Creating a Gameplay Module

Learn how to create a runtime module for your game from scratch.

Unreal Engine (UE) Modules are a useful way to organize your code, improve your project's build times, and configure its loading and unloading processes for systems and code. This guide will cover the steps necessary to implement a new runtime module from scratch in C++. This module will be separate from the **primary module** for your project.

The module in this example is named **ModuleTest**.

Required Setup

To follow this guide, start by creating a new project named **MyProject**, and make sure it is a **C++ project**. If you are following along with a Blueprint-only project, create a new piece of C++ code in Unreal Editor to convert it to a C++ project.

Set Up Your Directories

First, you need to set up directories to contain the module and its code.

1. Navigate to your project's root directory, then open the **Source** folder.

2. Inside your project's Source folder, make a new folder named **ModuleTest**. This will serve as your module's root directory.
3. Inside your module's root directory, create two new folders called **Private** and **Public**.

Your directory structure should resemble this:

- MyProject
  - Source
    - MyProject.Target.cs
    - MyProjectEditor.Target.cs
    - MyProject (Primary game module folder)
      - Private
      - Public
      - MyProject.Build.cs
      - Other C++ classes in your game module
    - **ModuleTest**
      - **Private**
      - **Public**

Create the Build.cs File

When **Unreal Build Tool (UBT)** looks through your project's directories for dependencies, it ignores your IDE solution file and instead looks at the `Build.cs` files in your Source folder. Every module must have a `Build.cs` file, or else UBT will not discover it.

To set up your `Build.cs` file, follow these steps:

1. Create a file called `ModuleTest.Build.cs` in your module's root directory. Open this file and add the following code:

   ModuleTest.Build.cs

```
using UnrealBuildTool;

public class ModuleTest: ModuleRules
{
public ModuleTest(ReadOnlyTargetRules Target) : base(Target)
{
```

```
7
8 }
9 }
10
```

Copy full snippet

This will make your module visible to the Unreal build system.

2. Edit the constructor so that it reads as follows:

   ModuleTest.Build.cs

```
1 public ModuleTest(ReadOnlyTargetRules Target) : base(Target)
2 {
3
4 PrivateDependencyModuleNames.AddRange(new string[] {"Core",
  "CoreUObject", "Engine"});
5
6 }
7
```

Copy full snippet

This will add the `Core`, `CoreUObject`, and `Engine` modules as private dependencies for your module. This will make several classes available to this module for later steps in this guide.

Now when you add code to this module's folders, that code will be discovered both when UBT builds the project and whenever you generate your IDE project files.

Implement Your Module in C++

While the `Build.cs` file makes your module discoverable to UBT, your module also needs to be implemented as a C++ class so that the engine can load and unload it.

Fortunately, Unreal Engine includes macros that can streamline this process for most common implementations. To make a quick implementation file, follow these steps:

1. Navigate to your module's root directory and open the `Private` folder. Create a new file called `ModuleTestModule.cpp` inside this folder. It is not necessary to make a `.h` file

for this class.

> [ModuleName]Module is the typical naming convention for module implementation files in Unreal Engine source code. This is useful for keeping track of them in a large codebase.

2. Inside `ModuleTestModule.cpp`, add the following code:

   ModuleTestModule.cpp

```
#include "Modules/ModuleManager.h"

IMPLEMENT_MODULE(FDefaultModuleImpl, ModuleTest);

```

Copy full snippet

This provides a default implementation for your module that makes it available to C++ code.

You can create more detailed implementations by manually writing the class for your module, its constructor, and the Startup and Shutdown functions for it. However, for most gameplay modules, this default implementation will be sufficient to load and unload your module.

Compile Your Module

Compile Module as Part of a Project

After defining and implementing your module, the process to compile it is very simple.

1. Right-click `MyProject.uproject` and click **Generate Visual Studio Files** to regenerate your IDE solution. This will ensure that your new module becomes visible in your IDE and within Unreal Editor.

   > Any time you re-organize your module's code or change the contents of its `Build.cs` file, you should regenerate your IDE solution to make sure it stays up to date.

2. Compile your project in Visual Studio. Your new module will be compiled alongside it.

Any time you add a new module manually, change your module's directory structure, move or rename C++ files, or change your module's dependencies, you should regenerate your project files to update your Visual Studio solution, then compile your project again.

> Although you can compile your project while Unreal Editor is running, sometimes you will need to close Unreal Editor, then restart it after compiling for major changes or new classes to take effect.

Compile Only Your Module

To compile only your module and not your entire project, execute the following command from your UE root directory:

```
Engine\Build\BatchFiles\Build.bat -Target="<PROJECT_NAME> <PLATFORM> <BUILD_ST
```

Copy full snippet

For example, to build the Core module as part of the `LyraEditor` project on Windows, execute:

```
Engine\Build\BatchFiles\Build.bat -Target="LyraEditor Win64 Development" -Modu
```

Copy full snippet

Include Your Module In Your Project

Your module should be able to compile, but to use any code from your module in your project, you need to register it with your `.uproject file`, then make a dependency for your game's primary module.

1. Open the `MyProject` root directory, then open `MyProject.uproject` in a text editor and edit your "Modules" list as follows:

MyProject.uproject

```
"Modules": [

{

"Name": "MyProject",

"Type": "Runtime",

"LoadingPhase": "Default"

},

{

"Name": "ModuleTest",

"Type": "Runtime"

}

]

```

Copy full snippet

You can use this list entry to configure which **Loading Phase** it will load up on and its **Type**. This example is a runtime module, so it can be used when the project is running as a standalone application or in the editor.

By comparison, Editor modules can only run in the editor. It also uses the **Default** loading phase, which initializes after game modules are loaded. Depending on the specifics of your module, you might need to use earlier loading phases to make sure other modules that depend on it don't throw errors looking for unloaded code.

2. Navigate to your `MyProject/Source` folder, then open the `MyProject.Build.cs` file. Add `ModuleTest` to your `PublicDependencyModuleNames` list. It should as follows:

MyProject.Build.cs

```
PublicDependencyModuleNames.AddRange(new string[] { "Core",
"CoreUObject", "Engine", "InputCore", "ModuleTest" });

```

Copy full snippet

Now it will be possible to include code from your module in your primary game module.

7. Add Code to Your Module

You can add C++ files to the `Public` and `Private` directories in your module manually, or you can add them in Unreal Editor.

These steps will guide you through using the New Class Wizard to add code to a module:

1. Open your project in Unreal Editor.
2. In the Content Browser, click **Add**, then click **New C++ Class**.
3. Choose **Actor** as the parent class, then click **Next**.
4. Locate the dropdown next to the **Name** field. It should read **MyProject (Runtime)** by default. Click this dropdown, then change it to **ModuleTest (Runtime)**.

   > If you do not see MyModule (Runtime) as an option, review the previous sections to ensure you followed the steps correctly.

5. Name the class **ModuleActorBase**, set the **Class Type** to **Public**, then click **Create Class**.

   Your class's `.h` and `.cpp` files will open automatically in your IDE. Your class's `.cpp` file will be added to your module's `Private` folder, while its `.h` file will be added to the `Public` folder.
6. Open `ModuleActorBase.cpp`, then add the following line to the `AModuleActorBase::BeginPlay` function:

   ModuleActorBase.cpp

```
GEngine->AddOnScreenDebugMessage(0, 5.0f, FColor::Blue, TEXT("Hello,
world!"));
```

```
2
```

Copy full snippet

7. Save your code and compile your module.

This class defines a simple Actor that outputs an on-screen debug message when the game starts. You can test this by using the **Place Actors** tool in Unreal Editor.

8. Extending Code From Your Module

Finally, follow the steps below to test your new Actor class and your primary game module's ability to see it.

1. Open your project in Unreal Editor. Create a **New Blueprint Class**, then expand the **All Classes** list.
2. Choose **ModuleActorBase** as the parent class. Name your Blueprint class **ModuleActorBP**.
3. Drag a copy of ModuleActorBP from the Content Browser into your game's world. Click the **Play** button.

If you do not see ModuleActorBase in the list of classes, make sure its header is in the `Public` folder, that it has the `BlueprintType UCLASS` specifier, and that you have added the `ModuleTest` module to your project's dependencies.

Final Result

When you click Play, your project will start playing in editor, and the on-screen debug message "Hello, world!" will display in blue text. This Blueprint-based Actor extends a class that is defined in a separate gameplay module. When you write modules in the future, you will now be prepared to create extensible classes as a foundation for gameplay-specific code.