Unreal Engine 측에서의 작업은 Python 개발과는 다른 환경과 언어(주로 C++ 및 Blueprint)를 사용하며, UE 에디터 내에서의 설정 작업이 필수적입니다. Python 시스템이 클라이언트로서 UE 서버로 데이터를 보내므로, UE 는 이 데이터를 수신하고 처리하는 서버 역할을 해야 합니다.

Unreal Engine 에서 구성 및 개발해야 할 사항:

1. **C++ 모듈/플러그인 생성:** 저수준의 네트워크 통신(TCP/IP 소켓) 및 데이터 처리(메시지 프레이밍 파싱, JSON 파싱)는 UE 의 C++ API 를 사용하는 것이 가장 효율적입니다. 별도의 UE 모듈 또는 프로젝트 플러그인으로 구현하는 것이 좋습니다.
2. **통신 서버 컴포넌트 (C++ Actor Component 또는 World Subsystem):**
    - UE 레벨이 로드될 때 시작되고 언로드될 때 종료되는 라이프사이클 관리가 가능한 객체가 필요합니다. Actor Component 를 빈 액터에 붙이거나, World Subsystem (UE 4.22+ 권장)으로 구현하여 게임 모드나 레벨과 무관하게 항상 존재하도록 할 수 있습니다. World Subsystem 이 시스템 매니저 역할에 더 적합할 수 있습니다.
    - 이 컴포넌트는 Python physics_interface.py 로부터 TCP/IP 연결을 수락하고 유지하며, 수신되는 데이터를 관리하는 역할을 합니다.
3. **데이터 수신 및 파싱 로직 (C++/Blueprint):**
    - 수신 스레드 관리: UE 의 게임 스레드를 블록하지 않도록, 네트워크 수신은 별도의 백그라운드 스레드에서 처리해야 합니다 (FRunnable 인터페이스 사용).
    - 메시지 프레이밍 파싱: 수신 스트림에서 4 바이트 길이 접두사를 먼저 읽고, 그 길이만큼의 데이터를 정확히 읽어와 하나의 완전한 메시지를 복원하는 로직을 구현해야 합니다.
    - JSON 파싱: 복원된 메시지(JSON 문자열)를 UE 의 JSON 파싱 API(FJsonReader, FJsonObject)를 사용하여 파싱하고, Python 에서 보낸 "action", "robot_id", "parameters" 등의 필드를 추출합니다.
4. **명령 디스패치 로직 (C++/Blueprint):**
    - 파싱된 JSON 메시지의 "action" 필드를 기반으로 어떤 처리를 수행할지 분기합니다 (예: "set_robot_pose", "welding_visual_command", "run_simulation").
    - 이 처리는 게임 스레드에서 안전하게 수행되어야 하므로, 수신 스레드에서 파싱된 메시지를 게임 스레드로 전달하는 메커니즘(예: TQueue 및 Delegate)이 필요합니다.
5. **로봇 액터 관리 및 업데이트 (Blueprint/C++):**
    - **로봇 액터 클래스 생성:** 기존 로봇 에셋(스켈레탈 메시 등)을 사용하는 Blueprint Actor 또는 C++ Actor 클래스를 생성합니다.
    - **로봇 등록:** 통신 서버 컴포넌트가 로드될 때, UE 레벨에 배치된 로봇 액터들을 찾아서 (예: 특정 태그 사용, Actor Name 사용) Python 의 robot_id 와 매핑하여 내부 맵(Map)에 저장합니다 (예: TMap<int32, ARobotActor*>). Python 에서 오는 메시지의 robot_id 로 해당 로봇 액터를 빠르게 찾을 수 있게 합니다.

    - **자세 업데이트 로직:** 로봇 액터 클래스 내부에 "set_robot_pose" 명령을 처리하는 함수(예: SetRobotPoseFromJSON)를 구현합니다. 이 함수는 JSON 데이터에서 관절 각도 또는 TCP 트랜스폼 데이터를 추출하여, 로봇 스켈레탈 메시의 해당 본(Bone)들의 트랜스폼을 실시간으로 업데이트합니다. 애니메이션 블루프린트에서 제어 본(Control Bone)을 사용하거나, 직접 본의 상대/월드 트랜스폼을 설정하는 방식이 사용됩니다.
    - **시각 효과 제어 로직:** 로봇 액터 클래스 내부에 "welding_visual_command" 명령을 처리하는 함수(예: HandleWeldingVisualCommand)를 구현합니다. 이 함수는 "arc_on", "arc_off" 등의 command_type 을 받아 용접 토치 위치에 미리 준비된 파티클 시스템(아크), 라이트, 또는 비드 메시의 가시성을 제어하거나 파라미터를 업데이트합니다.
6. **UE 레벨 설정:** 통신 서버 컴포넌트가 붙을 매니저 액터를 레벨에 배치하고, 로봇 액터들을 원하는 위치에 배치합니다. 로봇 액터에 Python 에서 사용할 robot_id 를 식별할 수 있는 정보(Actor Tag, Actor Name 등)를 설정합니다.
7. **물리 시뮬레이션 로직 (C++/Blueprint):** "run_simulation" 명령을 받았을 때, 해당 파라미터를 사용하여 UE 물리 엔진 또는 커스텀 시뮬레이션 로직을 실행하고, 결과를 계산하여 Python 으로 반환하는 로직을 구현합니다.
8. **데이터 반환 로직 (C++/Blueprint):** "get_sim2real_ark_situation" 명령을 받았을 때, 요청된 데이터를 생성하거나 조회하여 JSON 응답 메시지를 만들어 Python 으로 전송하는 로직을 구현합니다.
9. **오류 처리 및 로깅:** 네트워크 오류, 파싱 오류, 유효하지 않은 명령/데이터, 로봇 액터 찾기 실패 등 다양한 오류 상황에 대해 UE 로그 시스템을 사용하여 기록하고, 필요한 경우 Python 클라이언트로 오류 응답을 반환합니다.

핵심 C++ 코드 개념 (예시):

UE 프로젝트에서 새로운 C++ Actor Component 클래스 (예: UPythonCommServerComponent)와 새로운 C++ Actor 클래스 (예: ARobotBaseActor 또는 기존 로봇 스켈레탈 메시 액터 기반 수정)를 생성해야 합니다.

1. 통신 서버 컴포넌트 (UPythonCommServerComponent) 개념:

```
    // Header file (PythonCommServerComponent.h)

#pragma once

#include "Components/ActorComponent.h"
#include "Networking.h" // For FSocket, FTcpSocketListener etc.
```

```
#include "Json.h" // For JSON parsing
#include "HAL/Runnable.h" // For background thread
#include "Containers/Queue.h" // For message queue

#include "PythonCommServerComponent.generated.h"

// Define a struct to hold parsed messages for processing on the game
thread
struct FPythonMessage
{
    FString Action;
    int32 RobotId; // Or FString RobotName
    TSharedPtr<FJsonObject> Parameters;
    // Add Sequence ID if you need to track responses
};

// Declare a delegate to pass received messages from the worker thread to
the game thread
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FOnPythonMessageReceived, const
FPythonMessage&, Message);

// Declare a delegate for connection events
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnPythonClientConnected);
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnPythonClientDisconnected);


UCLASS( ClassGroup=(DigitalTwin), meta=(BlueprintSpawnableComponent) )
class YOURUNREALMODULE_API UPythonCommServerComponent : public
UActorComponent
{
    GENERATED_BODY()

public:
    UPythonCommServerComponent();
    virtual void BeginPlay() override;
    virtual void EndPlay(const EEndPlayReason::Type EndPlayReason)
override;
    virtual void TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction) override; // Used to process
messages from queue

    // --- Configuration ---
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Python Comm")
```

```
    FString ListenIP = TEXT("127.0.0.1"); // IP address to listen on

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Python Comm")
    int32 ListenPort = 9999; // Port to listen on (must match
physics_interface.py)

    // --- Events (exposed to Blueprint) ---
    UPROPERTY(BlueprintAssignable, Category = "Python Comm")
    FOnPythonMessageReceived OnMessageReceived; // Event triggered when a
message is received and parsed

    UPROPERTY(BlueprintAssignable, Category = "Python Comm")
    FOnPythonClientConnected OnClientConnected;

    UPROPERTY(BlueprintAssignable, Category = "Python Comm")
    FOnPythonClientDisconnected OnClientDisconnected;


    // --- Public Interface (Blueprint Callable) ---
    UFUNCTION(BlueprintCallable, Category = "Python Comm")
    bool StartServer();

    UFUNCTION(BlueprintCallable, Category = "Python Comm")
    void StopServer();

    // --- Utility for Blueprint to find registered robots ---
    // Need a way to register robots with this component
    // Example: BP_MyRobot calls this on BeginPlay
    UFUNCTION(BlueprintCallable, Category = "Python Comm")
    void RegisterRobotActor(int32 RobotId, AActor* RobotActor);

    // --- Send Response Back to Python (if needed for specific actions) --
-
    UFUNCTION(BlueprintCallable, Category = "Python Comm")
    bool SendJsonResponse(int32 ClientConnectionId, const FString& Status,
const FString& Message, const TSharedPtr<FJsonObject>& Data);


private:
    FTcpSocketListener* TcpListener;
    FSocket* ClientSocket; // We'll assume one client connection for
simplicity first
```

```
    // For multiple clients, you would need TArray<FSocket*> ClientSockets
and a handler per socket

    // Worker thread for receiving data
    class FSocketReceiveWorker : public FRunnable
    {
    public:
        FSocketReceiveWorker(FSocket* InSocket, TQueue<FPythonMessage>&
InMessageQueue, FEvent* InStopEvent);
        virtual bool Init() override;
        virtual uint32 Run() override;
        virtual void Stop() override;
        virtual void Exit() override;

    private:
        FSocket* ClientSocket;
        TQueue<FPythonMessage>& MessageQueue; // Queue to send messages to
the game thread
        FEvent* StopEvent; // Event to signal the thread to stop
        FThreadSafeBool bIsRunning; // Flag to check if the thread should
be running
    };
    FSocketReceiveWorker* ReceiveWorker;
    FRunnableThread* ReceiveThread;
    FEvent* ReceiveStopEvent; // Event object for the worker thread

    TQueue<FPythonMessage> ReceivedMessageQueue; // Queue to pass parsed
messages to game thread

    // Map to store registered robot actors
    TMap<int32, AActor*> RegisteredRobots; // Mapping Robot ID (from
Python) to UE Actor

    // Internal function to handle new client connections
    bool OnSocketConnectionAccepted(FSocket* ClientSocket, const
FIPv4Endpoint& ClientEndpoint);

    // Internal function to process messages from the queue on the game
thread
    void ProcessReceivedMessages();

    // Internal helper to send framed data
    bool SendFramedData(const FString& DataToSend);
```

```
};

      // Source file (PythonCommServerComponent.cpp) - Key implementations

#include "PythonCommServerComponent.h"
#include "Kismet/GameplayStatics.h" // For finding actors
#include "TimerManager.h" // For TickComponent timer (alternative to
TickComponent)
#include "HAL/RunnableThread.h"
#include "Misc/Base64.h" // If sending/receiving binary data like images

// Include your Robot Actor base class header if you have one
// #include "YourRobotBaseActor.h"

// --- FSocketReceiveWorker Implementation ---
FSocketReceiveWorker::FSocketReceiveWorker(FSocket* InSocket,
TQueue<FPythonMessage>& InMessageQueue, FEvent* InStopEvent)
    : ClientSocket(InSocket), MessageQueue(InMessageQueue),
StopEvent(InStopEvent)
{
    bIsRunning = true;
}

bool FSocketReceiveWorker::Init()
{
    UE_LOG(LogTemp, Log, TEXT("FSocketReceiveWorker Init"));
    return true;
}

uint32 FSocketReceiveWorker::Run()
{
    UE_LOG(LogTemp, Log, TEXT("FSocketReceiveWorker Run"));

    // Set a receive timeout for the worker thread
    // This allows the thread to check the StopEvent periodically even if
no data is incoming.
    ClientSocket->SetReceiveTimeout(100); // 100 milliseconds timeout

    while (bIsRunning && !StopEvent->IsSignaled())
    {
        // --- Receive Message Size (4 bytes) ---
        uint32 Size = 0;
        int32 BytesRead = 0;
```

```
        TArray<uint8> SizeBytes;
        SizeBytes.SetNumUninitialized(4);

        // Use Recv with Peek flag first to check if data is available
without removing it
        // Or just use blocking Recv with timeout and handle the timeout.
        // Using blocking Recv with timeout is simpler.
        // Use ClientSocket->Recv(..., ESocketReceiveFlags::Peek) if you
want non-blocking check first.

        // Try to receive the 4-byte size prefix
        BytesRead = 0;
        while (BytesRead < 4)
        {
            int32 ThisRead = 0;
            // Recv is blocking and respects the socket timeout (100ms)
            if (!ClientSocket->Recv(SizeBytes.GetData() + BytesRead, 4 -
BytesRead, ThisRead))
            {
                // Handle receive failure (timeout or connection error)
                int32 ErrorCode =
ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)->GetLastErrorCode();
                if (ErrorCode == SE_EWOULDBLOCK || ErrorCode ==
SE_SOCKET_ERROR) // EWOULDBLOCK can happen with timeouts
                {
                    // Timeout occurred, check stop event and continue loop
                    if (StopEvent->IsSignaled() || !bIsRunning) break; //
Exit if stopping
                    continue; // Retry receive
                }
                else // Actual error or connection closed
                {
                    UE_LOG(LogTemp, Error, TEXT("FSocketReceiveWorker:
Error receiving message size: %d"), ErrorCode);
                    bIsRunning = false; // Signal to exit loop
                    // TODO: Signal main component/game thread about
disconnection
                    break;
                }
            }
            BytesRead += ThisRead;
        }
```

```
        if (BytesRead < 4) continue; // Didn't receive the full size prefix
(likely connection closed)


        // --- Convert size bytes to integer ---
        Size = FPlatformMisc::BSwap(*(uint32*)SizeBytes.GetData()); //
Convert big-endian to host byte order


        // --- Receive Message Payload ---
        TArray<uint8> PayloadBytes;
        PayloadBytes.SetNumUninitialized(Size);
        BytesRead = 0;
         while (BytesRead < Size)
        {
            int32 ThisRead = 0;
            if (!ClientSocket->Recv(PayloadBytes.GetData() + BytesRead,
Size - BytesRead, ThisRead))
            {
                // Handle receive failure (timeout or connection error)
                int32 ErrorCode =
ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)->GetLastErrorCode();
                 if (ErrorCode == SE_EWOULDBLOCK || ErrorCode ==
SE_SOCKET_ERROR)
                {
                    if (StopEvent->IsSignaled() || !bIsRunning) break;
                    continue;
                }
                else
                {
                    UE_LOG(LogTemp, Error, TEXT("FSocketReceiveWorker:
Error receiving message payload: %d"), ErrorCode);
                    bIsRunning = false;
                    // TODO: Signal main component/game thread about
disconnection
                    break;
                }
            }
            BytesRead += ThisRead;
        }
        if (BytesRead < Size) continue; // Didn't receive the full payload
(likely connection closed)
```

```
        // --- Process Received Message ---
        // Convert bytes to FString (UTF-8 assumed)
        FString ReceivedJsonString;
        FFileHelper::BufferToString(ReceivedJsonString,
PayloadBytes.GetData(), PayloadBytes.Num());

        // Parse JSON
        TSharedPtr<FJsonObject> JsonObject;
        TSharedRef<TJsonReader<TCHAR>> JsonReader =
TJsonReaderFactory<TCHAR>::Create(ReceivedJsonString);

        if (FJsonSerializer::Deserialize(JsonReader, JsonObject) &&
JsonObject.IsValid())
        {
            // Successfully parsed JSON, now extract relevant fields
            FString Action;
            int32 RobotId = -1; // Default or error value

            // Get "action" (string)
            if (!JsonObject->TryGetStringField(TEXT("action"), Action))
            {
                UE_LOG(LogTemp, Warning, TEXT("FSocketReceiveWorker:
Received JSON message missing 'action' field."));
                continue; // Skip message if missing action
            }

            // Get "robot_id" (integer) - Optional, might not be in all
messages
            JsonObject->TryGetNumberField(TEXT("robot_id"), RobotId); //
RobotId is optional for some actions

            // Get "parameters" (object) - Optional
            TSharedPtr<FJsonObject> ParametersObject = JsonObject-
>GetObjectField(TEXT("parameters"));
            // Note: GetObjectField returns nullptr if field is missing or
not an object.

            // Create FPythonMessage struct
            FPythonMessage ParsedMessage;
            ParsedMessage.Action = Action;
            ParsedMessage.RobotId = RobotId;
```

```
            ParsedMessage.Parameters = ParametersObject; // Keep the shared
pointer to the parameters object
            // Add Sequence ID parsing if needed: JsonObject-
>TryGetNumberField(TEXT("sequence_id"), ParsedMessage.SequenceId);

            // Put the parsed message into the queue for the game thread
            MessageQueue.Enqueue(ParsedMessage);
            UE_LOG(LogTemp, Verbose, TEXT("FSocketReceiveWorker: Parsed and
enqueued message with Action: %s"), *Action);
        }
        else
        {
            UE_LOG(LogTemp, Warning, TEXT("FSocketReceiveWorker: Failed to
parse JSON message: %s"), *ReceivedJsonString);
        }
    }

    // Clean up socket when loop finishes
    if (ClientSocket)
    {
        ClientSocket->Close();
        // The socket object is likely managed by the main component,
        // so avoid deleting it here if it's meant to be reused or managed
centrally.
    }

    UE_LOG(LogTemp, Log, TEXT("FSocketReceiveWorker Exit"));
    return 0;
}

void FSocketReceiveWorker::Stop()
{
    bIsRunning = false; // Signal the loop to stop
    StopEvent->Trigger(); // Trigger the event in case the thread is
blocked on Recv
    UE_LOG(LogTemp, Log, TEXT("FSocketReceiveWorker Stop requested."));
}

void FSocketReceiveWorker::Exit()
{
    // Clean up thread resources if any
    UE_LOG(LogTemp, Log, TEXT("FSocketReceiveWorker Exited."));
}
```

```
// --- UPythonCommServerComponent Implementation ---
UPythonCommServerComponent::UPythonCommServerComponent()
    : TcpListener(nullptr), ClientSocket(nullptr), ReceiveWorker(nullptr),
ReceiveThread(nullptr), ReceiveStopEvent(FPlatformProcess::GetOEvent())
{
    // Set this component to be ticked
    PrimaryComponentTick.bCanEverTick = true;
}

void UPythonCommServerComponent::BeginPlay()
{
    Super::BeginPlay();

    StartServer(); // Start the server when the game starts
}

void UPythonCommServerComponent::EndPlay(const EEndPlayReason::Type
EndPlayReason)
{
    Super::EndPlay(EndPlayReason);

    StopServer(); // Stop the server when the game ends
}

bool UPythonCommServerComponent::StartServer()
{
    if (TcpListener)
    {
        UE_LOG(LogTemp, Warning, TEXT("Python Comm Server already
started."));
        return false;
    }

    FIPv4Address Addr;
    FIPv4Address::Parse(ListenIP, Addr);

    FIPv4Endpoint Endpoint(Addr, ListenPort);

    // Create the TCP Listener
    TcpListener = new FTcpSocketListener(Endpoint, false); // false = non-
threaded listener
```

```
    // Bind the connection accepted delegate
    // The listener will call this function when a new connection is
accepted
    TcpListener->OnForConnectionAccepted().BindUObject(this,
&UPythonCommServerComponent::OnSocketConnectionAccepted);

    // Start the listener
    if (TcpListener->Start())
    {
        UE_LOG(LogTemp, Log, TEXT("Python Comm Server listening on %s:%d"),
*ListenIP, ListenPort);
        return true;
    }
    else
    {
        UE_LOG(LogTemp, Error, TEXT("Failed to start Python Comm Server
on %s:%d"), *ListenIP, ListenPort);
        delete TcpListener;
        TcpListener = nullptr;
        return false;
    }
}

void UPythonCommServerComponent::StopServer()
{
    if (TcpListener)
    {
        // Stop the listener
        TcpListener->Stop();
        delete TcpListener;
        TcpListener = nullptr;
        UE_LOG(LogTemp, Log, TEXT("Python Comm Server stopped."));
    }

    // Ensure client connection and worker thread are stopped
    if (ReceiveWorker)
    {
        ReceiveWorker->Stop(); // Signal the worker thread to stop
        // Optional: Wait for the thread to finish (join)
        if (ReceiveThread)
        {
            ReceiveThread->WaitForCompletion();
```

```
            delete ReceiveThread;
            ReceiveThread = nullptr;
        }
        delete ReceiveWorker;
        ReceiveWorker = nullptr;
         if (ReceiveStopEvent) {
             FPlatformProcess::FreeEvent(ReceiveStopEvent);
             ReceiveStopEvent = nullptr;
         }
    }

    if (ClientSocket)
    {
        ClientSocket->Close();
        ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)-
>DestroySocket(ClientSocket);
        ClientSocket = nullptr;
        UE_LOG(LogTemp, Log, TEXT("Python client connection closed."));
         OnClientDisconnected.Broadcast(); // Trigger disconnected event
    }
}

bool UPythonCommServerComponent::OnSocketConnectionAccepted(FSocket*
InClientSocket, const FIPv4Endpoint& ClientEndpoint)
{
    // This function is called by the listener thread when a new client
connects.
    // We need to handle this connection on the game thread or pass it to a
worker.

    UE_LOG(LogTemp, Log, TEXT("Python client connected from %s"),
*ClientEndpoint.ToString());

    // For simplicity, accept only one connection at a time.
    if (ClientSocket && ClientSocket->GetConnectionState() ==
SCS_Connected)
    {
        UE_LOG(LogTemp, Warning, TEXT("Another client is already connected.
Rejecting new connection."));
        // Close the new socket immediately
        ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)-
>DestroySocket(InClientSocket);
        return false; // Tell the listener not to keep this socket
```

```
    }

    // Accept the new connection
    ClientSocket = InClientSocket;

    // Signal the game thread about the connection (optional, could use a
queue)
    // Or trigger a delegate directly if it's safe / handled on game thread
    OnClientConnected.Broadcast(); // Trigger connected event

    // Start a new worker thread to receive data from this client socket
    ReceiveStopEvent = FPlatformProcess::GetOEvent(); // Create the event
object
    ReceiveWorker = new FSocketReceiveWorker(ClientSocket,
ReceivedMessageQueue, ReceiveStopEvent);
    ReceiveThread = FRunnableThread::Create(ReceiveWorker,
TEXT("PythonCommReceiveThread"));

    return true; // Tell the listener to keep this socket
}

void UPythonCommServerComponent::TickComponent(float DeltaTime, ELevelTick
TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickType);

    // Process received messages from the queue on the game thread
    ProcessReceivedMessages();
}

void UPythonCommServerComponent::ProcessReceivedMessages()
{
    FPythonMessage Message;
    // Dequeue messages. Use Peek to check without removing if needed, or
Dequeue with timeout.
    // DequeueAll is efficient if many messages might arrive per tick.
    while (ReceivedMessageQueue.Dequeue(Message))
    {
        // Process the received message on the game thread
        UE_LOG(LogTemp, Log, TEXT("Processing message on Game Thread:
Action = %s, RobotId = %d"), *Message.Action, Message.RobotId);

        // --- Dispatch Actions ---
```

```
        if (Message.Action == TEXT("set_robot_pose"))
        {
            // Handle Set Robot Pose command
            // Find the target robot actor and update its pose
            AActor** RobotActorPtr =
RegisteredRobots.Find(Message.RobotId);
            if (RobotActorPtr && *RobotActorPtr)
            {
                // Cast the actor to your specific Robot Actor class if
needed
                // AYourRobotBaseActor* RobotActor =
Cast<AYourRobotBaseActor>(*RobotActorPtr);
                // if (RobotActor) {
                //     RobotActor-
>SetRobotPoseFromJSON(Message.Parameters); // Call BlueprintCallable or C++
function
                // }
                // For simplicity, just log or call a generic function if
your base actor has one
                 UE_LOG(LogTemp, Log, TEXT("Received set_robot_pose for
Robot %d. Dispatching..."), Message.RobotId);
                 // You would implement dispatch logic here, potentially
calling a function on the Actor:
                 // UGameplayStatics::CallFunctionByName(*RobotActorPtr,
TEXT("SetRobotPoseFromJSON"), params); // Generic function call
                 // Or trigger a Blueprint event if using Blueprint
                 OnMessageReceived.Broadcast(Message); // Let Blueprint
handle dispatch via this event

            }
            else
            {
                UE_LOG(LogTemp, Warning, TEXT("Received set_robot_pose for
unknown or unregistered Robot ID: %d"), Message.RobotId);
            }
        }
        else if (Message.Action == TEXT("welding_visual_command"))
        {
            // Handle Welding Visual Command
            AActor** RobotActorPtr =
RegisteredRobots.Find(Message.RobotId);
            if (RobotActorPtr && *RobotActorPtr)
            {
```

```
                UE_LOG(LogTemp, Log, TEXT("Received welding_visual_command
for Robot %d. Dispatching..."), Message.RobotId);
                // Call a function on the Actor or trigger an event:
                OnMessageReceived.Broadcast(Message); // Let Blueprint
handle dispatch via this event
            }
             else
            {
                UE_LOG(LogTemp, Warning, TEXT("Received
welding_visual_command for unknown or unregistered Robot ID: %d"),
Message.RobotId);
            }

        }
        else if (Message.Action == TEXT("run_simulation"))
        {
             // Handle Run Simulation command
             UE_LOG(LogTemp, Log, TEXT("Received run_simulation
command."));
             // Implement simulation logic here or delegate to another
component
             // After simulation, send result back using SendJsonResponse
             OnMessageReceived.Broadcast(Message); // Let Blueprint or
another C++ component handle simulation
        }
        else if (Message.Action == TEXT("get_sim2real_ark_situation"))
        {
             // Handle Get Sim2Real command
             UE_LOG(LogTemp, Log, TEXT("Received get_sim2real_ark_situation
command."));
             // Implement data generation logic
             // Send data back using SendJsonResponse
             OnMessageReceived.Broadcast(Message); // Let Blueprint or
another C++ component handle generating data
        }
        else
        {
            UE_LOG(LogTemp, Warning, TEXT("Received unknown action: %s"),
*Message.Action);
            // Potentially send an error response back to Python
        }
    }
}
```

```
bool UPythonCommServerComponent::SendFramedData(const FString& DataToSend)
{
    if (!ClientSocket || ClientSocket->GetConnectionState() !=
SCS_Connected)
    {
        UE_LOG(LogTemp, Warning, TEXT("Cannot send data, client socket not
connected."));
        return false;
    }

    TArray<uint8> PayloadBytes;
    FTCHARToUTF8 Converter(*DataToSend);
    PayloadBytes.SetNum(Converter.Length());
    FMemory::Memcpy(PayloadBytes.GetData(), Converter.Get(),
Converter.Length());

    uint32 Size = PayloadBytes.Num();
    TArray<uint8> SizeBytes;
    SizeBytes.SetNumUninitialized(4);
    // Convert size to big-endian
    uint32 BigEndianSize = FPlatformMisc::BSwap(Size);
    FMemory::Memcpy(SizeBytes.GetData(), &BigEndianSize, 4);


    int32 BytesSent = 0;
    bool Success = false;

    // Use a lock if SendFramedData can be called from multiple threads
    // (Unlikely in this Tick-based model, but good practice if logic
changes)
    // with SendLock: // Define SendLock if needed
    {
        // Send size prefix
        Success = ClientSocket->Send(SizeBytes.GetData(), SizeBytes.Num(),
BytesSent);
        if (Success && BytesSent == SizeBytes.Num())
        {
            // Send payload
            Success = ClientSocket->Send(PayloadBytes.GetData(),
PayloadBytes.Num(), BytesSent);
            if (Success && BytesSent == PayloadBytes.Num())
```

```
            {
                UE_LOG(LogTemp, Verbose, TEXT("Sent %d bytes payload."),
Size);
                return true; // Successfully sent size and payload
            }
            else
            {
                UE_LOG(LogTemp, Error, TEXT("Failed to send payload bytes.
Sent: %d / %d"), BytesSent, Size);
                Success = false; // Ensure Success is false on payload
send failure
            }
        }
        else
        {
            UE_LOG(LogTemp, Error, TEXT("Failed to send size bytes.
Sent: %d / 4"), BytesSent);
             Success = false; // Ensure Success is false on size send
failure
        }
    }

    if (!Success)
    {
        // Handle send failure (likely connection error)
        int32 ErrorCode = ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)-
>GetLastErrorCode();
        UE_LOG(LogTemp, Error, TEXT("SendFramedData error: %d.
Disconnecting client."), ErrorCode);
        // TODO: Signal main component/game thread about disconnection
        // Calling Disconnect here directly might not be safe if this is
not on the game thread.
        // Need a way to safely signal disconnection to the game thread.
        // For simplicity in this component, let's assume SendFramedData is
called on the game thread.
        // If called from worker, add a queue/delegate to signal game
thread.
        StopServer(); // Simple approach: Stop everything on send failure
    }

    return Success;
}
```

```
bool UPythonCommServerComponent::SendJsonResponse(int32 ClientConnectionId,
const FString& Status, const FString& Message, const
TSharedPtr<FJsonObject>& Data)
{
    if (!ClientSocket || ClientSocket->GetConnectionState() !=
SCS_Connected)
    {
         UE_LOG(LogTemp, Warning, TEXT("Cannot send JSON response, client
socket not connected."));
         return false;
    }
    // NOTE: ClientConnectionId is ignored here in the single-client model.
    // In a multi-client setup, you'd use this ID to find the correct
FSocket.

    TSharedPtr<FJsonObject> ResponseObject = MakeShareable(new
FJsonObject());
    ResponseObject->SetStringField(TEXT("status"), Status);
    ResponseObject->SetStringField(TEXT("message"), Message);
    if (Data.IsValid())
    {
        ResponseObject->SetObjectField(TEXT("data"), Data); // Or SetField
if Data is not always an object
    }
    // TODO: Include sequence_id from the original request if the protocol
requires matching responses!

    FString ResponseString;
    TSharedRef<TJsonWriter<>> Writer =
TJsonWriterFactory<>::Create(&ResponseString);
    FJsonSerializer::Serialize(ResponseObject.ToSharedRef(), Writer);

    UE_LOG(LogTemp, Log, TEXT("Sending JSON Response: %s"),
*ResponseString);
    return SendFramedData(ResponseString);
}


void UPythonCommServerComponent::RegisterRobotActor(int32 RobotId, AActor*
RobotActor)
{
    if (!RobotActor)
    {
```

```
        UE_LOG(LogTemp, Warning, TEXT("Attempted to register null Robot
Actor for ID %d."), RobotId);
        return;
    }
    if (RegisteredRobots.Contains(RobotId))
    {
        UE_LOG(LogTemp, Warning, TEXT("Robot ID %d is already registered.
Overwriting."), RobotId);
    }
    RegisteredRobots.Add(RobotId, RobotActor);
    UE_LOG(LogTemp, Log, TEXT("Registered Robot Actor '%s' with ID %d."),
*RobotActor->GetName(), RobotId);
}
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. C++
IGNORE_WHEN_COPYING_END

2. 로봇 액터 클래스 (ARobotBaseActor 또는 BP_MyRobot) 개념:

이것은 로봇의 3D 모델(스켈레탈 메시)을 포함하고 Python 으로부터 받은 자세 데이터를 적용하며 용접 시각 효과를 제어하는 로직을 가집니다. Blueprint 또는 C++로 구현할 수 있습니다. C++로 기본 클래스를 만들고 Blueprint 에서 상속받는 것이 일반적입니다.

```
      // Header file (RobotBaseActor.h)

#pragma once

#include "GameFramework/Actor.h"
#include "Json.h" // For JSON parsing if you parse parameters here
#include "Components/SkeletalMeshComponent.h" // For accessing bones
#include "Particles/ParticleSystemComponent.h" // For welding arc
#include "RobotBaseActor.generated.h"

UCLASS()
class YOURUNREALMODULE_API ARobotBaseActor : public AActor
{
    GENERATED_BODY()

public:
    ARobotBaseActor();
```

```
    // Public variable to set Robot ID in the editor/Blueprint
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Robot")
    int32 RobotID = -1; // Unique ID matching the Python system's robot_id

    // Reference to the Skeletal Mesh Component
    UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Components")
    USkeletalMeshComponent* RobotMesh;

    // Reference to the Welding Arc Particle Component
    UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Components")
    UParticleSystemComponent* WeldingArcParticle;

    // Optional: Reference to a Static Mesh Component representing the
welding tip
    UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Components")
    UStaticMeshComponent* WeldingTipMesh;


    virtual void BeginPlay() override;

    // Function to set robot pose from JSON parameters (called from
PythonCommServerComponent)
    UFUNCTION(BlueprintCallable, Category = "Robot Control")
    void SetRobotPoseFromJSON(const TSharedPtr<FJsonObject>&
PoseParameters);

    // Function to handle welding visual commands from JSON parameters
    UFUNCTION(BlueprintCallable, Category = "Robot Control")
    void HandleWeldingVisualCommand(const TSharedPtr<FJsonObject>&
CommandParameters);


protected:
    // Internal function to apply joint angles
    UFUNCTION(BlueprintImplementableEvent, Category = "Robot Internal")
    void ApplyJointAngles(const TArray<float>& JointAngles);

    // Internal function to apply TCP transform
    UFUNCTION(BlueprintImplementableEvent, Category = "Robot Internal")
    void ApplyTCPTransform(const FVector& Location, const FQuat& Rotation);
// Or FRotator

    // Internal function to set arc visibility
```

```
    UFUNCTION(BlueprintImplementableEvent, Category = "Robot Internal")
    void SetArcVisibility(bool bVisible);

    // Internal function to update arc parameters (color, size, etc.)
    UFUNCTION(BlueprintImplementableEvent, Category = "Robot Internal")
    void UpdateArcVisuals(const TSharedPtr<FJsonObject>& VisualDetails);


private:
    // Add references to bones if needed (e.g., using FName)
    // FName JointBoneNames[6]; // Example for a 6-DOF robot

};

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. C++
IGNORE_WHEN_COPYING_END
      // Source file (RobotBaseActor.cpp) - Key implementations

#include "RobotBaseActor.h"
#include "JsonUtilities.h" // For FJsonObjectConverter (optional, for
complex object conversion)
#include "Animation/SkeletalMeshActor.h" // If inheriting from
SkeletalMeshActor
#include "Animation/AnimInstance.h" // If using Animation Blueprint

ARobotBaseActor::ARobotBaseActor()
{
    PrimaryActorTick.bCanEverTick = true; // Allow ticking if needed

    // Create and attach components (example - customize based on your
setup)
    RobotMesh =
CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("RobotSkeletalMesh"));
    RootComponent = RobotMesh;

    WeldingArcParticle =
CreateDefaultSubobject<UParticleSystemComponent>(TEXT("WeldingArcParticle")
);
    // Attach particle system to a socket on the mesh (e.g.,
"WeldingTipSocket")
```

```
    WeldingArcParticle->SetupAttachment(RobotMesh,
TEXT("WeldingTipSocket"));
    WeldingArcParticle->SetVisibility(false); // Start hidden
    WeldingArcParticle->bAutoActivate = false; // Don't activate
automatically

    // Optional: Welding tip mesh
    // WeldingTipMesh =
CreateDefaultSubobject<UStaticMeshComponent>(TEXT("WeldingTipMesh"));
    // WeldingTipMesh->SetupAttachment(RobotMesh,
TEXT("WeldingTipSocket")); // Attach to the same socket

    // Initialize bone names if using direct bone control
    // JointBoneNames[0] = TEXT("Joint1_Bone"); // Example names - MUST
match your skeletal mesh

    // Ensure the skeletal mesh is set up in the Blueprint inheriting from
this class
}

void ARobotBaseActor::BeginPlay()
{
    Super::BeginPlay();

    // Register this robot actor with the communication server component
    // Find the manager actor or component in the level
    UPythonCommServerComponent* CommComponent = nullptr;
    TArray<AActor*> FoundActors;
    UGameplayStatics::GetAllActorsOfClass(GetWorld(),
AActor::StaticClass(), FoundActors); // Find all actors
    for (AActor* Actor : FoundActors)
    {
        CommComponent = Actor-
>FindComponentByClass<UPythonCommServerComponent>();
        if (CommComponent)
        {
            // Found the component, register this robot
            CommComponent->RegisterRobotActor(RobotID, this);
            break;
        }
    }
    if (!CommComponent)
    {
```

```
        UE_LOG(LogTemp, Error, TEXT("ARobotBaseActor '%s':
PythonCommServerComponent not found in level! Cannot register."),
*GetName());
    }
}

void ARobotBaseActor::SetRobotPoseFromJSON(const TSharedPtr<FJsonObject>&
PoseParameters)
{
    if (!PoseParameters.IsValid())
    {
        UE_LOG(LogTemp, Warning, TEXT("SetRobotPoseFromJSON called with
invalid parameters for Robot %d."), RobotID);
        return;
    }

    // --- Extract and apply pose data based on the JSON structure ---
    // Python sends: {"joint_angles": [...]} or {"tcp_transform":
{"position": [...], "rotation": [...]}}

    const TArray<TSharedPtr<FJsonValue>>* JointAnglesArray;
    if (PoseParameters->TryGetArrayField(TEXT("joint_angles"),
JointAnglesArray))
    {
        // Handle Joint Angles
        TArray<float> JointAngles;
        for (const TSharedPtr<FJsonValue>& JsonValue : *JointAnglesArray)
        {
            double Angle;
            if (JsonValue->TryGetNumber(Angle))
            {
                JointAngles.Add(static_cast<float>(Angle));
            }
            else
            {
                UE_LOG(LogTemp, Warning, TEXT("Invalid joint angle value in
array for Robot %d."), RobotID);
                // Decide how to handle invalid data - skip this update?
Log error and continue?
                break; // Stop processing this message if invalid data
found
            }
        }
```

```
        if (JointAngles.Num() > 0 && JointAngles.Num() == 6) // Basic
validation: check if 6 angles received
        {
            // Call the Blueprint Implementable Event or C++ function to
apply joint angles
            ApplyJointAngles(JointAngles); // <-- This calls the Blueprint
function (if implemented)
             UE_LOG(LogTemp, Verbose, TEXT("Applied joint angles for
Robot %d."), RobotID);
        }
         else if (JointAngles.Num() > 0)
         {
              UE_LOG(LogTemp, Warning, TEXT("Received %d joint angles for
Robot %d, expected 6."), JointAngles.Num(), RobotID);
         }
    }

    const TSharedPtr<FJsonObject>* TcpTransformObject;
    if (PoseParameters->TryGetObjectField(TEXT("tcp_transform"),
TcpTransformObject) && TcpTransformObject && TcpTransformObject->IsValid())
    {
        // Handle TCP Transform (Position and Rotation)
        const TArray<TSharedPtr<FJsonValue>>* PositionArray;
        const TArray<TSharedPtr<FJsonValue>>* RotationArray; // Format
(Quaternion or Euler?) must match Python

        FVector Location = FVector::ZeroVector;
        FQuat Rotation = FQuat::Identity; // Or FRotator

        // Get Position (assumed [x, y, z])
        if ((*TcpTransformObject)->TryGetArrayField(TEXT("position"),
PositionArray) && PositionArray && PositionArray->Num() == 3)
        {
             // Need to convert coordinate systems if Python's Z is UE's Z
etc.
             Location.X = static_cast<float>((*PositionArray)[0]-
>AsNumber()); // Example mapping X
             Location.Y = static_cast<float>((*PositionArray)[1]-
>AsNumber()); // Example mapping Y
             Location.Z = static_cast<float>((*PositionArray)[2]-
>AsNumber()); // Example mapping Z
             // TODO: Implement coordinate system and unit conversion if
needed!
```

```
        } else { UE_LOG(LogTemp, Warning, TEXT("Missing or invalid
'position' array for TCP transform for Robot %d."), RobotID); }


        // Get Rotation (assumed Quaternion [x, y, z, w]) - MUST match
Python format!
        if ((*TcpTransformObject)->TryGetArrayField(TEXT("rotation"),
RotationArray) && RotationArray && RotationArray->Num() == 4)
        {
             // Need to convert coordinate systems and rotation order if
needed!
             Rotation.X = static_cast<float>((*RotationArray)[0]-
>AsNumber());
             Rotation.Y = static_cast<float>((*RotationArray)[1]-
>AsNumber());
             Rotation.Z = static_cast<float>((*RotationArray)[2]-
>AsNumber());
             Rotation.W = static_cast<float>((*RotationArray)[3]-
>AsNumber());
             Rotation.Normalize(); // Normalize Quaternion
             // TODO: Implement coordinate system and rotation conversion
if needed!
        } // Else: Missing or invalid rotation array, use default identity
rotation


        // Call the Blueprint Implementable Event or C++ function to apply
TCP transform
        ApplyTCPTransform(Location, Rotation); // <-- This calls the
Blueprint function (if implemented)
        UE_LOG(LogTemp, Verbose, TEXT("Applied TCP transform for Robot %d.
Location: %s"), RobotID, *Location.ToString());

    }

    // If neither joint_angles nor tcp_transform was successfully parsed,
log a warning.
    if (!JointAnglesArray && !TcpTransformObject)
    {
         UE_LOG(LogTemp, Warning, TEXT("Received 'set_robot_pose' message
but no valid 'joint_angles' or 'tcp_transform' found for Robot %d."),
RobotID);
    }
```

```
}


void ARobotBaseActor::HandleWeldingVisualCommand(const
TSharedPtr<FJsonObject>& CommandParameters)
{
    if (!CommandParameters.IsValid())
    {
        UE_LOG(LogTemp, Warning, TEXT("HandleWeldingVisualCommand called
with invalid parameters for Robot %d."), RobotID);
        return;
    }

    FString CommandType;
    if (!CommandParameters->TryGetStringField(TEXT("command_type"),
CommandType))
    {
        UE_LOG(LogTemp, Warning, TEXT("Received welding_visual_command
missing 'command_type' field for Robot %d."), RobotID);
        return;
    }

    UE_LOG(LogTemp, Log, TEXT("Robot %d received welding visual
command: %s"), RobotID, *CommandType);

    // --- Handle Specific Command Types ---
    if (CommandType == TEXT("arc_on"))
    {
        SetArcVisibility(true); // Call Blueprint Implementable Event or
C++ function
         // Optional: Update arc parameters if details are provided
        TSharedPtr<FJsonObject> Details = CommandParameters-
>GetObjectField(TEXT("details"));
        if (Details.IsValid()) {
            UpdateArcVisuals(Details); // e.g., update particle size/color
based on current/voltage
        }
    }
    else if (CommandType == TEXT("arc_off"))
    {
        SetArcVisibility(false); // Call Blueprint Implementable Event or
C++ function
    }
```

```
    // Add other visual commands as needed (e.g., "set_bead_visibility",
"update_bead_shape")
    else
    {
        UE_LOG(LogTemp, Warning, TEXT("Unknown welding visual command
type: %s for Robot %d."), *CommandType, RobotID);
    }
}

// Implement Blueprint Implementable Events in Blueprint Editor:
// Right-click on the Blueprint class -> Override Function -> Find
"ApplyJointAngles", "ApplyTCPTransform", etc.
```

IGNORE_WHEN_COPYING_START
content_copy download
Use code with caution. C++
IGNORE_WHEN_COPYING_END

3. Blueprint 설정 개념:

- **매니저 액터 블루프린트 (BP_PythonCommManager):**
  - 새로운 Blueprint Actor 클래스를 생성합니다 (BP_PythonCommManager).
  - 이 액터에 위에서 만든 C++ 컴포넌트(UPythonCommServerComponent)를 추가합니다.
  - Event Graph 에서 UPythonCommServerComponent 의 이벤트(OnClientConnected, OnClientDisconnected, OnMessageReceived)에 바인딩할 커스텀 이벤트 또는 함수를 생성합니다.
  - OnMessageReceived 이벤트가 발생하면, 전달받은 FPythonMessage 구조체의 Action (문자열), RobotId (정수), Parameters (JSON 오브젝트)를 사용하여 추가적인 Blueprint 로직을 구현합니다. 예를 들어, Action 이 "set_robot_pose"이면 RobotId 를 사용하여 레벨에서 해당 로봇 액터를 찾고, 그 액터의 SetRobotPoseFromJSON 함수를 호출하며, Parameters JSON 오브젝트를 전달합니다.
  - StartServer 및 StopServer Blueprint Callable 함수를 레벨 시작/종료 이벤트(Event BeginPlay, Event EndPlay)에 연결하여 서버를 자동 시작/종료하도록 합니다.
- **로봇 액터 블루프린트 (BP_MyRobot):**
  - 위에서 만든 C++ 기본 클래스(ARobotBaseActor)로부터 상속받는 Blueprint Actor 클래스를 생성합니다 (BP_MyRobot).
  - RobotID 변수 (상속받은 UPROPERTY)를 각 액터 인스턴스에 고유한 값으로 설정합니다 (예: 1, 2, 3, 4).

    - RobotMesh (스켈레탈 메시 컴포넌트)에 실제 로봇 스켈레탈 메시 에셋을 할당합니다.
    - WeldingArcParticle (파티클 시스템 컴포넌트)에 용접 아크 파티클 시스템 에셋을 할당하고, WeldingTipSocket (스켈레탈 메시 소켓)에 붙입니다.
    - **Implement Blueprint Implementable Events:** 상속받은 Blueprint Implementable Event (BIE) 함수들(ApplyJointAngles, ApplyTCPTransform, SetArcVisibility, UpdateArcVisuals)를 오버라이드하여 실제 로봇 모델을 제어하는 로직을 Blueprint 로 구현합니다.
        - ApplyJointAngles: 입력받은 관절 각도 배열을 사용하여 스켈레탈 메시의 각 관절 본의 상대 회전(Relative Rotation)을 설정합니다. 각 본의 초기 방향과 회전 기준 축을 고려하여 각도 값을 적용해야 합니다. 애니메이션 블루프린트를 사용하는 경우, 애니메이션 그래프에서 이 관절 각도 값을 받아 제어 본에 적용하도록 설정할 수 있습니다.
        - ApplyTCPTransform: 입력받은 위치(Location)와 회전(Rotation)을 사용하여 로봇 모델 전체 또는 특정 본의 트랜스폼을 설정합니다. 로봇 베이스 위치 및 방향, 그리고 TCP 위치 정의에 따라 복잡한 변환 로직이 필요할 수 있습니다.
        - SetArcVisibility: 입력받은 boolean 값에 따라 WeldingArcParticle 컴포넌트의 가시성을 켜거나 끕니다.
        - UpdateArcVisuals: 입력받은 JSON 파라미터(예: Python 에서 보낸 전류, 전압 값)를 사용하여 파티클 시스템의 색상, 크기, 방출 속도 등 파라미터를 동적으로 변경하여 아크의 시각적 표현을 실제 용접 조건에 맞게 변화시킵니다.
    - Event Graph 에서 Event BeginPlay 이벤트에 RegisterRobotActor 함수를 호출하는 로직을 추가합니다.
      PythonCommServerComponent 인스턴스를 찾아서 해당 함수를 호출하고 자신의 RobotID 및 자신(self)을 인자로 전달합니다.
- **UE 레벨:** BP_PythonCommManager 액터와 각 BP_MyRobot 액터(RobotID 설정 완료)를 레벨에 배치합니다.

개발 환경 설정:

1. **Visual Studio 설정:** Unreal Engine C++ 개발을 위해 Visual Studio (Windows) 또는 Xcode (macOS)를 설치하고 UE 와 연동 설정을 완료합니다.
2. **UE 프로젝트 설정:** 프로젝트 설정에서 Networking 모듈이 활성화되어 있는지 확인합니다.
3. **C++ 클래스 생성:** UE 에디터 내에서 새로운 C++ 클래스 마법사(New C++ Class)를 사용하여 UActorComponent 기반 클래스(UPythonCommServerComponent)와 AActor 기반

클래스(ARobotBaseActor)를 생성합니다. 이때 프로젝트의 C++ 모듈 이름(YOURUNREALMODULE_API 부분)을 확인합니다.

4. **코드 작성:** 위의 C++ 코드(.h 및 .cpp 파일)를 해당 클래스에 맞게 복사/붙여넣기 하고 필요한 헤더 파일(// Include ...)을 추가합니다.
5. **컴파일:** Visual Studio/Xcode 에서 프로젝트 솔루션을 열고 UE 에디터와 함께 컴파일합니다. 오류가 없을 때까지 반복합니다.
6. **Blueprint 생성:** UE 에디터에서 C++ 클래스를 부모로 하는 Blueprint 클래스를 생성합니다.
7. **Blueprint 로직 구현:** Blueprint 에디터에서 위에서 설명한 로직을 구현합니다.
8. **레벨 배치:** 매니저 액터와 로봇 액터들을 레벨에 배치하고 설정합니다.