Asset Management

Asset Loading and Unloading

Unreal Engine (UE) handles Asset loading and unloading automatically. This provides developers a method to communicate with the Engine on when each asset is going to be needed. However, there may be cases where you want precise control over when and how Assets are discovered, loaded, and audited. For these cases, the **Asset Manager** can help. The Asset Manager is a unique, global object that exists in the Editor and packaged games. It can be overridden and customized for any project and provides a framework for managing Assets that can divide content into chunks that make sense in the context of your project, without losing the advantages of Unreal's loose package architecture. The Asset Manager provides a set of tools to help audit disk and memory usage, giving you the information you need to optimize the organization of your Assets for Cooking and Chunking when deploying your game.

Primary and Secondary Assets

The Asset management system in Unreal breaks all Assets into two types, **Primary Assets** and **Secondary Assets**. Primary Assets can be manipulated directly by the Asset Manager from their **Primary Asset ID**, which is obtained by calling the function `GetPrimaryAssetId`. In order to designate Assets made from a specific `UObject` class as Primary Assets, you can override the `GetPrimaryAssetId` function to return a valid `FPrimaryAssetId` structure. Secondary Assets are not handled directly by the Asset Manager, but instead are loaded automatically by the Engine in response to being referenced or used by Primary Assets. By default, only `UWorld` Assets (levels) are Primary, all other Assets are Secondary. In order to make a Secondary Asset into a Primary Asset, the `GetPrimaryAssetId` function for its class must be overridden to return a valid `FPrimaryAssetId` structure. A Primary Asset ID has two parts, a unique Primary Asset Type that identifies a group of Assets, and the name of that specific Primary Asset, which defaults to the Asset's name as it appears in the **Content Browser**.

Blueprint Class Assets and Data Assets

The Asset Manager handles two different types of Assets: Blueprint Classes, and non-Blueprint Assets like Levels and Data Assets (Asset instances of the `UDataAsset` class). Each Primary Asset Type is associated with a certain base class, and specifies whether it stores blueprint classes or not in the configuration described below.

Blueprint Classes

To create a new Blueprint Primary Asset, go to the **Content Browser** and create a new Blueprint class that is a descendant of a class that overrides the `GetPrimaryAssetId` function. This base class could be Primary Data Asset or any of its children, or an Actor subclass that overrides `GetPrimaryAssetId`. To access Blueprint Primary Assets, call functions like `GetPrimaryAssetObjectClass` from C++ code, or use Blueprint Asset Manager functions that include the word "Class" in their names. Once you have the class, you can treat it like any other Blueprint class and use it to spawn new instances, or you can use the Get Defaults function to access read-only data from the Class Default Object associated with the Blueprint.

For Blueprint classes that you never need to instantiate, you can store your data in a Data-Only Blueprint that inherits from `UPrimaryDataAsset`. You can also derive child classes, including Blueprint-based children, from your base class. For example, you could create a base class like `UMyShape` that extends

`UPrimaryDataAsset` in C++, then make a Blueprint-based subclass called `BP_MyRectangle` with `UMyShape` as its parent, and then a Blueprint-based child of `BP_MyRectangle` called `BP_MySquare`. With the default settings, the PrimaryAssetId of the last class you created would be `MyShape:BP_MySquare`.

Non-Blueprint Assets

You can use non-Blueprint Assets in cases where your Primary Asset Type does not need to store Blueprint data. Non-Blueprint Assets are simpler to access in code, and are more memory-efficient. To create a new non-Blueprint Primary Asset in the editor, create a new Data Asset from the advanced content browser window, or use the custom UI for creating things like new levels. Creating an Asset in this way is not the same as creating a Blueprint class; the Asset you create is an instance of a class, and not a class itself. To access the class, load them with C++ functions like `GetPrimaryAssetObject`, or the Blueprint functions without Class in their name. Once loaded, you can access them directly to read their data.

Because these Assets are instances rather than classes, you cannot inherit classes or other Assets from them. If you need to do this, for example, if you want to create a child Asset that inherits its parent's values except for those it explicitly overrides, you should use a Blueprint Class instead.

Asset Manager and Streamable Managers

The Asset Manager object is a singleton that manages discovery and loading of Primary Assets. The base Asset Manager class that is included in the Engine provides basic management functionality, but can be extended to suit project-specific needs. A **Streamable Manager** structure, contained within the Asset Manager, performs the actual work of asynchronously loading objects, as well as keeping objects in memory via the use of **Streamable Handles** until they are no longer needed and can be unloaded. Unlike the singleton Asset Manager, there are multiple Streamable Managers in different parts of the Engine and for different use cases.

Asset Bundles

An **Asset Bundle** is a named list of specific Assets associated with a Primary Asset. Asset Bundles are created by tagging the `UPROPERTY` section of a `TSoftObjectPtr` or `FStringAssetReference` member of a `UObject` with the "AssetBundles" meta tag. The value of the tag will indicate the name of the bundle where the Secondary Asset should be stored. For example, the following Static Mesh Asset, stored in the member variable called `MeshPtr`, will be added to the Asset Bundle called "TestBundle" when the UObject is saved:

```
/** Mesh */
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = Display, AssetRegistrySearchable, meta = (AssetBundles = "TestBundle"))
TSoftObjectPtr<UStaticMesh> MeshPtr;

```

Copy full snippet

A second way to use Asset Bundles is by registering them at runtime with the project's Asset Manager class. In this case, a programmer will need to write code that fills in an `FAssetBundleData` structure and then passes that structure to the Asset Manager. You can do this by either by overriding the `UpdateAssetBundleData` function, or calling `AddDynamicAsset` with the Primary Asset ID you want to associate with the Secondary Assets in the bundle.

Registering and Loading Primary Assets From Disk

Most Primary Assets are visible in the **Content Browser** and exist as Asset files stored on disk, so that they can be edited by artists or designers. The easiest way for a programmer to create a class that can be used in this way is to inherit from the `UDataAsset` child class, `UPrimaryDataAsset`, which has built-in functionality for loading and saving Asset Bundle data. If you want to use a different base class, such as `APawn`, studying `UPrimaryDataAsset` is useful, since it is a minimal example of the features that you need to implment to get Asset Bundles working for your class. The following class is an example of how to specify a type of zone in a hypothetical game; the zone type tells the game what art Assets to use when building the visual representation of the world in the game's overall map screen:

```
/** A zone that can be selected by the user from the map screen */
UCLASS(Blueprintable)
class MYGAME_API UMyGameZoneTheme : public UPrimaryDataAsset
{
GENERATED_BODY()

/** Name of the zone */
UPROPERTY(EditDefaultsOnly, Category=Zone)
FText ZoneName;

/** The Level that will be loaded when entering this zone */
UPROPERTY(EditDefaultsOnly, Category=Zone)
TSoftObjectPtr<UWorld> LevelToLoad;

/** The Blueprint class used to represent this zone on the map */
UPROPERTY(EditDefaultsOnly, Category=Visual, meta=(AssetBundles = "Menu"))
TSoftClassPtr<class AGameMapTile> MapTileClass;
};

```

Copy full snippet

Because this class inherits from `UPrimaryDataAsset`, it has a working version of `GetPrimaryAssetId` that uses the Asset's short name and native class. For example, a `UMyGameZoneTheme` saved under the name "Forest" would have a Primary Asset ID of "MyGameZoneTheme:Forest". Whenever a `UMyGameZoneTheme` Asset is saved in the Editor, the `AssetBundleData` member of `PrimaryDataAsset` will be updated to include it as a Secondary Asset.

Registering and loading our Primary Assets requires the following actions:

1. **Make the Engine aware of the project's custom Asset Manager class, if one exists.** You only need to override the default Asset Manager class, `UAssetManager`, if your project requires special functionality. If your project does not require special functionality, you can skip this step. To override it,

modify the project's `DefaultEngine.ini` file, and set the `AssetManagerClassName` variable under the `[/Script/Engine.Engine]` section. The final value should be in the following format:

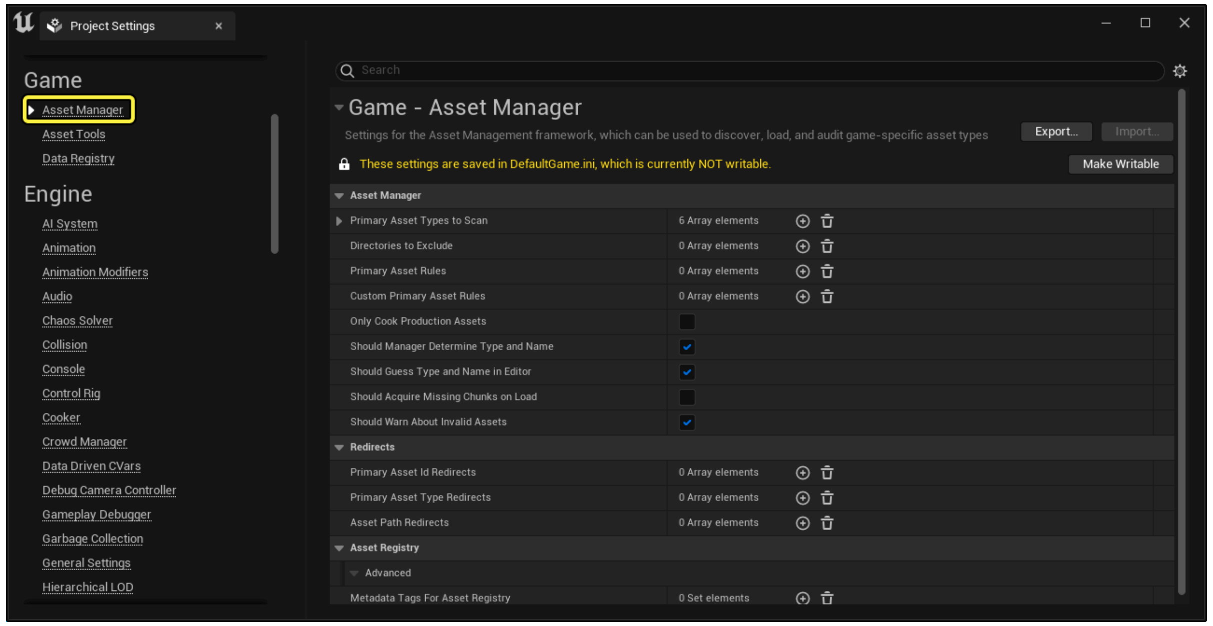
```
[/Script/Engine.Engine]
AssetManagerClassName=/Script/Module.UClassName
```

Copy full snippet

Where "Module" refers to your project's module name, and "UClassName" refers to the name of the `UClass` you want to use. In our example, the module name for the project is "MyGame", and the class we want to use is called `UFortAssetManager` (meaning its `UClass` name is `FortAssetManager`), so the second line would read:

~~~ AssetManagerClassName=/Script/FortniteGame.FortAssetManager ~~~

2. **Register your Primary Assets with the Asset Manager.** This can be done by configuring in the **Project Settings** menu, or by programming your Asset Manager class to register the Primary Assets during startup.
   - Configuring with **Project Settings** (under the **Game / Asset Manager** section) looks like this:



Game - Asset Manager

Settings for the Asset Management framework, which can be used to discover, load, and audit game-specific asset types

These settings are saved in DefaultGame.ini, which is currently NOT writable.

Asset Manager

Primary Asset Types to Scan	6 Array elements
Directories to Exclude	0 Array elements
Primary Asset Rules	0 Array elements
Custom Primary Asset Rules	0 Array elements
Only Cook Production Assets	
Should Manager Determine Type and Name	✓
Should Guess Type and Name in Editor	✓
Should Acquire Missing Chunks on Load	
Should Warn About Invalid Assets	✓

Redirects

Primary Asset Id Redirects	0 Array elements
Primary Asset Type Redirects	0 Array elements
Asset Path Redirects	0 Array elements

Asset Registry

Advanced

Metadata Tags For Asset Registry	0 Set elements
~~~

Paths to scan for primary Assets can be configured.

Setting	Effect
Primary Asset Types to Scan	Lists the types of Primary Assets to discover and register, as well as where to look for them and what to do with them.
Directories to Exclude	Directories that will explicitly not be scanned for Primary Assets. This is useful for excluding test Assets.
Primary Asset Rules	Lists the specific Rules Overrides, which dictate how Assets are handled. See Cooking and Chunking for more information.
Only Cook Production Assets	Assets designated as DevelopmentCook will cause errors during the cook process if this is checked. Good for ensuring that final, shipping builds are free of test Assets.
Primary Asset ID Redirects	When the Asset Manager looks up data about a Primary Asset whose ID appears in this list, the ID will be substituted for the alternate ID provided.
Primary Asset Type Redirects	When the Asset Manager looks up data about a Primary Asset, the type name provided in this list will be used instead of its native type.
Primary Asset Name Redirects	When the Asset Manager looks up data about a Primary Asset, the Asset name provided in this list will be used instead of its native name.

- If you want to register Primary Assets directly in code, override the `StartInitialLoading` function in your Asset Manager class and call `ScanPathsForPrimaryAssets` from there. In this case, we recommend that you put all Primary Assets of the same type in a single subfolder. This will make discovery and registration faster.

1. **Load the Asset.** Use the Asset Manager functions `LoadPrimaryAssets`, `LoadPrimaryAsset`, and `LoadPrimaryAssetsWithType` to begin loading Primary Assets at the appropriate time. Unload Assets later with `UnloadPrimaryAssets`, `UnloadPrimaryAsset`, and `UnloadPrimaryAssetsWithType`. When using these load functions, you can specify a list of Asset Bundles. Loading this way will cause the Asset Manager to load the Secondary Assets that those Asset Bundles reference as described above.

Registering and Loading Dynamically-Created Primary Assets

Primary Asset Bundles can also be dynamically registered and loaded at runtime. There are two Asset Manager functions that are useful to understand for doing this:

- `ExtractSoftObjectPaths` examines all `UPROPERTY` members of the `UScriptStruct` it is given, and identifies Asset references, which are then stored in an array of Asset names. This array can be used when creating an Asset Bundle. `ExtractSoftObjectPaths` parameters:

Parameter	Purpose
`Struct`	UStruct to search for Asset references.
`StructValue`	A void pointer to the struct.
`FoundAssetReferences`	Array used to return Asset references found in the struct.
`PropertiesToSkip`	Array of property names to exclusive from the return array.

- `RecursivelyExpandBundleData` will find all references to Primary Assets and recursively expands to find all of their Asset Bundle dependencies. In this case, it means that the TheaterMapTileClass referenced by the ZoneTheme above will be added to the AssetBundleData. It then registers the named dynamic Asset and starts loading it. `RecursivelyExpandBundleData` parameters:

Parameter	Purpose
`BundleData`	Bundle Data containing Asset references. These will be expanded recursively, and can be useful for loading a set of related assets.

For example, our "MyGame" project could use the following code in its custom Asset Manager class to construct and load Assets based on "theater" data that was downloaded during the game:

```cpp
// Construct the name from the theater ID
UMyGameAssetManager& AssetManager = UMyGameAssetManager::Get();
FPrimaryAssetId WorldMapAssetId = FPrimaryAssetId(UMyGameAssetManager::WorldMapInfoType, FName(*WorldMapData.UniqueId));

TArray<FSoftObjectPath> AssetReferences;
AssetManager.ExtractSoftObjectPaths(FMyGameWorldMapData::StaticStruct(), &WorldMapData, AssetReferences);

FAssetBundleData GameDataBundles;
GameDataBundles.AddBundleAssets(UMyGameAssetManager::LoadStateMenu, AssetReferences);

// Recursively expand references to pick up tile Blueprints in Zone
AssetManager.RecursivelyExpandBundleData(GameDataBundles);

// Register a dynamic Asset
AssetManager.AddDynamicAsset(WorldMapAssetId, FSoftObjectPath(), GameDataBundles);

// Start preloading
AssetManager.LoadPrimaryAsset(WorldMapAssetId, AssetManager.GetDefaultBundleState());
```

Copy full snippet