



## Introduction

At WisEngineering, we have multiple UnrealEngine 4 (UE4) projects in various states of production, at any given time. While UE4 is an excellent solution for teams large and small, managing UE4 projects can be daunting. Large teams will benefit from the Unreal Game Sync (UGS) System's vast resources, providing a fine solution for build distribution. However, for smaller groups or for groups that have in-depth remote access requirements, the UGS isn't an easy to implement solution. This white paper is for you.

This paper will clarify the process of managing and distributing builds to team members, establish a path for artists and programmers alike, and demonstrate how to setup the best solution for merging and upgrading.

## About the Author

Joe Wilcox is the Technical Director at WisEngineering and has been positioning WisE at the forefront of the AR/VR revolution through over 20 plus years' experience building gaming software in the Unreal Engine and shipping over 15 titles across 6 platforms.

## Different Types of UE4 Distributions

Before beginning to dissect how to manage projects with small teams, it's important to understand all three of the existing methods for distributing builds. Epic has a basic description of the various deploying methods [here](#). Let's break them down in more detail.

### 1. Checking builds in to source-control

This was the original method of distributing builds and was in use at Epic when I was working on UT Alpha. From a development point of view, checking builds into source control is simple. A programmer or a development machine creates a build and the newly created binaries are checked in.

Content developers download the precompiled build for their platform and utilize that build for their workload. Programmers typically recompile their own local builds of their project. Build labels are often used to give a finer control over what build a team member uses.

Sounds great huh? Well at first look, builds in perforce are simple, but, they have several downsides. First and foremost, the size of UE4 builds makes transferring them around difficult. If you are constantly checking in both the binaries for your project as well as the engine, you will be confronted with large sync times and huge file transfers.

Second, since each project has both its local source and full engine source, upgrading to newer engine drops can be a long process. developers must individually upgrade each project.

Finally, by using source control to distribute binaries, it's often possible to get a poison pill in the mix, i.e., a build that breaks game content.



## 2. Utilize the Unreal Game Sync system

The UGS system is an efficient way to manage builds in source control and it alleviates many of the downsides. It also requires a considerable investment in backend resources to make sure that builds can be properly tracked and distributed to end users. That being said, it simplifies the management of promoted builds and helps protect against poison builds.

The biggest downsides to using UGS as a small development house is the backend infrastructure needed to perform the management.

If you're interested in learning more about UGS, I suggest starting at this link:

[Unreal GameSync Documentation](#)

## 3. Distributing via custom Installed builds

Custom installed builds are like using the Epic Games Store to install a build of the engine and working from that. The main full install of the engine gets processed, in a way similar to making a stripped-down installed build. This build can then be checked into source control or distributed in some fashion.

There are several advantages to using custom install builds of the engine, but there are also some serious limitations. The biggest advantage of this method is that all of your projects are based on a fixed development build of the engine. This means upgrading to newer engine drops is a painless and quick process. However, this also emphasizes the biggest disadvantage.

Custom install builds are only suited for situations where minimal changes to the base engine are required and when those changes are to be shared across all projects.

Your chosen method should be determined by your circumstances. Here at WisEngineering, we have many projects in active development for multiple clients, but our changes to the engine are minimal and shared across all projects. So, the tight integration and faster compilation times on custom builds suits our development style.

This white-paper will layout our processes for installing, maintaining, and utilizing a custom install build for use in multiple projects. Let's begin.

## New Terminology

Before we can get into the nitty-gritty of setting up a custom install build system, we should make sure we are on the same page regarding terminology and directory layout. This will become important later.

**Custom Install Build** – A way to utilize pre-compiled UE4 engine binaries for development.

**Source Drop** – This is a copy of the engine in some form. There are two types of source drops, engine drops and development drops.

**Engine drop** – This is where you keep your base engine source code that you use to create your install build. There should only be one single engine drop in this system.

**Development drop** – This is a build of the engine that has been packaged into custom install build. Our system supports multiple development drops, in case we need to lock a project to a specific engine drop. Typically, there will be one engine drop and one development drop.



## Directory Structure

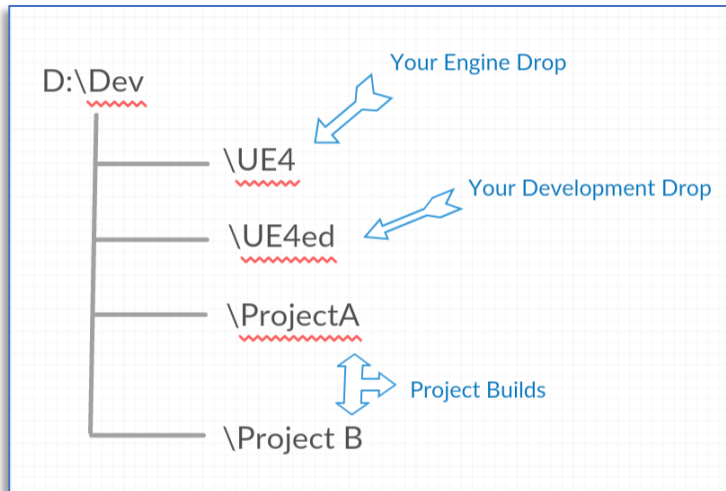


Figure 1.1 – Programmer Directory Structure

Figure 1.1 shows the typical programmer directory structure. Programmers at WisE have access to both the Engine drop and the Development drop to facilitate changes to our engine plugins. However, only someone tasked with creating the install build truly needs access to the engine drop. Our artists, have a directory structure as seen in Figure 1.2:

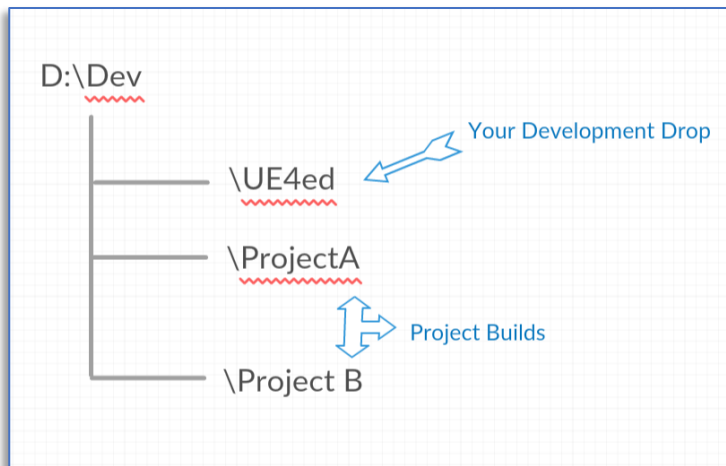


Figure 1.2 – Artist Directory Structure

Here, the engine drop is missing. Artists, some programmers, and blueprint scripters do not need to have access to the engine drop, so we restrict access using source control roles. This directory structure is perfect for managing project source. You can also utilize it to store common directories or even example code.


Setting up an efficient directory structure early will pay off in the long run. One of the benefits to this structure is that it makes utilizing streams in Perforce simple.

**NOTE:** Storing projects outside of the engine directory like this requires using fully qualified paths when loading the .UPROJECT file. Below, I'll show how to fix this at the engine level.



## Starting from Scratch

So how do we setup our environment? It all begins with downloading a source drop from Epic Game's GitHub page. You can find the source at the [UE4 GitHub Page](#).

The simplest way to get the latest source code is to click on the  button at the top of the GitHub page. This will download the latest approved build of the engine in .ZIP form. This is your Engine drop.

### Step 1- Installing the Engine Drop

Our next step is to install this drop. In the root of your development folder (in our case **D:\Dev**) you want to create a folder to hold the engine drop (in our case **D:\Dev\UE4**). What you call this folder isn't important, although you will need to remember its location for later.

### Step 2- Installing the Prerequisites

Once all files are unpacked from the GITHUB .zip file, you'll want to get all the needed prerequisite files installed. Epic has made this super easy. Use Explorer to browse to the root directory of your engine drop (in our case **D:\Dev\UE4**) and locate the file **Setup.bat**. Then execute!

### Step 3 – First compile

Now that engine drop is installed on your PC and you have installed all the required pre-reqs, you need to make your first compilation. To begin, you need to generate your project files. UE4 has a batch file for doing this located in root directory (in our case **D:\Dev\UE4**). Browse to that directory and run **GenerateProjectFiles.bat**. This will open a command prompt and close quickly.

You can now open the visual studio solution by opening **UE4.sln** that will be created in the same directory.

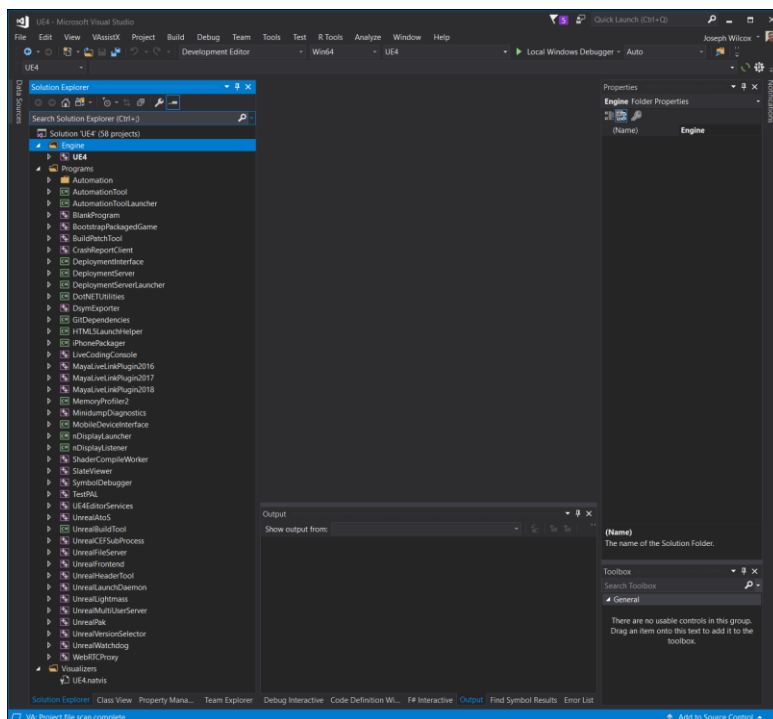


Figure 1.3 – The default solution



Once you have loaded Visual Studio, you will want to expand the Engine tab to show the UE4 project as seen in *Figure 1.4*:

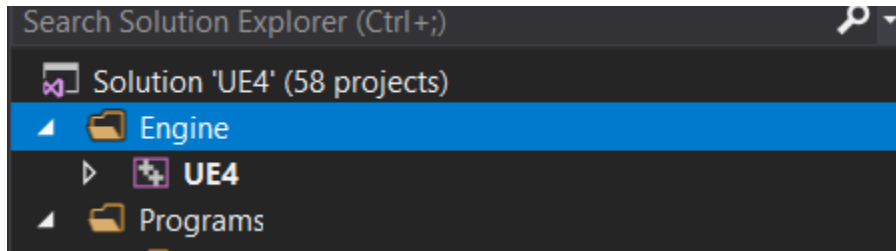


Figure 1.4 – The UE4 Project

Right click on **UE4** and select Rebuild. Now relax and go get a drink as Visual Studio creates a fresh set of binaries.

### Step 4 – Make UE4 more friendly to multiple projects

Once your initial compilation of the engine is completed, it's time to make UE4 behave better with multiple projects. By default, if you are using a custom install build you will have to always fully qualify the path to the **.uproject** file. Although small, this annoyance can be quickly resolved by making a single engine change to **LaunchEngineLoop.cpp**, which is located in the **Engine\Source\Runtime\Launch\Private** folder of your engine drop.

Open this file and find the function **ParseGameProjectFromCommandLine()**. This function's job is to take the command line and pull the uproject filename from it. By default, UE4 will look in any folder specific in the UE4Games.uprojectdirs file. However adding a reference to the parent directory ("../") doesn't work. In the "else" statement of the inner If/Then/Else block find the lines:

```
// Full game name is assumed to be the first token
OutGameName = MoveTemp(FirstCommandLineToken);
// Derive the project path from the game name. All games must have a uproject file, even if they are
// in the root folder.
OutProjectFilePath = FPaths::Combine(*FPaths::RootDir(), *OutGameName, *FString(OutGameName +
TEXT(".") + FProjectDescriptor::GetExtension()));
```

Directly after these two lines we want to add the following block of code:

```
if (!FPaths::FileExists(OutProjectFilePath))
{
    FString Root = FPaths::RootDir() + TEXT("../");
    FPaths::CollapseRelativeDirectories(Root);
    OutProjectFilePath = FPaths::Combine(*Root, *OutGameName, *FString(OutGameName + TEXT(".") +
FProjectDescriptor::GetExtension()));
}
```

This looks to see if UE4 can find the UPROJECT file using the normal system. If it can't, attempt to step back to the parent directory and use the UPROJECT filename (without extension) as a subdirectory name to build a fully qualified path to the uproject file. This works great with the directory structure I laid out in *Figure 1.1* and *Figure 1.2* and will save you a lot of typing.

**NOTE:** This is an unsupported change to the base engine. While I have been using it for more than a year, it still might have side-effects that I haven't determined, especially on other platforms.



### Step 5 – The automation tool

I could write a white paper on just this tool alone. For now, understand that the UE4 Automation Tool is what you will use to build the custom install build of our custom version of UE4. Before making this build, you're going to have to compile it as the binaries are not created as a part of the UE4 project.

Before leaving Visual Studio, expand the programs node of the solution and find the automation tool project as seen in *Figure 1.5*.

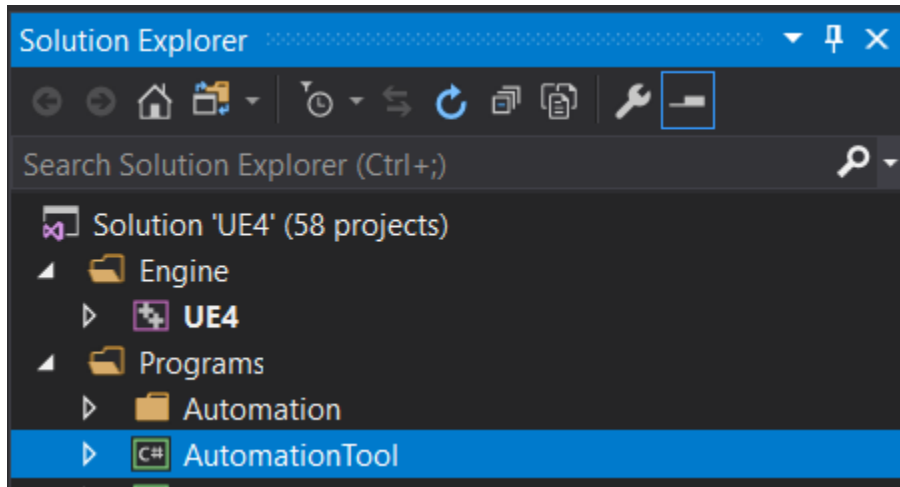


Figure 1.5 – The automation tool project

Once again, right click on this project and select Rebuild to create the automation tool. Unlike building this engine, this will be a quick compilation compared to the full rebuild above.

Our next step is to generate the install build!

**NOTE:** Before you make your custom install build, you'll want to install any and all engine plugins (i.e: Substance) either directly or be using the marketplace. Included plugins will be included in the install build.

## Generating the Install Build

Before we continue with our steps, it's important to briefly talk about the **UE4 Automation Tool (UAT)**, the **BuildGraph**, and the **InstallEngineBuild** script and how they work together. UAT is a special tool used to automate various tasks. It's a complex system and there are many scripts/usages. For more information on this system, I suggest you start reading the [Automation System Overview](#) as we will only be covering the tool to generate our development drop.

For generating an install build, we will utilize a custom **BuildGraph** task.



**BuildGraph** is a script-based build automation system that features graphs of building blocks common to Unreal Engine 4 (UE4) projects. BuildGraph integrates with UnrealBuildTool, AutomationTool, and the editor, and can be extended and customized for your projects.

Figure 2.1 – From the UE4 Documentation

Detailing all the features of the BuildGraph is far beyond the scope of this white paper. If you are looking to extend the build system, then I strongly suggest you start here at the [Build Graph Documentation](#) and really digest the scripts and code. Luckily for our purposes, Epic includes a predefined custom script for making install builds. You can find this script here:

***[EngineDrop]/Engine/Build/InstalledEngineBuild.xml***

This script is your gateway to install builds and starts with the standard XML header. The next 100 lines or so are the various options that we will use to configure the build. I'll cover how to utilize the options in a bit. The real meat of the script is found in the `<Agent>` and their `<Node>` tags. These define what type of action the script is going to take. Again, how to customize these scripts is beyond this document, however we are very much interested in three different nodes:

1. `<Node Name="Make Installed Build Win64" ...>` builds a Win64 custom install build.
2. `<Node Name="Make Installed Build Mac" ...>` builds a Mac custom install build.
3. `Node Name="Make Installed Build Linux" ...>` builds a Linux custom install build.

When you execute the BuildGraph you will use the node name that matches the desired platform for the install build in the `-TARGET=` parameter.

There are two methods to run the BuildGraph? First, you can utilize the ***RunUAT.bat*** file in the ***[EngineDrop]/Engine/Build/BatchFiles*** directory (feel free to add this to your path, as it makes life easier). This is the preferred way to execute the automation tool as it performs all the prerequisite checks and will even compile it if needed (and Visual Studio is present). You can also run the automation tool on its own. We will be using the ***RunUAT.bat*** approach.

The final thing we need to talk about are the available options. These are defined with the `<Option>` tag in the ***InstallEngineBuild.xml*** script. If you wish, you can set them directly in the xml or you can pass them via the command line using the `-set:[option]=<value>` format. A good description of each command can be found inside the ***InstallEngineBuild.xml*** script itself.



The default values for the platform options that Epic has chosen for their InstallEngineBuild.xml script are overkill for most developers. The inclusion of the Lumin platform (Magic Leap) causes real problems. Most developers can't just build an install build as is, instead they are left with either editing the xml script or knowing that they must disable Lumin support by adding

**"-Set:WithLumin:false"**

On the command line, I suggest always checking the xml script and fixing your defaults with each new engine since new hardware is always being added.

Now it's time to continue our steps to making our custom install build.

#### Step 6 – Making our Development Drop

Edit **[EngineDrop]/Engine/Build/InstalledEngineBuild.xml** and configure it for your needs. In my case here (and to save build time), I'm going to set all of the platform to false with the exception of **WithWin64** which I'll leave set to true. Remember, the more platforms that you enable the longer this step will be.

Once your configuration is ready, open a command prompt and navigate the following directory:

**[EngineDrop]/Engine/Build/Batchfiles**

```
Command Prompt
05/27/2019 10:05 PM <DIR> .
05/27/2019 10:05 PM <DIR> ..
05/21/2019 10:26 AM      840 Build.bat
05/21/2019 10:26 AM     990 BuildThirdPartyLibs.bat
05/21/2019 10:26 AM    2,390 Clean.bat
05/21/2019 10:26 AM     127 DocsPreviewPage.bat
05/21/2019 10:26 AM     439 FixDependencyFiles.bat
05/21/2019 10:26 AM    5,564 GenerateProjectFiles.bat
05/21/2019 10:26 AM    3,103 GetMSBuildPath.bat
05/21/2019 10:26 AM     883 GetVSCommToolsPath.bat
05/21/2019 10:26 AM <DIR> HTML5
05/21/2019 10:26 AM <DIR> Linux
05/21/2019 10:26 AM <DIR> Mac
05/21/2019 10:26 AM    2,625 MakeAndInstallSSHKey.bat
05/21/2019 10:26 AM     488 MSBuild.bat
05/21/2019 10:26 AM     477 Rebuild.bat
05/21/2019 10:26 AM    3,368 RunUAT.bat
05/21/2019 10:26 AM     310 RunUAT.command
05/21/2019 10:26 AM    3,164 RunUAT.sh
05/21/2019 10:26 AM     365 RunUATTest.bat
05/21/2019 10:26 AM     878 RunUAT_LocalTest.bat
05/21/2019 10:26 AM     633 SyncToRemotePC.bat
05/27/2019 10:05 PM      92 SyncToRemotePC.exclude
05/21/2019 10:26 AM    3,134 TestCrashes.bat
      19 File(s)      29,870 bytes
       5 Dir(s)  1,417,092,657,152 bytes free

D:\AssetPack\UE4_422\Engine\Build\BatchFiles>runuat BuildGraph -target="Make Installed Build Win64" -script=Engine/Build/InstalledEngineBuild.xml
```

Figure 2.2

In *Figure 2.2*, you can see the typical command line required for a custom install build. This command line assumes that you are using the script to configure the platforms you are building.





Press enter and go grab some dinner. Making a custom install build takes time, and the amount of time is dependent on how many platforms you are including. But even just building Win64/Win32 can take upwards of an hour on a fast PC. So seriously, get some coffee!

## What do you with the Development Drop?

Now that you have finished building your custom install build, how do you use it as your development drop? Well, it's almost as simple as copying some files... almost. Before we do that, let's talk about two things, how UE4 recognizes different builds of the engine and how-to setup prerequisites.

### *.UPROJECT ASSOCIATIONS*

As Windows10 is my primary development platform, the following technical information is regarding that platform only. Mac / Linux users may have different mechanisms.

UE4 registers **.UPROJECT** files with windows and passes it through to a **UnrealVersionSelection.exe**. This is a custom utility developed by Epic Games that opens the uproject file and determines what build of the engine to use. But how does it know?

Every **.UPROJECT** file has an option **EngineAssociation**. This option tells UnrealVersionSelection what build of the engine to use. The whole process relies on two specific branches in the Windows Registry.

The first branch is located at **HKEY\_LOCAL\_MACHINE\SOFTWARE\EpicGames\Unreal Engine** and this branch describes all Epic official launcher builds. If you install a version of the engine via the Epic Games Store (EGS), the important information regarding this build will be found under the above key.

The second branch is located at **HKEY\_CURRENT\_USER\Software\Epic Games\Unreal Engine\Builds** and any custom install builds will be found here. Before you can use a custom install build it has to be registered with the OS. This happens as a part of the prerequisite steps that I will lay out in a bit.

So how does the process work? When you double-click on a **.uproject** file, Windows will execute a version of the **UnrealVersionSelection** passing the **.UPROJECT** file in as parameter. Which version is determined by typical windows file association rules and the key can be found in

**HKEY\_CLASSES\_ROOT\Unreal.ProjectFile**.

UVS will open the uproject file and read in the EngineAssociation value. It first looks at officially installed versions of the engine to see if there is a match and if there is, it loads that version. Next, it looks at the custom installed builds. If no match is found in either branch, UVS will open up a dialog that allows you to assign the **.uproject** file to an existing engine.

**NOTE:** When Epic associates a custom installed build with a uproject, it uses a standard format GUID. A GUID is a unique identifier (guaranteed by the OS) so it makes sense, but you don't have to use it. UE4 performs a standard string / text comparison to find what build to use! So, I suggest using values that make sense instead!



## *UE4 Prerequisites*

The development drop will need to be installed onto a new machine or a machine that hasn't been configured for UE4 development. In order to make this work, you'll need to make sure that all of the prerequisite SDKs and software are installed. It also ensures that a given build of the engine is registered with the OS.

When you install a build via the Epic Games Store this step will happen for you. Likewise, you'll remember from Step 2 of "Starting From Scratch" that when you do a fresh engine drop from Github, you executed **setup.bat** which is included to install the pre-reqs. But what about development drops?

It's a mixed bag. The UE4 Prerequisite redistributable is included in the install build, but the **GITDependencies**, **UnrealVersionSelection** and **Setup.xxx** files are not. So, in order to create a completely installable build, you'll need to manually copy some files.

**[EngineDrop]/Setup.bat**

**[EngineDrop]/Engine/Binaries/Win64/UnrealVersionSelector-Win64-Shipping.exe**

**[EngineDrop]/Engine/Binaries/DotNET/GitDependencies.exe**

**[EngineDrop]/Engine/Binaries/DotNET/GitDependencies.exe.config**

The **GITDependencies** is optional if you are using P4 or some other source control system and the setup.bat file already skips it nicely if it's not installed. Once these are copied into place, you now have a fully functional development drop that can be packaged for distribution as you see fit.

## *Step 7 – Installing the Development Drop*

Now it's time to do something with your new development drop. Here at WisE, we utilize perforce to distribute the drop, making it easy for artists to be at the topmost level. We simply check in the development drop and tell user when they need to execute setup.bat.

How you distribute and install the drop is entirely up to you and your needs. Just remember that **setup.bat** needs to be run at least once, each time there is an engine update.



## APPENDIX A – Custom Tools

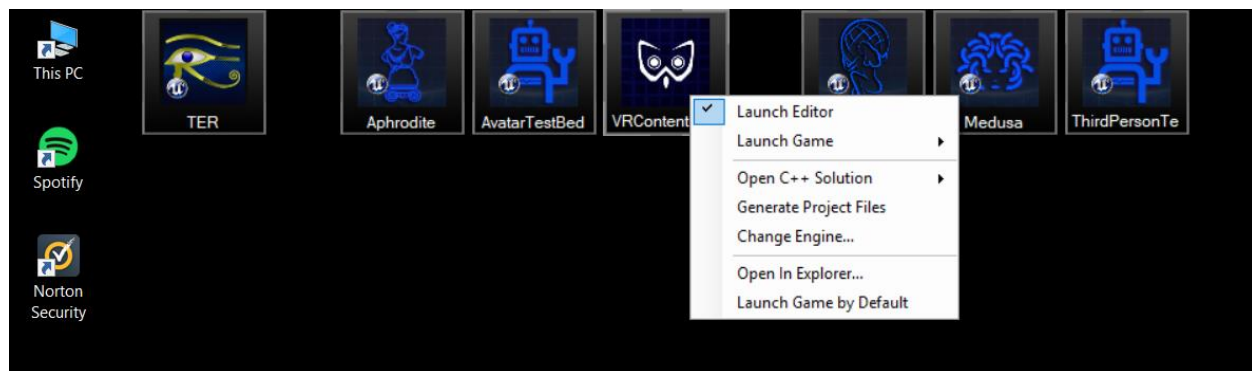
Now that you are setup to use your own custom install build, WisE has a couple of tools for you. These tools will make managing multiple UE4 projects much easier.

### UE4Register

The first tool we have is UE4Register. Manually editing the registry can be a pain and UE4Register takes it away. It allows you to quickly register multiple versions of the engine at one time. When you execute it, it will look at all of the sub-folders underneath itself and register any copies of the engine it finds. Of course, you can include a **UE4Register.key** text file in your engine directory to define what key to register the engine under.

### UE4Desktop

Our second tool is UE4Desktop. UE4Desktop is a lightweight tool that creates desktop like icons on your Windows desktop for managing UE4 projects.



Common actions for the project are provided via a context-menu popup. UE4Desktop also supports opening multiple copies of your project with complete control over where the windows open and has many more features. You should definitely check it out.

For more information about UE4Register or UE4Desktop, check out our tools page at:

[The WisE Engineering Resources Page](#)

## APPENDIX B – FAQ:

I will attempt to keep this section updated with answers to questions.

**Q: When I try to build my custom install build, I get an error that “Unreal to find installation of PDBCOPY.EXE”.**

A: These tools aren't for some reason pre-installed with Visual Studio. You'll need to install them yourself. Just run the debugger at the [Windows 10 SDK Download Page](#)