UE4 Modules

Ari Arnbjörnsson

Get these slides at

HOUSEMARQUE

https://ari.games

@flassari

What are modules?

Collections of classes (think DLLs).

UE4’s code is split among 1000+ modules.

Why use modules?

- Better code practices / encapsulation.
- Re-use code easily.
- Only ship the modules you use.
- Faster compile and linking time.
- Better control of what gets loaded when.

Creating modules: **B.U.I.L.D.**

Build

Use

Implement

Load

Depend

Creating modules: **B.U.I.L.D.**

Build

Use

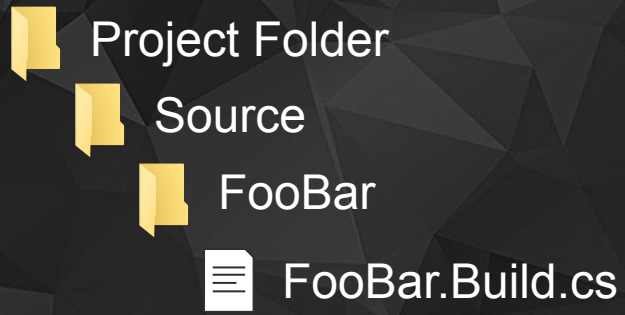
Implement

Load

Depend

HOUSEMARQUE

Build



Let’s create module “FooBar”!

HOUSEMARQUE

Build

`[YourModuleName].Build.cs`

- Describes how to build your module.
- Defines your module's dependencies.
- And more...

Projects are built according to `.Target.cs` and `.Build.cs` files, not solution files.

- UBT ignores solution files.
  - They're mostly for your convenience.
- To generate solution files:
  - Run `GenerateProject.bat`.
  - Right click [ProjectName].uproject -> Generate [..] Project Files.
  - Click File -> Refresh [..] Project.
  - Good practice after changing `.Build.cs` files or moving source files around.

Build

Minimum implementation

```
using UnrealBuildTool;

public class FooBar : ModuleRules
{
  public FooBar(ReadOnlyTargetRules Target) : base(Target)
  {
     PrivateDependencyModuleNames.AddRange(new string[] {"Core"});
  }
}
```

Creating modules: **B.U.I.L.D.**

Build

Use

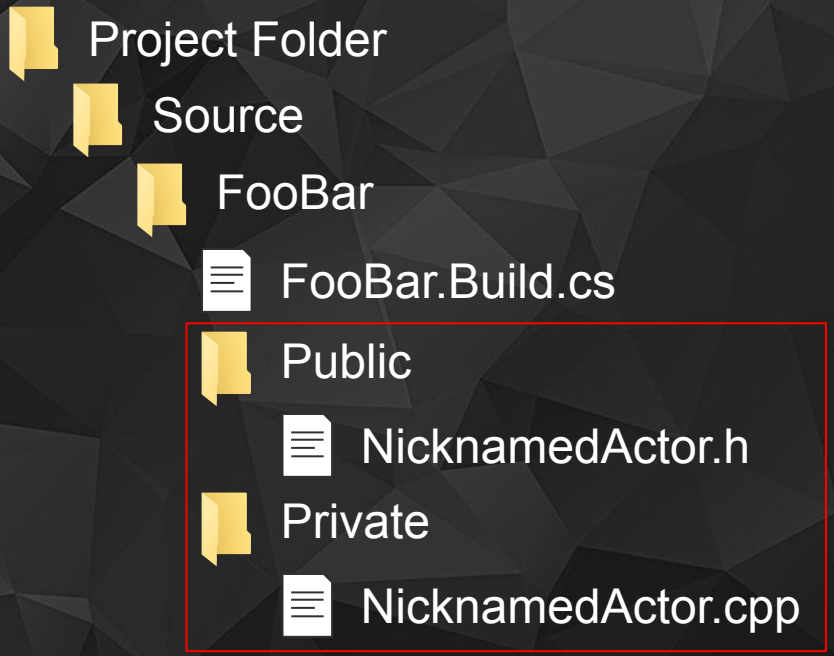
Implement

Load

Depend

HOUSEMARQUE

Use


Project Folder
Source
FooBar
FooBar.Build.cs
Public
NicknamedActor.h
Private
NicknamedActor.cpp

Use

Code from your module is not exposed to other modules by default, you need to mark each function or class explicitly for export.

Headers not meant for use by other modules can go into the `Private/` folder.

No need for `private/public` folders for your primary game module if you don’t plan on having other modules depend on it.

Use

NicknamedActor.h

```
#pragma once
#include "GameFramework/Actor.h"
#include "CoreMinimal.h"
#include "NicknamedActor.generated.h"

UCLASS(Blueprintable)
class ANicknamedActor : public AActor
{
	GENERATED_BODY()

public:
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	FString Nickname;

	UFUNCTION(BlueprintCallable)
	void SayNickname();
};
```

Use

NicknamedActor.h

```
#pragma once
#include "GameFramework/Actor.h"
#include "CoreMinimal.h"
#include "NicknamedActor.generated.h"

UCLASS(Blueprintable)
class ANicknamedActor : public AActor
{
	GENERATED_BODY()

public:
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	FString Nickname;

	UFUNCTION(BlueprintCallable)
	void SayNickname();
};
```

Exposed to the engine/editor only.

C++ classes in other modules can’t cast to it or call its functions.

Use

NicknamedActor.h

```
#pragma once
#include "GameFramework/Actor.h"
#include "CoreMinimal.h"
#include "NicknamedActor.generated.h"

UCLASS(Blueprintable, MinimalAPI)
class ANicknamedActor : public AActor
{
	GENERATED_BODY()

public:
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	FString Nickname;

	UFUNCTION(BlueprintCallable)
	void SayNickname();
};
```

Exposes type information to other modules. Other modules can:

- Cast to the class.
- Extend the class.
- Use inline functions.

Use

NicknamedActor.h

```
#pragma once
#include "GameFramework/Actor.h"
#include "CoreMinimal.h"
#include "NicknamedActor.generated.h"

UCLASS(Blueprintable, MinimalAPI)
class ANicknamedActor : public AActor
{
	GENERATED_BODY()

public:
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	FString Nickname;

	UFUNCTION(BlueprintCallable)
	FOOBAR_API void SayNickname();
};
```

Exposes a function to other modules.

`[YourModuleName]_API`.

Use

NicknamedActor.h

```
#pragma once
#include "GameFramework/Actor.h"
#include "CoreMinimal.h"
#include "NicknamedActor.generated.h"

UCLASS(Blueprintable)
class FOOBAR_API ANicknamedActor : public AActor
{
	GENERATED_BODY()

public:
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	FString Nickname;

	UFUNCTION(BlueprintCallable)
	void SayNickname();
};
```

Exposes everything in the class.

Use

```
fatal error C1083: Cannot open include file: 'GameFramework/Actor.h': No such file or directory
```

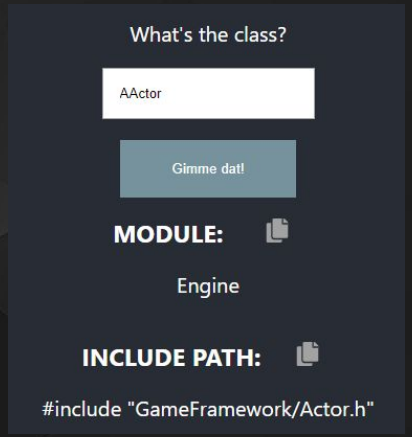
`Engine/Source/Runtime/Engine/Classes/GameFramework/Actor.h`

Module name — Include path

To use classes from other modules you need to:

- Include the header in your `.h/.cpp` file.
  - Whole path, starting from `Classes/` or `Public/` folder.
  - (`Classes/` folder is legacy, always put new classes in `Public/`)
- Include the header's module as a dependency of your module.
  - From the header's path, the folder before `Classes/` or `Public/`.

Check the header's path, docs, or use tools like http://classifier.celdevs.com



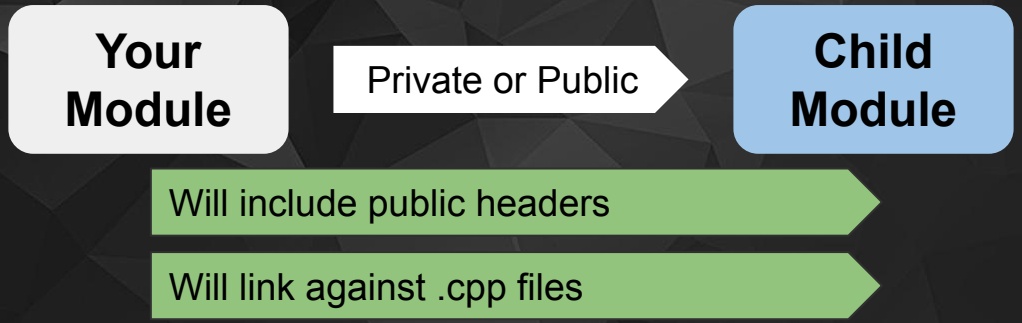
Use

```
using UnrealBuildTool;

public class FooBar : ModuleRules
{
  public FooBar(ReadOnlyTargetRules Target) : base(Target)
  {
     PublicDependencyModuleNames.AddRange(new string[] {"Engine"});
     PrivateDependencyModuleNames.AddRange(new string[] {"Core"});
  }
}
```

Use

Private / Public Dependency

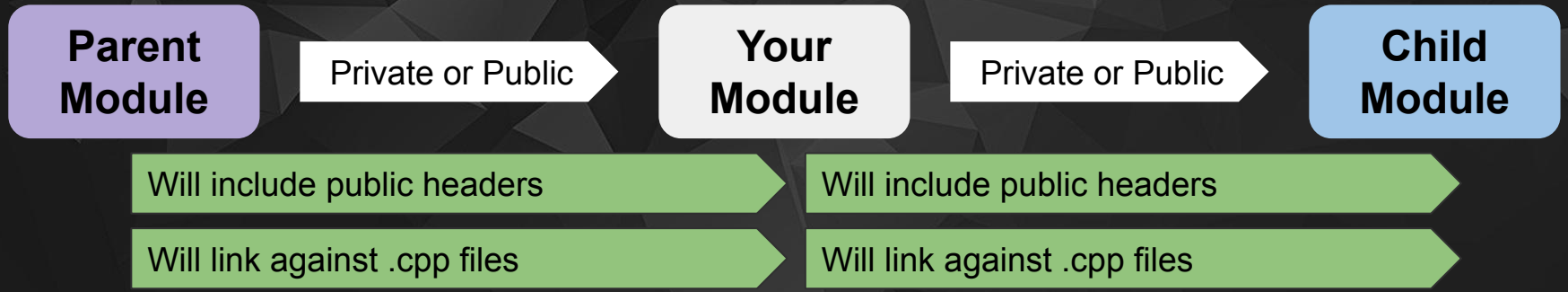


Adding a `Child Module` as either a private or public dependency:

- Adds include paths to `Child Module`'s public headers.
- Links `Child Module`'s exposed classes/functions/variables against your `.cpp` files.

Use

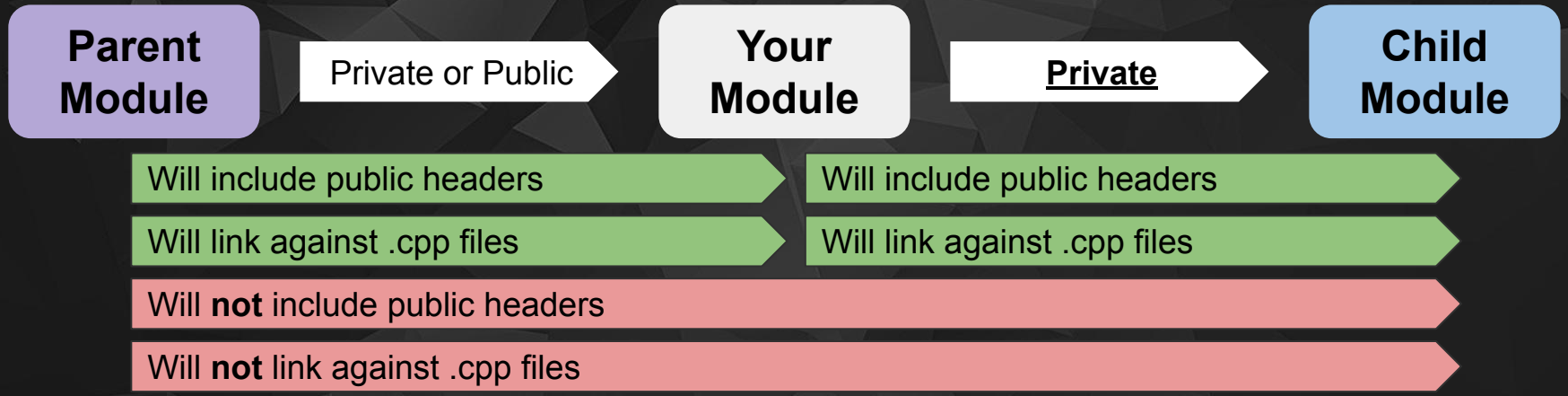
Private / Public Dependency



A `Parent Module` can also add **`Your Module`** as a private or public dependency.

Use

Private / Public Dependency



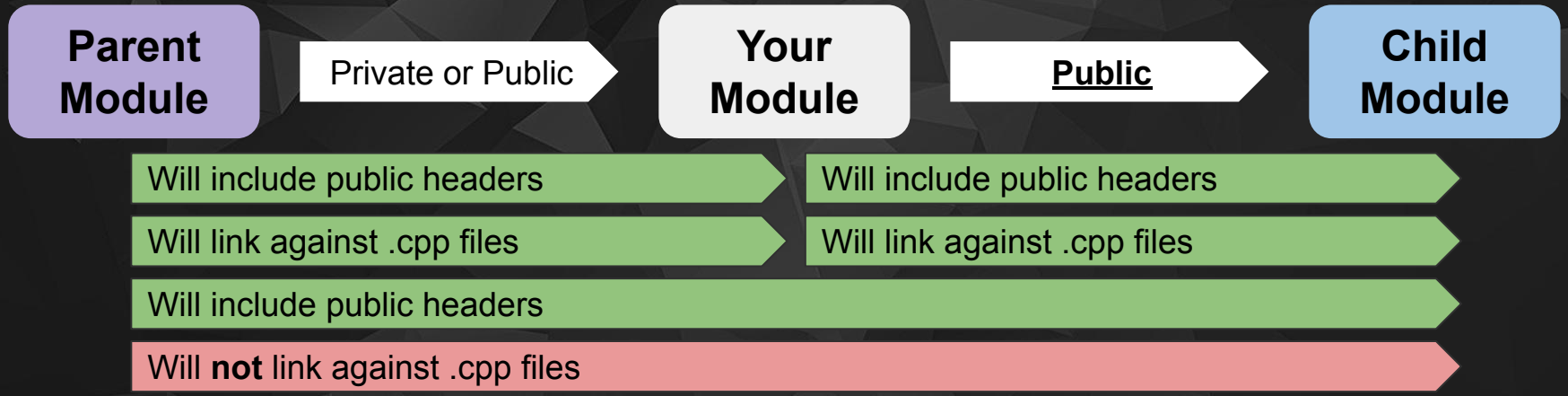
If `Your Module` privately depends on `Child Module` then `Parent Module` will not include headers or link against `Child Module`'s .cpp files.

If `Parent Module` includes `Your Module`'s headers which include `Child Module`'s headers, you'll get a File Not Found error.

HOUSEMARQUE

Use

Private / Public Dependency



If **`Your Module`** publicly depends on **`Child Module`** then **`Parent Module`** will include headers from **`Child Module`**, but still not link against it.

Every module must always depend directly on the modules they want to link against!

Use

Private / Public Dependency

If only your module's `.cpp` or private `.h` files use a dependency's headers, make it private. Private dependencies are preferred as they reduce compile times.

Forward declare when you can (when convenient).

Missing module dependencies will produce compiler or linking errors.

Use

Demystifying linker errors

SomeActor.cpp

```
void ASomeActor::Test()
{
    NicknamedActor->SayNickname();
}
```

Error!

```
SomeActor.cpp.obj : error LNK2019: unresolved external symbol "public: void __cdecl
ANicknamedActor::SayNickname(void)" (?SayNickname@ANicknamedActor@@QEAAXXZ)
referenced in function "public: void __cdecl ASomeActor::Test(void)" (?Test@ASomeActor@@QEAAXXZ)
```

Use

Demystifying linker errors

SomeActor.cpp

```
void ASomeActor::Test()
{
     NicknamedActor->SayNickname();
}
```

Error!

```
SomeActor.cpp.obj : error LNK2019: unresolved external symbol
"public: void __cdecl ANicknamedActor::SayNickname(void)"
     (?SayNickname@ANicknamedActor@@QEAAXXZ)
referenced in function
"public: void __cdecl ASomeActor::Test(void)"
     (?Test@ASomeActor@@QEAAXXZ)
```

Use

Demystifying linker errors

SomeActor.cpp

```
void ASomeActor::Test()
{
    NicknamedActor->SayNickname();
}
```

Error!

```
SomeActor.cpp.obj : error LNK2019: unresolved external symbol
"public: void __cdecl ANicknamedActor::SayNickname(void)"
    (?SayNickname@ANicknamedActor@@QEAAXXZ)
referenced in function
"public: void __cdecl ASomeActor::Test(void)"
    (?Test@ASomeActor@@QEAAXXZ)
```

The file that the error occurred in

Use

Demystifying linker errors

SomeActor.cpp

```
void ASomeActor::Test()
{
    NicknamedActor->SayNickname();
}
```

Error!

```
SomeActor.cpp.obj : error LNK2019: unresolved external symbol
"public: void __cdecl ANicknamedActor::SayNickname(void)"
    (?SayNickname@ANicknamedActor@@QEAAXXZ)
referenced in function
"public: void __cdecl ASomeActor::Test(void)"
    (?Test@ASomeActor@@QEAAXXZ)
```

What it couldn't find

HOUSEMARQUE

Use

Demystifying linker errors

SomeActor.cpp

```
void ASomeActor::Test()
{
    NicknamedActor->SayNickname();
}
```

Error!

```
SomeActor.cpp.obj : error LNK2019: unresolved external symbol
"public: void __cdecl ANicknamedActor::SayNickname(void)"
    (?SayNickname@ANicknamedActor@@QEAAXXZ)
referenced in function
"public: void __cdecl ASomeActor::Test(void)"
    (?Test@ASomeActor@@QEAAXXZ)
```

Where it referenced it

Use

Demystifying linker errors

SomeActor.cpp

```
void ASomeActor::Test()
{
    NicknamedActor->SayNickname();
}
```

Error!

```
SomeActor.cpp.obj : error LNK2019: unresolved external symbol
"public: void __cdecl ANicknamedActor::SayNickname(void)"
    (?SayNickname@ANicknamedActor@@QEAAXXZ)
referenced in function
"public: void __cdecl ASomeActor::Test(void)"
    (?Test@ASomeActor@@QEAAXXZ)
```

Almost always because of a missing [ModuleName]_API specifier or module dependency.

HOUSEMARQUE

Creating modules: **B.U.I.L.D.**

Build

Use

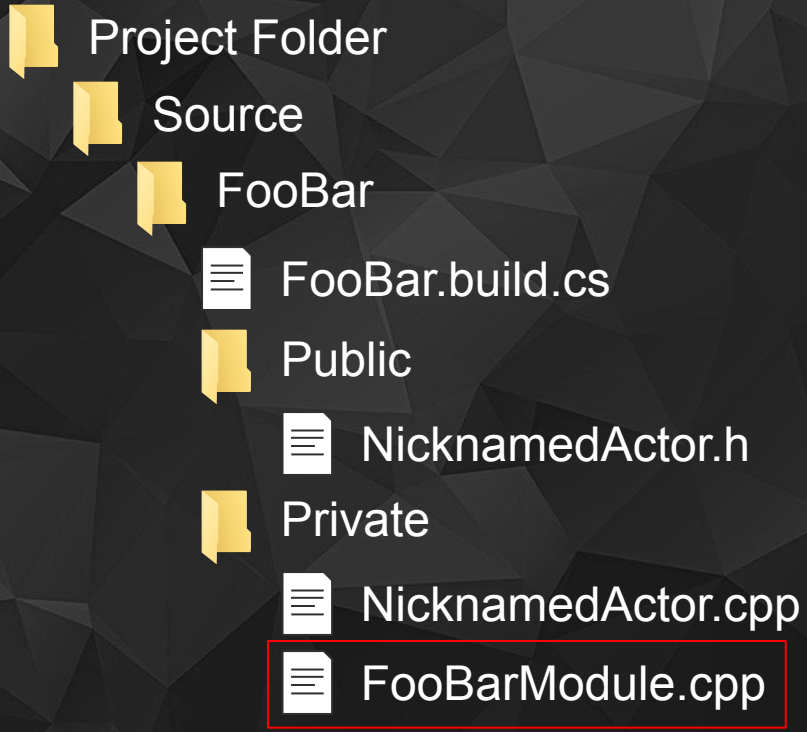
Implement

Load

Depend

HOUSEMARQUE

Implement


Project Folder
Source
FooBar
FooBar.build.cs
Public
NicknamedActor.h
Private
NicknamedActor.cpp
FooBarModule.cpp

Implement

Call `IMPLEMENT_MODULE` after any declarations in `[YourModuleName]Module.cpp`.

- By convention. Could in theory be anywhere in any `.cpp` file in your module.
- Exposes your module's "main class" to the rest of the engine.

```
#include "Modules/ModuleManager.h"
IMPLEMENT_MODULE(FDefaultModuleImpl, FooBar);
```

ModuleManager is in "Core" module (that's why it's always a minimum dependency).

"Main class" should extend `IModuleInterface` (Modules/ModuleInterface.h)

- A module main class is a class that shares the lifetime of the module itself.
- Can be your own custom class, can also just be the empty `FDefaultModuleImpl`.
- `IMPLEMENT_GAME_MODULE` for game modules.
- `IMPLEMENT_PRIMARY_GAME_MODULE` for the primary game module.

Implement

Creating your own module main class enables you to implement:

- `virtual void StartupModule()`
  - Called right after the module DLL has been loaded and the module object has been created.
- `virtual void ShutdownModule()`
  - Called before the module is unloaded, right before the module object is destroyed.

See other overridable functions in `Core/Public/Modules/ModuleInterface.h`.

Other code can use a module's main class from anywhere.

```
FModuleManager::Get().LoadModuleChecked<FFooBarModule>(TEXT("FooBar")).DoFoo();
```

Implement

Module Lifetime

During shutdown, modules are destroyed in the reverse order they were created.

That means you can load other modules during your module's startup.
E.g. `FModuleManager::Get().LoadModuleChecked(TEXT("HTTP"));`

You can then be sure any modules loaded in `StartupModule()` will be available in `ShutdownModule()` as well.

Implement

Gameplay modules

Usually you have only one gameplay module, your primary gameplay module.

You should only mark a module as a gameplay module when it depends on another gameplay module.

- Sets up hot-reloading support.

You can mark a module as gameplay modules:

- In that module's main class, override `virtual bool IsGameModule() const` from `IModuleInterface` to return true.
- Call `IMPLEMENT_GAME_MODULE` instead of `IMPLEMENT_MODULE`.

Gameplay modules have extra hot-reloading overhead, avoid when possible.

Creating modules: **B.U.I.L.D.**

Build

Use

Implement

Load

Depend

HOUSEMARQUE

Load

Module Descriptor

Modules need to be described in the `.uproject` or `.uplugin` file.
Defines when the module is loaded and on what targets / platforms.

```
"Modules": [
    {
        "Name": "FooBar",
        "Type": "Runtime",
    }
]
```

See more info in ModuleDescriptor.h/.cpp in Projects module.

Runtime	CookedOnly
Editor	UncookedOnly
Program	RuntimeNoCommandlet
EditorAndProgram	RuntimeAndProgram
ServerOnly	EditorNoCommandlet
ClientOnly	DeveloperTool
Developer	ClientOnlyNoCommandlet

Load

Module Descriptor

Modules need to be described in the `.uproject` or `.uplugin` file.
Defines when the module is loaded and on what targets / platforms.

```
"Modules": [
    {
        "Name": "FooBar",
        "Type": "Runtime",
        "LoadingPhase": "Default"
    }
]
```

See more info in ModuleDescriptor.h/.cpp in Projects module.

EarliestPossible	PreDefault
PostConfigInit	**Default**
PostSplashScreen	PostDefault
PreEarlyLoadingScreen	PostEngineInit
PreLoadingScreen	None

Load

Module Descriptor

Modules need to be described in the `.uproject` or `.uplugin` file.
Defines when the module is loaded and on what targets / platforms.

```
"Modules": [
    {
        "Name": "FooBar",
        "Type": "Runtime",
        "LoadingPhase": "Default",
        "WhitelistPlatforms": ["Win64"]
    }
]
```

See more info in ModuleDescriptor.h/.cpp in Projects module.

[Black/White]listPlatforms
[Black/White]listTargets
[Black/White]listTargetConfigurations
[Black/White]listPrograms

Creating modules: **B.U.I.L.D.**

Build

Use

Implement

Load

Depend

HOUSEMARQUE

Depend

Only modules in the dependency chain get compiled.

You can add your module to the chain:

- If another module depends on it: in its `.Build.cs` file:
  `[Private/Public]DependencyModuleNames` arrays.
  - Preferred if you don't want your module to be compiled if nothing depends on it.
- If no other module depends on it: in your `.Target.cs` files:
  `ExtraModuleNames` array.
  - When it should compile even if no other module depends on it.
  - Usually the case for the primary game module and custom editor modules.

Creating modules: **B.U.I.L.D.**

Build

Use

Implement

Load

Depend

Extra content!

Precompiled Headers

Include What You Use

DefaultBuildSettings

Module Logging

Plugins

Precompiled Headers

PCH for short.

- Normal header files aren’t compiled on their own.
  - They are included and compiled into every `.cpp` file.
- Lots of duplicate compiling.
- If you always include the same x headers, why not compile them only once?
- Enter PCHs.
  - Define one header file that includes your most common header files.
  - Gets compiled before other files.
  - Doesn’t compile again unless any of its included headers change.
    - But then all other cpp files in the module need to be compiled.
    - Best for engine headers or very rarely changing code.

Precompiled Headers

Private PCH

- A custom PCH you create yourself for your module.
- Define it in your `.Build.cs` file.
  - `PrivatePCHHeaderFile = "FooBarPrivatePCH.h";`
- Never include it yourself in your `.h/.cpp` files.
  - UBT will automatically inject it for all compiled files in your module.
- PCHs should be considered an optimization layer.
  - Don’t treat it as an easy “include all”, still include only what you use.
  - Your code should compile even if PCHs are turned off.

Precompiled Headers

Shared PCHs

- Instead of defining your own PCH you can use a shared PCH.
  - A shared PCH is when a module defines a PCH for other depending modules to use.
  - Exists in some foundational often-used UE4 modules.
    - UnrealEd, Engine, Slate, CoreUObject, and Core to be specific.
  - Only engine modules can create shared PCHs.
- A shared PCH will only get compiled once.
  - Even if multiple modules use it.
- UE4 will choose the “highest priority” shared-PCH to use for you.
  - Sorted by how many other modules with a shared PCH it depends on.
  - The list above is sorted by that priority.

Precompiled Headers

When to use what PCHs?

You have three options for precompiled headers:

- Create your own private PCH.
  - Good for modules with very big codebases.
    - Often the case with the primary game module on bigger games.
  - You have to decide what to put in there and how to balance it.
- Use a shared engine PCH.
  - Good for all smaller modules.
- Don’t use a PCH.
  - Not really practical.

Precompiled Headers

PCH build settings

Set in the module's `.Build.cs` file.

Two relevant settings:

- `PCHUsage` property, takes a `PCHUsageMode` enum.
- `PrivatePCHHeaderFile` property, string path to the header.

Precompiled Headers

`PCHUsageMode`: which setting to use?

- `Default`
- `NoSharedPCHs`
- `UseSharedPCHs`
- `UseExplicitOrSharedPCHs`
- `NoPCHs`

Precompiled Headers

`PCHUsageMode`: which setting to use?

- `Default`
- `UseSharedPCHs`
- `UseExplicitOrSharedPCHs` <- Always use this one.
- `NoPCHs`

LEGACY

Uses a shared PCH by default, or a private PCH if it's set via `PrivatePCHHeaderFile`.

Is default in new projects from 4.24.2 onwards. This enum will probably get phased out in the future.

HOUSEMARQUE

Include What You Use

IWYU for short.

- Only **I**nclude **W**hat **Y**ou **U**se.
  - `.h` and `.cpp` files should only include their required dependencies.
- Will warn you if you include a monolithic header (`Engine.h`, `UnrealEd.h`, etc).
  - They're legacy, just stop using them.
- Will warn if `.cpp` files don't include their matching `.h` file first.
  - Is to warn older code that includes a PCH first, which used to be the standard.
  - Instead define the private PCH with `PrivatePCHHeaderFile` in your `.Build.cs` file.
- Your `.cpp` files should compile fine without PCHs and in non-Unity builds.
  - Unity builds concatenate many `.cpp` files into bundles for more efficient compiling.
    - Can sometimes hide missing include errors.

Include What You Use

Optional up to 4.23.

- Turn on by setting `PCHUsage` to `PCHUsageMode.UseExplicitOrSharedPCHs` in your `.Build.cs` file.

Enabled by default in 4.24.2.

- Comes with `DefaultBuildSettings` defaulting to `BuildSettingsVersion.V2` in new projects.
  - Also affects other settings.

DefaultBuildSettings

`BuildSettingsVersion.V2` is the new default in 4.24.2.

Older projects can upgrade by setting `DefaultBuildSettings = BuildSettingsVersion.V2;` in `.Target.cs` or `.Build.cs`.

- `PCHUsage` gets set to `PCHUsageMode.UseExplicitOrSharedPCHs`;
  - Enables IWYU-style build settings.
- `bLegacyPublicIncludePaths` gets set to false;
  - Omits subfolders from public include paths to reduce compiler command-line length.
  - Means you now have to have every include path correct!
  - **Can be a huge refactor for big projects!**
  - Use UAT's `RebasePublicIncludePaths` command-line tool to help the migration.
- `ShadowVariableWarningLevel` gets set to `WarningLevel.Error`;
  - Treats shadowed variable warnings as errors.

Module Logging

It’s good practice to have your module use its own log category for easier filtering.

Declare the log category type.

- `DECLARE_LOG_CATEGORY_EXTERN(CategoryName, DefaultVerbosity, CompileTimeVerbosity);`
  - Commonly (`Log[ModuleName]`, `Display`, `All`), see `Logging/LogVerbosity.h` for more.
- Declares a category class that extends `FLogCategory`.
- Most practical to put it in its own header file.

Initialize the log category with your module.

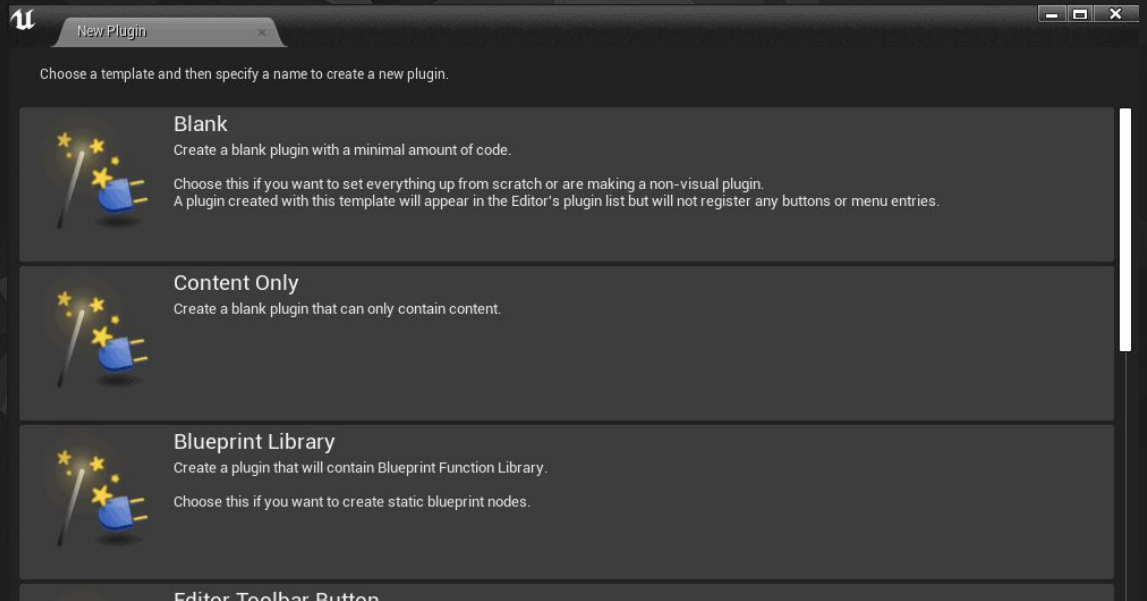
- `DEFINE_LOG_CATEGORY(CategoryName);`
- Instantiates an instance of that log category class, which registers itself with the log suppression system in the constructor.
- Put it in the same place where you called `IMPLEMENT_MODULE`.

Use the log category.

- `UE_LOG(LogFooBar, Display, TEXT("A wild log appeared!"));`

Plugins

- Basically just a collection of modules.
- Example plugin in `Engine/Plugins/Developer/BlankPlugin`.
- Select Edit -> Plugins -> New Plugin which has many good starting options.



HOUSEMARQUE

Creating modules: **B.U.I.L.D.**

Build	Create a `.Build.cs` file that builds the module.
Use	Expose functions/classes via macros to **use** in engine or other modules.
Implement	Call `IMPLEMENT_MODULE` in `[module name]Module.cpp`.
Load	Set when the module **loads** in the `.uproject` or `.uplugin` file.
Depend	Add it as a **dependency** of another module or target so it gets compiled.

HOUSEMARQUE

Questions?

- Build
- Use
- Implement
- Load
- Depend

- Precompiled Headers
- Include What You Use
- DefaultBuildSettings
- Module Logging
- Plugins

Get these slides at

https://ari.games

HOUSEMARQUE

@flassari