

Unreal Engine Modules

Modules are the building blocks of Unreal Engine's software architecture. You can organize your code into modules to create more efficient and maintainable projects.



Modules are the basic building block of **Unreal Engine's (UE)** software architecture. These encapsulate specific editor tools, runtime features, libraries, or other functionality in standalone units of code.

All projects and plugins have their own **primary module** by default, however, you can define other modules outside of these to organize your code.

This page provides an overview of how modules are structured and how they can benefit your Unreal Engine projects.



Unreal Engine modules are not related to C++ 20 modules.

Benefits of Using Modules

Organizing your project with modules provides the following benefits:

- Modules enforce good code separation, providing a means to encapsulate functionality and hide internal parts of the code.
- Modules are compiled as separate compilation units. This means only modules that have changed will need to compile, and build times for larger projects will be significantly faster.
- Modules are linked together in a dependency graph and limit header includes to code that is actually used, per the [Include What You Use \(IWYU\)](#) standard. This means modules that are not used in your project will be safely excluded from compilation.
- You can control when specific modules are loaded and unloaded at runtime. This provides a way to optimize the performance of your project by managing which systems are available and active.
- Modules can be included or excluded from your project based on certain conditions, such as which platform the project is being compiled for.

In summary, if you observe best practices with modules, your project's code will be better organized, will compile more efficiently, and will be more reusable than if you put all of your project's code into a single module.

Setting Up a Module

The following is an overview of how to build and implement a module from scratch. If you follow these steps, you will create a gameplay module separate from the primary module that your project includes by default.

1. Create a directory for your module at the top level of your project's **Source** folder. This directory should have the same name as your module.



You can place modules in any subdirectory within your Source folder, at any number of levels deep. This makes it possible to use subdirectories to group modules.

2. Create a `[ModuleName].Build.cs` file inside your module's root directory and use it to define dependencies with other modules. This makes it possible for the Unreal build system to discover your module.
3. Create **Private** and **Public** subfolders inside your module's root directory.

4. Create a `[ModuleName]Module.cpp` file in the Private subfolder for your module. Use this to provide methods for starting up and shutting down your module, as well as other common functions that Unreal Engine uses to manage modules.
5. To control how and when your module loads, add configuration information for your module in your `.uproject` or `.uplugin file`. This includes the name, type, compatible platforms, and loading phase of the module.
6. List your module as a dependency in the `Build.cs` file for any module that will need to use it. This may include the `Build.cs` file for your project's primary module.
7. Generate the solution files for your IDE any time you modify your `[ModuleName].Build.cs` file or move source files between folders. You can use any of the following methods to do this:
 - a. Run `GenerateProjectFiles.bat`.
 - b. Right-click the `.uproject` file for your project, then click **Generate Project Files**.
 - c. In the Unreal Editor, click **File > Refresh Visual Studio Project**.

For more details on these components and how to configure them, continue reading this page. For a more detailed walkthrough of setting up a module, refer to [Creating a Gameplay Module](#).

Understanding the Structure of a Module

All modules should be placed in the **Source** directory for either a plugin or project. The module's root folder should have the same name as the corresponding module.

There should also be a `[ModuleName].Build.cs` file for each module in its root folder, and its C++ code should be contained in **Private** and **Public** folders.

The following is an example of the recommended folder structure for a module:

- [ModuleName]
 - Private
 - [ModuleName]Module.cpp
 - All .cpp files and private headers
 - Public
 - All public headers
 - [ModuleName].Build.cs

Configuring Dependencies in the Build.cs File

The Unreal build system builds projects according to `Target.cs` files in your projects and the `Build.cs` files in your modules, not according to the solution files for your IDE.

The IDE solution is generated automatically when editing code, but the Unreal Build Tool (UBT) will ignore it when compiling projects.

All modules require a `[ModuleName].Build.cs` file placed in the module's root directory for the Unreal build system to recognize them.

Inside your `[ModuleName].Build.cs` file, you need to define your module as a class inherited from the `ModuleRules` class. Below is an example of a simple `Build.cs` file:

Sample ModuleTest.Build.cs File

```
1 using UnrealBuildTool;
2
3 public class ModuleTest: ModuleRules
4
5 {
6
7     public ModuleTest(ReadOnlyTargetRules Target) : base(Target)
8
9     {
10
11         PrivateDependencyModuleNames.AddRange(new string[] { "Core" });
12
13     }
14
15 }
16
```

 Copy full snippet

When configuring your `Build.cs` files, you will mainly use the `PrivateDependencyModuleNames` and `PublicDependencyModuleNames` lists. Adding module names to these lists will set the modules that are available to your module's code.

For example, if you add the "Slate" and "SlateUI" module names to your private dependencies list, you will be able to include Slate UI classes within your module.

Private and Public Dependencies

You should use the `PublicDependencyModuleNames` list if you use the classes from a module publicly, such as in a public `.h` file. This will make it possible for other modules that depend on your module to include your header files without issues.

You should put a module's name in the `PrivateDependencyModuleNames` list if they are only used privately, such as in `.cpp` files. Private dependencies are preferred wherever possible, as they can reduce your project's compile times.



You can make many dependencies private instead of public by using forward declarations in your header files.

Using the Private and Public Folders

If your module is a regular C++ module (meaning the `ModuleType` is not set to `External` in your `.uproject` or `.uplugin`), its C++ files should be placed in the Private and Public subfolders inside your module's root directory.

These do not have any relation to the `Private`, `Public`, or `Protected` access specifiers in your C++ code. Instead, they control the availability of the module's code to other modules. If you use these folders, all `.cpp` files should be placed in the Private folder. Header (`.h`) files should be placed in the Private and Public folders per the guidelines below.

If you place a header file in the Private folder, its contents will not be exposed to any modules outside its owning module. Classes, structs, and enums in this folder will be accessible to other classes inside the same module, but they will not be available to classes in other modules.

If you place a header in the Public folder, the Unreal build system will expose its contents to any other module that has a dependency on the current module. Classes in outside modules will be able to extend classes contained in the Public folder, and you will be able to create variables and references using classes, structs, and enums in the Public folder as well. The `Private`, `Public`, and `Protected` specifiers will take effect as normal for functions and variables.

If you are working on a module that will not be made a dependency for others, you do not need to use the `Private` and `Public` folders. Any code outside of these folders will behave as if it were Private. A typical example of this would be your game's primary module, which will likely be at the end of the dependency chain.

You can further organize your code by creating subfolders within the `Public` and `Private` folders. For any new folder you create within `Public`, create a corresponding folder with the same name in `Private`. Similarly, for any header file you place in `Public`, make sure its corresponding `.cpp` files are always in the corresponding folder in `Private`.

If you create new classes with the New Class Wizard in Unreal Editor, it will automatically ensure parallel construction between these folders.

Implementing the Module in C++

To expose a module to the rest of your C++ project, you need to create a class extending `IModuleInterface`, then provide that class to the `IMPLEMENT_MODULE` macro.

For the simplest implementation, you can create a `.cpp` file in the module's Private directory, and name it `[ModuleName]Module.cpp`, where `[ModuleName]` is the name of your module. Go on to call the `IMPLEMENT_MODULE` macro after all other `#include` declarations, and provide `FDefaultModuleImpl` as the class.

ModuleTestModule.cpp

```
1 #include "Modules/ModuleManager.h"
2
3 IMPLEMENT_MODULE(FDefaultModuleImpl, ModuleTest);
4
```

 Copy full snippet

`FDefaultModuleImpl` is an empty class that extends `IModuleInterface`. For a more detailed implementation, you can write your own class to implement in this `.cpp` file.

`IModuleInterface` features several functions that trigger when your module loads and unloads, similar to the `Startup` and `Shutdown` functions in the `GameInstance**` class.

Using Modules in Your Projects

Anytime you make a new Unreal Engine project or plugin, it will automatically set up a primary module of its own, which you can find in the project's `Source` folder. You can include outside modules in your project by adding them to the `Build.cs` file for your project's primary module.

For example, to use the Gameplay Tasks system in a project titled MyProject, you need to open `MyProject.Build.cs`, then add the "GameplayTasks" module as a dependency.

To optimize compilation speeds, Unreal Build Tool only compiles modules found in the dependency chain for your project. This means that if a module isn't included in any `Build.cs` files that are used by your project, that module will be skipped during compilation.

Controlling How Modules Load

Your `.uproject` and `.uplugin` files contain a `Modules` list defining which modules are included in your project and how they will load.

When you regenerate your project files, entries for your modules will be added to this list automatically if they are not already present, provided that you have included them in the dependency chain. Entries in this list will look similar to the following:

```
1  "Modules": [  
2  
3  {  
4  
5    "Name": "ModuleTest",  
6  
7    "Type": "Runtime",  
8  
9    "LoadingPhase": "Default",  
10  
11  },  
12  
13  {  
14  
15    "Name": "ModuleTestEditor",
```

```

16
17 "Type": "Editor",
18
19 }
20
21 ]
22

```


 Copy full snippet

Most gameplay modules will simply list their **Name**, while their **Type** will be set to `Runtime`. If their **LoadingPhase** is not defined it will be set to `Default`. There are a variety of other module types, loading phases, and additional parameters that control which platforms a module will and won't load on.

For information about the available module types, refer to the API documentation for [EHostType::Type](#).

The most common module types are `Runtime` and `Editor`, which are used for in-game classes and editor-only classes, respectively.

For more information about loading phases, refer to the API documentation for [ELoadingPhase::Type](#).

 While the `Default` loading phase is suitable for most gameplay modules in your project, plugins sometimes need to load earlier. If you frequently see errors from Unreal Editor trying to find C++ classes in a plugin, try setting them to `PreDefault`.

The other parameters used by this list include the following:

Parameter	Description
IncludelistPlatforms / ExcludelistPlatforms (Array)	Include or exclude the module for compiling on the listed platforms. Example platform strings include <code>Win32</code> , <code>Win64</code> , <code>Mac</code> , <code>Linux</code> , <code>Android</code> and <code>IOS</code> .
IncludelistTargets / ExcludelistTargets (Array)	Include or exclude the module for compiling on the listed build targets. The available build

Parameter	Description
	targets are <code>Game</code> , <code>Server</code> , <code>Client</code> , <code>Editor</code> , and <code>Program</code> .
IncludelistTargetConfigurations / ExcludelistTargetConfigurations (array)	Include or exclude the module for compiling on the listed build configurations. The available target configurations are <code>Debug</code> , <code>DebugGame</code> , <code>Development</code> , <code>Shipping</code> , and <code>Test</code> .
IncludelistPrograms / ExcludelistPrograms (array):	Include or exclude the module for compiling under specific program names.
AdditionalDependencies (array):	Specifies additional dependencies needed by the module. You should specify this in your <code>Build.cs</code> files instead.