Automation Spec

Overview covering a new type of automation test, known as a 'Spec', which has been added to our existing automation testing framework.

We have added a new type of automation test to our existing automation testing framework. This new type is known as a **Spec**. "Spec" is a term for a test that is built following the Behavior Driven Design (BDD) methodology. It is a very common methodology used in web development testing, which we adapted to our C++ framework.

There are several reasons to start writing Specs, including that they:

- Are self-documenting
- Are fluent and often DRYer

DRY (Don't Repeat Yourself)

- Are much easier when writing threaded or latent test code
- Isolate expectations (tests)
- Can be used for nearly all flavors of tests (Functional, Integration, and Unit)

How to Set up a Spec

There are two methods for defining the header for your spec, and both are similar to the existing method we use to define test types.

The easiest method is to use the `DEFINE_SPEC` macro, which takes the exact same parameters as all the rest of the test define macros.

```
DEFINE_SPEC(MyCustomSpec, "MyGame.MyCustomSpec", EAutomationTestFlags::ProductFilter | EAutomationTestFlags::ApplicationContextMask)
void MyCustomSpec::Define()
{
//@todo write my expectations here
}

```

Copy full snippet

The only other alternative is to use the `BEGIN_DEFINE_SPEC` and `END_DEFINE_SPEC` macros. These macros allow you to define your own members as part of the test. As you will see in the next section, there is value to have things relative to the this pointer.

```
BEGIN_DEFINE_SPEC(MyCustomSpec, "MyGame.MyCustomSpec", EAutomationTestFlags::ProductFilter |
EAutomationTestFlags::ApplicationContextMask)
```

```
TSharedPtr<FMyAwesomeClass> AwesomeClass;
END_DEFINE_SPEC(MyCustomSpec)
void MyCustomSpec::Define()
{
//@todo write my expectations here
}

```

Copy full snippet

The only other callout is that you need to write the implementation for the `Define()` member of your Spec class, instead of the `RunTests()` member - as you would for any other test type.

Specs should be defined in a file with the `.spec.cpp` extension and not have the word "Test" in the name. For example, the `FItemCatalogService` class might have the files `ItemCatalogService.h`, `ItemCatalogService.cpp`, and `ItemCatalogService.spec.cpp`.

> This is a suggested guideline and not a technical restriction.

How to Define Your Expectations

A big part of BDD is that instead of testing a specific implementation, you are testing expectations of a public API. This makes your test much less brittle and thus easier to maintain, and more likely to work if multiple different implementations of the same API ever crop up.

In a Spec, you define your expectations via the use of two different primary functions, `Describe()` and `It()`.

Describe

`Describe()` is used as a way to scope complicated expectations, such that they are more readable and DRYer. Using `Describe()` makes your code DRYer based on the interaction it has with other supporting functions such as `BeforeEach()` and `AfterEach()`, which is covered below:

```
void Describe(const FString& Description, TFunction<void()> DoWork)


```

Copy full snippet

`Describe()` takes a string that describes the scope of the expectations within it, and a lambda that defines those expectations.

You can cascade `Describe()` by putting a `Describe()` in another `Describe()`.

Keep in mind that `Describe()` is not a test and is not executed during an actual test run. They are only executed once when first defining the expectations (or tests) within the Spec.

It

`It()` is the bit of code that defines an actual expectation for the Spec. You can call `It()` from the root `Define()` method or from within any `Describe()` lambda. `It()` should be used ideally to just assert the expectation, but can also be used to do the final bits of setup for the scenario being tested.

Generally, it is a best practice to start an `It()` call description string with the word "should", which implies "it should".

Defining a Basic Expectation

Here is an example of putting it all together to define a very simple expectation:

```
BEGIN_DEFINE_SPEC(MyCustomSpec, "MyGame.MyCustomClass", EAutomationTestFlags::ProductFilter |
EAutomationTestFlags::ApplicationContextMask)
TSharedPtr<FMyCustomClass> CustomClass;
END_DEFINE_SPEC(MyCustomSpec)
void MyCustomSpec::Define()
{
Describe("Execute()", [this]()
{
It("should return true when successful", [this]()
{
TestTrue("Execute", CustomClass->Execute());
});

It("should return false when unsuccessful", [this]()
{
TestFalse("Execute", CustomClass->Execute());
});
});
}

```

Copy full snippet

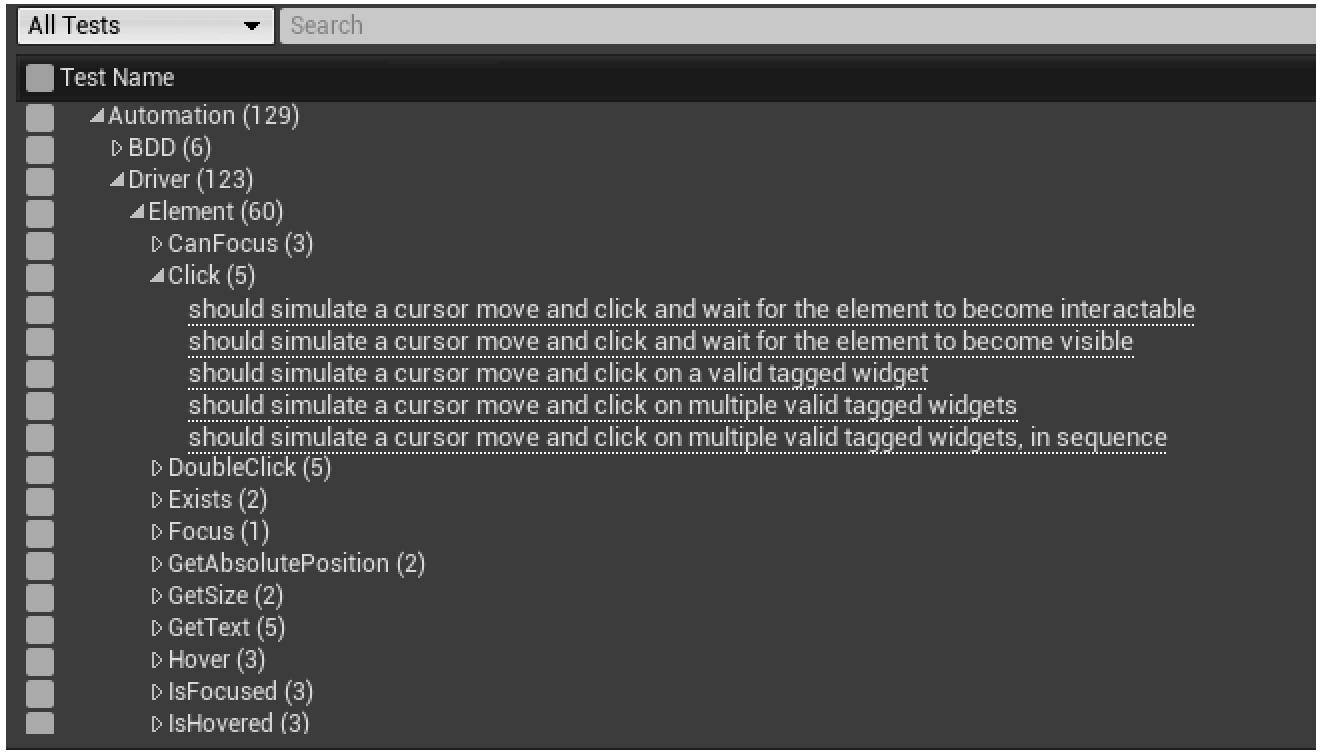
As you can see, this causes the tests to be self-documenting, especially if the programmer takes the time to describe the expectation correctly without combining disparate expectations together. It is intended that combining all the `Describe()` and the `It()` calls should make a mostly readable sentence, for example:

```
Execute() should return true when successful
Execute() should return false when unsuccessful


```

Copy full snippet

The following is a more complicated example of what a mature Spec currently looks like in the Automation Test UI:



```
Automation (129)
  BDD (6)
  Driver (123)
    Element (60)
      CanFocus (3)
      Click (5)
        should simulate a cursor move and click and wait for the element to become interactable
        should simulate a cursor move and click and wait for the element to become visible
        should simulate a cursor move and click on a valid tagged widget
        should simulate a cursor move and click on multiple valid tagged widgets
        should simulate a cursor move and click on multiple valid tagged widgets, in sequence
      DoubleClick (5)
      Exists (2)
      Focus (1)
      GetAbsolutePosition (2)
      GetSize (2)
      GetText (5)
      Hover (3)
      IsFocused (3)
      IsHovered (3)
```

In this example, `Driver`, `Element`, and `Click` are each `Describe()` calls, with the various "should..." messages being defined by `It()` calls.

Each of these `It()` calls becomes an individual test to be executed, and thus can be executed in isolation if one fails while others do not. This makes maintaining tests easier because it is less troublesome to debug them. Also because the tests are self documenting and isolated, when one fails, the person reading the test report has a much more specific understanding as to what is not working - rather than just knowing that a very large bucket named Core failed. This means problems get routed to the right person quicker, and less time is spent investigating issues.

Finally, clicking on any of the above tests will take you directly to the `It()` statement that defined it.

How a Spec Expectation Translates to a Test

The following is a detailed explanation; however, understanding the underlying behavior of the Spec test type will make some of the following complex features easier to understand.

The Spec test type executes the root `Define()` function once, but not until it is needed. As this runs, it collects every non-`Describe` lambda. After the `Define()` finishes, it then goes back through all the lambdas or code blocks it collected, and generates an array of latent commands for each `It()`.

Therefore, every `BeforeEach()`, `It()`, and `AfterEach()` lambda code block is put together in a chain of execution for a single test. When asked to run a specific test, the Spec test type will queue all the commands for that particular test for execution. When this happens, each block does not continue until the previous block has signaled it has finished executing.

Additional Features

The Spec test type offers several other features that make it easier to write complicated tests. In particular, it generally removes the need to directly use the Automation Test Framework's latent command system, which is both powerful and cumbersome.

Here are a list of features supported by the Spec test type that can help with more complicated scenarios:

- BeforeEach and AfterEach
- AsyncExecution
- Latent Completion
- Parameterized Tests

- Redefine
- Disabling Tests

BeforeEach and AfterEach

`BeforeEach()` and `AfterEach()` are core functions to writing anything beyond the most trivial Spec. `BeforeEach()` enables you to run code before the subsequent `It()` code runs. The `AfterEach()` does the same thing, but will run the code after the `It()` code runs.

> Remember that each "test" is only composed of a single It() call.

For example:

```
BEGIN_DEFINE_SPEC(AutomationSpec, "System.Automation.Spec", EAutomationTestFlags::SmokeFilter | EAutomationTestFlags::ApplicationContextMask)
FString RunOrder;
END_DEFINE_SPEC(AutomationSpec)
void AutomationSpec::Define()
{
Describe("A spec using BeforeEach and AfterEach", [this]()
{
BeforeEach([this]()
{
RunOrder = TEXT("A");
});

It("will run code before each spec in the Describe and after each spec in the Describe", [this]()
{
TestEqual("RunOrder", RunOrder, TEXT("A"));
});

AfterEach([this]()
{
RunOrder += TEXT("Z");
TestEqual("RunOrder", RunOrder, TEXT("AZ"));
});
});
}

```

Copy full snippet

In our example, the code blocks are executed from top to bottom, due to the `BeforeEach()` being defined, then the `It()`, then the `AfterEach()`. While it is not a requirement, we suggest you maintain this logical ordering of the calls. But you could mix up the order of the above three calls and the result would always produce the same test.

Also in the above example, it is checking an expectation in `AfterEach()`, and this is very abnormal and a side-effect of testing the Spec test type itself. As such, we do not recommend using the `AfterEach()` for anything other than clean up.

You can also make multiple `BeforeEach()` and `AfterEach()` calls, and they will be called in the order that they are defined. As with the first `BeforeEach()` call being executed before the second `BeforeEach()` call, `fterEach()` behaves much the same way — with the first call executing before the subsequent call.

```
BeforeEach([this]()
{
RunOrder = TEXT("A");
});

BeforeEach([this]()
{
```

```
RunOrder += TEXT("B");
});

It("will run code before each spec in the Describe and after each spec in the Describe", [this]()
{
TestEqual("RunOrder", RunOrder, TEXT("AB"));
});

AfterEach([this]()
{
RunOrder += TEXT("Y");
TestEqual("RunOrder", RunOrder, TEXT("ABY"));
});

AfterEach([this]()
{
RunOrder += TEXT("Z");
TestEqual("RunOrder", RunOrder, TEXT("ABYZ"));
});

```

Copy full snippet

Additionally, `BeforeEach()` and `AfterEach()` are affected by the `Describe()` scope they are called in. Both will only execute for `It()` calls that are within the scope in which they are also called.

Here is a complicated example, with improperly ordered calls, which all work out correctly.

```
BEGIN_DEFINE_SPEC(AutomationSpec, "System.Automation.Spec", EAutomationTestFlags::SmokeFilter | EAutomationTestFlags::ApplicationContextMask)
FString RunOrder;
```

```
END_DEFINE_SPEC(AutomationSpec)
void AutomationSpec::Define()
{
Describe("A spec using BeforeEach and AfterEach", [this]()
{
BeforeEach([this]()
{
RunOrder = TEXT("A");
});

AfterEach([this]()
{
RunOrder += TEXT("Z");

// Can result in
// TestEqual("RunOrder", RunOrder, TEXT("ABCYZ"));

// or this, based on which It() is being executed
// TestEqual("RunOrder", RunOrder, TEXT("ABCDXYZ"));
});

BeforeEach([this]()
{
RunOrder += TEXT("B");
});

Describe("while nested inside another Describe", [this]()
{
AfterEach([this]()
{
RunOrder += TEXT("Y");
});

It("will run all BeforeEach blocks and all AfterEach blocks", [this]()
{
TestEqual("RunOrder", RunOrder, TEXT("ABC"));
});

BeforeEach([this]()
{
RunOrder += TEXT("C");
});

Describe("while nested inside yet another Describe", [this]()
{
It("will run all BeforeEach blocks and all AfterEach blocks", [this]()
{
TestEqual("RunOrder", RunOrder, TEXT("ABCD"));
});

AfterEach([this]()
{
RunOrder += TEXT("X");
});

BeforeEach([this]()
{
RunOrder += TEXT("D");
});
});
});
});
}

```

Copy full snippet

AsyncExecution

The Spec test type also enables you to easily define how a single code block should be executed. This is done by simply passing the appropriate `EAsyncExecution` type into the overloaded version of `BeforeEach()`, `It()`, and/or `AfterEach()`.

For example:

```
BeforeEach(EAsyncExecution::TaskGraph, [this]()
{
// set up some stuff
));

It("should do something awesome", EAsyncExecution::ThreadPool, [this]()
{
// do some stuff
});

AfterEach(EAsyncExecution::Thread, [this]()
{
// tear down some stuff
));

```

Copy full snippet

Each of the above code blocks will execute differently but in a guaranteed sequential order. The `BeforeEach()` block will run as a task in the `TaskGraph`, the `It()` will run on an open thread in the thread pool, and the `AfterEach()` will spin up its own dedicated thread just to run a block of code.

These options are extremely handy when having to simulate scenarios that are thread sensitive, such as with the Automation Driver.

The `AsyncExecution` feature can be combined with the `Latent Completion` feature.

Latent Completion

Sometimes, you need to write a test needing to perform an action that takes multiple frames, such as when performing a query. In these scenarios, you can use the overloaded `LatentBeforeEach()`, `LatentIt()`, and `LatentAfterEach()` members. Each of these members are identical to the non-latent variations, except their lambdas take a simple delegate called `Done`.

When using the latent variations, the Spec test type will not continue execution to the next code block in the test sequence until the actively running latent code block invokes the Done delegate.

```
LatentIt("should return available items", [this](const FDoneDelegate& Done)
{
BackendService->QueryItems(this, &FMyCustomSpec::HandleQueryItemComplete, Done);
});

void FMyCustomSpec::HandleQueryItemsComplete(const TArray<FItem>& Items, FDoneDelegate Done)
{
TestEqual("Items.Num() == 5", Items.Num(), 5);
Done.Execute();
}

```

Copy full snippet

As you can see in the example, you can pass the `Done` delegate as a payload to other callbacks to make it accessible to the latent code. So when the above test is executed, it will not continue to execute any `AfterEach()` code blocks for the `It()` until the `Done` delegate is executed, even though the `It()`

code block has finished execution already.

The `Latent Completion` feature can be combined with the `AsyncExecution` feature.

Parameterized Tests

Sometimes, you need to create tests in a data-driven way. And at times, this means reading inputs from a file and generating tests from those inputs. Other times, it may simply be ideal to reduce code duplication. Either way, the Spec test type allows for parameterized tests in a very natural way.

```
Describe("Basic Math", [this]()
{
for (int32 Index = 0; Index < 5; Index++)
{
It(FString::Printf(TEXT("should resolve %d + %d = %d"), Index, 2, Index + 2), [this, Index]()
{
TestEqual(FString::Printf(TEXT("%d + %d = %d"), Index, 2, Index + 2), Index + 2, Index + 2);
});
}
});

```

Copy full snippet

As you can see in the above example, all you need to do to create parameterized tests is to dynamically call the other Spec functions passing the parameterized data as part of the lambda payload while generating a unique description.

In some cases, using parameterized tests could just create test bloat. It may be reasonable to simply execute all the scenarios from the input as part of a single test. You should consider the number of inputs and the resulting tests that are produced. The major benefit to creating your data-driven tests in a parameterized way is that each test gets to run in isolation, making reproduction easy.

Redefine

When working with parameterized tests, it can sometimes be convenient at runtime to make a change to an external file driving the inputs and have the tests automatically refresh. `Redefine()` is a member of the Spec test type, which, when called, re-performs the `Define()` process. This causes all the code blocks for the tests to be re-gathered and collated.

The most convenient method to do the above would be to create a bit of code that listens for the input file changes and calls `Redefine()` on the test as needed.

Disabling Tests

Every `Describe()`, `BeforeEach()`, `It()`, and `AfterEach()` member of the Spec test type has a variation with a preceding 'x'. For example, `xDescribe()`, `xBeforeEach()`, `xIt()`, and `xAfterEach()`. These variations are a simpler way of disabling a code block or `Describe()`. If `xDescribe()` is used, then all code within `xDescribe()` is also disabled.

This can be easier than commenting out expectations that need iteration.

Mature Examples

You can find mature examples of the Spec test type in `Engine/Plugins/Tests/AutomationDriverTests/Source/AutomationDriverTests/Private/AutomationDriver.spec.cpp`. This spec currently includes over one hundred and twenty expectations and makes use of most of the advanced features at some point.

Our Launcher team also has multiple mature uses of the Spec framework, and one of the most mature uses are the Specs written around `BuildPatchServices`.