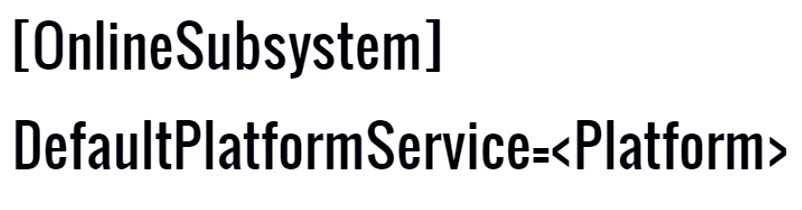
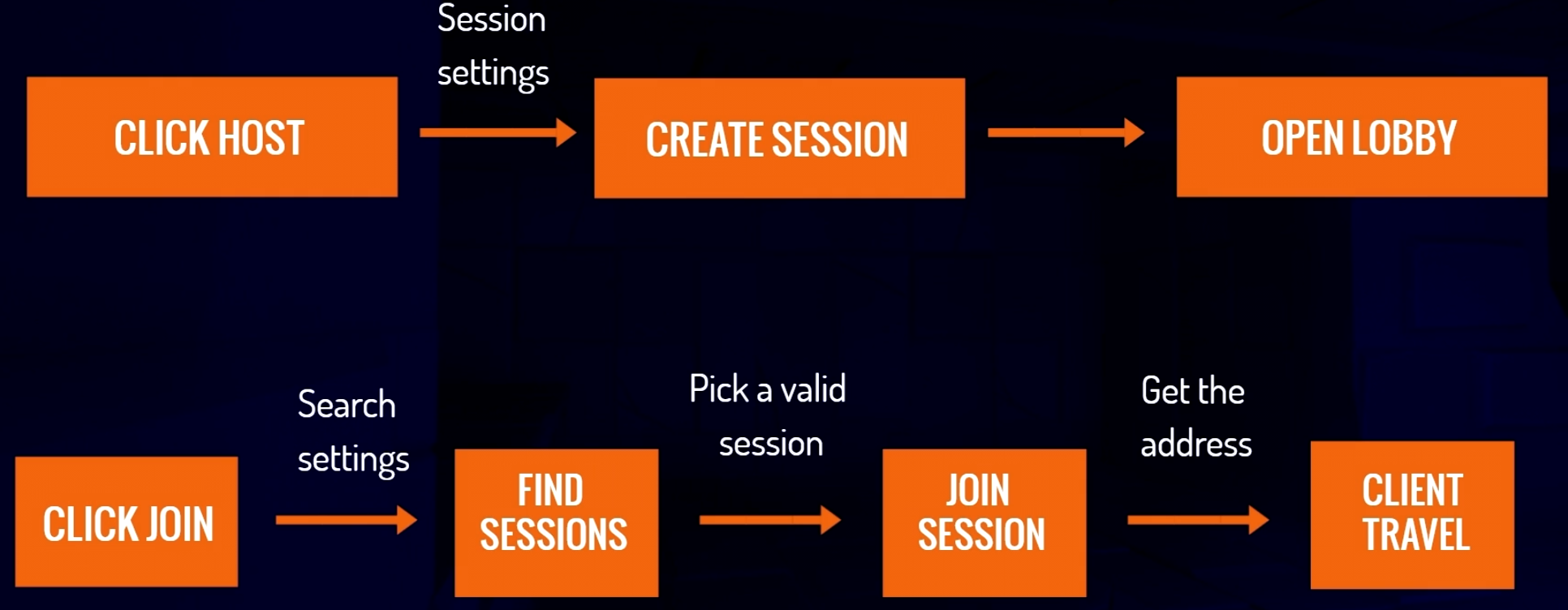
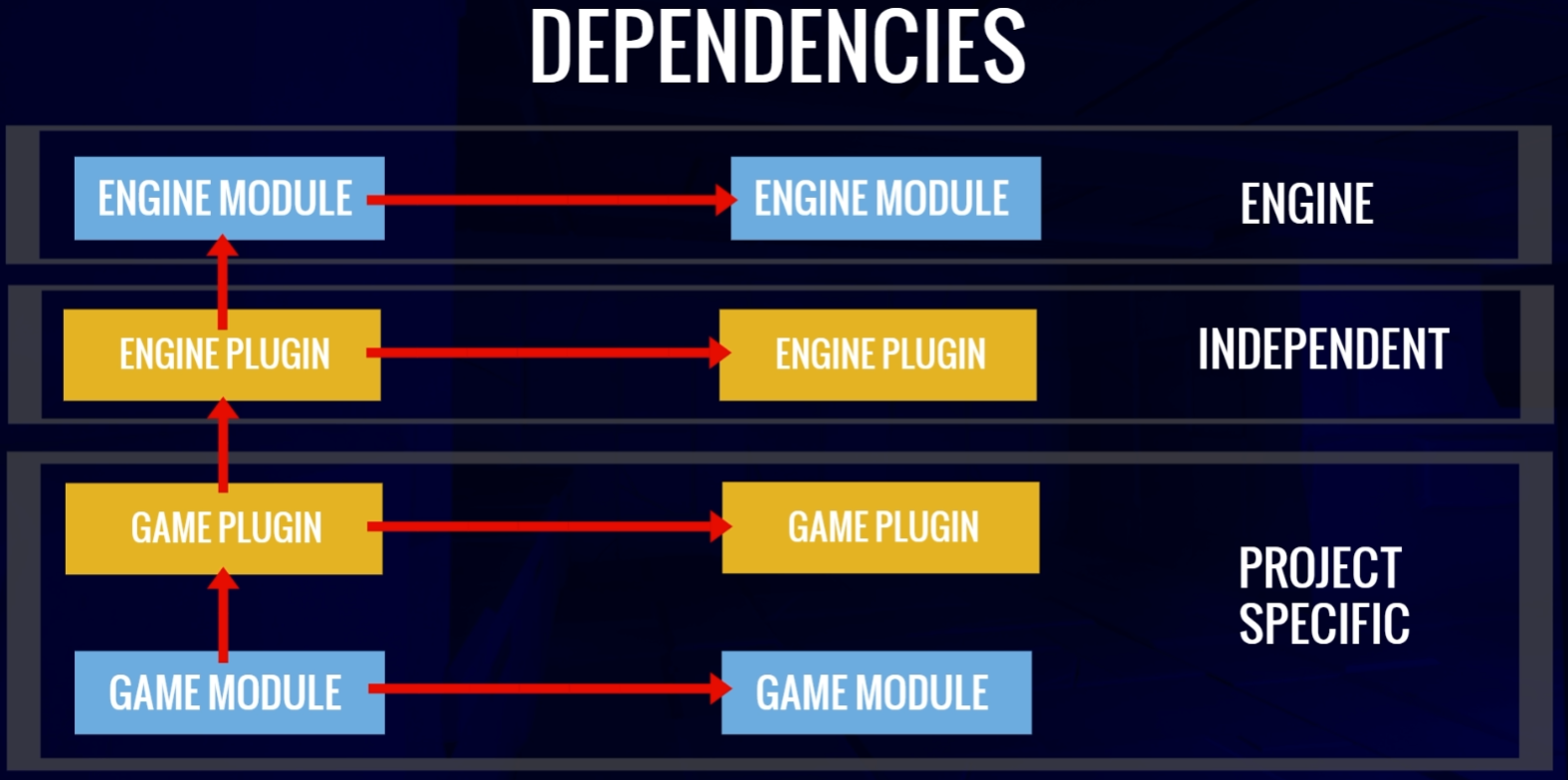
Blaster – Multiplayer Shooter

Unreal Engine 5 Multiplayer Tutorial by Stephen Ulibarri – documentation

* C++ objects
  + Multiplayer Plugin
    - UGameInstanceSubsystem
      * UMultiplayerSessionsSubsystem
    - UUserWidget
      * UMenu
  + Blaster
    - ACharacter
      * ABlasterCharacter
    - UDataAsset
      * UInputConfigDataAsset
    - UAnimInstance
      * UBlasterAnimInstance
    - AGameMode
      * ALobbyGameMode
* UE objects
  + Multiplayer Plugin
    - UMenu
      * WBP\_Menu
  + Blaster
    - ABlasterCharacter
      * BP\_BlasterCharacter
    - ALobbyGameMode
      * BP\_LobbyGameMode
    - UBlasterAnimInstance
      * ABP\_BlasterCharacter
    - UInputConfigDataAsset
      * DA\_InputConfig
    - UInputAction
      * IA\_Move
      * IA\_Look
      * IA\_Jump
    - UInputMappingContext
      * IMC\_Blaster
* Course Topics
  + Online Subsystem
    - Provides a way to access functionality of online platform (steam, xbox live, etc.) services
    - Each of platform has its own set of services upport for things like friends, achievements, setting up matchmaking sessions, etc. Online Subsystem contains a set of interfaces designed to handle these diefferent services for each platform
    - We can use Online Subsystem to handle our use of these interfaces no matter which service we choose, all we must do is configure our project for a particular platform
      * We specify default platform service in Engine.ini file
    - The Online Subsystem is a class of type IOnlineSubsystem, which can be accessed through the static IOnlineSubsystem\* IOnlineSubsystem::Get() function
    - Session Interface
      * Handles creating, managing, destroying, searching game sessionsand other matchmaking functionality
      * Session – An instance of the game running on the server.
        + Advertised (players can join)
        + Private (invite only)
      * Lifetime of a session
        + Create session
        + Wait for players to join
        + Register each one as they come in
        + Start session
        + Play
        + End session
        + Unregister players
        + Update/Destroy session
    - Game plan
  + Plugins
    - Collection of code and data
    - Easy to enable/disable per project
    - Runtime gameplay functionality
    - Editor functionality
    - Made of modules
      * A distinct unit of C++ code with its own build file
      * Encapsulate their funcionality
      * Just code, no other assets like meshes or textures
    - Dependencies
      * Our project itself is a module
      * Whenever we enable a plugin, it’s added to our project’s .uproject file
      * Plugin itself can also depend on other plugins, these are added to .uplugin file
      * Plugins and modules are broken into hierarchical levels and can only depend on other plugins and modules at their same level or higher
      * Our game project module can depend on an engine module, but and engine module cannot depend on our game (the engine modulest must be able to be built without any project)
  + Game Instance class
    - Spawned at game creation
    - Not destroyed until the game is shut down
    - Persist between level
  + [Subsystems in Unreal Engine](https://dev.epicgames.com/documentation/en-us/unreal-engine/programming-subsystems-in-unreal-engine?application_version=5.3)
    - Automatically instanced classes with managed lifetimes
    - Provide easy to use extension points, where the programmers can get Blueprint and Python exposure right away while avoiding the complexity of modifying or overriding engine classes
    - Supported subsystems
      * Engine
      * Editor
      * GameInstance
      * LocalPlayer
  + Delegate handles
    - Steps of creating delegate
      * Construct delegate
      * Bind delegate
      * Add delegate to the delegate list
    - Session interface functions used to add delegate return FDelegateHandle which we can store to unbind when we don’t need it anymore
      * AddOn...Delegate\_Handle()
      * ClearOn...Delegate\_Handle()
  + Game Mode and State
    - Game Mode
      * Rules for the game
      * Moving players to levels
      * Selecting spawn locations
      * Tracking when players enters or leaves the game
        + PostLogin(APlayerController\*) - called whenever player joins the game
        + Logout(AController\*) - called when player leaves the game
    - Game State
      * Holds state information about the game not individual players
      * Contains array of player states (player specific information)
      * Accessed by Game Mode
  + Ways to travel
    - Non-seamless vs seamless
      * Non-seamless
        + Not prefferd way to travel in Unreal Engine
        + Client disconnects from its current server and reconnects. This takes time and there can be issues when reconnecting.
        + Occurs when loading a map for a first time,connecting to a server for the first time and when multiplayer match/game ends and a new one begins
      * Seamless
        + Smoother experience – client doesn’t have to disconnect from the server
        + Avoids any reconnecting issues – being unable to find the server or the server having too many players and not allowing the player to rejoin
        + Seamless travel can be enabled in a game mode – *bUseSeamlessTravel = true;*
        + Simple transition map/level is needed, because a map must always be loaded at any given point in time

Travel from one map to another without a transition map, we would have to load the second map while the first one is still loaded – it needs more resources

Transition map is used and loaded before deconstructing the original map that we were in already and after deconstructing we can load the new map and two large maps are not exisiting simultaneously

* + - Travel in Multiplayer
      * Server Travel
        + *UWorld::ServerTravel*
        + For the server only, the server machine calls server travel and the server jumps to a new level
        + All clients will follow to the map the server is traveling to. The server does this by getting all connected player controllers and calling the function *APlayerController::ClientTravel*
      * Client Travel
        + When called from a client, the client will travel to a new server. The clinet must provide an address to travel to
        + When called from a server, the client will travel to the new map specified by the server
  + Network Role
    - In multiplayer game, there are multiple versions of any given character controlled by player
    - If client is connected to a server, then the character that it is currently controlling exists on its machine but there also exists a version of its character on the host’s machine and if there are more clients then its character exists on each client’s machine (3 player game – 3 copies of character, one on each machine)
    - It is important to know how distinguish which version of the character we are dealing with in the code – here comes Unreal Engine’s concept of Role – *ENetRole*
    - *ENetRole is an enum that has several enum constants that we can use to identify the role of any given character or pawn*
      * *ENetRole::ROLE\_Authority*, assigned to pawns that exist on the server machine
      * *ENetRole::ROLE\_SimulatedProxy*, assigned to pawns that exist on a machine that is not controlling them
      * *ENetRole::ROLE\_AutonomousProxy*, assigned to pawn that exist on a machine that is controlling it
      * *ENetRole:ROLE\_None*, for actors that don’t have any defined role
      * Disclaimer:
        + These above are for local role

Remote role for non server players are authority

Remote role for server player are Autonomous/Simulated Proxy

* + - * + On server every pawn is authority (Pawn controlled by host)