

# Write C++ Tests

Programmer's guide to creating engine-level Automation Tests.



## Automation Testing

**Automation Testing** is the lowest level of automated testing, and is best suited for low-level tests of the core functionality of **Unreal Engine**. This system exists outside of the `UObject` ecosystem, so it is not visible to **Blueprints** or the Engine's **Reflection System**. These tests are built in code and can be run from the **Console Command Line**, either in the **Editor** or with **Command-Line Parameters** from your operating system. Automation Tests can be broken into two types: **Simple** and **Complex**. Both types are implemented as derived classes based on `FAutomationTestBase`.

# Creating a New Automation Test

Automation Tests are declared by macros, and implemented by overriding virtual functions from the `FAutomationTestBase` class. Simple Tests are declared using the `IMPLEMENT_SIMPLE_AUTOMATION_TEST` macro, while Complex Tests require the `IMPLEMENT_COMPLEX_AUTOMATION_TEST` macro. Both macros feature the same three parameters, in the following order:

Parameter	Description
<code>TClass</code>	The desired class name of the test. The macro will create a class of this name, e.g. <code>FPlaceholderTest</code> , in which your test can be implemented.
<code>PrettyName</code>	A string specifying the hierarchical test name that will appear in the UI. For example, "TestGroup.TestSubgroup.Placeholder Test" in our minimal example (below).
<code>TFlags</code>	A combination of <code>EAutomationTestFlags</code> values, used for specifying Automation Test requirements and behaviors. See <a href="#">EAutomationTestFlags</a> for details.

Once the new Automation Test class has been declared by one of the two macros, its functionality can be implemented. The following functions must be written:

Function	Parameter	Description
<code>RunTest</code>		This function performs the actual test, returning <code>true</code> to indicate that the test passed, or <code>false</code> to indicate that it failed.
	<code>Parameters</code>	This parameter can be parsed or passed through to other functions as appropriate for the specific Functional Test.

Function	Parameter	Description
--		
<code>GetTests</code>		This function must be overridden for Complex Tests only; Simple Tests have an auto-generated version of this function built into their declaration macro.
	<code>OutBeautifiedNames</code>	This array of strings must be populated with the UI-visible <code>PrettyName</code> of each individual Test.
	<code>OutTestCommands</code>	This array of strings is expected to be parallel to <code>OutBeautifiedNames</code> and must be populated with the <code>Parameters</code> to be passed into <code>RunTest</code> .

Source File Locations

The current convention is to put all Automation Tests into the `Private\Tests` directory within the relevant module. When an Automation Test matches one-to-one with a particular class, please name the test file `[ClassFilename]Test.cpp`, e.g. a test that applies only to `FText` would be written in `TextTest.cpp`.

Minimal Example

The smallest and simplest test possible would be a Simple Test that automatically succeeds (or fails). Building a test like this can be useful as a first step to ensuring that your testing setup is correct before building more meaningful tests.

```
1 IMPLEMENT_SIMPLE_AUTOMATION_TEST(FPlaceholderTest, "TestGroup.TestSubgroup.Placeholder Test", EAutomationTestFlags::EditorContext |
  EAutomationTestFlags::EngineFilter)
2
```

```
3 bool FPlaceholderTest::RunTest(const FString& Parameters)
4 {
5     // Make the test pass by returning true, or fail by returning false.
6     return true;
7 }
```

 Copy full snippet

## Simple Tests

**Simple Tests** are used to describe single atomic tests, and are useful as unit or feature tests. For example, simple tests can be used to test if the current map loads in Play In Editor or whether text wrapping is working properly in Slate. The following example tests the functionality of the `SetRes` command:

```
1 IMPLEMENT_SIMPLE_AUTOMATION_TEST( FSetResTest, "Windows.SetResolution", ATF_Game )
2
3 bool FSetResTest::RunTest(const FString& Parameters)
4 {
5     FString MapName = TEXT("AutomationTest");
6     FEngineAutomationTestUtilities::LoadMap(MapName);
7
8     int32 ResX = GSystemSettings.ResX;
9     int32 ResY = GSystemSettings.ResY;
10    FString RestoreResolutionString = FString::Printf(TEXT("setres %dx%d"), ResX, ResY);
11
12    ADD_LATENT_AUTOMATION_COMMAND(FEngineWaitLatentCommand(2.0f));
13    ADD_LATENT_AUTOMATION_COMMAND(FExecStringLatentCommand(TEXT("setres 640x480")));
14    ADD_LATENT_AUTOMATION_COMMAND(FEngineWaitLatentCommand(2.0f));
15    ADD_LATENT_AUTOMATION_COMMAND(FExecStringLatentCommand(RestoreResolutionString));
16
```

```
17 return true;
18 }
```

 Copy full snippet

## Complex Tests

**Complex Tests** are used to run the same code on a number of inputs. These are generally content stress tests. For instance, loading all maps or compiling all Blueprints would be good fits for complex automation tests. Note that both the `RunTest` and `GetTests` functions must be overridden. The following example tests the loading all of a project's maps:

```
1 IMPLEMENT_COMPLEX_AUTOMATION_TEST(FLoadAllMapsInGameTest, "Maps.LoadAllInGame", ATF_Game)
2
3 void FLoadAllMapsInGameTest::GetTests(TArray<FString>& OutBeautifiedNames, TArray<FString>& OutTestCommands) const
4 {
5     FEngineAutomationTestUtilities Utils;
6     TArray<FString> FileList;
7     FileList = GPackageFileCache->GetPackageFileList();
8
9     // Iterate over all files, adding the ones with the map extension..
10    for( int32 FileIndex=0; FileIndex< FileList.Num(); FileIndex++ )
11    {
12        const FString& Filename = FileList[FileIndex];
13
14        // Disregard filenames that don't have the map extension if we're in MAPONLY mode.
15        if ( FPaths::GetExtension(Filename, true) == FPackageName::GetMapPackageExtension() )
16        {
17            if (!Utils.ShouldExcludeDueToPath(Filename))
18            {
```

```
19 OutBeautifiedNames.Add(FPaths::GetBaseFilename(Filename));
20 OutTestCommands.Add(Filename);
21 }
22 }
23 }
24 }
25
26 bool FLoadAllMapsInGameTest::RunTest(const FString& Parameters)
27 {
28     FString MapName = Parameters;
29
30     FEngineAutomationTestUtilities::LoadMap(MapName);
31     ADD_LATENT_AUTOMATION_COMMAND(FEnqueuePerformanceCaptureCommands());
32
33     return true;
34 }
```

 Copy full snippet



The `Parameters` argument can be built and parsed in whatever way is needed for a specific testing situation. For Complex Tests, this is the intended way to test several data points using the same code.

## Latent Commands

**Latent Commands** can be queued up during `RunTest`, causing sections of the code to run across multiple frames. To create a Latent Action, it must be defined via the `DEFINE_LATENT_AUTOMATION_COMMAND` macro. This macro takes one parameter, known as `CommandName`, which defines the class name that will

be created for this type of Latent Command. To finish creating the Latent Command, the new class will require a function body for its `Update` function. Here is an example of a simple Latent Command which runs until the **Unit Test Manager** is finished running tests:


```
1 DEFINE_LATENT_AUTOMATION_COMMAND(FNUTWaitForUnitTests);
2
3 bool FNUTWaitForUnitTests::Update()
4 {
5     return GUnitTestManager == NULL || !GUnitTestManager->IsRunningUnitTests();
6 }
```

 Copy full snippet

If the Latent Command you wish to create requires an argument, such as a parameter string, the `DEFINE_LATENT_AUTOMATION_COMMAND` macro can be used. This macro takes two additional parameters, known as `ParamType` and `ParamName`, which define the type and name of the parameter to be passed in. In this example, a Latent Command is used to start connecting to a source control provider and then wait until the connection attempt finishes:

```
1 DEFINE_LATENT_AUTOMATION_COMMAND_ONE_PARAMETER(FConnectLatentCommand, SourceControlAutomationCommon::FAsyncCommandHelper, AsyncHelper);
2
3 bool FConnectLatentCommand::Update()
4 {
5     // Attempt a login and wait for the result.
6     if(!AsyncHelper.IsDispatched())
7     {
8         if(ISourceControlModule::Get().GetProvider().Login( FString(), EConcurrency::Asynchronous, FSourceControlOperationComplete::CreateRaw(
9             &AsyncHelper, &SourceControlAutomationCommon::FAsyncCommandHelper::SourceControlOperationComplete ) ) != ECommandResult::Succeeded)
10        {
11            return false;
12        }
13        AsyncHelper.SetDispatched();
14    }
```


```
14  
15 return AsyncHelper.IsDone();  
16 }
```

 Copy full snippet

When the `Update` function returns `true`, the Latent Command is considered complete. A return value of `false` indicates that the Automation Test should stop executing immediately and try again next frame. To use a Latent Command in your test code, invoke the `ADD_LATENT_AUTOMATION_COMMAND` with the constructor of the Latent Command you wish to run. If the Latent Command was established with a parameter, pass in the value that the parameter should take as a constructor argument. In your `RunTest` function, the following code will wait for all Unit Tests to finish, then attempt to connect to a previously-named source control provider:

```
1 ADD_LATENT_AUTOMATION_COMMAND(FNUTWaitForUnitTests());  
2 ADD_LATENT_AUTOMATION_COMMAND(FConnectLatentCommand(SourceControlAutomationCommon::FAsyncCommandHelper()));
```

 Copy full snippet

 Activities related to loading or streaming data, or anything else that isn't guaranteed to run in a single frame, are candidates for Latent Command usage. For example, in the Editor, loading a map happens immediately, but in game, loading a map happens on the next frame, so Latent Commands must be used to ensure consistent behavior when an Automation Test needs to load a map.