# Gameplay Modules

Collections of gameplay classes belonging to a game project compiled into DLLs.

In the same manner that the engine itself is composed of a collection of modules, each game is made up of one or more gameplay modules. These are similar to packages in previous versions of the engine in that they are containers for a collection of related classes. In Unreal Engine, since gameplay is all done in C++, modules are actually DLLs instead of proprietary package files.

> If your game project is not already created, see Creating a Game Project for instructions on creating and setting up a project for your game.

# Module Creation

Gameplay modules must contain, at the very least, a header file (.h), C++ file (.cpp), and a build file (*.Build.cs). The header file must be located in the Public folder of the module's directory, i.e. [GameName] \Source\ [ModuleName]\Public. This file includes any header files - including the module's auto-generated headers - necessary to compile the classes contained within the module.

```
#include "Engine.h"
#include "EnginePrivate.h"
#include "<ModuleName>Classes.h"

```

Copy full snippet

The C++ file is placed in the Private folder of the module's directory, i.e. [GameName] \Source\ [ModuleName]\Private, registers and implements the module.

At least one module in your game must be registered using `IMPLEMENT_PRIMARY_GAME_MODULE`. Additional modules can use the alternative `IMPLEMENT_GAME_MODULE` method. See the Multiple Gameplay Modules section for more details on using more than one gameplay module in a game project.

```
// Include our game's header file
#include "<ModuleName>.h"

// Designate the module as primary
IMPLEMENT_PRIMARY_GAME_MODULE( <ModuleName>, "<GameName>" );

```

Copy full snippet

The build file is placed in the root directory of the gameplay module, i.e. [GameName] \Source\ [ModuleName], and it defines certain pieces of information used by the UnrealBuildTool to compile the module.

```
using UnrealBuildTool;

public class <ModuleName> : ModuleRules
{
public <ModuleName>( TargetInfo Target )
{
PublicDependencyModuleNames.AddRange( new string[] { "Core", "Engine" } );
PrivateDependencyModuleNames.AddRange( new string[] { "RenderCore" } );
}
}

```

Copy full snippet

# INI File Setup

Because the new gameplay modules will have UObject code, some configuration is necessary.

1. The module needs to be added in several areas of the DefaultEngine.ini file:

   The `EditPackages` array in the `[UnrealEd.EditorEngine]` section:

   ```
   [UnrealEd.EditorEngine]
   +EditPackages=<ModuleName>

   ```

   Copy full snippet

   The `[Launch]` section:

```
[Launch]
Module=<ModuleName>

```

Copy full snippet

The `NativePackages` array of the `[/Script/Engine.UObjectPackages]` section:

```
[/Script/Engine.UObjectPackages]
+NativePackages=<ModuleName>
```

Copy full snippet

# Multiple Gameplay Modules

There are philosophical choices about game DLL modularity. Splitting a game up into a lot of DLL files may be more trouble than it is worth, but this is a decision to be made by each individual team based on their needs and principles. Using multiple gameplay modules will result in better link times and faster code iteration, but with more modules you will need to deal with DLL exports and/or interface classes more often. This trade-off is the correct one for engine and editor code, but it is questionable for gameplay.

You can create a primary game module, then any number of additional game-specific modules. You can create *.Build.cs files for these new modules, then add references to these modules to your game's Target.cs file (OutExtraModuleNames array). Be sure to use the appropriate macro for the game modules in the C++ code. At least one module must use the `IMPLEMENT_PRIMARY_GAME_MODULE` macro, while all others should use the `IMPLEMENT_GAME_MODULE` macro. UBT should then automatically discover the modules and compile additional game DLL files.

# Limitations

We do support creating modules that are cross-dependent (both export and import functions and data from each other -- e.g., Engine and UnrealEd modules), but this is not ideal for compile-times and may sometimes cause problems with static initialization of variables. Gameplay modules that are not cross-dependent are harder to design and maintain, but the code may be cleaner for it.