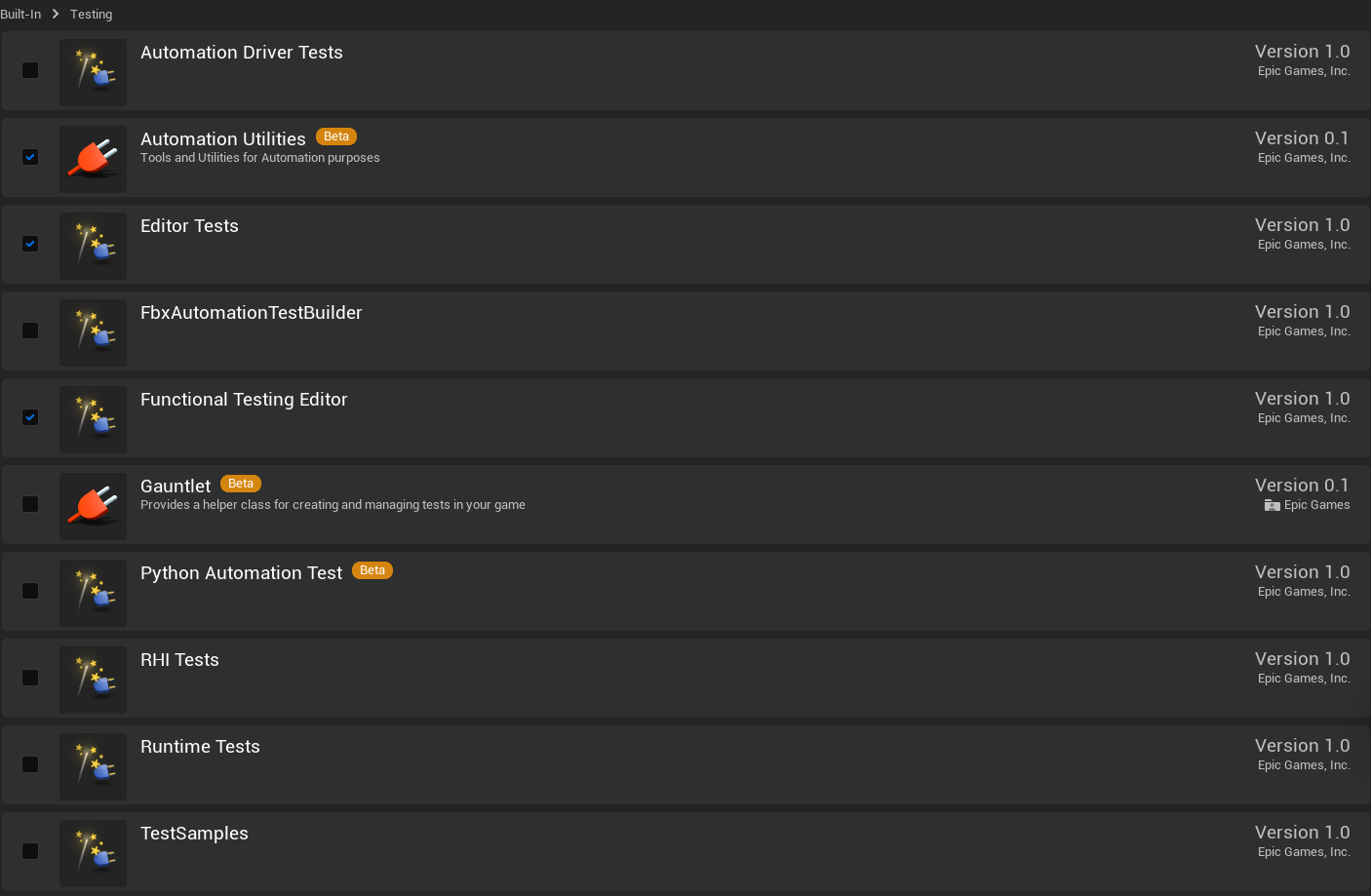
Creating automated functional tests using session frontend

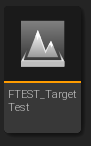
# Enabling the plugins

In order to make use of automated testing tools, we will have to enable certain plugins like “Automation utilities” and “Editor tests” under the “Testing” category. This will allow us to create our own custom functional tests within the session frontend.



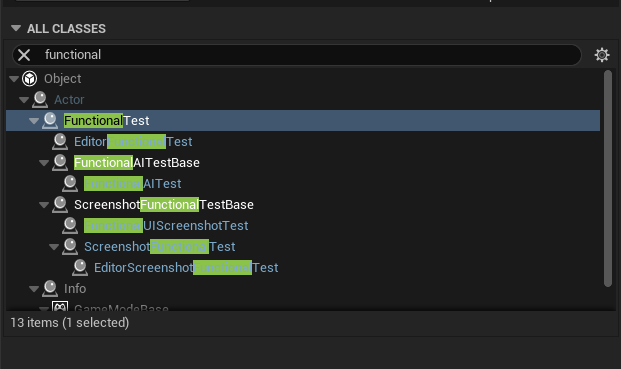
# Creating an environment for the tests

It is ideal that we create our own custom levels to run our functional tests. They can be either a copy of a previous level or a new one. **We will assign the suffix** “FTEST\_” to all levels of this type. All test levels will be located under the “Levels” folder.



# Creating the functional tests

Under “FunctionalTest” we will find different sub-classes with different uses for testing, but for now, the base functional test class should be enough for the majority of tests.



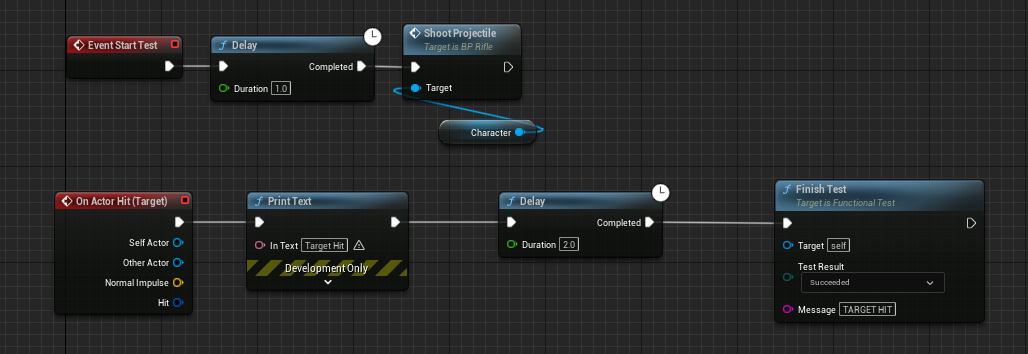
## Inside the functional test blueprint

The base functional test class provides us with 3 default main events to conduct testing.

* On test Start: This delegate is called when the Functional Testing Manager begins the test. Bind this to your test functionality, and be sure to call FinishTest at the end.
* Prepare Test: This function is the first to run, along with the OnTestPrepare delegate, and can be overridden to perform the initial setup required for your test. If this setup requires multiple frames, for example, if the setup requires streaming data to be loaded into the Level, building pathing data, or connecting to a server, this function should start those processes.
* On test Finished: Once the test ends, this delegate will be called. Because testing often affects the Level or Actors within the Level in a way that impacts subsequent tests, using this opportunity to clean up can be vitally important to maintaining a usable testing environment.

# Test 1: Detecting hits on targets

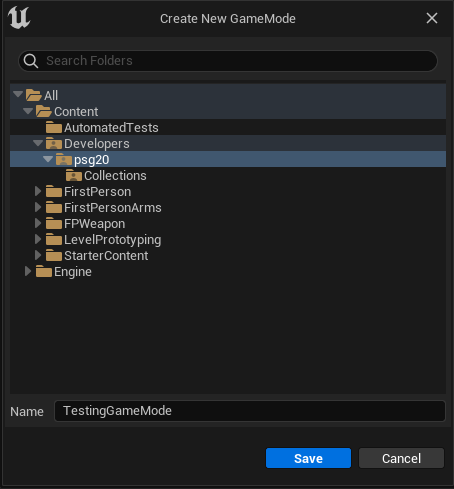
On this test we are shooting a projectile with functionality built in the first-person template. Once we shoot a projectile in a certain direction, we are evaluating if the projectile hits a target and printing a message of “success” if the condition is met.



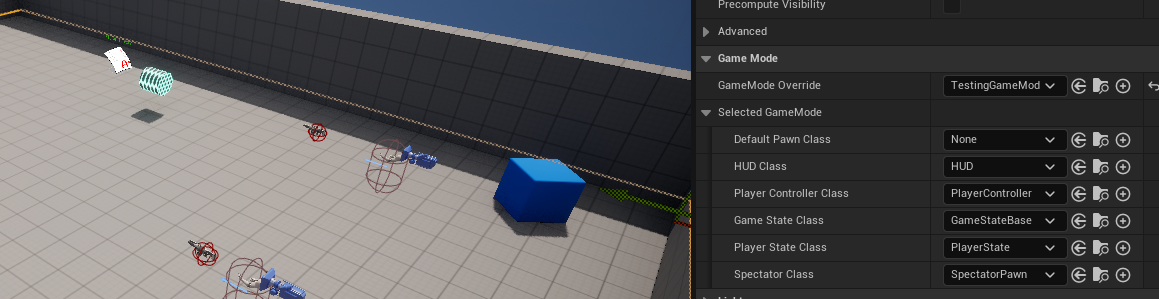
I am using 2 variables for this, one being the “Target” and the other being “Character” which will be the rifle shooting.

# Setting up the test

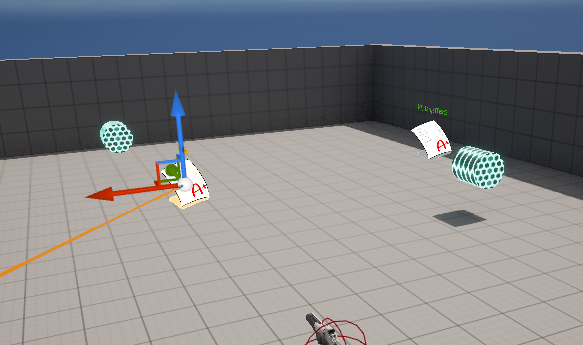
We want to create our own game mode for testing, in order for external input to not affect the results of our test in any way.



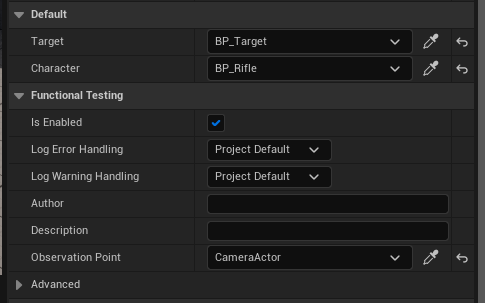
We are not going to be possessing our character during the actual test so “Default pawn class” should be assigned to none:



Then we can drag and drop our functional test blueprints inside of our level:

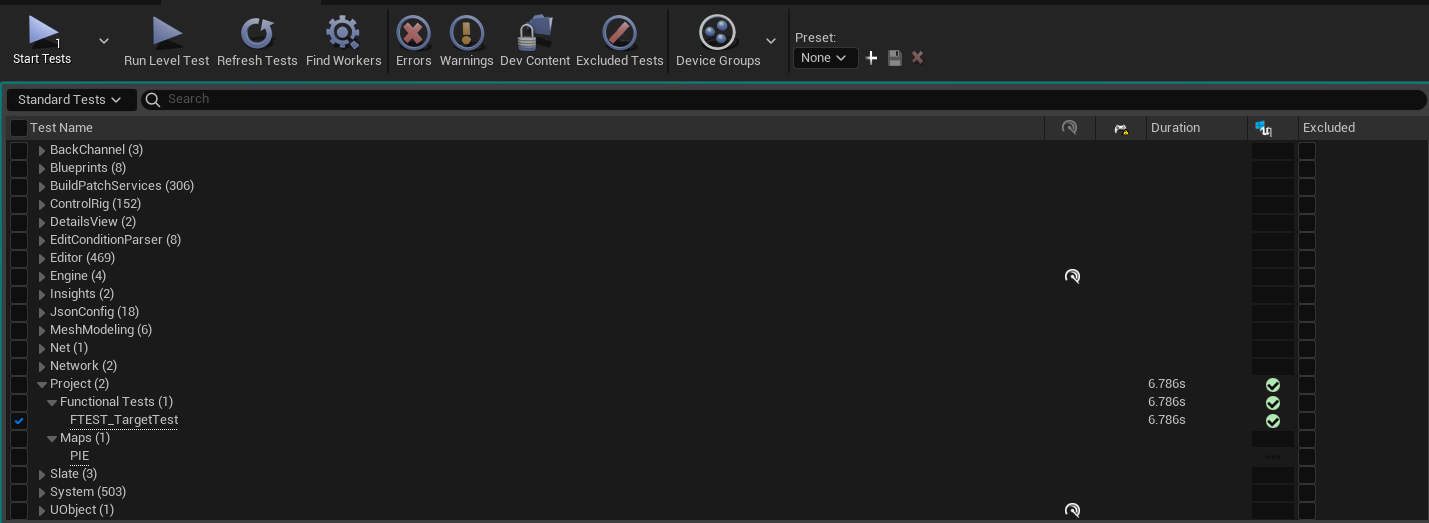


Once that is done, we can assign our variables with the elements that we placed in the world. The target will be a small cylinder like object with a collider, and the character will be one of the rifles in the level. On another hand we can assign an observation point to our test which will be a top-down camera overlooking the entire level:

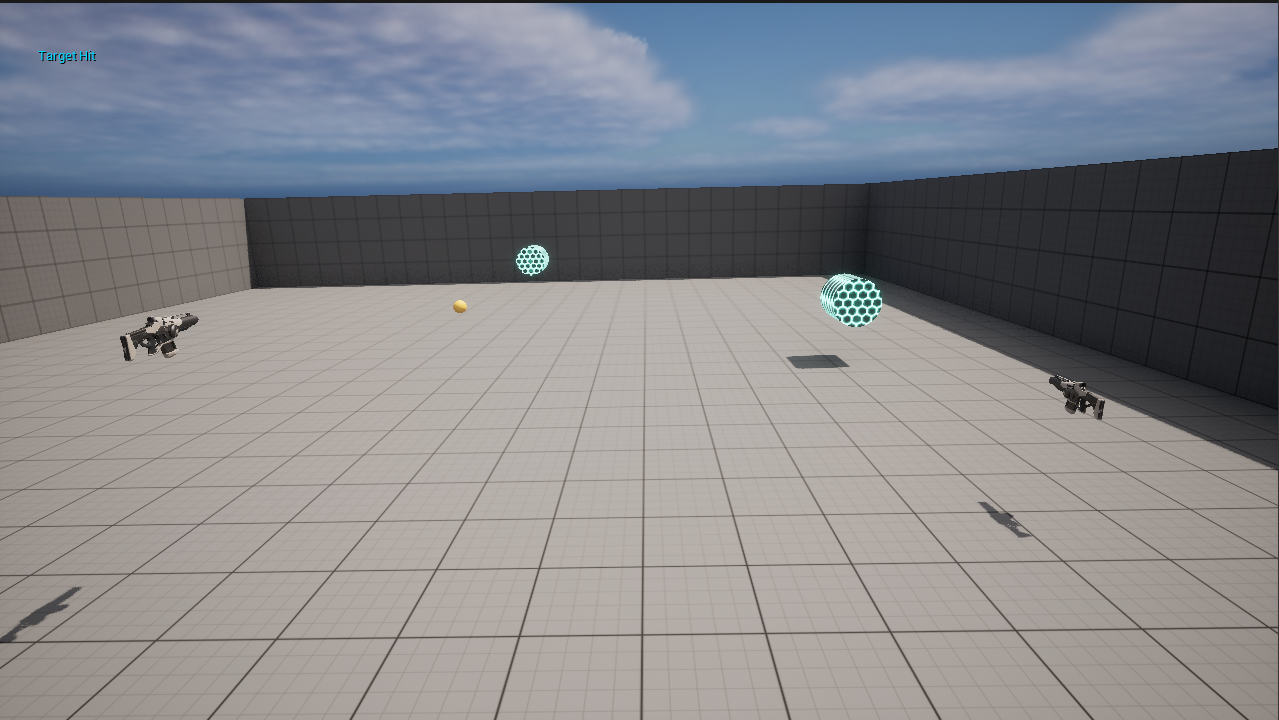


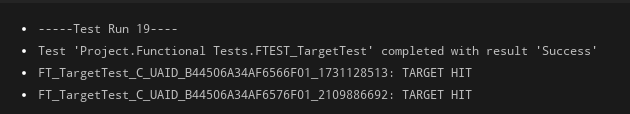
# Configuring the session front-end

Once everything is set up, we can open the session frontend under Tools->Session Frontend. Here we can find our current session as well as all the tests that we can run. Our custom tests will be located under Project->Functional tests->[Name of the level].



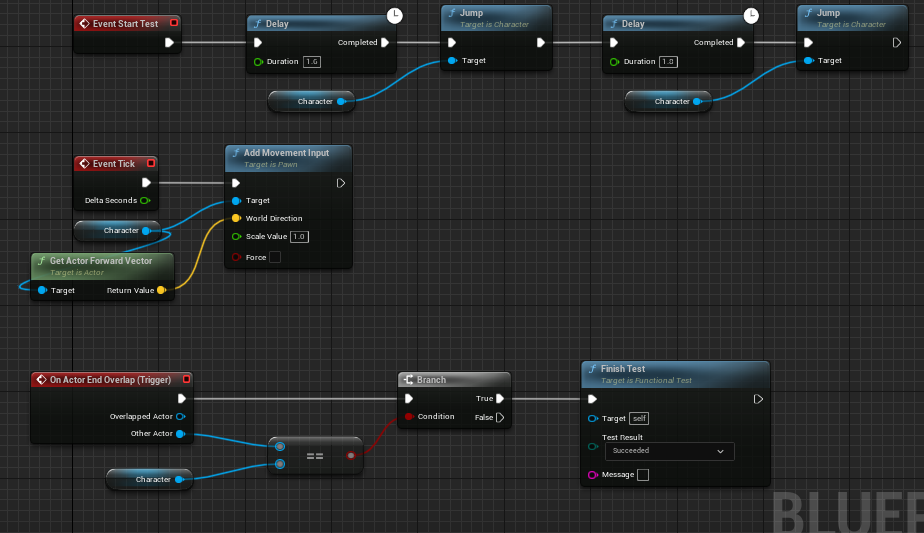
Then we can start the test. If the condition we set before (hitting the target) is met, then the test will succeed. In my case I set up two different rifles inside the scene, shooting at targets in different distances:



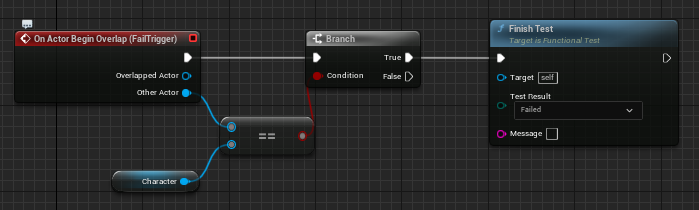


# Test #2: Experimenting with player locomotion

In this test we are experimenting with player locomotion. The player will be sprinting and them jumping on a small obstacle course in order to reach the goal.



* Firstly, we are adding constant movement to the player in the direction of its forward vector
* Then we make the character jump twice, after a certain delay
* Lastly, we added a trigger box which will determine the outcome of the test. If the character reaches this box, then the test will succeed.



On Another hand, we have a failing condition. A second trigger box was added in a gap between the blocks, so if the player falls down, the test results in failure.

# My Personal Experience

The Unreal engine automation system is a powerful and versatile tool which I believe, has great potential for making long and tedious QA processes faster and more effective; but is very poorly documented. Epic Games documentation for this topic is confusing and hard to follow, because little to no context is given for the majority of instructions and explanations. Reading this provided me with some basic useful knowledge, but I found better luck in forums and external resources. This leads into another problem; unfortunately, there are barely any guides/resources online that handle automation testing within UE, making it hard to harness its full capacity. In despite of this I was able to figure out tests for basic functionalities, which can have useful and larger applications within the circumstances of a real project.

# Larger applications

* Automated tests require little to no human interaction if set up right. Test engineers strive to catch defects before any product is released, but they can always creep in or re-surface. Automation is an authentic way to increase coverage and effectiveness so that scenarios like this, are less common, and results more consistent.
* Automated tests can be running at any point during the day. Meaning they can be running overnight, catching bugs/defects for builds that are not tested during the day.
* Automated testing can simulate multiple users interacting within the application/game at the same time. This paves the way for smaller QA teams, as single tests require less testers, and they can be shifted to tasks that require the human eye and attention to detail.
* Automated testing is a solution that fits well with continuous delivery systems. Tests can run automatically whenever source code changes are pushed, and can notify the programmers or team if they fail. Features like these save developers time and increase their confidence.