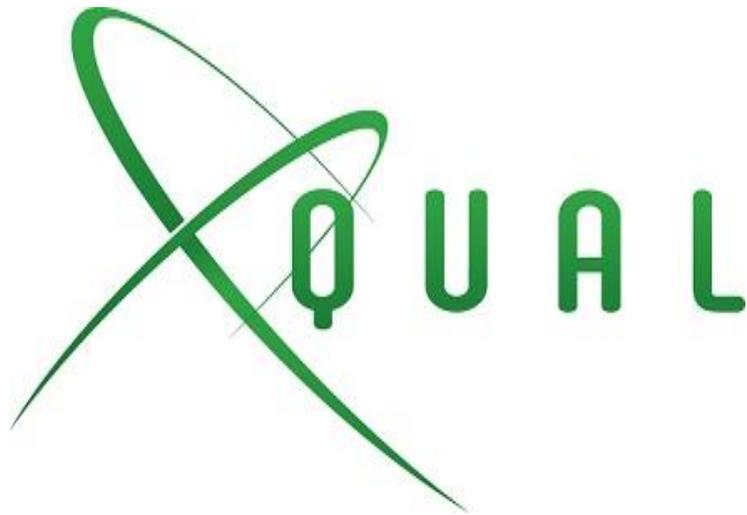


XSTUDIO LAUNCHERS SDK



01/12/2017

V3.2 (draft)

This document presents the necessary information de
customize or create a new launcher for XStudio.

Table des matières

Table des matières.....	1
OPEN-SOURCE LAUNCHERS SDK.....	2
What is the SDK made for?.....	2
Licensing.....	2
Support.....	2
What's in?.....	2
Windows, Linux.....	2
Getting the complete SDK.....	2
WHAT'S IN A LAUNCHER.....	3
Files used in launcher.....	3
DEVELOPER'S GUIDE.....	5
Preparation.....	5
Implementation.....	5
Implementing the main java class.....	5
5 methods to implement.....	6
initialize.....	6
preRun.....	6
run.....	7
postRun.....	7
terminate.....	7
The mandatory APIs.....	7
Build structured messages to be returned.....	7
Add attachments.....	8
The expert APIs.....	9
Identifiers.....	9
Helpers to parse objects passed.....	10
Campaign session content.....	10
Defects.....	10
Attachments.....	11
Test cases custom fields.....	11
SUT custom fields.....	11
Build returned messages including rich-text and embedding images generated on-the-fly.....	12
Writing the configuration template.....	12

Open-Source launchers SDK

What is the SDK made for?

The launchers that are delivered with XStudio are all customizable.

A launcher is a small java code that conforms to an interface defined by XQual. Hence it can be adapted in any way you need. You can create your own one if you wish to drive your home brewed test framework.

Licensing

Launchers are licensed under the GNU General Public License v2 (GPLv2).



Note: some Launchers are provided by the community of XStudio's users and may have a different license.

Support

XQual does not provide any support neither bear any guaranty on these launchers unless you purchased a valid commercial license.

If you develop your own launcher and would like it to be included in the distribution, just contact us at sales@xqual.com. We will review your code and make it part of our distribution **This is free**.

What's in?

The SDK contains all launchers including:

- source code (in the `src` folder),
- configuration file (in the `src` folder),
- binaries and libraries (in the `lib` folder),
- and build scripts (.bat) for Windows™ (in the `build` folder).
- After running the build script, the launcher binary will be located in the `bin` folder

Windows, Linux

The SDK is initially targeted to be used under MS Windows™. You can easily adapt it to be used under Linux.

For this you need to:

- Edit the files you want to modify with your favorite text editor and save them converting the end-Of-Line to your target OS.
- Create a “.sh” based on the `buildLauncher.bat` example

Getting the complete SDK

The SDK is included in XStudio, under `XStudio/sdk`, if you selected the Software Developer Kit package during the install.

Alternatively:

- you can also download it from our Download section.
- You are download the Windows 'fat' client and use this.

What's in a Launcher

Files used in launcher

A launcher is a package including:

- A **JAR** file (a zip containing some Java classes)
- An **XML** document describing the form template to be used to create an execution configuration for this launcher



The role of the launcher is to:

- **Read** the execution configuration parameters (provided by XStudio or XAgent)
- **Control** (setup, tear down, start, stop, etc.) the test scripts
- **Retrieve** the results, messages, log, traces, screenshots, etc. from the test execution
- **Store** all those information in XStudio's database so that all the metrics are calculated real-time

XStudio or XAgent (depending whether the tests are to be executed locally or remotely), starts the launcher and passes all the required information:

- All fields from Test, Test Case, Procedures,SUT and sessions
- Custom fields
- Test Attributes
- Test Case Parameters
- Configuration field value for the session

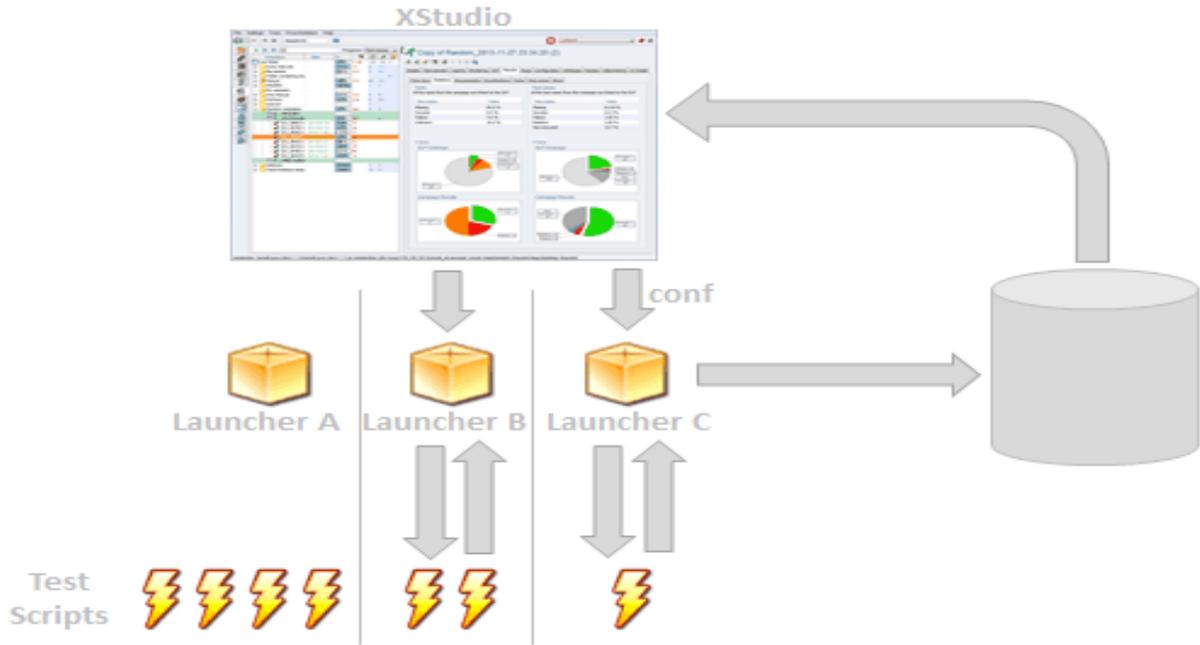
The launcher is then responsible to use (or not) all this information as per its needs.

Then once the tests are executed, the launcher gathers, interprets and stores the information back into XStudio's database. It can use:

- Return code from executable or scripts
- Any files.

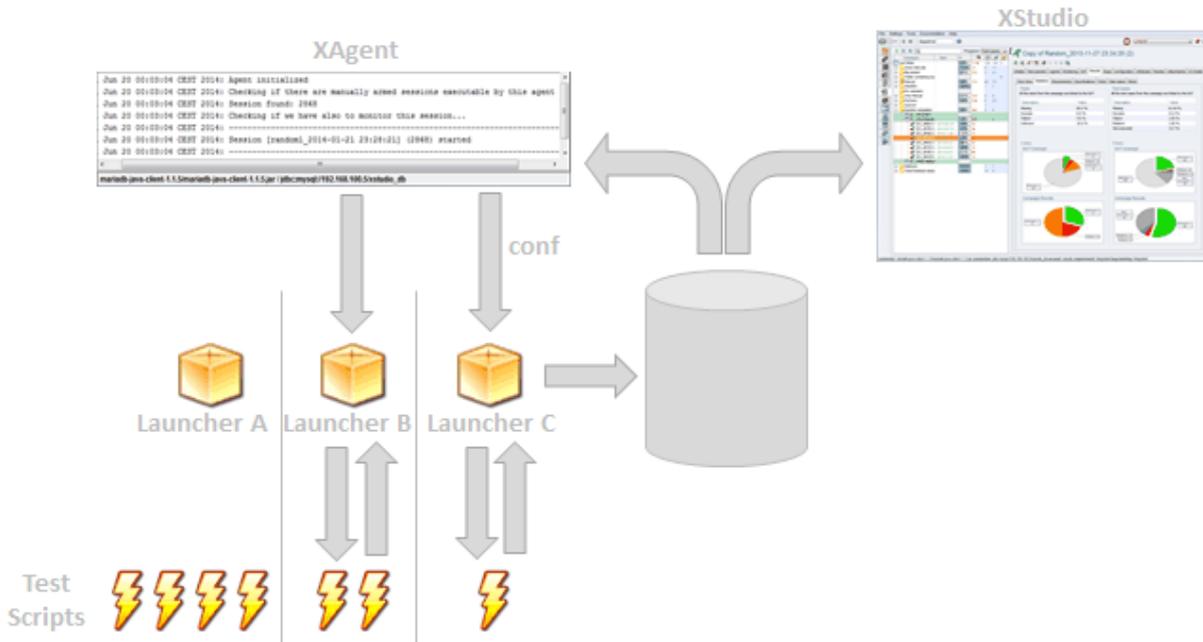
Note: launchers can scan existing test scripts that reside on the test servers. This enables to upload tests descriptions into XStudio's DB, saving the time to describe all existing automated tests and test cases. Not all launchers have this implemented out-of-the-box. But all can support it, provided that there's a clear signature for each test script in the test framework you use.

- When executing tests locally



- When executing tests on remote agents

The execution is controlled by XAgent and not XStudio but the process remains basically the same:



Developer's Guide

This part documents how to develop a launcher from scratch:

- How to prepare the environment to build a template launcher
- How to implement the launcher

Preparation

We describe the case where you code a new launcher from scratch. But the learnings are essentially the one if you just want to update an existing one.

The best approach is to duplicate one of the two following templates provided in the SDK:

- random launcher (returning random results)
- success launcher (returning only success results)

Let's assume we want to create an **"imaginary"** launcher. The preparation consists in doing the following:

- Copy the folder `src/com/xqual/launcher/random` to `src/com/xqual/launcher/imaginary`
- Rename `src/com/xqual/launcher/random.xml` to `src/com/xqual/launcher/imaginary.xml`
- Modify the package and the static variable `TRACE_HEADER` in the `CLauncherImpl.java` source file to match those of your imaginary launcher
- Make sure the `JAVA_HOME` variable in `BuildLauncher.bat` (or your `'sh'`) points to your JDK
- Run `buildLauncher.bat imaginary` (same for your `'sh'`)
- You should get a file `imaginary.jar` and its associated `configuration.xml` file in the `..\bin` folder
- XStudio or XAgent will expect all the launchers to be present in the `xstudio\launchers` folder
- To make the new launcher usable by XStudio or XAgent you need to add it into the `xstudio/tomcat/studio/launchers\launchers.xml`

Implementation

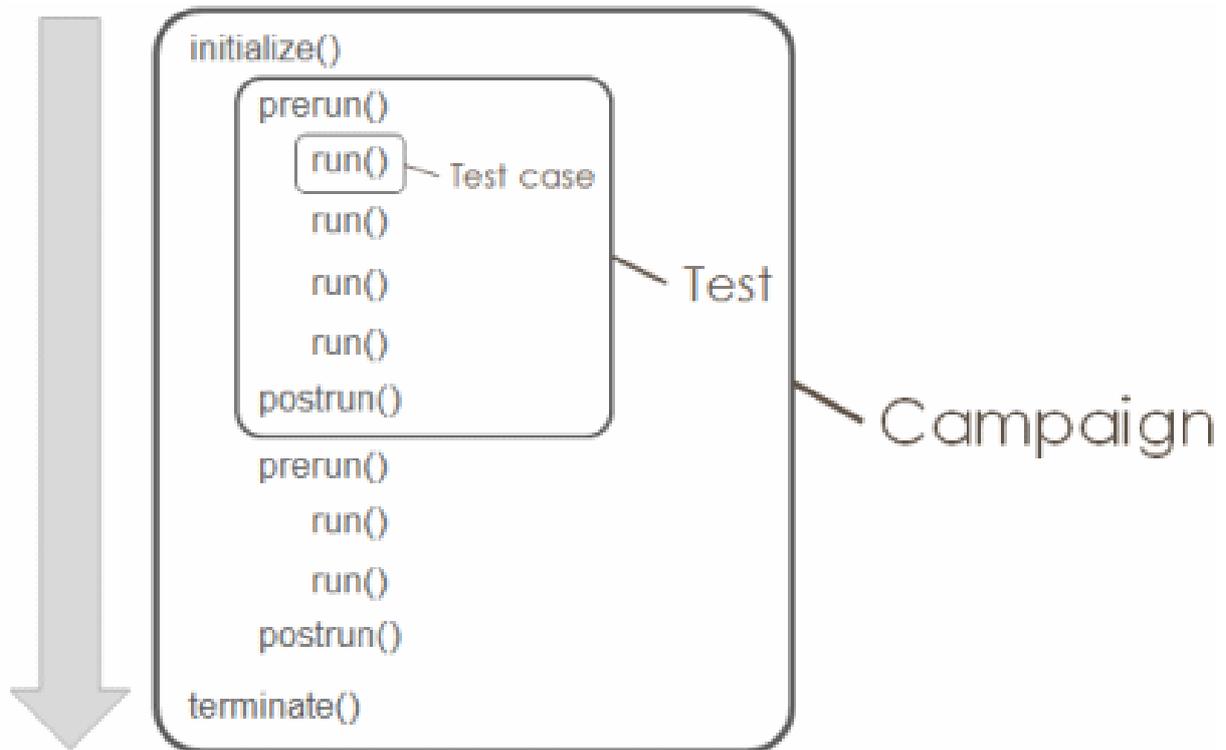
Developing a new launcher essentially consists in:

- Implementing the main java class (extending `CLauncher`)
- Writing the configuration template

Implementing the main java class

The main java class will have to extend the class `com.xqual.xagent.launcher.CLauncher`

There will be 5 methods to implement. These methods will be automatically called by XStudio or XAgent's execution scheduler:



5 methods to implement

initialize

```
CReturnStatus initialize(int sutId, String sutName, String sutVersion)
```

The initialize method is executed once at the instantiation of the launcher. You put there all initialization operations that needs to be done before everything else.

This method is called by XStudio or XAgent when the session starts. This is the right place to setup the test environment. Sometimes it is useful to know which SUT is going to be tested so all the SUT information are passed to the launcher at this point. This is also where we generally parse the configuration information with the `getXXXParamValue()` methods.

preRun

```
CReturnStatus preRun(int testId, String testPath, String testName,  
Vector<CTestAttribute> attributes, String additionalInfo)
```

preRun is executed before any testcase is run.

You put there all initialization operations that needs to be done before any testcase is executed.

It is called when a new test is about to start. All the attributes of the test are passed (as well as the additional info string) so that the launcher can eventually use them.

run

```
CReturnStatus run(int testId, String testPath, String testName, int
testcaseId, int testcaseIndex, String testcaseName, Vector<CTestcaseParam>
params, String additionalInfo)
```

`run` is the main entry point to execute a particular testcase. This is in this method that you will have to execute your test case.

The testcase must handle correctly all cleanup tasks (apart from operations previously executed in `preRun`

postRun

```
CReturnStatus postRun(int testId, String testPath, String testName)
```

`postRun` is executed after all the testcases have been executed.

You can put there all operations of cleanup necessary to bring back the SUT to its initial state. It's a good idea to rollback what were previously done in the `preRun` step.

This is called when a new test is about to be terminated: all test cases have been executed. We're leaving this test and are going to get to the next one.

terminate

```
CReturnStatus terminate()
```

The `terminate` method is executed once when the launcher is not anymore used.

You can put there all operations of termination that needs to be done after all the tests have been executed.

Basically, you do here all the cleanup.

The mandatory APIs

Build structured messages to be returned

The messages returned are include in the `CReturnStatus` object returned by the 5 methods to implement described above.

Most of the time you will build a `CReturnStatus` object using this kind of code:

```
Vector executionSteps = new Vector();

executionSteps.add(new CExecutionStep(CCalendarUtils.getCurrentTime(),
RESULT_INFO, "This is a message you want to see in the test case execution
details"));
```

```
executionSteps.add(new
CExecutionStep(CCalendarUtils.getCurrentTime(),RESULT_SUCCESS, "This is a
message informing about a successful step"));

executionSteps.add(new
CExecutionStep(CCalendarUtils.getCurrentTime(),RESULT_FAILURE, "This is a
message informing about a failed step"));

// in this case, RESULT_FAILURE is the global result given to the test case
execution:

return new CReturnStatus(RESULT_FAILURE, executionSteps);
```

for more details about the result codes, check out the SDK API reference.

Add attachments

```
void addSessionMessageAttachment(File file)
```

- Add an attachment file to the session being executed.

```
void addSessionMessageAttachments(Vector files)
```

- Add several attachment files to the session being executed.

```
void addSessionMessageAttachmentFolder(File file)
```

- Add all attachment files contained in a folder to the session being executed.

```
void addTestcaseMessageAttachment(File file)
```

- Replaces the deprecated method `addAttachment(File file)`
- Add an attachment file to the testcase being executed in the session.

```
void addTestcaseMessageAttachments(Vector files)
```

- Replaces the deprecated `addAttachments(Vector files)` method)
- Add several attachment files to the testcase being executed in the session.

```
void addTestcaseMessageAttachmentFolder(File file)
```

- Replaces the deprecated `addAttachmentFolder(File file)`method)
- Add all attachment files contained in a folder to the testcase being executed in the session.

Auto-extractable html zip files: In many places, you can attach some files to objects in XStudio.

When you attach a zip file and open it later, the default application for zip file (configurable in your OS's settings) is used to open it.

This is fine but in certain cases (i.e. html structure), it could be useful to unzip the file on the fly and open a specific entry point file in the zip.

You can do this now just by naming a specific way your zip file.

Let's imagine you attach a file named my_report.html.zip that contains several files (css, images, html etc.).

When the user will open this attachment from XStudio, XStudio will recognize the double extension (html + .zip) so will start by unzipping the file in a temporary folder then **XStudio** will open directly the my_report.html file that is included in the zip. Hence you can read an entire complex HTML reports (including images etc.) directly without having to unzip and open manually the right index file.

The expert APIs

In each of these 5 methods implementation you can use any API included in the SDK.

The included APIs include many libraries allowing to:

- execute processes synchronously or asynchronously,
- monitor a task using a timer,
- parse XML files,
- download files from shared folders/ftp server,
- add attachments to the test case execution in **XStudio**
- many utilities (conversion, file/date/time manipulation, etc.)

In addition, some methods allow accessing data related to the whole campaign being run. You should not need to use those methods but in particular cases this may be useful.

Identifiers

```
int getCampaignSessionId()
```

- Returns the id of the campaign session being executed.

```
int getInstanceId()
```

- Returns the instance id of the thread being executed. An instance is the campaign session/agent association.

Helpers to parse objects passed

- **Test's attributes**

```
Vector<String> getStringAttributeValues (Vector<CTestAttribute> attributes,  
String attributeName)
```

```
String getStringAttributeValue (Vector<CTestAttribute> attributes, String  
attributeName)
```

```
Vector<Integer> getIntegerAttributeValues (Vector<CTestAttribute>  
attributes, String attributeName)
```

```
Integer getIntegerAttributeValue (Vector<CTestAttribute> attributes, String  
attributeName)
```

```
boolean getBooleanAttributeValue (Vector<CTestAttribute> attributes, String  
attributeName)
```

- **Test case's params**

```
String getStringParamValue (String formName, String paramName, int index)
```

```
int getIntegerParamValue (String formName, String paramName, int index)
```

```
Double getDoubleParamValue (String formName, String paramName, int index)
```

```
boolean getBooleanParamValue (String formName, String paramName, int index)
```

Campaign session content

```
Vector<CTestExecution> getTestExecutionVector()
```

- Returns the complete list of tests and test cases included in the campaign session being executed.

```
Vector<Integer> getTestIdVector()
```

- Returns the complete list of test ids that are going to be executed in this campaign session.

```
Vector<Integer> getTestcaseIdVector()
```

- Returns the complete list of test case ids that are going to be executed in this campaign session.

Defects

```
Vector<CIIdAndNameDescriptor> getTestDefects (int testId)
```

- Returns the ids and names of all bugs attached to the test passed as argument.

Attachments

```
Vector<CIdAndNameDescriptor> getTestAttachments(int testId)
```

- Returns the ids and names of all files attached to the test passed as argument.
- The class `CIdAndNameDescriptor` includes 2 getter methods `getId()` and `getName()`.

```
Vector<CIdAndNameDescriptor> getTestcaseAttachments(int testcaseId)
```

- Returns the ids and names of all files attached to the test case passed as argument.

Test cases custom fields

```
Vector<CNameAndValueDescriptor> getTestcaseBooleanCustomFieldsHashtable(int testcaseId)
```

- Returns the names and values of all the boolean custom fields of the test case passed as argument.
- The class `CNameAndValueDescriptor` includes 2 getter methods `getName()` and `getValue()`.

```
Vector<CNameAndValueDescriptor> getTestcaseIntegerCustomFieldsHashtable(int testcaseId)
```

- Returns the names and values of all the integer custom fields of the test case passed as argument.

```
Vector<CNameAndValueDescriptor> getTestcaseStringCustomFieldsHashtable(int testcaseId)
```

- Returns the names and values of all the string custom fields of the test case passed as argument.

```
Vector<CNameAndValueDescriptor> getTestcaseHtmlCustomFieldsHashtable(int testcaseId)
```

- Returns the names and values of all the formatted string custom fields of the test case passed as argument.

```
Vector<CNameAndValueDescriptor> getTestcaseStringChoiceCustomFieldsHashtable(int testcaseId)
```

- Returns the names and values of all the drop-down custom fields of the test case passed as argument.

SUT custom fields

```
Vector<CNameAndValueDescriptor> getSutBooleanCustomFieldsVector()
```

- Returns the names and values of all the boolean custom fields of the SUT being tested.
- The class `CNameAndValueDescriptor` includes 2 getter methods `getName()` and `getValue()`.

```
Vector<CNameAndValueDescriptor> getSutIntegerCustomFieldsVector()
```

- Returns the names and values of all the integer custom fields of the SUT being tested.

```
Vector<CNameAndValueDescriptor> getSutStringCustomFieldsVector()
```

- Returns the names and values of all the string custom fields of the SUT being tested.

```
Vector<CNameAndValueDescriptor> getSutHtmlCustomFieldsVector()
```

- Returns the names and values of all the formatted string custom fields of the SUT being tested.

```
Vector<CNameAndValueDescriptor> getSutStringChoiceCustomFieldsVector()
```

- Returns the names and values of all the drop-down custom fields of the SUT being tested.

Build returned messages including rich-text and embedding images generated on-the-fly

If your test-automation framework generates some images automatically (i.e. screenshots), you can of course upload the images as attachment of the testcase execution using the standard

```
addTestcaseMessageAttachment.
```

You can also reference this image within your rich-text messages.

In your launcher, just do the following:

upload the images as usual:

```
addTestcaseMessageAttachment(new  
File("C:\\test_repository\\data\\screenshot.png"))
```

ii) Insert a message referencing the image using the "OTF_" prefix:

```
executionSteps.add(new CExecutionStep(CCalendarUtils.getCurrentTime(),  
RESULT_INFO, "This is a [b]nice/embedded[/b] image:  
[img]OTF_screenshot.png[/img]"));
```

Writing the configuration template

The configuration template file is an XML file that describes the form displayed to the user when he creates the execution configurations.

All the information you are going to describe here are going to be filled by the user and are going to be passed to the launcher at run time.

The user will be able to create some configuration once and reuse them later.

From the launcher you can get those values by calling the methods `getXXXParamValue()`