## Test Modularization

Modularizing a test has multiple benefits. Improving extensibility and maintainability being a few of them. It also reduces the scripting time as you might not have to repeat the steps already coded and perfected. You can modularize the tests based on your application requirements. Some ways being:

* Modularizing page wise
* **Modularizing functionality wise**
* Modularizing validations wise

**In any case, it is suggested to modularize login and logout.**

**As by experience, Functionality Wise Modularization works best with PEGA applications.**

Page Wise Modularization:

In the scenarios where the complete scenarios consist of different screens repeated in different sequence we can make each screen as a different test and call them under a parent test.

If there are any changes on any of the screen that are to affect all the scenarios, the only action required would be to update a single test. Also, if a new functionality comes up in between, all that would be required is to create a test for that functionality and add it to the parent test.

**Functionality Wise Modularization:**

If a same functionality like creating a customer, selecting current application from menu or approving a request is being utilized multiple times in different scenarios, we can save them as separate test and call them when required.

If the flow is upgraded for any functionality, we might not have to update all the tests. Updating just one test would handle all.

Validation Wise Modularization:

In a scenario where similar kind of validations appear again and again on flows, we can create a separate test as validations. Whenever we want to use the validations, we can just call the test as a sub test. This is difficult to create as it requires you to create highly dynamic objects.

## Data Driving (Optional)

To make the tests highly maintainable, it is recommended to data drive as many steps as possible. One benefit being, we can execute the same test with different data which is a good testing practice. Other than that, if in future, if there are any data related changes in the flow, they can be handled by just changing the excel.

To add to the maintainability, we can data drive steps like verifications and wait to look for specific data as per an excel. Even if the data is changed, the test script will continue functioning as required by updating the excel.

## Excel Formulas

To make the job of updating the excel every time something changes, we can create one sheet in the excel which has the recurring data like URL, userid’s and passwords for different portals etc. Other sheets using these information could be referenced from this sheet using simple excel formulas.

It would be very beneficial if we expect this information to change any time as all we need to do is update this sheet and all other sheets follow suite.

## Adding Comments and Annotations Generously

Though seen as an overhead, comes in very handy when updating the scripts. Usually, there should be a comment at the start and end of every action like filling of a form on a screen. This improves the readability while updating.

## Naming of Test, Objects and Steps

The default name provided by Telerik would mostly not depict the sense of the subject under consideration. For example: Default name “WebTest” for a new test, “Click On Menu Item” for a step and “ItemMiddleCell” for an object might not clearly imply that it’s a Login test, or a step to click on logout from the menu or a hyperlink object pointing to new user.

It is recommended to use proper naming whenever working with any of the above so that it makes it easy to be updated at a later date.

# URL Independence

URL independence or the ability to run the test on any environment can be hindered by the following:

* Pages
* Frames
* Popups

## Pages

They hold the top position in the object hierarchy in Telerik test Studio. They can be recognized by using BaseURL, FullPath, Full Path and Query, Relative path, Title etc. These modes can be set for default under the project settings or can also be done after recording of objects.

When using anything other than relative path or Title, the url becomes a part of page recognition property. The two possible ways to remove this dependency are:

Relative Path

Select the compare mode as relative path and remove the dynamic part from the url mentioned in path. This method has issues like whenever you add a new object from the same page, it will create a new page to add it. You will have to manually set the same relative path for the page to merge it with the previous one.

Title

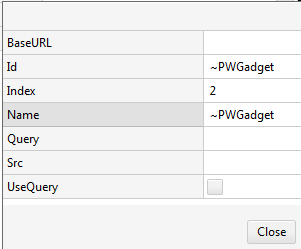
Mostly, the title of the page remains same even after the environment is changed. This compare mode can be safely used to make the test url independent. Also, every new object will automatically put itself under the right page.

You can also use partial title by putting a ‘~’ symbol at the starting of title and removing the dynamic part. But in this case, you will have to manually do this for pages whenever an object is added.

## Frames

Most of the pega applications would comprise of lot of frames. These frames have two properties which have url dependency in them. These are “BaseURL” and “Query”.

It is safe to remove these properties. The frame recognition will then work using ID, Name and Index in the same sequence of priority. Also, the ID and Name can be made partial by using ‘~’ symbol before it.



## Popups

When a new popup is opened or closed from the main application, it records the URL of the popup which usually have a part of parent URL in it. We can simply make it partial by selecting the “IsPartial” checkbox and removing the dynamic part from it.

# Reaching Out For Help:

* Post the issue on mesh pages:
  + <https://mesh.pega.com/community/pega/verticalsolutionframeworks/verticalsolutionframeworksqa>
  + <https://mesh.pega.com/groups/engineering-development-infrastructure/projects/automated-testing-at-pega>
  + <https://mesh.pega.com/community/pega/verticalsolutionframeworks/sustainingengineering/projects/verticals-frameworks-test-automation>
* Drop a mail to Vertical Solution Frameworks – QA <[VerticalSolutionFrameworks-QA@pega.com](mailto:VerticalSolutionFrameworks-QA@pega.com)> and anand.chikoti@in.pega.com