Server Administration Plugin UE 4.26

CONTENTS

1. FOREWORD

This guide will demonstrate the setup of the Server Admin plugin and shows you where in your game's code and Blueprints it needs to be integrated to make full use of its functionality. We are going to be using the ShooterGame code as an example. So, wherever a ShooterGame class is changed in this guide, make these same changes to your corresponding game classes.

The fully prepared version of ShooterGame is available for download on the server admin plugin page, too, if you desire to look at the code there.

2. UPROJECT & BUILD FILES

UPROJECT

Let's start by installing the plugin to your engine, which is done via the Epic launcher. Once that's done, you might want to move it to the GitHub version of the engine. This is not strictly needed but a requirement for building dedicated servers, so you might end up taking that step anyway.

After installing it, it is available to the engine but needs to be referenced by your project. Open your uproject file and add the following lines to the plugins list, right below Gauntlet in our example:

Alternatively, you can add the plugin through the plugins dialogue in UE4.

```
72          "Name": "Gauntlet",
73          "Enabled": true
74      },
75      {
76          "Name": "TIServerAdmin",
77          "Enabled": true,
78          "MarketplaceURL": "com.epicgames.launcher://ue/marketplace/content/7d3b7754dab44c4ebf57eeb926f0cc11"
79      }
```

Note: You do not need line 78 with the MarketPlaceURL if you are adding this definition manually.

SHOOTERGAME.BUILD.CS

In the ShooterGame build file in line 33, add the following line to the PublicDependencyModuleNames: "TIServerAdmin".

```
26          PublicDependencyModuleNames.AddRange(
27              new string[] {
28                  "Core",
29                  "CoreUObject",
30                  "Engine",
31                  "OnlineSubsystem",
32                  "OnlineSubsystemUtils",
33                  "AssetRegistry",
34                  "NavigationSystem",
35                  "AIModule",
36                  "GameplayTasks",
37                  "Gauntlet",
38                  "TIServerAdmin",
39              }
```

3. PREPARING YOUR GAME CLASSES

Depending on the type of game you want to add this plugin to, you'll want to expose some of your game's functionality, so the plugin has access to your game logic.

EVENTS

For the plugin to know what is happening in your game, some game specific logic needs to be setup to call the plugin's functions, such as the Event that requests the next map, or records a team kill for the TK punishment measures to kick in. In this guide, this will be achieved by adding the required hooks in the form of Blueprintable Events into the game mode of ShooterGame.

SHOOTERGAMEMODE.H

We cannot easily call the plugin functions from ShooterGame's code directly, as that would create build dependencies between the project and the plugin that go against the idea of the UE4 plugin architecture.

So, at the ShooterGameMode header's bottom, add the three shown function declarations, which will provide is with three Blueprint Event nodes once we compile the example game.

```
162     // ServerAdmin
163     // Event for when a match is over and the next map needs determining
164     UFUNCTION(BlueprintImplementableEvent, Category = "GameMode")
165         void OnNextMap();
166
167     // Blueprint getter for reading the value of the Game Mode component's TK setting
168     UFUNCTION(BlueprintImplementableEvent, Category = "GameMode")
169         bool GetTeamKillsEnabled() const;
170
171     // Event for when a team kill occurred
172     UFUNCTION(BlueprintImplementableEvent, Category = "GameMode")
173         void OnTeamKill(AController* Killer);
174
```

Second, add two more function declarations, as shown below. They're the Blueprint Getter and Setter for the game mode's round time and make this variable read/write accessible via BP. Without them, neither the plugin, nor your BP_GameMode can see or modify it.

```
177
178     // Getter for the roundtime
179     UFUNCTION(BlueprintPure, Category = "GameMode")
180         float GetRoundTime() { return RoundTime; };
181
182     // Setter for the roundtime
183     UFUNCTION(BlueprintCallable, Category = "GameMode")
184         void SetRoundTime(float NewRoundTime);
185 };
```

SHOOTERGAMEMODE.CPP

In the ShooterGameMode class, we will implement the calls to the Events that we defined earlier.

1. In ::DefaultTimer(), line 138, replace the function call *RestartGame()* with *OnNextMap()*

```
101     if (MyGameState->RemainingTime <= 0)
102     {
103         if (GetMatchState() == MatchState::WaitingPostMatch)
104         {
105             OnNextMap();
106         }
```

2. In the ::Killed() function, near line 310, implement the call *OnTeamKill()*, if both the Killer's and Killed's team are identical.

```
310         // Call the ServerAdmin parent function to increase the TK Counter
311         if (KillerPlayerState->GetTeamNum() == VictimPlayerState->GetTeamNum())
312             OnTeamKill(Killer);
313     }
```

3. Go to the ::CanDealDamage() function and enhance it with the code shown below. These few lines are checking whether team kills are enabled and returns the appropriate value based on the result. This is where the GetTeamKillsEnabled() call comes into play. In our example, it will go to the GameMode blueprint and query the TIServerAdmin game mode component for the setting's value.

```
514 bool ARnLGameMode::CanDealDamage(class ARnLPlayerState* DamageInstigator, class ARnLPlayerState* DamagedPlayer) const
515 {
516     // Check whether team kills are enabled in the server admin settings
517     const bool TeamKillsEnabled = GetTeamKillsEnabled();
518
519     if (DamagedPlayer && DamageInstigator && DamageInstigator->GetTeam() == DamagedPlayer->GetTeam() && !TeamKillsEnabled)
520         return false;
521
522     return true;
523 }
```

4. Below the three events, let's add the function that modifies the GameMode's round time:

```
575
576 void AShooterGameMode::SetRoundTime(float NewRoundTime)
577 {
578     if (NewRoundTime >= 0.0f)
579         RoundTime = NewRoundTime;
580 }
581
```

5. Something worth noting in this class still is, that the DefaultTimer() function, the GameMode's tick does not run fully in the editor. So, if you were to test the plugin in the editor, you'd get the game stuck. To remedy this, locate line 93 and comment out the return; call.

```
85      // don't update timers for Play In Editor mode, it's not real match
86      if (GetWorld()->IsPlayInEditor())
87      {
88          // start match if necessary.
89          if (GetMatchState() == MatchState::WaitingToStart)
90          {
91              StartMatch();
92          }
93          //return;
94      }
```

SHOOTERGAMESTATE.H

In the ShooterGameState, we need to make the RemainingTime BP read/write accessible, so the plugin can modify those via BP. This can be achieved purely in C++, too, but would require casting to Game Classes which may or may not exist in your game, so we'll go the BP route instead here.

```
41
42      //////////////////////////////////////////////////////////////////////////
43      // ServerAdmin
44      //////////////////////////////////////////////////////////////////////////
45
46      // Getter for the remaining time in a round
47      UFUNCTION(BlueprintPure, Category = "GameState")
48          float GetRemainingTime() { return RemainingTime; };
49
50      // Setter for the remaining time in a round
51      UFUNCTION(BlueprintCallable, Category = "GameState")
52          void SetRemainingTime(float NewRemainingTime);
```

Declare the Getter and setter for the RemainingTime variable, as we did above for the RoundTime.

SHOOTERGAMESTATE.CPP

And now the corresponding function:

```
100
101 void AShooterGameState::SetRemainingTime(float NewRemainingTime)
102 {
103     if (NewRemainingTime >= 0)
104         RemainingTime = NewRemainingTime;
105 }
106
```

OPTIONAL: SHOOTERGAMEINSTANCE

If you are not using ShooterGame's slate UI implementation, but went with UMG instead, the Server Admin plugin offers an interface function to call UMG UI in the case of a new Admin Message. This is an optional implementation shown below.

Note, if you stick with Slate and implement this function below, too, you may end up with overlapping UI elements.

The game instance is the only class that we cannot add a Blueprint component to, so instead we will use an interface to call the TIServerAdmin functions. You only need to include the interface' header, as the interface itself is implemented in the TIServerAdmin class that it's called on.

```
3  /*=============================================================================
4      ShooterGameInstance.cpp
5  =============================================================================*/
6
7  #include "ShooterGame.h"
8  #include "ShooterGameInstance.h"
9  #include "ShooterMainMenu.h"
10 #include "ShooterWelcomeMenu.h"
11 #include "ShooterMessageMenu.h"
12 #include "ShooterGameLoadingScreen.h"
13 #include "OnlineKeyValuePair.h"
14 #include "ShooterStyle.h"
15 #include "ShooterMenuItemWidgetStyle.h"
16 #include "ShooterGameViewportClient.h"
17 #include "Player/ShooterPlayerController_Menu.h"
18 #include "Online/ShooterPlayerState.h"
19 #include "Online/ShooterGameSession.h"
20 #include "Online/ShooterOnlineSessionClient.h"
21 #include "OnlineSubsystemUtils.h"
22 #include "TIServerAdminIGameInstance.h"
23
```

To continue the optional implementation we started above, locate the ::BeginMessageMenuState() function around line 800. Add the code below, to call the interface function described above. This will call the interface implemented in the ServerAdmin game instance, if you chose to go with that one or the BP interface function that will be described further down the line.

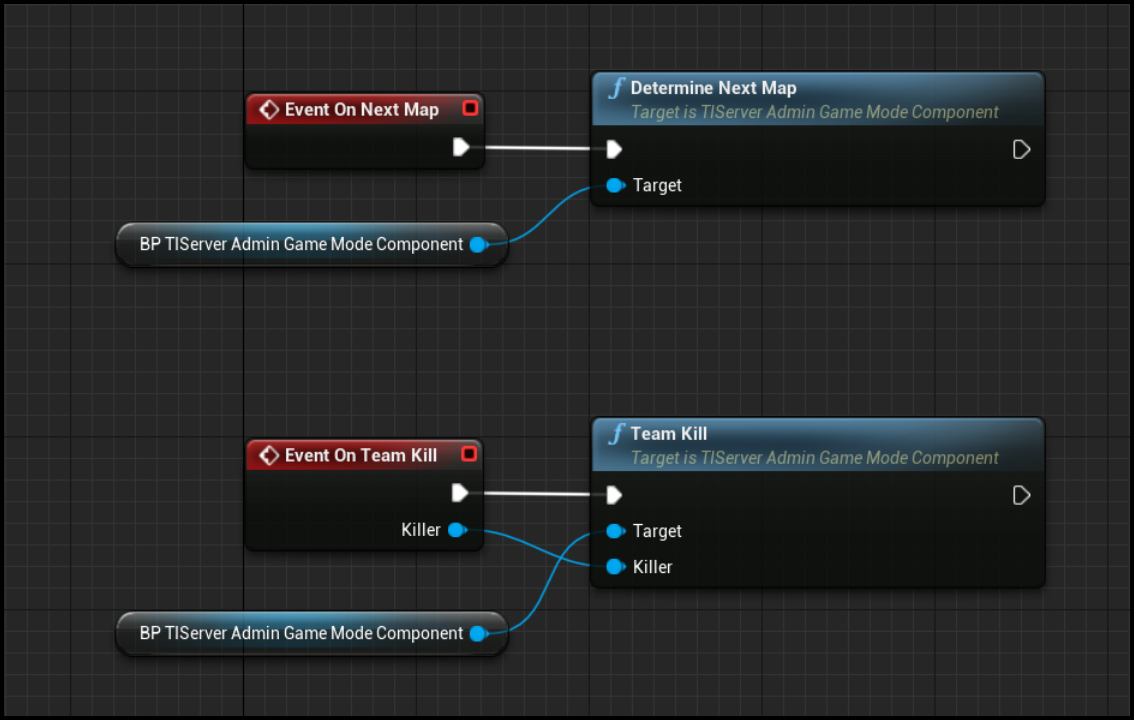
```
808     /**
809     * Server Admin: Optional, Send the new message to the Game Instance Blueprint or class
810     * where it can be received via this interface call and displayed
811     * in case you chose not to use ShooterGame's Slate UI implementation but UMG instead
812     */
813     if (GetClass()->ImplementsInterface(USAIGameInstance::StaticClass()))
814     {
815         ISAIGameInstance::Execute_OnNewAdminMessage(this, PendingMessage.DisplayString,
816             PendingMessage.OKButtonString, PendingMessage.CancelButtonString, PendingMessage.NextState);
817     }
818
```

This concludes the work needed in C++ and we can move on to the Blueprints portion of this guide. At this point, the plugin should be implemented fully and your project compiles and launches.

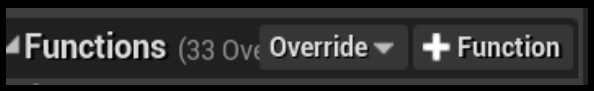
BP_GAMEMODE

The last step of exposing game functions to the plugin is to make the three Events we defined in the GameMode class earlier work in Blueprint.

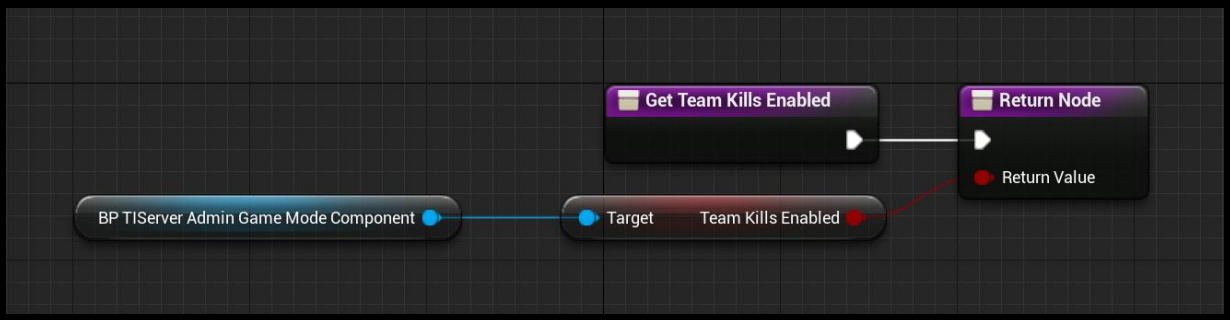
Two of the Events go into the Event Graph and call two functions in the plugin, which is done as shown below:



And the third Event, which has a return value is a function override, which you can add by clicking the Override button in the functional panel of your GameMode Blueprint:



And choosing *GetTeamKillsEnabled.* With that function overridden, go ahead and read the plugin's *TeamKillsEnabled* variable value for the return value:



This concludes the work that is needed to expose the ShooterGame default functionality to BP for the plugin to hook into it.

BLUEPRINT FUNCTIONS AND DELEGATES

Now we start hooking the plugin back into your game by interfacing several of the plugin functions with your game code. Let's get started by creating three blueprints:

1. **BP_GameMode**: Inherits from the **ShooterGameTeamDeathMatch** game mode
2. **BP_GameState**: Inherits from **ShooterGameState**
3. **BP_GameInstance**: Inherits from **BP_TIServerAdminGameInstance**
4. **BP_PlayerController**: Inherits from **ShooterGamePlayerController**

When those are done, add the Blueprint ServerAdmin components to each:

1. **BP_GameMode:** Add the **BP_TIServerAdminGameMode_Component** to it
2. **BP_GameState:** Add the **BP_TIServerAdminGameState_Component** to it
3. **BP_PlayerController:** Add the **BP_TIServerAdminActor_Component** to it

Note: If you prefer to have your implementation done in C++, this is also possible. You'll have to take the steps described here in BP in C++ instead and create delegates that do the work.

BP_GAMEINSTANCE

You might have noticed that BP_GameInstance has no component and that its parent, BT_TIServerAdminGameInstance has ShooterGameInstance as parent.

The reason for this is, that the GameInstance is the only class that we cannot enhance with a component, neither in C++ nor in BP. So, we need to have a TIServerAdmin game instance class that is layered between ShootergameInstance and your own BP_GameInstance.

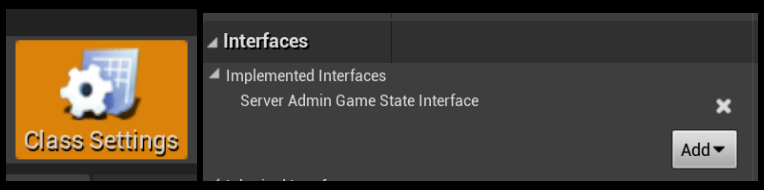
Note: You could work with the BP_TIGameInstance class in your project directly. That is an option, but opens your project up for conflicting changes in future updates of the TIServerAdmin plugin.

If you prefer to use C++ here, too, the plugin offers a ServerAdminGameInstance C++ class that is identical with the Blueprint version.

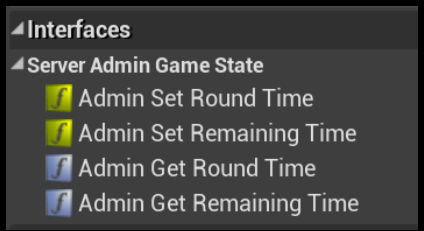
BP_GAMESTATE

In the BP_GameState, we'll hook up an interface that gets called by the server admin plugin. It's responsible for getting and setting the remaining time and round timer.

To do this, open the Blueprint class and go to the class settings, which will open the settings panel. There, add the *ServerAdmin Game State Interface*:

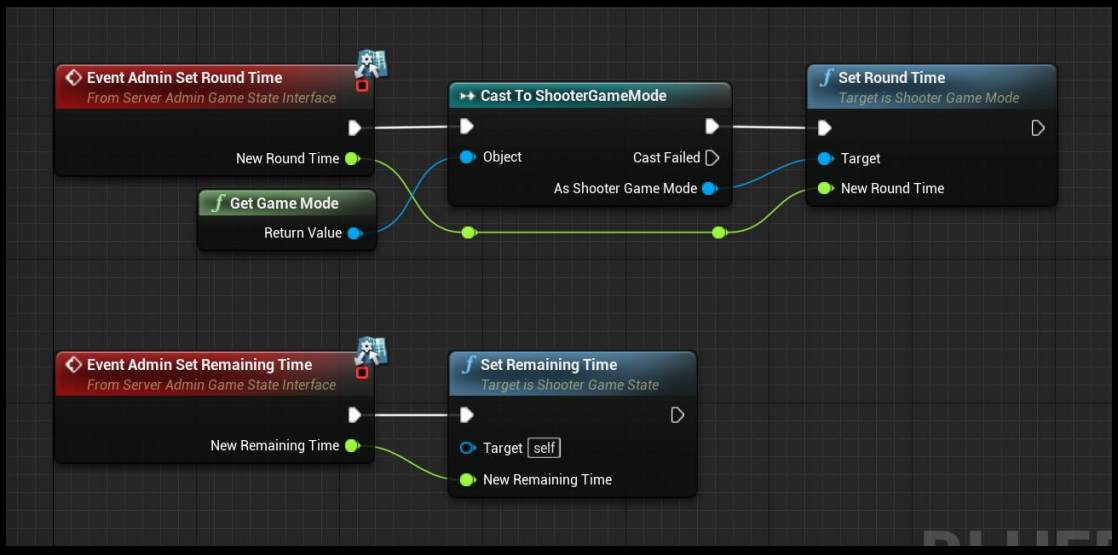


This will add four functions, as shown below:

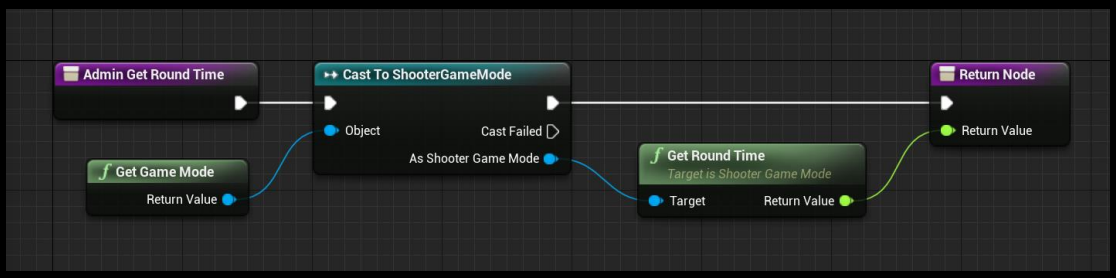


Now, to implement the four functions. This is where the C++ getters and setters we made earlier come into play.

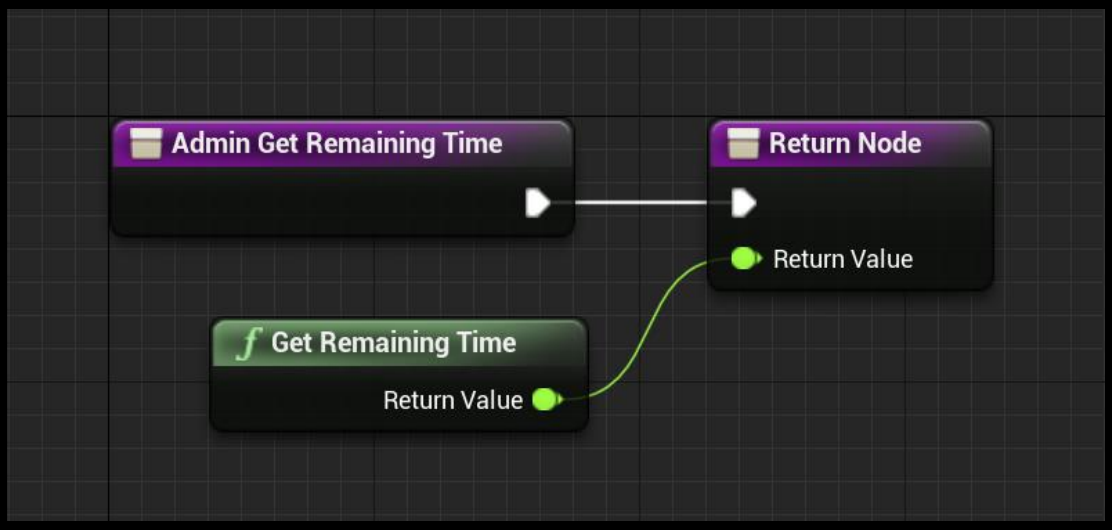
The two setters are implementable Events in the EventGraph of the BP_GameState class. Add them as shown below:



The two getters are functions instead, as they return a value. First, the Round Time setter:



And then the Remaining time setter:



GAME MODE AND .INI SETTINGS

Last but not least, your BP_GameMode class settings need to be modified to use the newly created BP classes, as shown here:

- GameState is set to BP_GameState
- PlayerController is set to BP_PlayerController
- In your game settings, make sure BP_GameMode is the new default, or set in the world override of the map you want to test this in.
- Finally, in the DefaultGame.ini, replace the PlatformPlayerControllerClass in line 9 and the GameInstance class in the DefaultEngine.ini in line 221.

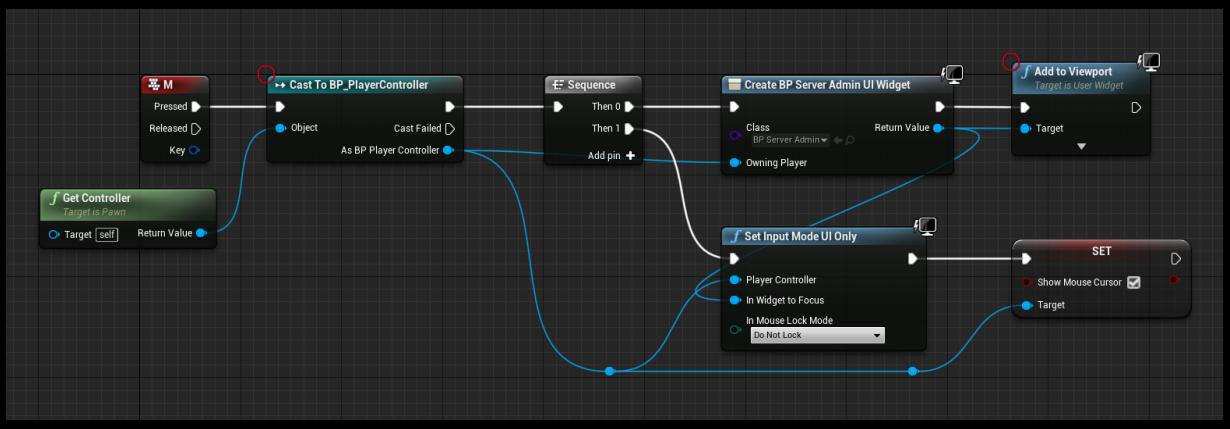
```
9  ;PlatformPlayerControllerClass=Class'/Script/ShooterGame.ShooterPlayerController'
10 PlatformPlayerControllerClass=Class'Game/Blueprints/GameMode/BP_PlayerController.BP_PlayerController_C'
11
```

```
221 ;GameInstanceClass=/Script/ShooterGame.ShooterGameInstance
222 GameInstanceClass=/Game/Blueprints/GameMode/BP_GameInstance.BP_GameInstance_C
```

4. CALLING THE SERVER ADMIN MENU IN YOUR GAME

This last step will show you how to call the TIServerAdmin UMG menu. This implementation is “quick and dirty” and it is strongly encouraged to integrate it into your own in-game menu structure instead. However, for a proof of concept the following will do.

Let’s open ShooterGame’s PlayerPawn Blueprint and add this bit of logic:



Grab the Pawn’s Controller, cast it to the PlayerController class and then place a *CreateWidget* node. In the *Create Widget* node, reference **BP_ServerAdmin_UI** as class.

This logic will create and open the TIServerAdmin menu when you press the key “M” while you are playing in game.

Important note: This plugin is made for a server-client environment. Make sure to tick the “Run Dedicated Server” checkbox when testing the functionality in the editor!

5. CONFIGURATION

Now, with all the logic done, it's time to configure the GameMode.ini to store the TIServerAdmin configuration. This is where your game's server owners will be able to create their own configurations, prior to using the plugin.

Note: In one of the future releases, the plugin will get its own configuration file, but for the time being, it's using the game.ini.

```
47
48  [ServerAdmin]
49  ; Every server admin can conveniently be added here.
50  ; Enter their 64bit SteamID to the list of ServerAdminSteamIDs
51  ; and they will automatically be verified when they join.
52  +ServerAdminSteamID=12345678910111213
53  ;+ServerAdminSteamID=12345678910111213
54  ; Use the password option if you cannot or do not want to use any of the
55  ; unique OnlineSubSystem user IDs.
56  ; Note: Adding a ServerAdminSteamID will disable the password prompt.
57  AdminPassword="ThisIsMyPassword"
58  ; Friendly Fire enabled or disabled
59  TeamKillsEnabled=1
60  ; Team Kill threshold until the TKer is punished
61  TeamKillLimit=5
62  ; Team Kill punishment: 0 = kick, 1 = ban
63  TeamKillPunishment=0
64  ; Team Kill count decay. Decreases the count of accumulated TKs per player by one per X minutes
65  TeamKillCountDecay=1
66  ; Configurable path for looking up the game's maps'
67  MapsPath="/Game/Maps"
68  ; Maps excluded from showing up in the admin menu
69  ; such as the entry map or submaps that live in the Game/Maps folder
70  ; Note: It is recommended to store sublevels in a separate folder of Game/Maps/.
71  ; That will automatically exclude them from the map list in the admin menu.
72  +ExcludedMaps="ShooterEntry"
73  +ExcludedMaps="Highrise_Audio"
74  +ExcludedMaps="Highrise_Gameplay"
75  +ExcludedMaps="Highrise_Lights"
76  +ExcludedMaps="Highrise_Meshing"
77  +ExcludedMaps="Highrise_Vista"
78  +ExcludedMaps="Highrise_Collisions_Temp"
79  ; Map Rotation List. Add all maps that are supposed to run by default here
80  +MapRotation="/Game/Maps/Highrise"
81  +MapRotation="/Game/Maps/Sanctuary"
82
```

1. The first entries are the server admins' UniqueNetIDs. In the case of using the Steam subsystem they are the 64bit version of the player's SteamIDs
2. The second entry is the optional password. Keep in mind three things:
    a. The password method is only active if no Admin IDs are defined, otherwise it's inactive
    b. The password is transmitted unencrypted, so it's potentially unsafe and it's highly recommended to only use it in networks that you trust.
    c. Providing neither a password nor Steam IDs will make the TIServerAdmin controls accessible to any player in your server. This is useful for debug purposes but should not be used on a public server.
3. The third, fourth and fifth entry are the setting for the TeamKill feature being enabled, the TK limit beyond which the punishment kicks in, the type of punishment and the TK counter decay per minute.
4. The Maps path defines the location on your server, where the plugin should look for maps to display as options to switch to. Note, the plugin does not check folders recursively.
5. The Excluded maps list holds all maps and sublevels that you do not want to show up in the map selection.
6. And last but not least, your server's map rotation is defined at the bottom. The server will indefinitely loop through those maps.