

Analysis

The CloudConnect software quality assurance (SQA) engineers at Symantec test server software that verifies and handles consumer product licensing and activations. While several quality assurance team members manually test the software, others write tests handled by an automation tool known as *AutoFLEX*. In order to emulate user experiences in different environments, the CloudConnect SQA engineers must understand how the tool works in collaboration with a startup program that installs the necessary dependencies and executes the tests. This program consists of several scripts written using Python and Python libraries part of the *Atrium* framework. As a result of the complexity of *Atrium* and *AutoFLEX*, the team has created this manual to train incoming CloudConnect quality assurance personnel.

A good user manual explains the usage of a product to a target audience and should be split into multiple sections covering the details of each aspect. It needs to be clear enough such that the lowest common denominator of the target audience can understand how the entire product works. In addition, high quality manuals include pictures, tables, and figures to address the needs of every type of learner. While the combined *AutoFLEX* and *Atrium* documentation fulfills each of these requirements, improvements can be made. First, only material relevant to the CloudConnect team is covered. This may satisfy current requirements, but if features are added or changed in the future, then the current content may not be exhaustive enough to provide assistance for team members. Another issue is that the present documentation is written in Markdown and HTML. Although these features make formatting and adding resources to the documentation simple, it is not as straightforward as creating a text-based document. If the formatting is considered too complicated for other SQA engineers, the manual may have to be rewritten, taking time away from more important tasks.

Even though *AutoFLEX* and *Atrium* have a steep learning curve, they are powerful products that make test automation manageable and convenient for software developers and SQA engineers on the CloudConnect team alike. Test automation can be approached in many different ways; however, at the time of writing, it still requires some human interaction. *AutoFLEX* helps users keep this interaction minimal by providing remote virtual machines to run tests on. Furthermore, it has a built-in database to store resources in a common location. Once the *Atrium* script is set up and the proper resources uploaded to the *AutoFLEX* database, users can set a trigger to run tests at any time of the day, even when the office is closed. Consequently, teams can focus on test writing and other high-priority tasks without having to use time executing automated tests manually. The *Atrium* framework works closely with *AutoFLEX* and standardizes how communication between the two pieces of software occurs. Having a standard method of communication between the project and automation tools prevents confusion that may arise between team members, possibly creating delays.

Because *AutoFLEX* and *Atrium* are specifically designed for Symantec teams, CloudConnect engineers have decided that using these together is the best option to improve automated testing efficiency. In addition, because *AutoFLEX* is a powerful tool, it has become an important asset for storing CloudConnect resources and executing tests remotely. The *Atrium* framework has also improved testing conditions. Because it is compatible with Python, a programming language commonly used by the SQA team, and integrates easily with *AutoFLEX*, the framework allows for considerable feature expansions in the future. For example, tests could be uploaded and executed individually without the need to re-upload the entire project every time, something that can be done with Python and *Atrium*, but not with *AutoTask*, the previous tool used to create testing projects.

Autoflex Navigation and Script Usage

Contents

- Autoflex Navigation and Script Usage
 - Contents
 - Introduction
 - A Brief Explanation of Atrium
 - A Look at the AutoFLEX tool
 - Automation Walkthrough
 - Resources
 - Projects
 - Scenarios
 - Administration Walkthrough
 - Channels
 - Triggers
 - Automation Scripts Explained
 - Important Code Snippets
 - Driver Script
 - The “Action” Scripts
 - Script and Attribute Explanation
 - Useful Commands and Tips

Introduction

As you may know, the **CloudConnect** team has recently moved to a new automation framework. Previously, the automation script handling the CC tests in **AutoFLEX** was written using the **AutoTask** framework and program. However, having expressed interest in future-proofing the automation process, the team has decided to switch to the **Python** scripting language and **Atrium** framework. This document will assist and train members of the **CloudConnect** team to navigate the **AutoFLEX** tool and further maintain the **Atrium** automation script.

A Brief Explanation of Atrium

Note: This section only covers terminology and directories relevant to the current **Atrium** script located in **AutoFLEX**. To view and learn about other **Atrium** resources and their uses, please visit the official [Atrium documentation page](#) or the `help` folder in the **Atrium** root directory.

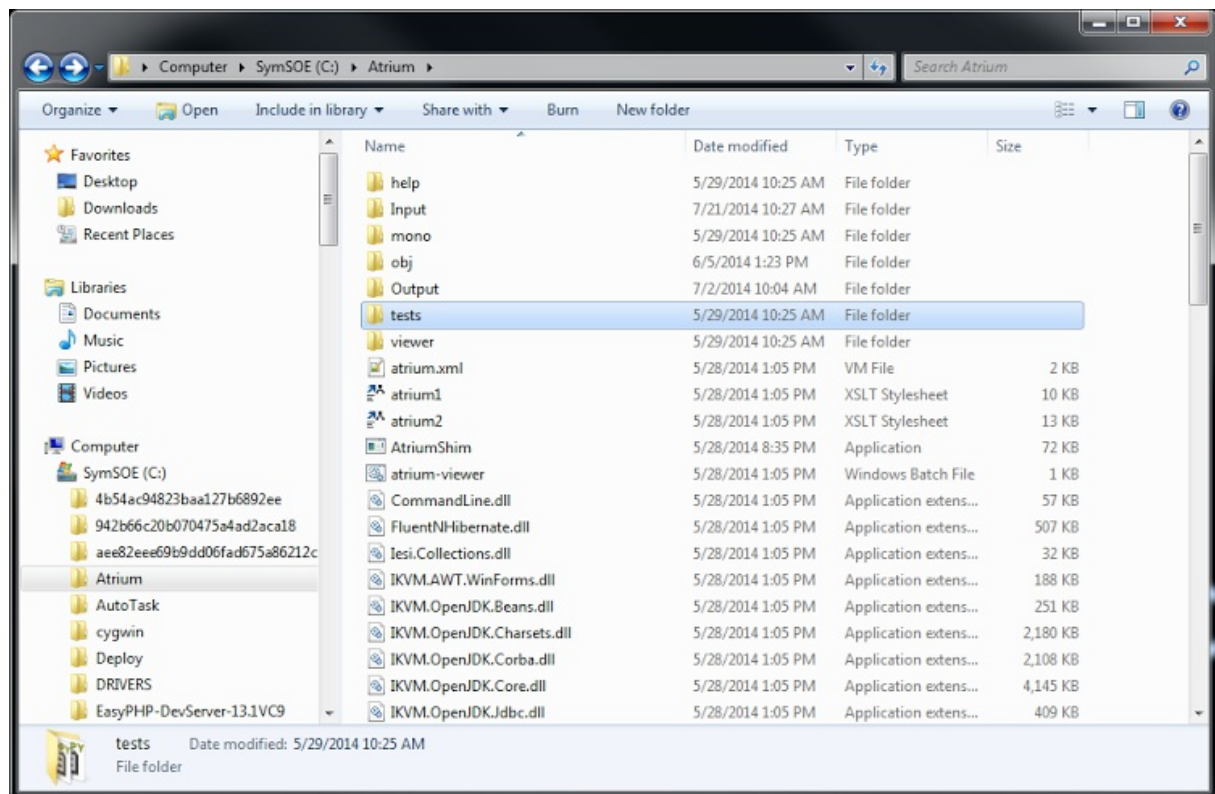
The **Atrium** framework works closely with the **AutoFLEX** automation tool.

Currently, the framework supports two languages, **Java** and **Python**, for authoring test scripts. To download the **Atrium** framework, see the box below.

How to Download Atrium

- Visit this [link](#)
- In the *Deployment Package* section, choose either link under Downloads depending on your operating system

After downloading the framework, you will be presented with a root directory containing the files that make up **Atrium**. The two important programs here are *AtriumShim* and *atrium-viewer*. *AtriumShim* is called from the command line when running the automation driver and also handles importing dependency frameworks (such as **IronPython**); therefore, the user does not have to worry about including dependencies in their script(s). *Atrium-viewer* is used to view the contents of the *.db* returned by **AutoFLEX**.



The Atrium root directory

Click on the `Input` folder. This is where the *Driver* script (see [Automation Scripts Explained](#) section) and user-provided resources are located. Click on the `_scripts` folder and then the `IronPython` folder. This is where user scripts are placed. It is important to note that if the scripts are not placed here, they will NOT be run.

A Look at the AutoFLEX tool

The **AutoFLEX** tool has been mentioned in this document many times, but what exactly is it? **AutoFLEX** is used to maintain and handle test scripts as well as create custom testing scenarios to run on any of the virtual OS disk images provided by the **AutoFLEX** team. In short, this is where the **CloudConnect** team stores its automation scripts and testing scenarios for multiple environments.

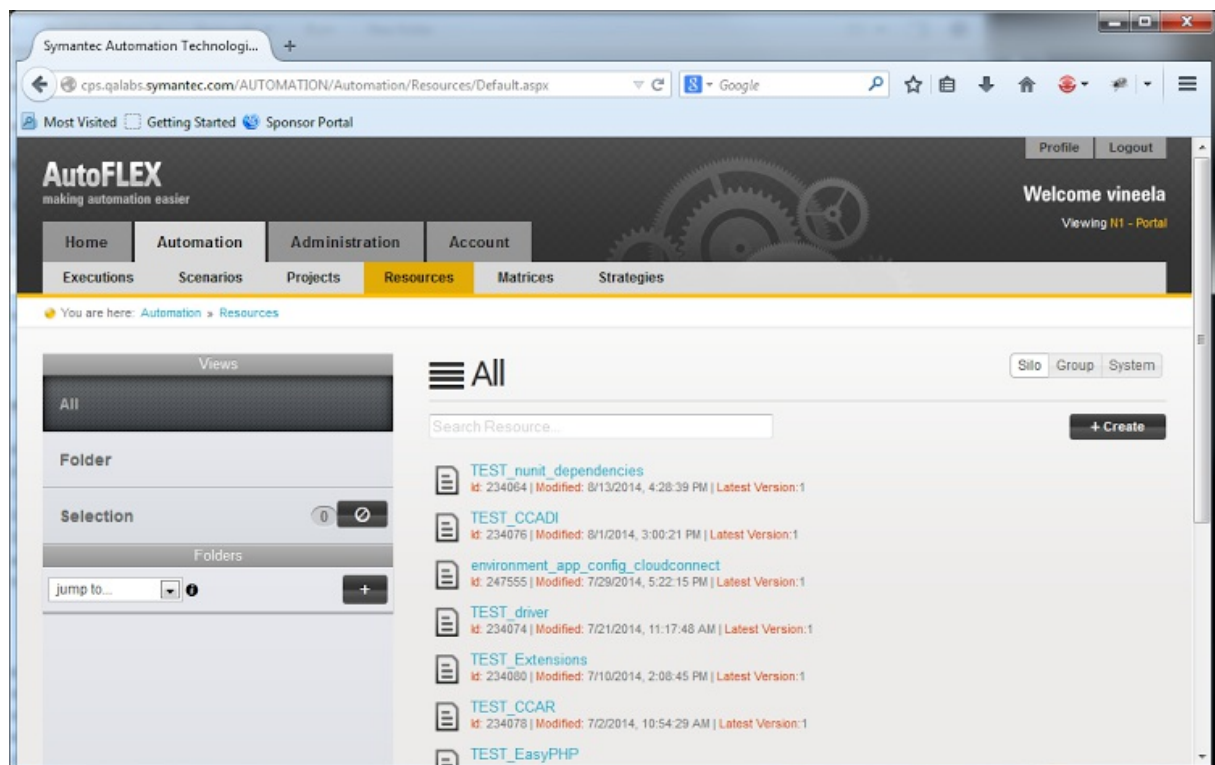
Automation Walkthrough

Using and navigating **AutoFLEX** can be somewhat complicated. Again, the content covered here does not necessarily explain every feature **AutoFLEX** provides. If your intent is to cover the tool in depth, please consult the [SymExchange](#) forums.

After logging in, four tabs appear at that top: *Home*, *Automation*, *Administration*, and *Account*. Click on the *Automation* tab. The page should now display a history of scenario executions. Below these tabs, there are multiple links: *Scenarios*, *Projects*, *Resources*, *Matrices*, and *Strategies*. The *Matrices* and *Strategies* links are not covered in this document.

Resources

Click on the *Resources* link. The *Resources* page displays all the files and directories uploaded to the **AutoFLEX** database. See the below box for uploading resources.



The Resources page in AutoFLEX

Uploading a Resource

When creating a new resource, there are multiple ways of uploading: *Versioned Resource*, *Container Resource*, and *Dynamic Resource*. The *Dynamic Resource* is not relevant to the **CloudConnect** team's scripts and

hence, will not be covered in this manual..

Versioned Resource

Use a *Versioned Resource* when uploading a single file.

Container Resource

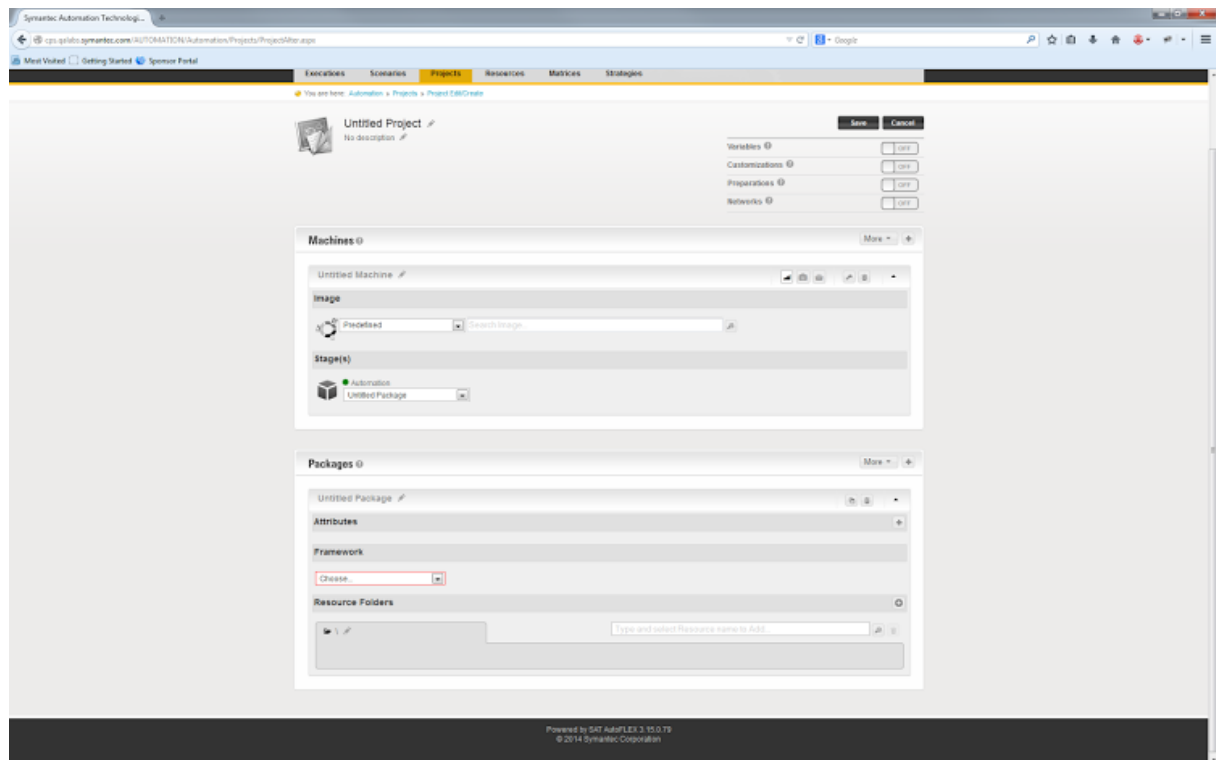
A *Container Resource* allows for a multiple-file upload.

Besides upload capacity, the two resource types are similar when filling out the settings. After filling out the relevant information, click **Add** and then **Upload**. Once the file is done uploading, a **Save** button will appear at the top of the resource window.

Note: It is EXTREMELY important to mark your driver script (this will be covered soon) as **Invokeable**. Otherwise, your script will not work.

Projects

Next, go to the *Projects* page. Projects differ from a scenario because they contain resources directly related to our scripts rather than the preferences for running the tests. Setting up a project is not relatively straightforward and can be confusing.



The Project creation page with most options expanded

When creating a project, there are several concepts to remember. Variables are NOT the same as attributes. Before we cover the differences, we must know what an **AutoFLEX** variable is. These variables are used only by the tool to communicate with the scenario preferences and are not seen by the automation scripts at all (the concept will be further examined in the [Scenarios](#) subsection).


On the other hand, attributes are elements that get passed into the automation scripts. This way, **AutoFLEX** and the scripts can communicate with each other, which is very helpful when setting file directory and environment names from **AutoFLEX**

The next project setup step asks what disk image(s) the scripts should run on. To add a machine, click the + button in the *Machines* section. This will reveal several settings, including the ability to name a machine, what disk image to use and what package (this will be covered shortly) it should correlate to. Although there are multiple columns to choose from in the drop-down menu under *Image*, the predefined setting should be chosen.

Tip: Returning Test Information

You may have noticed the group of three buttons on the right-hand side of the *Untitled Machine* box above. These buttons are actually very important, despite their lack of emphasis. The first button collects the test results and stores them in the report format that is set on the *Scenarios* page. The second button with the camera icon sets whether **AutoFLEX** should take screenshots of the testing process. Creating screenshots is very helpful when debugging your test script, and it is highly recommended that this option is enabled. Finally, you have the option of archiving the machine. In case bugs in the scripts are not visible through screenshots, you have the option of cloning the last state of the disk image and linking it to **VMWare Workstation**. Enabling this option is also highly recommended for debugging purposes.

The last step is to create a package. A package bundles all of the resources and attributes the automation scripts relies on. Choose **Atrium** from the drop-down menu under *Framework*. Finally, add the resource folders that your project will use. Select the magnifying glass by the input box and a screen will show the resources uploaded to the database earlier. Finally, click the `Save` button at the top.

Note: The resource folder root is NOT the same as the Atrium root folder when downloading the framework to a local computer. In this case, it is the `Input` folder mentioned in [Brief Explanation of Atrium](#). To add files to a subdirectory of `Input` (e.g. `_scripts`), add another resource box by clicking the + button and then edit the relative path by the  icon

Scenarios

The last link in the navigation bar is the *Scenarios* link. Scenarios handle the execution settings. While our project setup is handled on the *Projects* page, the *Scenario* page is where environment-specific settings, post-execution actions, and reporting is handled.

Scenario Terminology

While creating a scenario is similar to creating a project, there are several extra settings that need to be filled out.

- Mappings
 - Mappings bind a custom **AutoFLEX** variable to a value. In other words, this is the value that will be assigned to the the variable. For this project, mappings are strings representing the environment that is currently being tested.
- Channel
 - A channel handles notifications after a test has finished running.

Channels are set in the *Administration* tab under the *Channels* link (see [Administration Walkthrough](#))

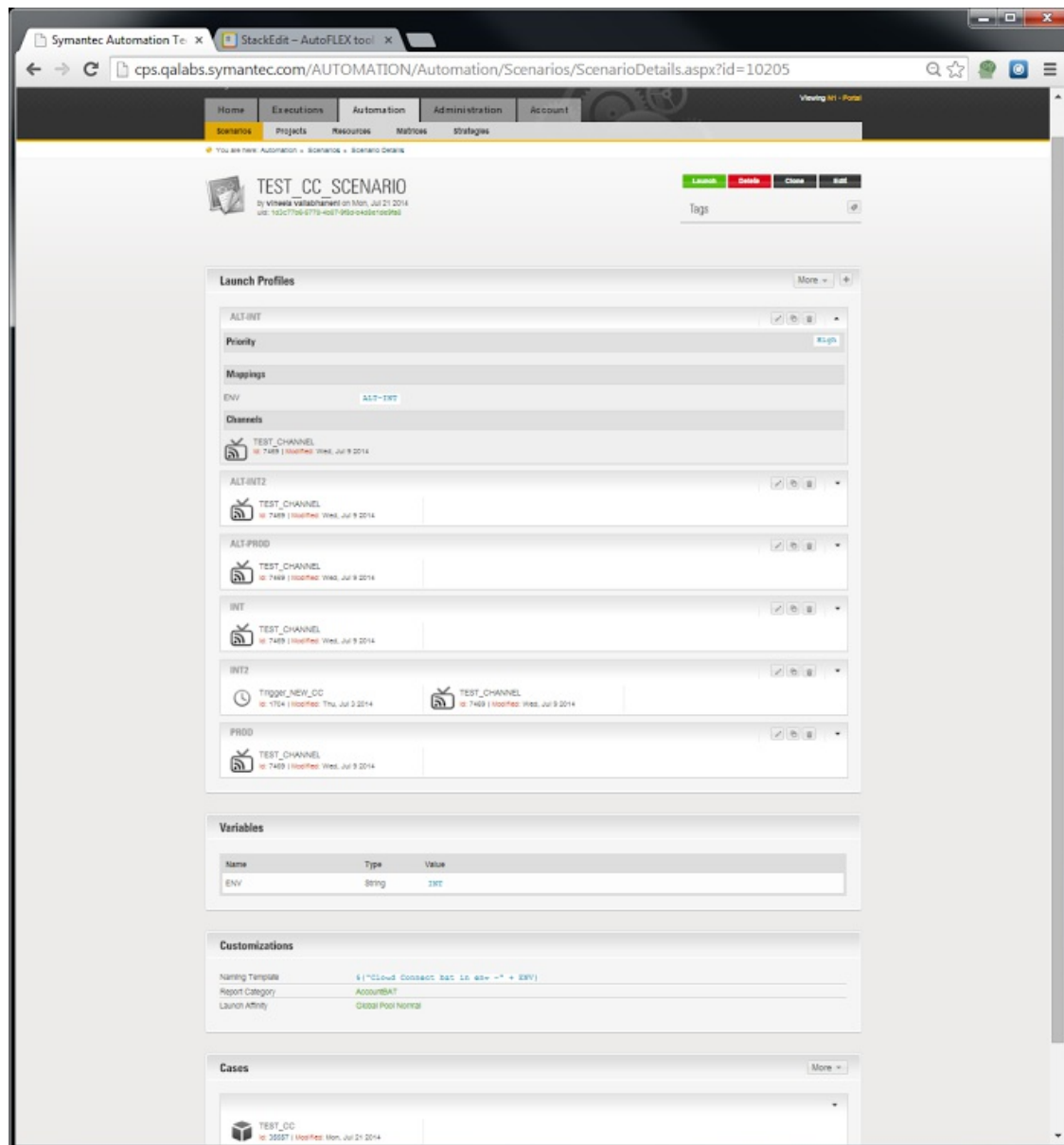
- Trigger
 - Triggers are set in the *Administration* tab under the *Triggers* link (see [Administration Walkthrough](#)) and control when the scenario should be executed
- Launch Profile
 - Launch Profiles contain the core customizations for a specific testing environment.

Launch profiles can only be set after a test has been created. First, turn on variables and customizations with the respective toggles in the top right corner. Next, press the + button next to *Cases*. We can ignore the *Strategy* and *Condition* tab and continue to add our project. Once the relevant project has been added, we can add a custom **AutoFLEX** variable. Set this to `ENV` and set the value to `INT`. Because our launch profiles will map the correct environment to the `ENV` variable, we just need to specify a default value. Next, we must set the customization preferences. The naming template refers to the name of our report in the email sent to the **CloudConnect** team. For the current test script, it is set to `${"Cloud Connect bat in env -" + ENV}`. Set the report category to `AccountBAT` and the launch affinity to `Global Pool Normal`. There are many different report categories, but this is the one that is used by the **CloudConnect** team.

The Flow of AutoFLEX variables

Now that all the variable settings have been explained, it is a good time to explain how the **AutoFLEX** variables are passed around. When a variable is first created in a scenario, it needs to be mapped to a variable in the project in order to pass the information to the project. As an example, let us take a the `ENV` variable in `TEST_CC_SCENARIO`. We first create the variable with a default value. When we map this variable to a value passed to us from the launch profile, the `ENV` value changes from the default value to the new value. Next, we create a mapping under the *Cases* box to a variable that has been created in projects (in this case, called `Environ`). When the **AutoFLEX** project runs, the variable in *Projects* receives the value of the variable in *Scenarios*. Optionally, we can create an attribute to hold the value of the *Project* variable to pass this value into our scripts.

Setting launch profiles is relatively simple. Save the scenario, add a launch profile. And `Priority` to `high`. The rest of the settings are straightforward and we will leave that up to the reader to handle.



Scenario page displaying the CloudConnect test suite project settings

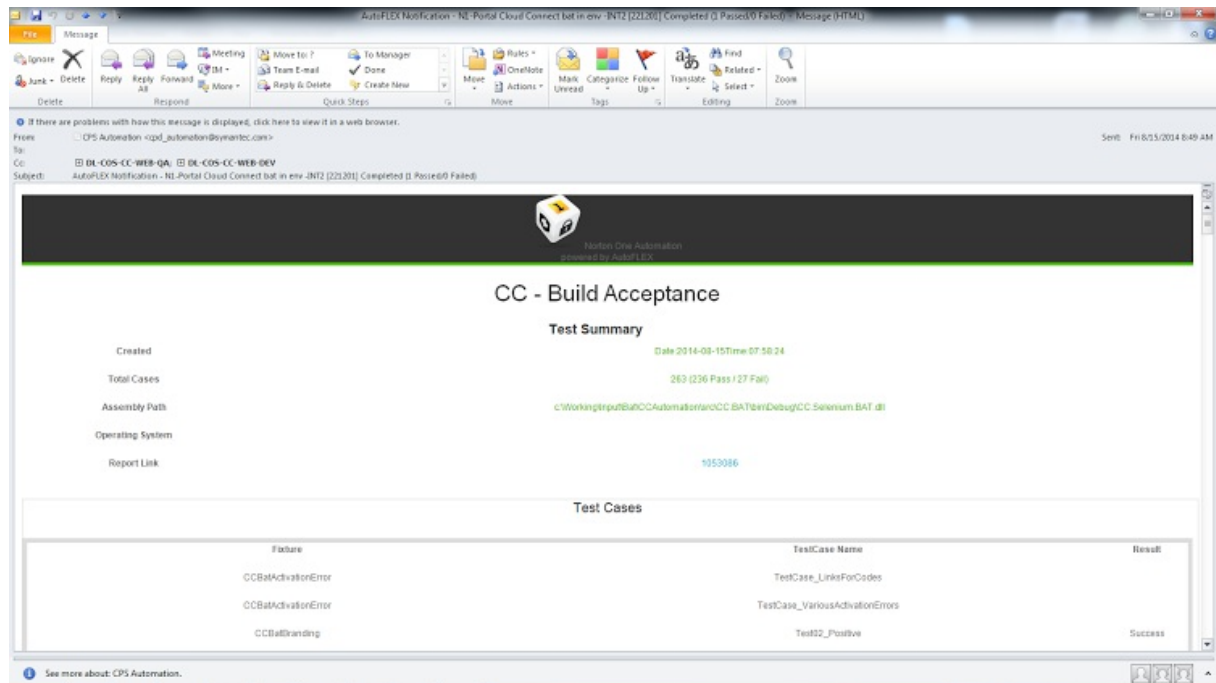
The *Administration* tab is covered in the next section. This is where triggers and channels are created. Finally, to execute a scenario, click the **Launch** button and choose the appropriate environment.

Administration Walkthrough

Now that we have gone over the important links under the *Automation* tab, we should also cover the *Channels* and *Triggers* links under the *Administration* tab. There are several more links to choose from, but they will not be covered as they are not used in current **CloudConnect** testing with the exception of the *Contacts* link (setting up contacts is rather obvious, however, and does not need further clarification).

Channels

As mentioned in the [Automation Walkthrough](#) subsection, channels handle notifications following the execution set completion. The *Channels* page shows a list of all the channels stored in the **AutoFLEX** database. Upon clicking on **Create**, you will notice that setting up a channel is very easy. Create a stream, set **Event** to **Execution Set Completion**, and set **Dispatcher** to **Email** (This is the only setting at the time of writing. Currently, the only way to dispatch notifications is through email), and set the **Format** to **N1 Transclusion Email Format**.



The N1 Transclusion Email Format

Once the channel has been created, we can search the **Contacts** list and add who will receive an email once the tests have finished running.

Triggers

Like the *Channels* page, the *Triggers* page displays a list of triggers in the **AutoFLEX** database. Add a trigger by clicking on the **Create** button. After naming the trigger, we have several options to specify the type of trigger under the **Add Trigger** drop-down menu. The only trigger we need to use here is the **Schedule Trigger**. The other options are not used.

Automation Scripts Explained

Before examining each automation script, we must become familiar with some of the library methods that *Atrium* provides. For documentation that covers methods beyond the scope of this manual, please see [this page](#) in the **Atrium** framework documentation.

Important Code Snippets

Most of the code below is taken directly out of the **Atrium** framework documentation. Further explanations will be provided for any non-obvious code snippets.

Driver Script

```
IDriverMetaData.Id
metadata.Id = "Script.Simple.HelloWorld.Test" # Unique ID
```

```
IDriverMetaData.Name
metadata.Name = "Simple Hello World Script" # Name of script
```

```
IDriverMetaData.Parameters.AddInput[str]
metadata.Parameters.AddInput[str]("test", "", True);

# This code creates an Atrium Variable (not to be confused with a Python variable)
#
# The first argument is the name of the variable
# The second argument gives a default value
#
# The third argument is important. If it is True, it will retrieve a
# variable with the name of THIS variable (first argument) from AutoFLEX
# if False, it will create a new Atrium variable with the default value set in
# the second argument
```

```
IDriverMetaData.Scripts.Add
metadata.Scripts.Add("Name_of_script", "ID_of_script", True)

# This method adds a relation between another script and the driver script
# First argument is the name of the script
# Second argument is the script identifier
# The third argument is set to True. What this argument does is not known
```

```
IDriverContext.CreateScriptExecutionBlock
hwScript = context.CreateScriptExecutionBlock(context.Scripts.Get("HWldScript"))
# Creates a script block for metadata.Scripts.Add("HWldScript", "Script.Simple.HelloWorld.Test", True)

# The code here creates a block of executable code and should be set to variables
# that represent other scripts that intend to be executed
#
# The parameter asks for a script object that was added to the driver's list of known scripts
# IDriverContext.Scripts.Get("name") asks for the script name set as an argument to the
# IDriverMetaData.Scripts.Add method
```

```
IDriverContext.CreateScriptExecutionBlock(...).SetInputParameterValue
hwScript.SetInputParameterValue("name_of_target_variable_in_script", driverContext.Parameters.GetInputValue[str]("test_path"))

# This method takes the name of an Atrium variable in another script as the first argument
# It will then get the value of an Atrium variable in the current driver script as a second argument
# Finally it will pass the variable in the driver script to another script (which we set as the execution block # prior to calling this method)
```

```
IDriverContext.ActionBlocks.CreateSequence
actionBlock = context.ActionBlocks.CreateSequence() # Returns an action block

#To add to the documentation, an action block represents the entire project to
be executed
#We proceed to queue "code blocks" next
```

```
IDriverContext.ActionBlocks.CreateReboot
rebootBlock = context.ActionBlocks.CreateReboot() # Creates a reboot block tha
t can be enqueued in an action block
```

```
IDriverContext.ActionBlocks.CreateSequence(...).Enqueue
context.ActionBlocks.CreateSequence().Enqueue(rebootBlock) # Enqueues a reboot
block
or actionBlock.Enqueue(rebootBlock) # Another way to enqueue

#Adding the executions blocks to a queue in the order they are to be run
```

```
IDriverContext.ScheduleActionBlock
context.ScheduleActionBlock(actionBlock) # Schedules an action block for exec
ution

#This method signals to the driver to run our scripts in the ordered they are
queued
```

The “Action” Scripts

The **Atrium** framework library methods used in the non-driver scripts are very similar to the driver script code. Please refer to the [Driver Script](#) subsection to see how this code works.

However, there are two predefined Atrium variables that are not used in the driver:

```
IScriptContext.Environment.InputDirectory
dir = context.Environment.InputDirectory # dir = C:\Working\Input if base fold
er is C:\Working
```

```
IScriptContext.Environment.OutputDirectory
dir = context.Environment.OutputDirectory # dir = C:\Working\Output if base fo
lder is C:\Working
```

Note: Although it appears that `Working` is the root directory here,
REMEMBER THAT THIS IS NOT THE CASE IN **AutoFLEX**

Script and Attribute Explanation

This subsection will cover the three automation scripts used to handle the suite of tests used in **CloudConnect** testing. At the time of this writing, there are three automation scripts: *CCADI.py*, *CCAR.py*, and *CCADriver.py*. *CCADI* stands for “**CloudConnect** Automation Download and Install” and does exactly what it sounds like. This first script installs all the necessary test suite dependencies on the virtual machine. The *CCAR* (**CloudConnect** Automation Runner) script prepares **AutoFLEX** reporting and executes the test suite using **NUnit**.

For convenience, the three scripts are well commented to explain what each block of code does. To prevent redundancy, the code will not be posted here. Python

documentation can be viewed [here](#) in case the comments are not entirely clear.

Although the names of the attributes explain what information they hold, the box below will cover the attributes used in each script.

List of Attributes Used in the CloudConnect Automation Scripts

- `firefoxPath`
 - Refers to the relative path location of the FireFox setup file
- `dotnetCommandPath`
 - Refers to the location of the .net 4 framework command line installer
- `nunitPath`
 - Refers to the location of the NUnit program zip file
- `extensionsZipPath`
 - The zip file containing any FireFox extensions
- `projectZipPath`
 - Contains the relative location of the test suite and project dependency zip file
- `nunitConsolePath`
 - Refers to the location of *nunit-console.exe*
- `seleniumBatPath`
 - Refers to the location of the Selenium dynamically linked library file
- `easyphpZipPath`
 - Contains the relative location of the **EasyPHP** program zip file
- `appConfigRelativePath`
 - This attribute refers to the relative path of the directory containing the custom `app.config` files (NOT the actual files themselves!)

Useful Commands and Tips

The final section in this manual will cover some useful commands and tips when creating and debugging future test scripts.

Running AtriumShim to execute the driver locally

Sometimes, testing locally is more convenient than testing on a virtual machine that the **AutoFLEX** tool provides. To run your driver locally, run the following commands:

```
cd [insert Atrium root folder absolute path here]
AtriumShim -s [driver path]
```

Linking Base Images and an Archived Machine Image

When you want to link a base image to an archived machine image, first download the *VMLink* image under the *Media* box in the test summary page (this can be found by clicking on the executed test under the *Executions* tab). Once you have downloaded the *VMLink* program, run the following commands:

```
cd [path to VMLink program]
VMLink [clone image path to *.vmx] [base image path to *.vmx]
```

The base image is the original Windows image, while the clone image refers to the archived machine image.

Tip: This can only be run ONCE with the base and clone images. Attempting to run the command again will result in an error. To prevent having to re-download the images every time, make copies of your base and archived machine images!

Local Testing with Attributes

It is also possible to pass in attributes to the automation scripts locally. To do so, create a txt file with your mock attributes in the form of key=value. DO NOT use quotation marks around the value like in a C string. Then call the following commands:

```
cd [insert Atrium root folder absolute path here]
AtriumShim.exe -s C:\Working\Input\MyDriver.py --test-inputs-file [path to the txt file]
```