

Runtime Asset Management

Prior to 4.16, UE4 has not provided much support for runtime loading/unloading of assets. There were bits and pieces in StreamableManager, ObjectLibrary, and the Map streaming code, but there were no examples or documentation. Our internal games have developed their own systems for asset management, but those have their own problems. The goal of the Asset Management system added in 4.16 is to provide the basic structure for runtime asset management and allow individual games to customize things as needed.

Goals

- Improve the existing low level engine systems for async loading assets
- Create a system for managing assets that divides content into understandable chunks, without losing the advantages of a loose package architecture
- Provide a distinction between “top level” assets that are manually loaded, and other assets that are loaded automatically as needed
- Provide a centralized, easy to use interface for async loading assets and managing their memory
- Provide tools to help package related assets into separately cooked/downloaded bundles
- Provide tools to help audit disk and memory usage

Concepts

- **Assets** already exist, and are objects that are viewed in the Content Browser. Maps, texture, sounds, blueprints, etc are assets
- The **Asset Registry** already exists and is a repository for useful information about specific assets that is extracted at package save time
- **Streamable Managers** already exist and are native structs responsible for loading objects and keeping them in memory
- **Primary Assets** are assets that can be manually loaded/unloaded based on changes in game state. This includes maps and game-specific objects such as inventory items or character classes.
- **Secondary Assets** are all other assets, such as textures, sounds, etc. These are loaded automatically based on use by Primary Assets
- An **Asset Bundle** is a named list of Assets that can be loaded as a group at runtime
- The **Asset Manager** is a new, optional, global singleton that manages information about primary assets and asset bundles that is useful at runtime.

Primary Assets

A Primary Asset is a UObject that returns something valid from **GetPrimaryAssetId()**. This could be a PrimaryDataAsset/UObject subclass that is directly edited, a Blueprint class, or a runtime-only UObject that represents other data sources.

Primary Assets are referenced by **FPrimaryAssetId**, which has these properties:

- **PrimaryAssetType**: An FName describing the logical type of this object, usually the name of a base UClass. For instance if we have blueprints AWeaponRangedGrenade_C and AWeaponMeleeHammer_C that inherit from native class AWeapon, they would both have the same primary asset type of "Weapon"
- **PrimaryAssetName**: An FName describing this asset. This is usually the short name of the object, but could be a full asset path for things like maps
- **PrimaryAssetType:PrimaryAssetName** forms a unique pair across the entire game, and violations of this will cause errors. This is the string you would use to identify an item when talking to a persistent back end. For instance, Weapon:BattleAxe_Tier2 represents the same object as /Game/Items/Weapons/Axes/BattleAxe_Tier2.BattleAxe_Tier2_C
- Type and Name are saved directly as AssetRegistry tags, so once a primary asset has been saved to disk once you can search for it directly in the asset registry
- **Maps** inside /Game/Maps are set to be Primary Assets by default, everything else needs to be set up for your specific game
- **PrimaryAssetLabels** also live at the engine level and are special Primary Assets that are used to label other assets for chunking and cooking

Streamable Manager

FStreamableManager is a native structure that handles async loading objects and keeping them in memory until they are not needed. There are multiple Streamable Managers that are used for different use cases. This struct already existed, but has been enhanced to work with Streamable Handles that keep the loaded objects in memory as long as they are needed:

- **FStreamableHandle** is a struct tracked by shared pointer that is returned from streaming operations. An active handle keeps the referenced assets loaded into memory, and all handles are active while loading. Once loading is finished, handles are active until canceled or released.
- **FStreamableHandle::ReleaseHandle()** can be explicitly called to release references, and will implicitly get called when all shared pointers to the handle are destroyed
- **FStreamableHandle::CancelHandle()** can be called to abort the load and stop the complete callback from happening

- **FStreamableHandle::WaitUntilComplete()** blocks until the requested assets have loaded. This pushes the requested asset to the top of the priority list, but does not flush all async loading so is usually faster than a LoadObject call
- **RequestAsyncLoad** is the primary streamable operation. If you pass in a list of StringAssetReferences it will attempt to load them all, call a callback when complete, and return a Streamable Handle for later use
- **RequestSyncLoad** is the synchronous version. It will either start an async load and call WaitUntilComplete, or call LoadObject directly, whichever is faster
- **LoadSynchronous** is a version of RequestSyncLoad that returns a single asset, and has templated type safe versions
- These functions support **bManageActiveHandle**, which if set will cause the streamable manager itself to hold an active reference to the request handle, until queried for with **GetActiveHandles** and released manually

Asset Manager

The AssetManager is a singleton UObject that provides operations for scanning for and loading Primary Assets at runtime. It is meant to replace the functionality that ObjectLibraries currently provide, and wraps a FStreamableManager to handle the actual async loading. The engine asset manager provides basic management, but more complicated things like caching could be implemented by game-specific subclasses. Basic operations for Asset Manager:

- **Get()**: Static function to return the active asset manager. Use **IsValid()** before if unsure
- **ScanPathsForPrimaryAssets(Type, Paths, BaseClass)**: This functions scans the disk (or cooked asset registry) and parses FAssetData for primary assets of a specific type
- **GetPrimaryAssetPath(PrimaryAssetId)**: Converts Primary Asset to object path
- **GetPrimaryAssetIdForPath(StringReference)**: Converts an object path into a Type:Name pair if that path refers to a Primary Asset
- **GetPrimaryAssetIdFromData(FAssetData)**: Figures out the Type:Name based on the FAssetData returned from the Asset Registry
- **GetPrimaryAssetData(PrimaryAssetId)**: Returns FAssetData for a Type:Name combo
- **GetPrimaryAssetDataList(Type)**: Returns a list of all FAssetData for a PrimaryAssetType
- **GetPrimaryAssetObject(PrimaryAssetId)**: Returns the in memory UObject for a Type:Name combo if loaded into memory
- **LoadPrimaryAssets(AssetList, BundleState, Callback, Priority)**: Asynchronously loads the list of primary assets and any assets referenced by BundleState. Returns FStreamableRequest to allow polling or waiting, and calls callback when complete. This can also be used to change the asset state for already loaded assets
- **UnloadPrimaryAssets(AssetList)**: Drops hard GC references to the primary assets. Objects may still be referenced via other systems
- **ChangeBundleStateForPrimaryAssets(AssetList, Add, Remove)**: This operates on a list of assets and can be used to change bundle state in a more complex way than Load

- **GetPrimaryAssetsWithBundleState()**: This runs a query to get the list of assets that match the search criteria, useful to get a list to pass into `ChangeBundleState`
- Some generally useful utility functions for runtime asset management, such as redirector support for `PrimaryAssetIds`

Asset Bundles

An Asset Bundle is a globally named explicit list of assets, associated with a Primary Asset. “All Player buildable walls”, “First Generation Heroes” and “ArcBlade InGame” are good examples of bundles:

- At a core level, a Bundle is a Name to `TArray<FStringAssetReferences>` map that exists in the asset manager and can be queried at editor and runtime. All bundles are associated with a specific Primary Asset id, but this can be a dynamic asset
- Asset Bundles tied to a real Primary Asset are created by adding a `FAssetBundleData` UStruct to your object and filling it out at save time. This then gets written to the asset registry tags and parsed when that asset data is loaded into the registry
- The `AssetBundles` meta tag can be set on specific `AssetPtr` or `StringAssetReference` members to cause those references to be added to the named bundle at save time
- Other Asset Bundles are registered by calling `AddDynamicAsset` with a filled in `FAssetBundleData`, and are useful for data sets determined at runtime
- Anything in an asset bundle is considered to be “part of” that Primary Asset, which will be useful when setting up Chunking using Labels as described below

On Disk Example

Most Primary Assets will have an on-disk representation because they are directly edited by artists or designers. The easiest way to get this functionality up and running is by inheriting from **PrimaryDataAsset**. For non-blueprint assets you can directly inherit from it and your class will show up in the content browser when making a New Data Asset, for Blueprints you can inherit from it and it will show up in the New Blueprint class list. Here’s an example of a blueprintable data-only class from Fortnite:

```

UCLASS(Blueprintable, hidecategories = (Object, Actor, Advanced, Navigation))
class FORTNITEGAME_API UFortZoneTheme : public UPrimaryDataAsset
{
    GENERATED_UCLASS_BODY()

public:

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

    /** The class that is used for this theme in the new world flow */
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = Visual, meta = (AssetBundles = "Menu"))
    TAssetSubclassOf<class AFortTheaterMapTile> TheaterMapTileClass;
}

```

Because this class inherits from PrimaryDataAsset, it automatically acquires a working version of GetPrimaryAssetId using the asset's short name and native class. If there was a FortZoneTheme called Forest, it would have a primary asset id of **FortZoneTheme:Forest**. The AssetBundles tag specifies that the asset reference will be automatically loaded as part of the Menu bundle. Whenever a Fort Zone Theme is saved, the AssetBundleData member of PrimaryDataAsset will be updated to include the TheaterMapTileClass asset reference.

Now that the native class is setup, designers can start adding instances of this class. To make the parsing more efficient and for better organization, all Zone Themes should end up in the same directory hierarchy. So, in our example the designers added the Zone Themes to sub directories inside World in the content browser. So the next step is to Scan that directory to load in the asset's metadata. To do this from native you would add code like this:

```

void UFortAssetManager::StartInitialLoading()
{
    Super::StartInitialLoading();

    FName ZoneThemeType = UFortZoneTheme::StaticClass()->GetFName();
    ScanPathForPrimaryAssets(ZoneThemeType, TEXT("/Game/World"), UFortZoneTheme::StaticClass(), true);
}

```

FortAssetManager is the Fortnite-specific subclass of AssetManager, it is spawned by specifying "**AssetManagerClassName=/Script/FortniteGame.FortAssetManager**" in DefaultEngine.ini. StartInitialLoading gets called when the game engine starts up for the editor, commandlets, standalone games, and cooked games. The 4th parameter says that this type is a Blueprint as opposed to a raw object. So, when the editor starts up it scans the /Game/World directory for any instances of UFortZoneTheme that say they have the primary asset type of FortZoneTheme, and will add them to the Asset Manager's dictionary. All of the query functions mentioned above such as GetPrimaryAssetPath will now work.

Now that the primary asset is in the dictionary, we need to actually load it at runtime. Here's some example code:

```

TSharedPtr<FStreamableHandle> UFortAssetManager::LoadZone(FName ZoneName)
{
    FPrimaryAssetId ZoneId(UFortZoneTheme::StaticClass()->GetFName(), ZoneName);
    TArray<FName> BundleStates;
    if (IsInMenu())
    {
        BundleStates.Add(FName(TEXT("Menu")));
    }

    return LoadPrimaryAsset(ZoneId, BundleStates);
}

```

This code calls some game-specific logic to figure out if the Menu state should be enabled, and then passes in a desired list of BundleStates to the LoadPrimaryAsset function. This means that if you're in the menu the TheaterMapTileClass reference will get loaded for you, while if you're not it will only load the base ZoneTheme asset. The Load function returns a StreamableHandle that can then be polled or otherwise managed

Dynamic Asset Example

The other type of Primary Asset is called a Dynamic Asset and is added by calling the AddDynamicAsset function. Dynamic Assets can be registered at runtime from data downloaded off a server or anywhere else, and are very flexible. Here's an example from Fortnite, where we download "theater" information off of a database and use that to construct an asset at runtime:

```

// Construct the name from the theater GUID
UFortAssetManager& AssetManager = UFortAssetManager::Get();
FPrimaryAssetId TheaterAssetId = FPrimaryAssetId(UFortAssetManager::FortTheaterInfoType, FName(*TheaterData.UniqueId));

// Extract references from the downloaded data
TArray<FStringAssetReference> AssetReferences;
AssetManager.ExtractStringAssetReferences(FFortTheaterMapData::StaticStruct(), &TheaterData, AssetReferences);
AssetManager.ExtractStringAssetReferences(FFortAvailableTheaterMissions::StaticStruct(), &MissionData, AssetReferences);

// Create a Bundle with the asset list
FAssetBundleData GameDataBundles;
GameDataBundles.AddBundleAssets(UFortAssetManager::LoadStateMenu, AssetReferences);

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

// Register a dynamic asset
AssetManager.AddDynamicAsset(TheaterAssetId, FStringAssetReference(), GameDataBundles);

// Start preloading
AssetManager.LoadPrimaryAsset(TheaterAssetId, AssetManager.GetDefaultBundleState());

```

ExtractStringAssetReferences is similar to the function that parses the AssetBundles metadata, but is usable at runtime as well as editor save time. **RecursivelyExpandBundleData** will find all references to Primary Assets and recursively expands to find all of their bundle dependencies. In this case it means that the TheaterMapTileClass referenced by the ZoneTheme above will get added to the AssetBundleData. It then registers the named dynamic asset, then starts loading it.

Once a Primary Asset is loaded, either on disk or dynamic, it will stay in memory until being unloaded. You can also change the active BundleState for an asset, and it will drop references to old assets and async load any new asset references for the new state. Here's an example in Fortnite of what happens when you change maps:

```
void UFortAssetManager::HandlePreLoadMap(const FString& MapName)
{
    // Unload some cached types that are loaded on demand
    UnloadPrimaryAssetsWithType(FortTheaterInfoType);

    TArray<FName> OldBundles = DefaultBundleState;
    TArray<FName> NewBundles;

    // Make new Bundle State list
    if (MapIsMenu(MapName))
    {
        NewBundles.Add(FName(TEXT("Menu")));
    }

    DefaultBundleState = NewBundles;

    TArray<FPrimaryAssetId> AssetsToChange;
    if (GetPrimaryAssetsWithBundleState(AssetsToChange, TArray<FName>(), OldBundles))
    {
        ChangeBundleStateForPrimaryAssets(AssetsToChange, NewBundles, OldBundles);
    }
}
```

First, this code unloads any theaters that we previously created as dynamic assets. Then this code figures out what the new game-global BundleState should be, finds any assets that were in the old state, then tells those assets to transition to the new one. This allows transitioning all assets at once between “menu” and “zone” states as an example

Game-Specific Configuration

It's possible to use the manual method above to scan for Primary Assets, but if your game is complicated enough to have multiple asset types you will want to use the

PrimaryAssetTypesToScan member of **AssetManagerSettings**. This is editable from the [/Script/Engine.AssetManagerSettings] section of DefaultGame.ini, or via the Asset Manager Project Settings tab. Here's an example of the Fortnite Asset Manager settings (reformatted for legibility, each + should start it's own line):

```
[/Script/Engine.AssetManagerSettings]
!PrimaryAssetTypesToScan=ClearArray

+PrimaryAssetTypesToScan=(PrimaryAssetType="FortGameData",
AssetBaseClass=/Script/FortniteGame.FortGameData,
bHasBlueprintClasses=False, bIsEditorOnly=False,
SpecificAssets=("/Game/Balance/DefaultGameData.DefaultGameData"),
Rules=(Priority=1000,CookRule=AlwaysCook,ChunkId=0))
```

```

+PrimaryAssetTypesToScan=(PrimaryAssetType="Weapon",
AssetBaseClass=/Script/FortniteGame.FortWeaponItemDefinition,
bHasBlueprintClasses=False, bIsEditorOnly=False,
Directories=((Path="/Game/Items/Weapons"), (Path="/Game/Abilities/Player")),
Rules=(Priority=2,CookRule=AlwaysCook,ChunkId=0))

+PrimaryAssetTypesToScan=(PrimaryAssetType="Map",
AssetBaseClass=/Script/Engine.World,
bHasBlueprintClasses=False,bIsEditorOnly=True,
Directories=((Path="/Game/Maps"))))

+PrimaryAssetTypesToScan=(PrimaryAssetType="FortZoneTheme",
AssetBaseClass=/Script/FortniteGame.FortZoneTheme,
bHasBlueprintClasses=True, bIsEditorOnly=False,
Directories=((Path="/Game/World/ZoneThemes")),
Rules=(Priority=0))

+PrimaryAssetTypesToScan=(PrimaryAssetType="PrimaryAssetLabel",
AssetBaseClass=/Script/Engine.PrimaryAssetLabel,
bHasBlueprintClasses=False, bIsEditorOnly=False,
Directories=((Path="/Game/Labels"))))

```

This example first clears the values initialized in BaseGame.ini, then adds information for an always loaded GameData object, Weapons, Maps, Zone Themes and Labels. Here's what some of those fields mean:

- **PrimaryAssetType**, **AssetBaseClass**, **bHasBlueprintClasses**, and **bIsEditorOnly** are parameters passed to ScanPathsForPrimaryAssets
- **SpecificAssets** and **Directories** are combined to create the paths passed to ScanPathsForPrimaryAssets
- **Priority** is used to determine which Primary Assets will "Manage" referenced secondary assets. A higher priority is applied first, so the startup data has a priority of 1000 to make sure it goes first. -1 is the default priority, and acts like 1 if not overridden
- **CookRule** says rather something should be Always Cooked, Never Cooked, or only cooked in development build. Setting AlwaysCook is equivalent to adding something to the AlwaysCookMaps list in DefaultEditor
- **ChunkId** specifies that the specific type goes into a certain chunk by default, it can be overridden per asset as described below

Once your INI is set up, there's a good chance you'll want to override some virtual functions in your game specific subclass of AssetManager. Here are a few good places to start:

- **StartInitialLoading** gets called when the asset manager is initially created early in Engine init, and this is where it does the scanning specified in the ManagerSettings. This is a good time to do extra scanning

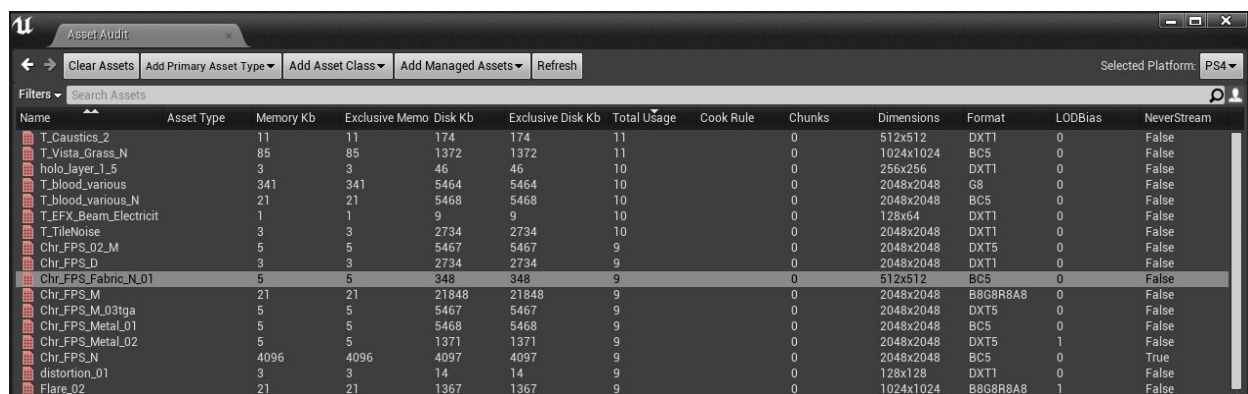
- **FinishInitialLoading** gets called at the end of Engine init right before the actual game starts. This is a good time to finalize any loading
- **PostInitialAssetScan** gets called either from FinishInitialLoading in game, or when the async asset registry scan finishes in the editor
- **ApplyPrimaryAssetLabels** gets called whenever the editor needs to set up the Management database for cooking or the audit tools described below. It load and applies PrimaryAssetLabels and is a good place to do game-specific chunk overrides
- **ModifyCook** is called by the cooker, and by default will schedule loading all AlwaysCook assets, but you can modify it to do what you want
- **PreBeginPIE** is called right before PIE starts, it is a good time to preload expensive assets you didn't want to load on editor startup

Asset Audit UI

Once you've set up your game to scan Primary Assets, you can use the Asset Audit UI to inspect your primary and secondary assets and audit them for memory use, chunking, or general type-specific metadata. This window is a specialized version of the content browser that is designed for showing assets in a list format, and with extra information useful in auditing. There are 3 ways to get to the window:

1. Window->Developer Tools->Asset Audit opens an empty one
2. From Content Browser, select assets, right click, asset actions, audit assets
3. From Reference viewer, select a node and audit assets

Once you have the window open, you can add assets to the audit window with the buttons, and select a platform using the drop down in the upper right. That drop down will be populated with any local cooked asset registries that are available to the editor. Here's an example of using the window to audit textures in Shooter Game on PS4, which is a Bottom-Up way of auditing:

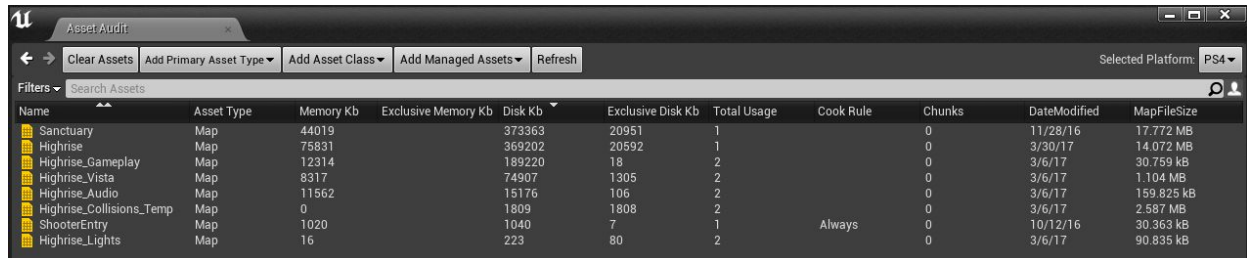


Name	Asset Type	Memory Kb	Exclusive Memo Disk Kb	Exclusive Disk Kb	Total Usage	Cook Rule	Chunks	Dimensions	Format	LODBias	NeverStream
T_Caustics_2		11	11	174	174	11	0	512x512	DXT1	0	False
T_Vista_Grass_N		85	85	1372	1372	11	0	1024x1024	BC5	0	False
holo_layer_1_5		3	3	46	46	10	0	256x256	DXT1	0	False
T_blood_various		341	341	5464	5464	10	0	2048x2048	G8	0	False
T_blood_various_N		21	21	5468	5468	10	0	2048x2048	BC5	0	False
T_EFX_Beam_Electricit		1	1	9	9	10	0	128x64	DXT1	0	False
T_TileNoise		3	3	2734	2734	10	0	2048x2048	DXT1	0	False
Chr_FPS_02_M		5	5	5467	5467	9	0	2048x2048	DXT5	0	False
Chr_FPS_D		3	3	2734	2734	9	0	2048x2048	DXT1	0	False
Chr_FPS_Fabrie_N_01		5	5	343	343	9	0	512x512	BC5	0	False
Chr_FPS_M		21	21	21848	21848	9	0	2048x2048	B8G8R8A8	0	False
Chr_FPS_M_03tga		5	5	5467	5467	9	0	2048x2048	DXT5	0	False
Chr_FPS_Metal_01		5	5	5468	5468	9	0	2048x2048	BC5	0	False
Chr_FPS_Metal_02		5	5	1371	1371	9	0	2048x2048	DXT5	1	False
Chr_FPS_N		4096	4096	4097	4097	9	0	2048x2048	BC5	0	True
distortion_01		3	3	14	14	9	0	128x128	DXT1	0	False
Flare_02		21	21	1367	1367	9	0	1024x1024	B8G8R8A8	1	False

To get to this view I cleared assets, clicked "Add Asset Class", selected Texture2d, then sorted by Total Usage. Total Usage is only available in this window, and the higher the number, the

more Primary Assets use the texture. It's not a strict count, as it weights by Priority. In this example T_Caustics_2 is the most commonly used texture, as 11 separate maps use it.

Here's an example of auditing Maps, which is a Top-Down way of auditing:



Name	Asset Type	Memory Kb	Exclusive Memory Kb	Disk Kb	Exclusive Disk Kb	Total Usage	Cook Rule	Chunks	DateModified	MapFileSize
Sanctuary	Map	44019	373363	20951	1	1	Always	0	11/28/16	17.772 MB
Highrise	Map	75831	369202	20592	1	1		0	3/30/17	14.072 MB
Highrise_Gameplay	Map	12314	189220	18	2	2		0	3/6/17	30.759 kB
Highrise_Vista	Map	8317	74907	1305	2	2		0	3/6/17	1.104 MB
Highrise_Audio	Map	11562	15176	106	2	2		0	3/6/17	159.825 kB
Highrise_Collisions_Temp	Map	0	1809	1808	2	2		0	3/6/17	2.587 MB
ShooterEntry	Map	1020	1040	7	1	1		0	10/12/16	30.363 kB
Highrise_Lights	Map	16	223	80	2	2		0	3/6/17	90.835 kB

To get here I cleared assets, hit "Add Primary Asset Type", selected Map, then sorted by Disk Kb. Disk Kb is a count of how much space is used by this Primary Asset as well as anything else it Manages, where Exclusive Disk Kb is just for the specific asset. So in this case Sanctuary and Highrise are both about the same exclusive size on disk at 20 Mb, but Sanctuary is managing more textures and meshes so takes a total of 373 Mb on disk. Management can be shared by types that share the same priority, so a good amount of that 373 Mb is also used by Highrise. If you set up a type like "SharedAssets" with a higher priority, the Disk Size listed here would only include assets not managed by SharedAssets.

Managing Chunking and Cook Rules

When you cook your game for release or test on non-editor platforms, Chunking can be used to split your data into several pak files that can then be independently deployed. Chunking has existed in the engine for several years, but setting the rules up relied on creating a game-specific callback or using an experimental UI. Chunking is now integrated into the Asset Manager, and it can be set up using Labels or Rules Overrides. The old system will continue to work If you have a game-specific AssignStreamingChunkDelegate set, but eventually we will deprecate that path in favor of the Asset Manager.

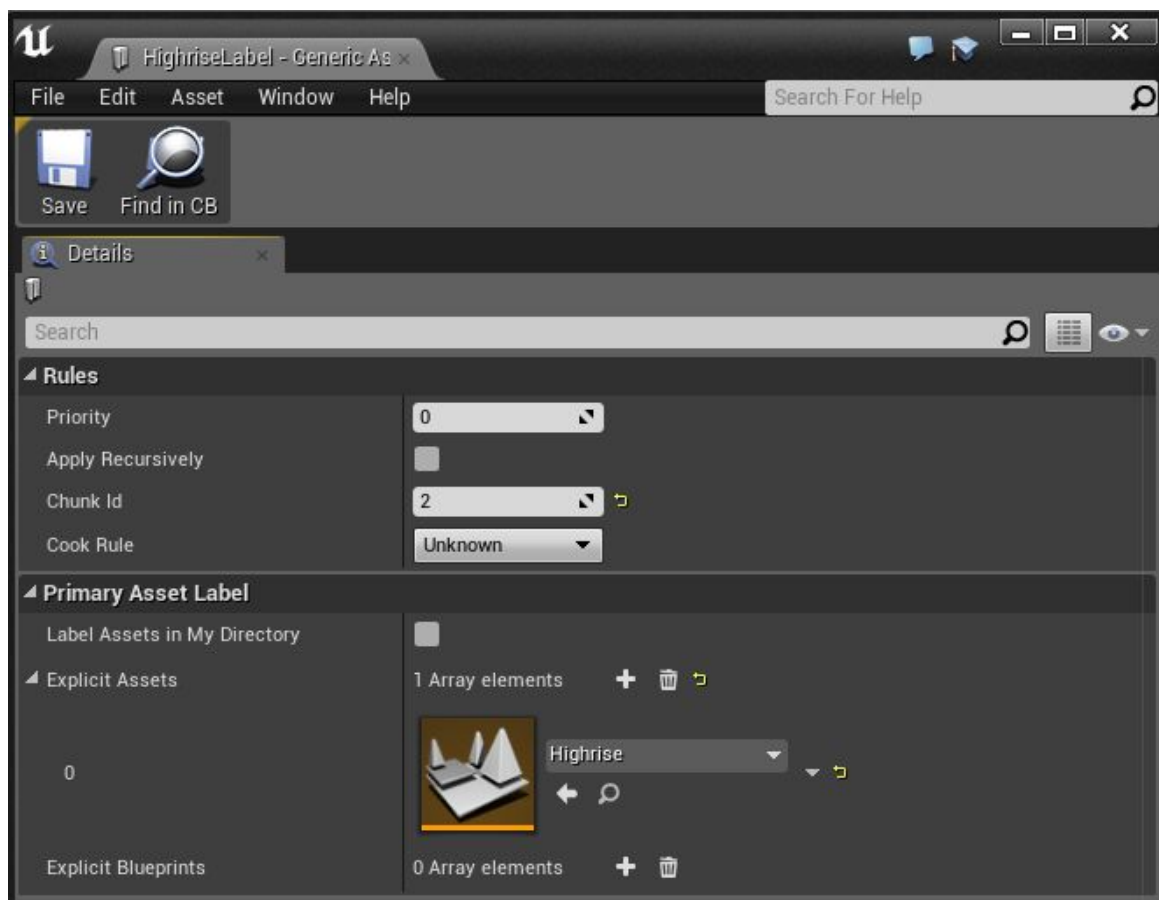
The best example of how to do this is in the ShooterGame sample, which was modified to use Asset Manager for Chunking. The chunking setup for ShooterGame is simple, where it wants the Sanctuary map to be in chunk 1, Highrise to be in chunk 2, and everything else in the default Chunk 0. Because Maps are now Primary Assets, this is very easy to setup. The first approach is to use Rule Overrides. These allow setting the priority/chunk settings for a specific primary asset. Here's how this would work for Shooter Game:

```
[/Script/Engine.AssetManagerSettings]
+PrimaryAssetRules=(PrimaryAssetId="Map:/Game/Maps/Sanctuary",
Rules=(Priority=-1,ChunkId=1,CookRule=Unknown))
+PrimaryAssetRules=(PrimaryAssetId="Map:/Game/Maps/Highrise",
Rules=(Priority=-1,ChunkId=2,CookRule=Unknown))
```

```
+PrimaryAssetRules=(PrimaryAssetId="Map:/Game/Maps/ShooterEntry",  
Rules=(Priority=-1,ChunkId=0,CookRule=AlwaysCook))
```

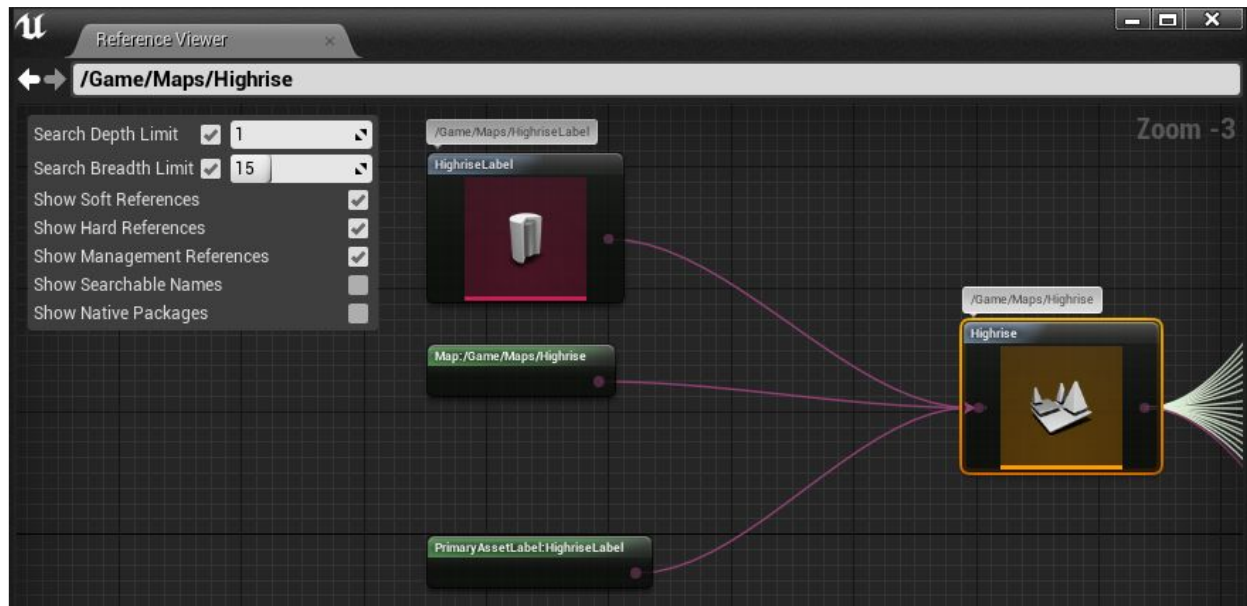
This sets specific maps to be in specific chunks, and all of their references will automatically be added to those chunks as well, with the special rule that anything in the default chunk 0 is only added to that chunk. This isn't how chunking is actually set up in ShooterGame, though:

Instead of using ini overrides, you can also set up the chunking rules using **PrimaryAssetLabels**. Labels are special assets that aren't generally directly loaded at runtime, but can be used to collect assets into functional groups using the Asset Bundle system described above. Each Label has it's own Rules that are applied to each of the labelled assets, so they are great for setting chunking. Here's the Highrise label from ShooterGame:



In this case the Highrise map is explicitly added to the Label's ExplicitAsset list, which means that the map /Game/Maps/Highrise.Highrise is managed by "PrimaryAssetLabel:HighriseLabel" in addition to being managed by "Map:/Game/Maps/Highrise". When it goes to decide the Chunking/Cook rules, it follows this hierarchy. Because there's no Chunk set for Map:/Game/Maps/Highrise it uses the ChunkId from the label, which is set to 2. This causes all of the assets in Highrise to end up in Chunk 2. If you want to inspect the results of the

Management dependency resolution, you can use the Reference viewer, with show Management References enabled:



This shows the relationship described above, but if you were to double click to center on Map:/Game/Maps/Highrise it would show it as being managed by PrimaryAssetLabel:HighriseLabel, resulting in the “inheritance” mentioned above. Setting up chunking is still a fairly complicated process, but by using the asset manager you can easily try out different management scenarios, and then load up the Asset Audit window and look at the Chunk column to see where it will end up when it is eventually cooked. Also, all of the techniques described here can also be used to set the NeverCook or DevelopmentCook flag, which can be very useful to for instance stop the released game from cooking things in the Developers folder.

