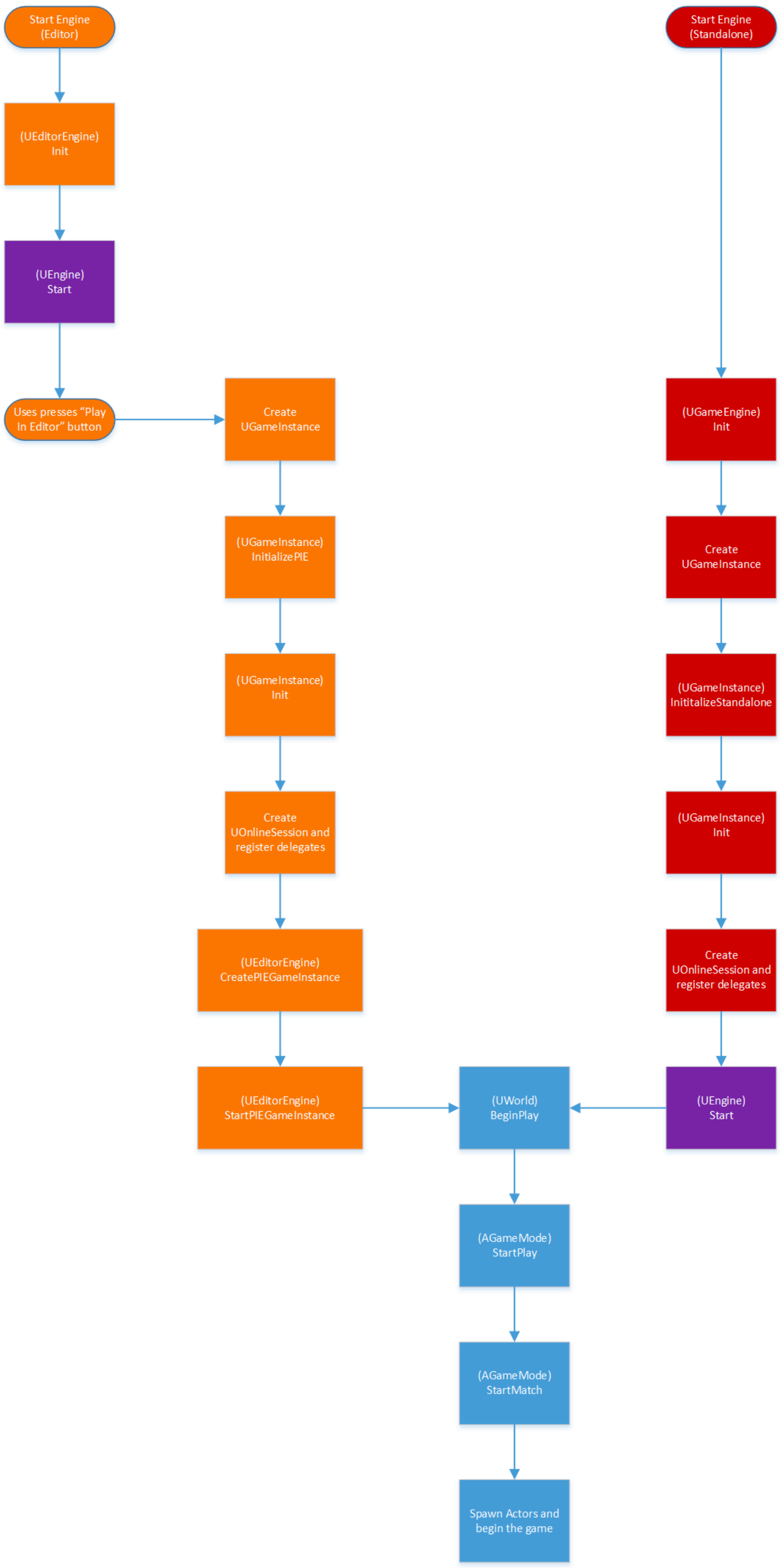
UNREAL
ENGINE


Start Engine (Editor)
(UEditorEngine) Init
(UEngine) Start
Uses presses "Play In Editor" button
Create UGameInstance
(UGameInstance) InitializePIE
(UGameInstance) Init
Create UOnlineSession and register delegates
(UEditorEngine) CreatePIEGameInstance
(UEditorEngine) StartPIEGameInstance
Start Engine (Standalone)
(UGameEngine) Init
Create UGameInstance
(UGameInstance) InititalizeStandalone
(UGameInstance) Init
Create UOnlineSession and register delegates
(UEngine) Start
(UWorld) BeginPlay
(AGameMode) StartPlay
(AGameMode) StartMatch
Spawn Actors and begin the game

https://docs.unrealengine.com/Images/Gameplay/Framework/GameFlow/GameFlowChart.png

Standalone: In Standalone mode, which is the mode used by games played outside of the editor, the objects needed to play the game are created and initialized immediately following engine initialization on startup. Objects such as the GameInstance are created and initialized before starting the engine (distinct from creating and initializing the Engine), and the starting map is loaded immediately after the engine's start function is called. Gameplay officially begins at that point with the level creating the appropriate Game Mode and Game State , and then the other Actors .

Editor: In Editor mode, which is used by Play In Editor and Simulate In Editor, a different flow is used. The engine initializes and starts immediately, as it is needed to run the editor, but creation and initialization of objects such as the GameInstance are deferred until the user presses the button to launch the PIE or SIE session. In addition, the Actors in the level are duplicated so that in-game changes do not affect the level in the editor, and every object, including the GameInstance object, has a separate copy for each PIE instance. The editor path rejoins the standalone path with the beginning of gameplay in the UWorld class.

UObjectBase

Low level implementation of UObject , should not be used directly in game code

```
class UObjectBase
```

UObjectBaseUtility

Provides utility functions for UObject , this class should not be used directly

```
    class UObjectBaseUtility : public UObjectBase
```

UObject

The base class of all UE4 objects.

```
    class UObject : public UObjectBaseUtility
```

UEngine

Abstract base class of all Engine classes, responsible for management of systems critical to editor or game systems.

```
    class UEngine :
        public UObject ,
        public FExec
```

UEditorEngine

Engine that drives the Editor. Separate from **UGameEngine** because it may have much different functionality than desired for an instance of a game itself.

```
    class UEditorEngine :
        public UEngine ,
        public FGCObject
```

UGameEngine

Engine that manages core systems that enable a game.

```
    class UGameEngine : public UEngine
```

UEngine::Start

Start the game, separate from the initialize call to allow for post initialize configuration before the game starts.

```
    virtual void Start()
```

UGameInstance

GameInstance: high-level manager object for an instance of the running game.

```
class UGameInstance :
    public UObject ,
    public FExec
```

UGameInstance::UGameInstance

Spawned at game creation and not destroyed until game instance is shut down. Running as a standalone game, there will be one of these. Running in PIE (play-in-editor) will generate one of these per PIE instance.

```
    UGameInstance(
        const FObjectInitializer & ObjectInitializer)
```

UGameInstance::Init

Virtual function to allow custom GameInstances an opportunity to set up what it needs

```
    virtual void Init()
```

UGameInstance::InitializeStandalone

Called to initialize the game instance for standalone instances of the game

```
    void InitializeStandalone(
        const FName InWorldName,
        UPackage * InWorldPackage)
```

UEditorEngine::CreatePIEGameInstance

Deprecated. Please override CreatePlayInEditorGameInstance instead. PIE (Play in editor)

UGameInstance::StartPIEGameInstance

Deprecated. Please override StartPlayInEditorGameInstance instead.

UGameInstance::InitializePIE

Deprecated. Please override InitializeForPIE instead. lease override StartPlayInEditorGameInstance instead.

UOnlineSession

Handles online GameSession

```
class UOnlineSession : public UObject
```

UWorld

The World is the top level object representing a map or a sandbox in which Actors and Components will exist and be rendered.

```
class UWorld :
     public UObject ,
     public FNetworkNotify
```

* A World can be a single Persistent Level with an optional list of streaming levels that are loaded and unloaded via volumes and blueprint functions or it can be a collection of levels organized with a World Composition.

* In a standalone game, generally only a single World exists except during seamless area transitions when both a destination and current world exists. In the editor many Worlds exist: The level being edited, each PIE instance, each editor tool which has an interactive rendered viewport, and many more.

UWorld::BeginPlay

Start gameplay.

```
void BeginPlay()
```

* This will cause the game mode to transition to the correct state and call BeginPlay on all actors

AGameMode

GameMode is a subclass of GameModeBase that behaves like a multiplayer match-based game.

```
class AGameMode : public AGameModeBase
```

AGameModeBase

The GameModeBase defines the game being played.

```
class AGameModeBase : public AInfo
```

*It is only instanced on the server and will never exist on the client.

* A GameModeBase actor is instantiated when the level is initialized for gameplay C++
in UGameEngine::LoadMap() .

*The class of this GameMode actor is determined by (in order) either the URL ?game=xxx, the GameMode Override value set in the World Settings, or the DefaultGameMode entry set in the game's Project Settings.

AInfo

Info, the root of all information holding classes.

```
class AInfo : public AActor
```

*Doesn't have any movement / collision related code. Info is the base class of an Actor that isn't meant to have a physical representation in the world, used primarily for "manager" type classes that hold settings data about the world, but might need to be an Actor for replication purposes.

AActor

Actor is the base class for an Object that can be placed or spawned in a level.

```
class AActor : public UObject
```

*Actors may contain a collection of ActorComponents, which can be used to control how actors move, how they are rendered, etc.

*The other main function of an Actor is the replication of properties and function calls across the network during play.

*Actor initialization has multiple steps, check those virtual functions in documentation.

AGameMode::StartPlay

Transitions to calls BeginPlay on actors.

```
     virtual void StartPlay()
```

APawn

Pawn is the base class of all actors that can be possessed by players or AI.

```
     class APawn :
          public AActor ,
          public INavAgentInterface
```

*They are the physical representations of players and creatures in a level.

ACharacter

Characters are Pawns that have a mesh, collision, and built-in movement logic.

```
     class ACharacter : public APawn
```

*They are responsible for all physical interaction between the player or AI and the world, and also implement basic networking and input models.

*They are designed for a vertically-oriented player representation that can walk, jump, fly, and swim through the world using CharacterMovementComponent.

Networking and Multiplayer

Replication is the name for the process of synchronising data and procedure calls between clients and servers.

Client-Server Model

An overview of the role of the server in multiplayer.

Starting a Server or Client

Listen Server	`UE4Editor.exe ProjectName MapName?Listen -game`
Dedicated Server	`UE4Editor.exe ProjectName MapName -server -game -log`
Client	`UE4Editor.exe ProjectName ServerIP -game`

Server Gameplay Flow

Gameplay state and flow are generally driven through the **GameMode** actor. Only the server contains a valid copy of this actor (the client doesn't contain a copy at all).

To communicate this state to the client, there is a **GameState** actor, that will shadow important state of the **GameMode** actor.

This **GameState** actor is marked to be replicated to each client. Clients will contain an approximate copy of this **GameState** actor, and can use this actor as a reference to know the general state of the game.

Game Mode and Game State

There are two main classes which handle information about the game being played: **Game Mode** and **Game State**.

AGameModeBase

InitGame	The `InitGame` event is called before any other scripts (including `PreInitializeComponents`), and is used by `AGameModeBase` to initialize parameters and spawn its helper classes. This is called before any Actor runs `PreInitializeComponents`, including the Game Mode instance itself.
PreLogin	Accepts or rejects a player who is attempting to join the server. Causes the `Login` function to fail if it sets `ErrorMessage` to a non-empty string. `PreLogin` is called before `Login`, and a significant amount of time may pass before Login is called, especially if the joining player needs to download game content.
PostLogin	Called after a successful login. This is the first place it is safe to call replicated functions on the `PlayerController`. `OnPostLogin` can be implemented in Blueprint to add extra logic.
HandleStartingNewPlayer	Called after `PostLogin` or after seamless travel, this can be overridden in Blueprint to change what happens to a new player. By default, it will create a pawn for the player.
RestartPlayer	This is called to start spawning a player's Pawn. There are also `RestartPlayerAtPlayerStart` and `RestartPlayerAtTransform` functions available if you want to dictate where the Pawn will be spawned. `OnRestartPlayer` can be implemented in Blueprint to add logic after this function is finished.
SpawnDefaultPawnAtTransform	This is what actually spawns the player's Pawn, and can be overridden in Blueprints.
Logout	Called when a player leaves the game or is destroyed. `OnLogout` can be implemented to do Blueprint logic.

The Game Mode is not replicated to any remote clients that join in a multiplayer game; it exists only on the server, so local clients can see the stock Game Mode class (or Blueprint) that was used, but cannot access the actual instance and check its variables to see what has changed as the game progresses.

If players do need updated information relating to the current Game Mode, that information is easily kept in sync by being stored on an `AGameStateBase` Actor, one of which will be created along with the Game Mode and then replicated out to all remote clients.

AGameMode

The Game Mode is not replicated to any remote clients that join in a multiplayer game; it exists only on the server, so local clients can see the stock Game Mode class (or Blueprint) that was used, but cannot access the actual instance and check its variables to see what has changed as the game progresses. If players do need updated information relating to the current Game Mode, that information is easily kept in sync by being stored on an **`AGameStateBase`** Actor, one of which will be created along with the Game Mode and then replicated out to all remote clients.

Game State

The **Game State** is responsible for enabling the clients to monitor the state of the game. Conceptually, the Game State should manage information that is meant to be known to all connected clients and is specific to the Game Mode but is not specific to any individual player.

Game State is not the best place to keep track of player-specific things like how many points one specific player has scored for the team in a Capture The Flag match because that can be handled more cleanly by **Player State**. In general, the GameState should track properties that change during gameplay and are relevant and visible to everyone. While the Game mode exists only on the server, the Game State exists on the server and is replicated to all clients, keeping all connected machines up to date as the game progresses.

`AGameStateBase` is very commonly extended in C++ or Blueprints to contain additional variables and functions that are needed to keep players informed of what's going on in the game.

Connection Process

When a new client connects for the first time, a few things happen. First, the client will send a request to the server to connect. The server will process this request, and if the server doesn't deny the connection, will send a response back to the client, with proper information to proceed.

The major steps are:

1. Client sends a connect request.
2. If the server accepts, it will send the current map.
3. The server will wait for the client to load this map.
4. Once loaded, the server will then locally call **AGameModeBase::PreLogin**
    - This will give the **GameMode** a chance to reject the connection
5. If accepted, the server will then call **AGameModeBase::Login**
    - The role of this function is to create a **PlayerController**, which will then be replicated to the newly connected client. Once received, this PlayerController will replace the clients temporary **PlayerControlle**r that was used as a placeholder during the connection process.
    - Note that **APlayerController::BeginPlay** will be called here. It should be noted that it is NOT yet safe to call RPC functions on this actor. You should wait until **AGameModeBase::PostLogin** is called.
6. Assuming everything went well, **AGameModeBase::PostLogin** is called.
    - At this point, it is safe for the server to start calling RPC functions on this **PlayerController**.

Remote Procedure Calls (RPCs)

Remote Procedure Calls are also referred to as replicated functions. They can be called from any machine, but will direct their implementation to happen on a specific machine that is connected to the network session.

There are three types of RPCs available:

- **Server:** Called only on the server that is hosting the game.
- **Client:** Called only on the client that owns the Actor that the function belongs to. If the Actor has no owning connection, this logic will not be executed.
- **NetMulticast:** Called on all clients that are connected to the server as well as the server itself.

Networking Tips

- Use as few RPCs or replicated Blueprint functions as possible. If you can use a RepNotify instead, you should.
- Use Multicast functions especially sparingly, as they create extra network traffic for each connected client in a session.
- Server-only logic does not necessarily have to be contained in a server RPC if you can guarantee that a non-replicated function will only execute on the server.
- Be cautious when binding Reliable RPCs to player input. Players can repeatedly press buttons very rapidly, and that will overflow the queue for Reliable RPCs. You should use some way of limiting how often players can activate these.
- If your game calls an RPC or replicated function very often, such as on Tick, you should make it Unreliable.
- Some functions can be recycled by calling them in response to gameplay logic, then calling them in response to a RepNotify to ensure that clients and servers have parallel execution.
- You can check an Actor's network role to see if it is **`ROLE_Authority`** or not. This is a useful method for filtering execution in functions that activate on both server and client.
- You can check if a Pawn is locally controlled by using the **`IsLocallyControlled`** function in C++ or the Is Locally Controlled function in Blueprint. This is useful for filtering execution based on whether it is relevant to the owning client.
- Avoid using **`IsLocallyControlled`** in constructor scripts, as it is possible for a Pawn not to have a Controller assigned during construction.