The principles behind the asset system are the following:

- **Every piece of data in the game belongs to an asset**
    - **This means if you are adding new things to be edited by users, you need to integrate with existing asset types or create a new asset type.**
- An asset is an "indivisible" piece of work: only one person can work on it at one time. It can, for example, have exclusive locking rights in the source control system.
- An asset is composed of one or multiple files, and files on disk only belong to one asset at a time.
- Assets reference each other, i.e. depend on each other, by explicit file path.
- An asset may be able to be opened in an editor in the sandbox, and can only be opened in one editor at a time.
- Rare case, but asset editors may be able to handle different types of assets, for example, the Mesh Importer can handle Skinned Mesh and Mesh.

Ultimately, the asset system is a way to hide the real files to the user, so the engine can use more creative solutions when it comes to file management, while the users are only concerned with the concept of an asset, i.e. a piece of data that can be created, modified and submitted to source control.

The user documentation of the Asset System can be found here: Asset System 5.5.2

As a programmer in the Sandbox, what this means for you is that the asset system is primarily a unified file management service that provides the following features to you "for free":

- File management - Asset Browser
    - Browsing for all assets in the Asset Browser.
    - This offers a simpler view of the project as several files are grouped together within the same asset, the resulting view is much simpler than looking at the files on disk.
    - Creating an object by drag&drop from the Asset Browser into the level viewport.
- Editor framework providing all basic features.
    - Opening, editing, saving, recent files...
    - Inherits from CEditor framework and all of its functionality.
- Source control integration (git and perforce are planned).
- Dependency tracking: assets that use another asset or depend on another asset, like a model using a texture, will be tracked.
- Deployment and packaging for the final game (planned).
- Platform-specific variations of an asset (planned).
- More...

This means you should not have to do any of that work except implement the relevant methods in your specific AssetType and AssetEditor, i.e. the specific part of your system.

Let's go over the most important use case of having to interact with the asset system: adding a new editable asset type.

Adding an asset type

Create a class and inherit from CAssetType. Implement all the virtual methods to describe your type, implement importing if necessary, and editing if necessary.

A good example of this can be found in the SamplePlugin in SampleAssetType.h/cpp. These files will be updated if the API changes.

Adding an asset editor

Create an editor class inheriting from CAssetEditor. This is an advanced version of CEditor that offers similar services and then some more related to editing assets. Implement all the virtual methods necessary and you are done, your asset can now be edited.

A good example of this can be found in the SamplePlugin in SampleAssetEditor.h/cpp. These files will be updated if the API changes.

Asset Importer

If your asset is imported from an external tool such as a DCC, you may also need to write an Asset Importer.