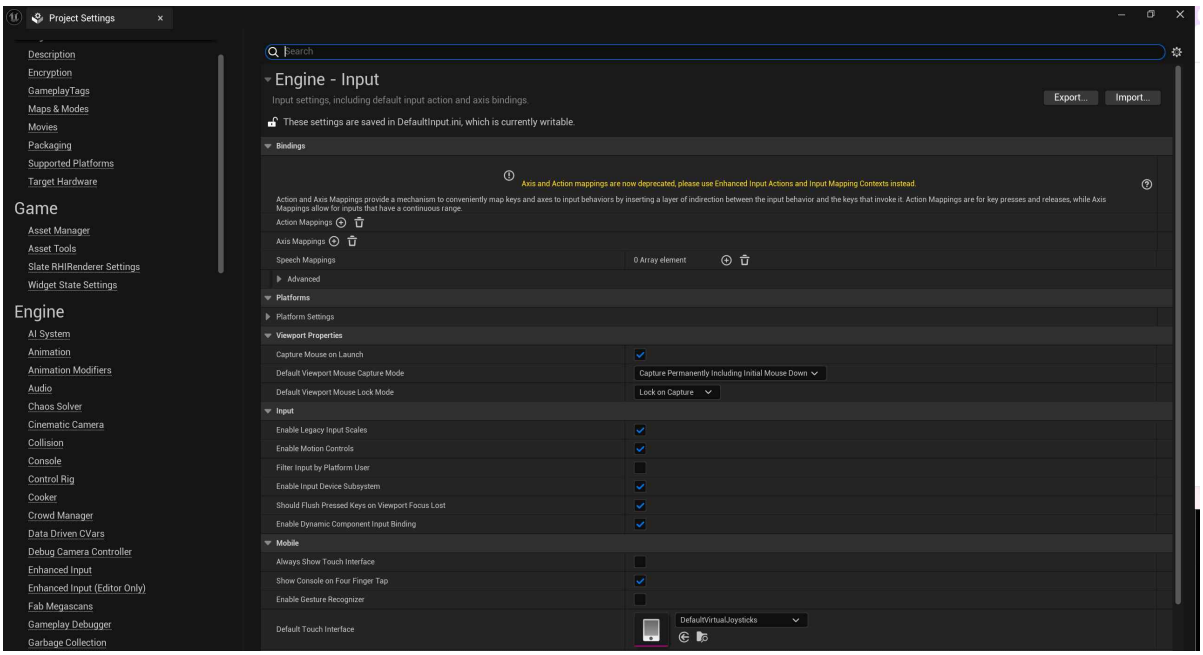
09_Enhanced Input System

1. Legacy Input System

1) Legacy Input System

- 기존 입력 시스템은 언리얼 엔진의 초기부터 제공된 입력 처리 방식으로, Project Settings > Input에서 설정합니다.
- 키보드, 마우스, 컨트롤러 등의 입력을 Action Mappings 및 Axis Mappings으로 분리하여 처리합니다.



2) 특징

① Action Mappings

- 버튼 클릭(예: Jump, Fire) 같은 단일 이벤트를 처리합니다.
- 키가 눌렸을 때(Pressed)와 뗐을 때(Released)의 이벤트를 제공합니다.

② Axis Mappings

- 마우스 이동, 아날로그 스틱 입력처럼 연속적인 값(Float)을 처리합니다.

3) 장점

- 간단함: 설정과 사용이 직관적이며, 소규모 프로젝트나 간단한 입력 처리에 적합합니다.
- 안정성: 오랜 시간 사용되어 테스트된 안정적인 시스템.

4) 단점

- 확장성 부족: 복잡한 입력 로직, 입력 체계의 확장(예: 다중 입력 디바이스 지원)에 한계가 있음.
- 데이터 중심적 접근 부족: 입력 상태를 저장하고 다양한 방식으로 조작하는 데 제약이 있음.

2. Enhanced Input System

1) Enhanced Input System

- 향상된 입력 시스템은 언리얼 엔진 5에서 도입된 새로운 입력 처리 방식으로, 더 강력하고 유연한 입력 설정과 확장을 지원합니다.
- Enhanced Input Plugin을 활성화하여 사용할 수 있습니다.

2) 특징

① Input Mapping Context

- 키보드, 마우스, 컨트롤러 입력을 매핑 컨텍스트를 통해 정의하며, 각 컨텍스트는 서로 다른 입력 우선순위를 가질 수 있습니다.
- 예: 플레이어의 기본 입력과 차량 조작 입력을 분리.

② Input Actions

- 각 입력에 대해 행동(Action)과 축(Axis)을 정의하고, 이를 동적으로 연결 및 해제할 수 있습니다.

③ 모듈식 처리

- 각 입력 이벤트를 델리게이트 기반으로 처리하여 복잡한 입력 로직을 유연하게 설계 가능.

3) 장점

- 유연성과 확장성: 멀티플레이어 게임, 다양한 입력 장치 지원, 다이내믹 입력 체계 변경 등에 적합.
- 데이터 중심 설계: 입력 상태를 관리하는 데 뛰어난 데이터 중심 접근 방식을 제공.
- 동적 입력 변경: 상황(예: 캐릭터 상태, 차량 모드)에 따라 입력 체계를 동적으로 전환할 수 있음.
- 키 리매핑: 플레이어가 입력을 자유롭게 설정할 수 있는 리매핑 기능을 쉽게 구현 가능.

4) 단점

- 학습 곡선: 기존 입력 시스템보다 복잡하며, 사용법을 익히는 데 시간이 더 필요.
- 성능 오버헤드: 일부 간단한 입력 처리에는 불필요하게 복잡한 구현일 수 있음.

※ 기존 입력 시스템과 향상된 입력 시스템 장단점 비교

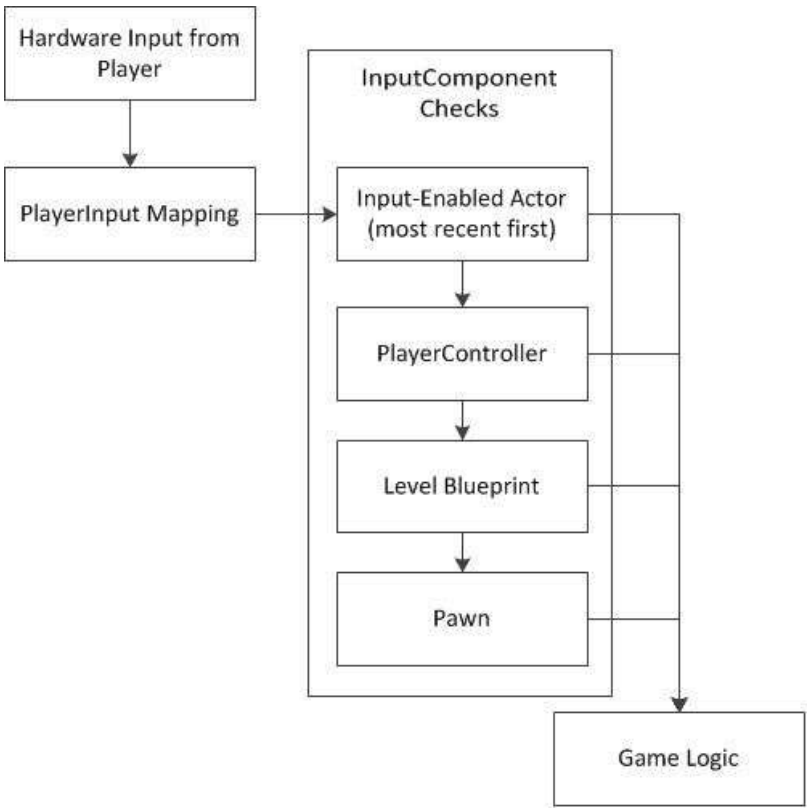
특징	기존 입력 시스템	향상된 입력 시스템
구조적 단순성	간단하고 직관적	더 복잡하지만 유연한 구조
확장성	제한적	매우 유연하며 확장 가능
입력 리매핑	기본 지원 없음	내장 지원
멀티 디바이스 지원	제한적	다양한 입력 장치 지원
동적 입력 전환	수동으로 처리	컨텍스트 기반으로 동적 처리 가능
러닝 커브	낮음	높음

5) 입력 컴포넌트

- 입력 컴포넌트(InputComponent)는 보통 폰과 컨트롤러에 있지만 필요한 경우 다른 액터나 레벨 스크립트에서도 설정할 수 있다. 입력 컴포넌트는 프로젝트의 축 매핑과 액션 매핑을 C++ 코드 또는 블루프린트 그래프로 게임 액션에 연결한다.
- 입력 컴포넌트의 입력 처리 우선순위 스택은 다음과 같은 우선순위를 따른다.

① 최소로 최근에 사용한 것부터 최대로 최근에 사용한 것까지 "입력 허용"이 활성화된 모든 액터
② 컨트롤러
③ 레벨 스크립트.
④ 폰

6) 입력 프로세싱 절차



7) 입력 전달의 핵심 클래스

① PlayerController

- 입력 데이터를 감지하고 관리.
- Possessed Pawn으로 입력 데이터를 전달.

② InputComponent

- 입력 이벤트와 Character의 함수를 연결.
- 실제 입력 데이터를 처리하고 함수 호출.

③ ACharacter

- 입력 데이터를 처리한 결과로 이동, 회전, 점프 등의 동작을 수행.

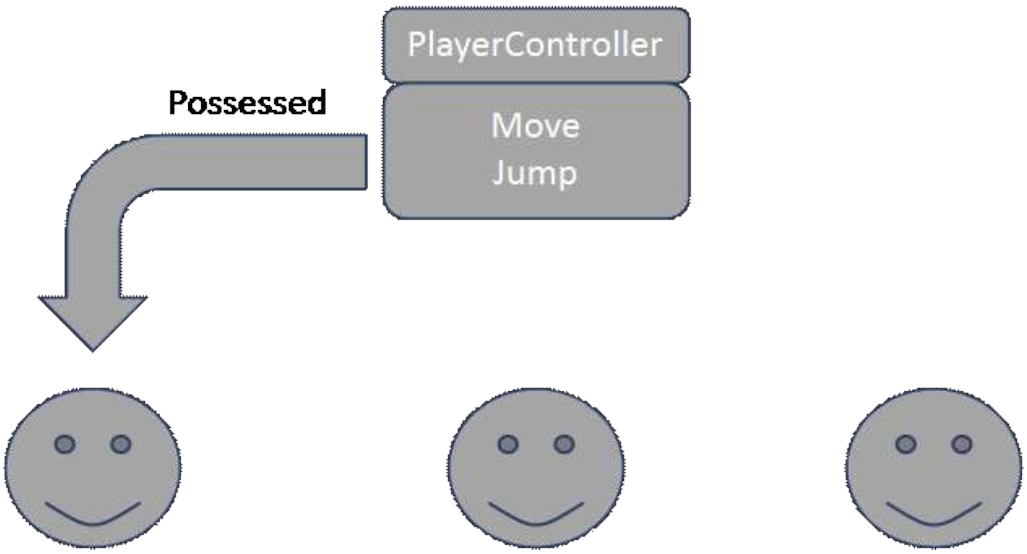
3. Input 구현하기

1) Input 구현하기

- 입력 데이터가 감지되었을 때, PlayerController가 처리할지 Pawn(Character)가 처리할지 선택한다.
- 게임을 개발할 때, 주인공 캐릭터의 특징에 맞춰서 구현해 주면 된다.

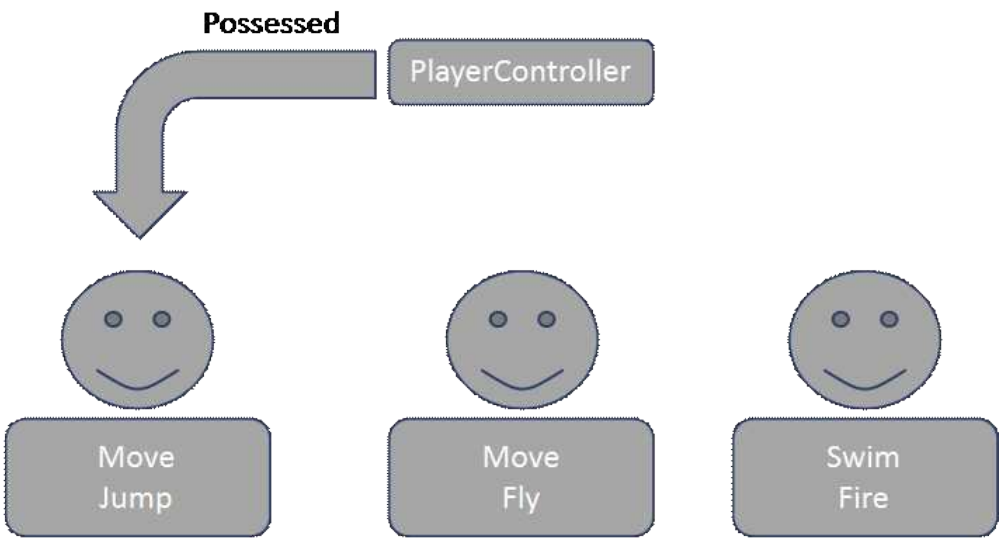
① PlayerController

- 빙의(Possess)하는 캐릭터와 상관없이 언제나 같은 입력 처리 가능
- 특히, 주인공 캐릭터가 탈 것을 탄 경우 기존의 캐릭터의 움직임을 제어해야하는 경우 등 하나의 시스템에서 제어가 가능하다.



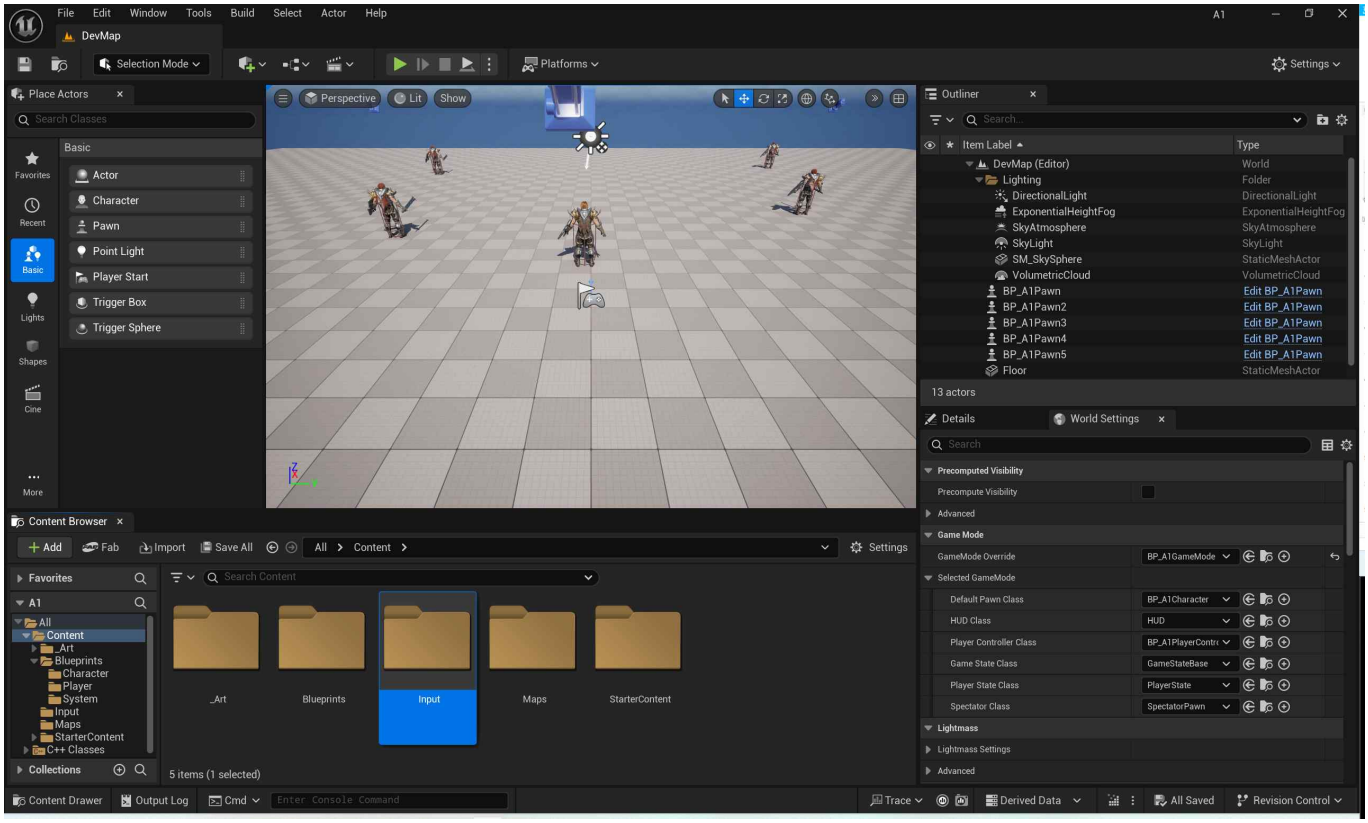
② Pawn(Character)

- 빙의(Possess)하는 캐릭터가 가진 입출력 이벤트처리를 사용하는 방식이다.
- 여러명의 주인공을 PlayerController가 빙의할 때마다 다른 이벤트 처리가 필요할 때 사용하면 좋은 구조이다.

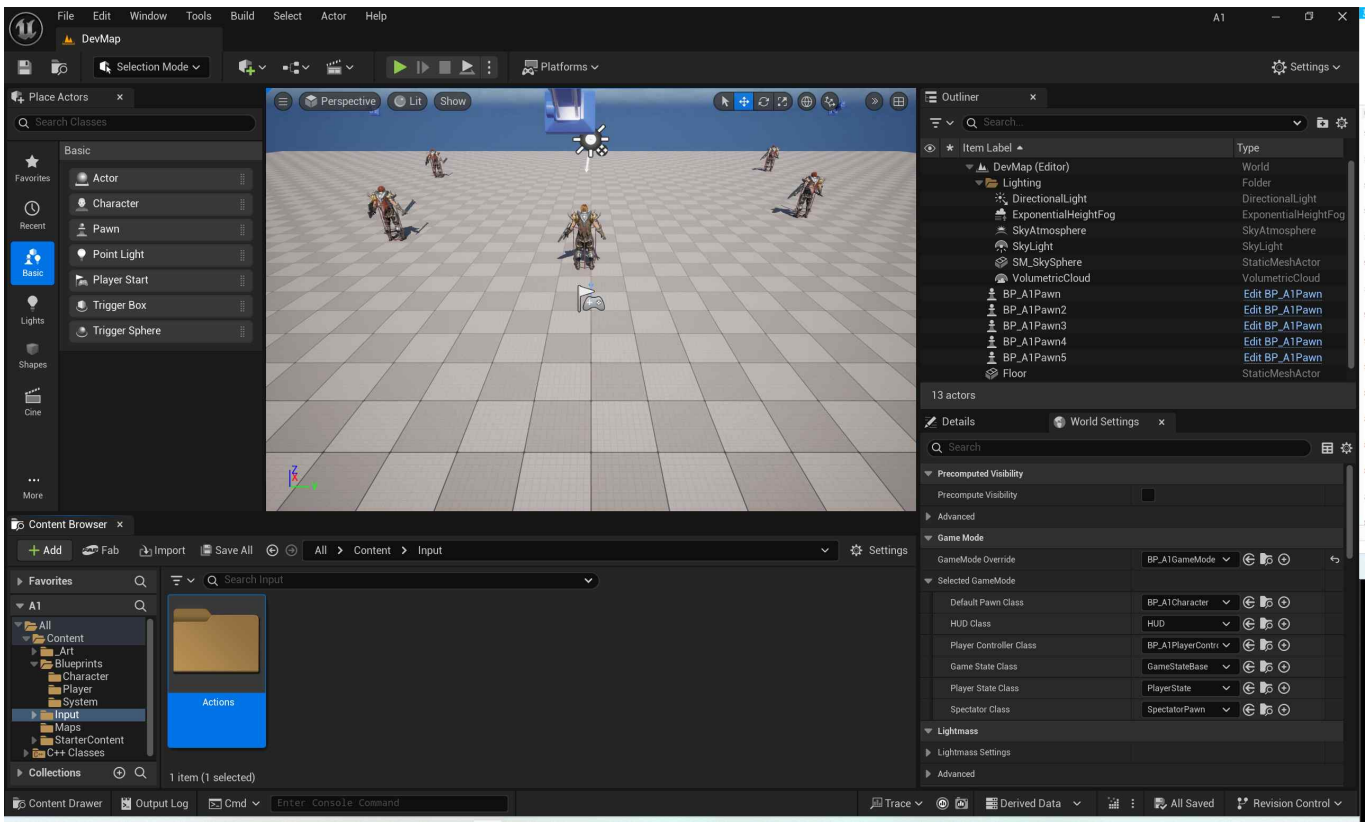


2) Enhanced Input 관련 에셋

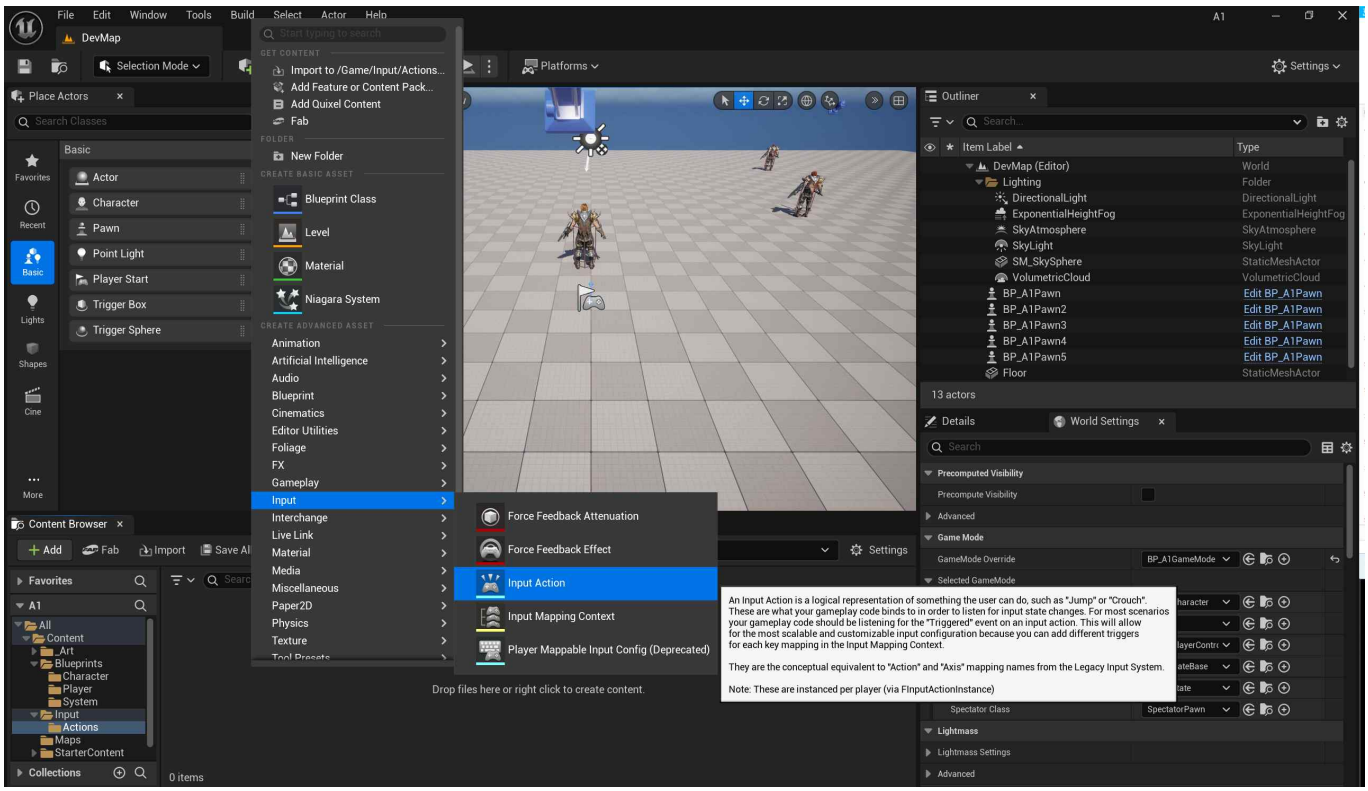
- 언리얼 에디터의 **Content 폴더**에 **Input 폴더**를 하나 생성하자.



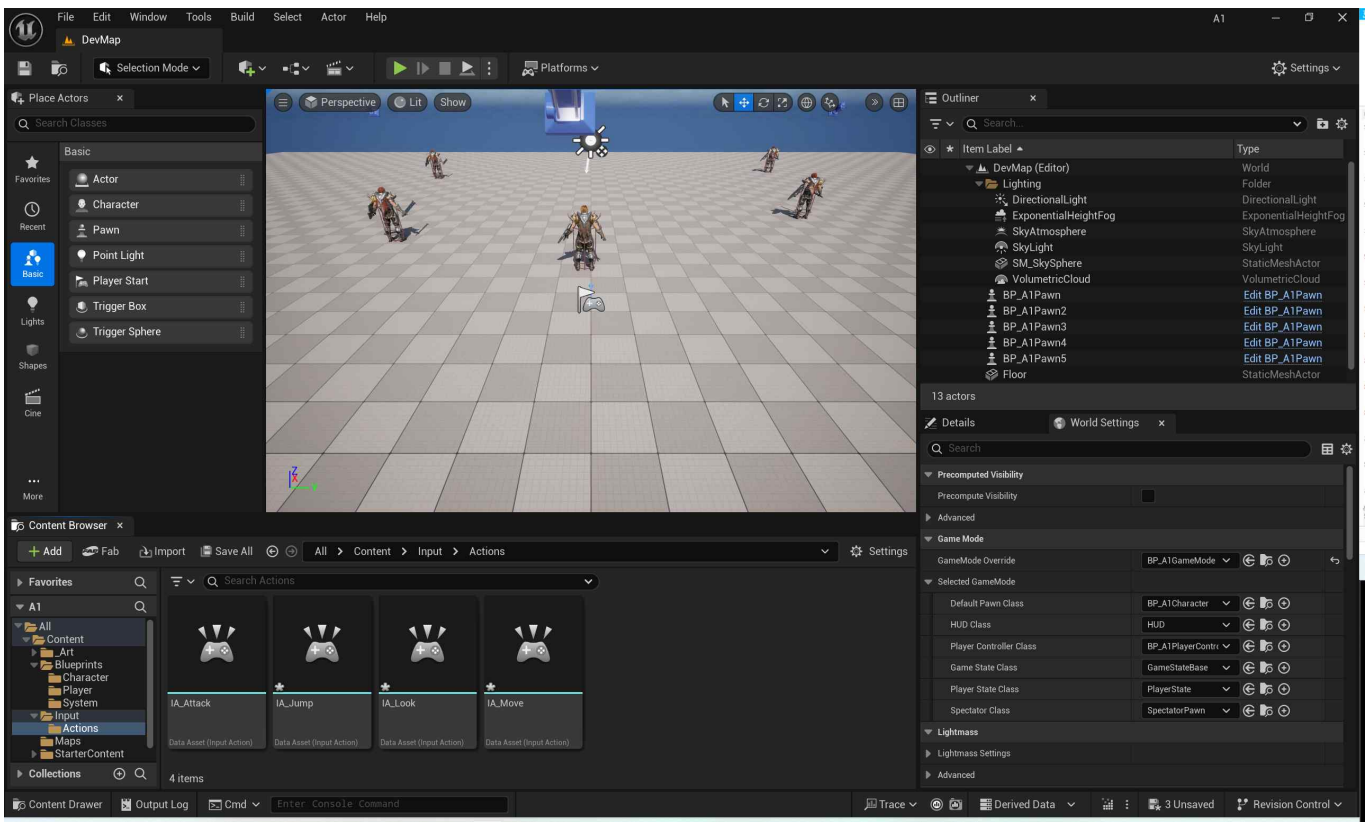
- Input 폴더 하단에 **Actions 폴더**를 하나 추가하자.



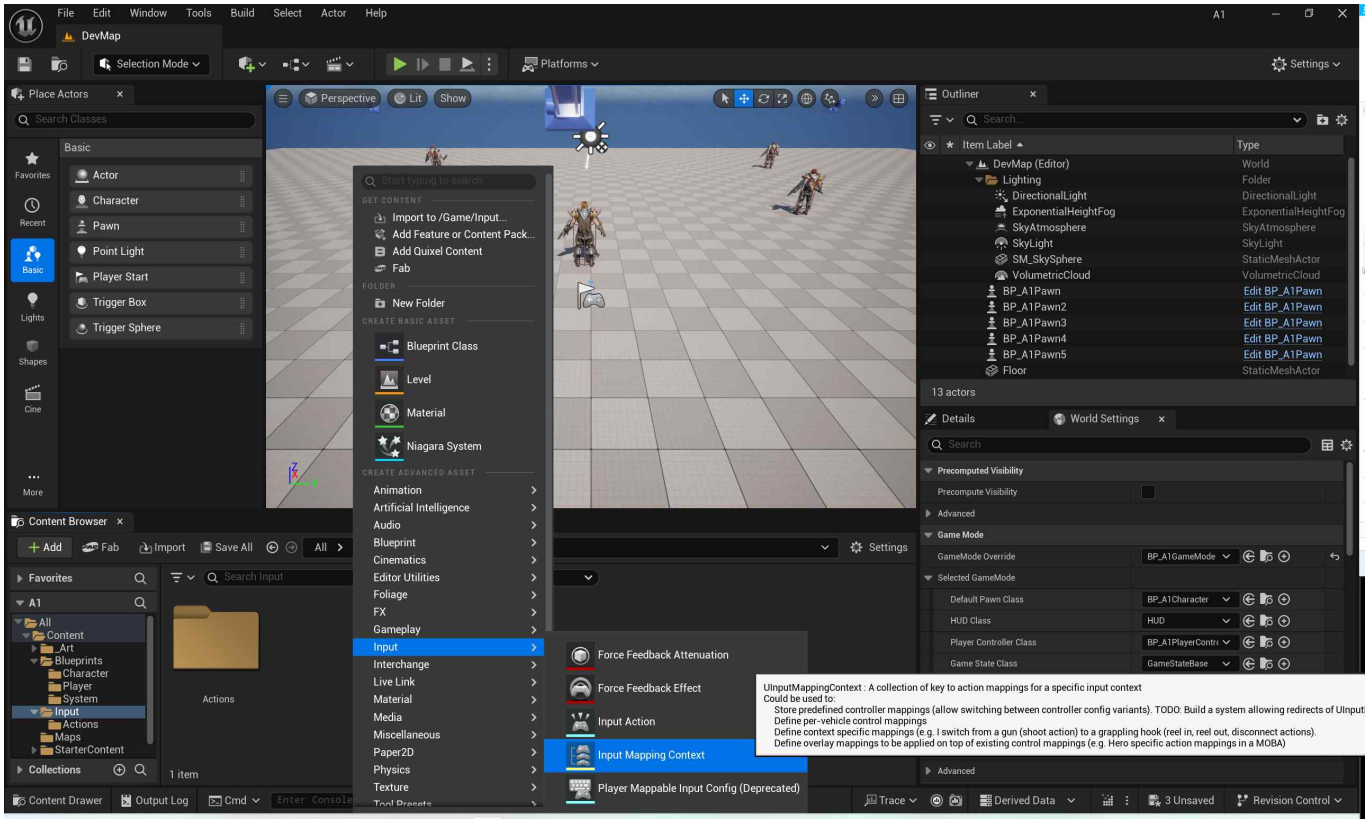
- InputAction(입력 액션)은 액션이 할당되는 부분이다.
- Input Action에서는 특정 키가 연결되지 않고, 역할에 대한 정보만을 구성하게 된다.
- 입력받는 정보는 bool, float, Vector 2D, Vector 3D 를 받을 수 있게 되며, 내가 어떤 입력값을 받을지에 따라 이를 정할 수 있다.
- **Actions 폴더**에서 마우스 오른쪽 메뉴의 **Input 항목**에서 **Input Action**을 네 개 추가하자.



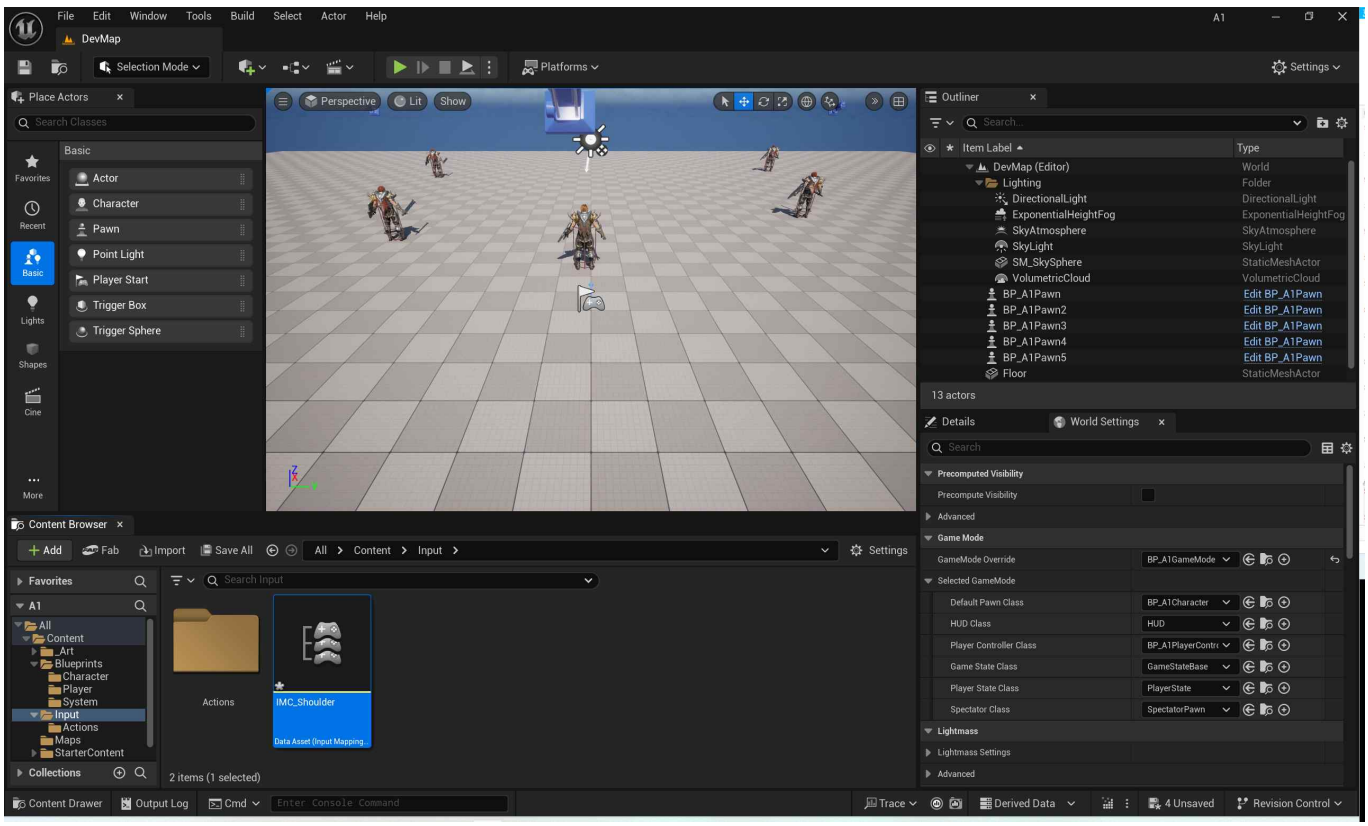
- 추가한 Input Action의 이름은 **IA_Attack, IA_Jump, IA_Look, IA_Move**로 설정하자.



- 추가한 Input Action과 입력 장치를 연결하는 Input Mapping Context를 추가하자.
- **Input 폴더**로 이동해서 마우스 오른쪽 메뉴를 통해 Input의 **Input Mapping Context**를 추가하자.

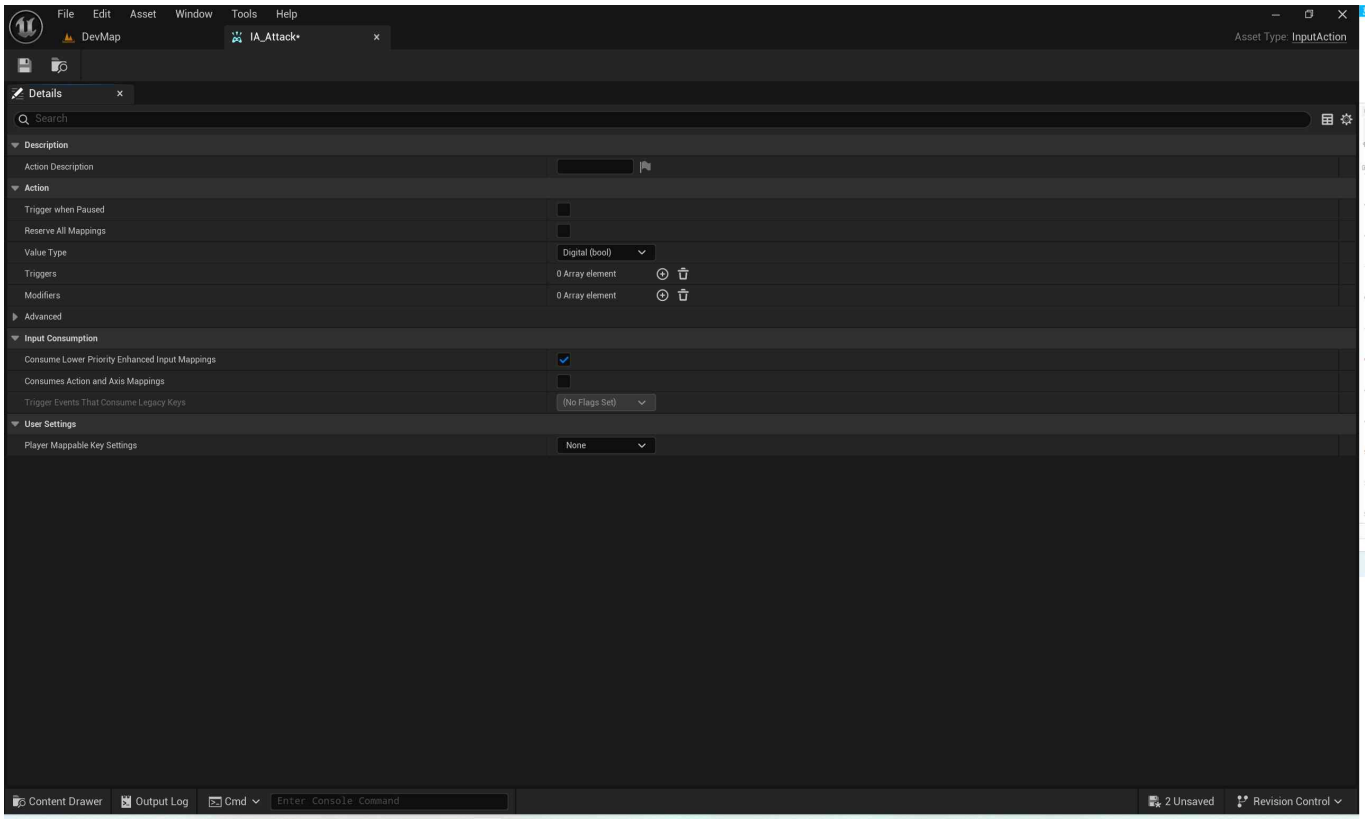


- 추가한 Input Mapping Context의 이름은 **IMC_Shoulder**로 설정하자.

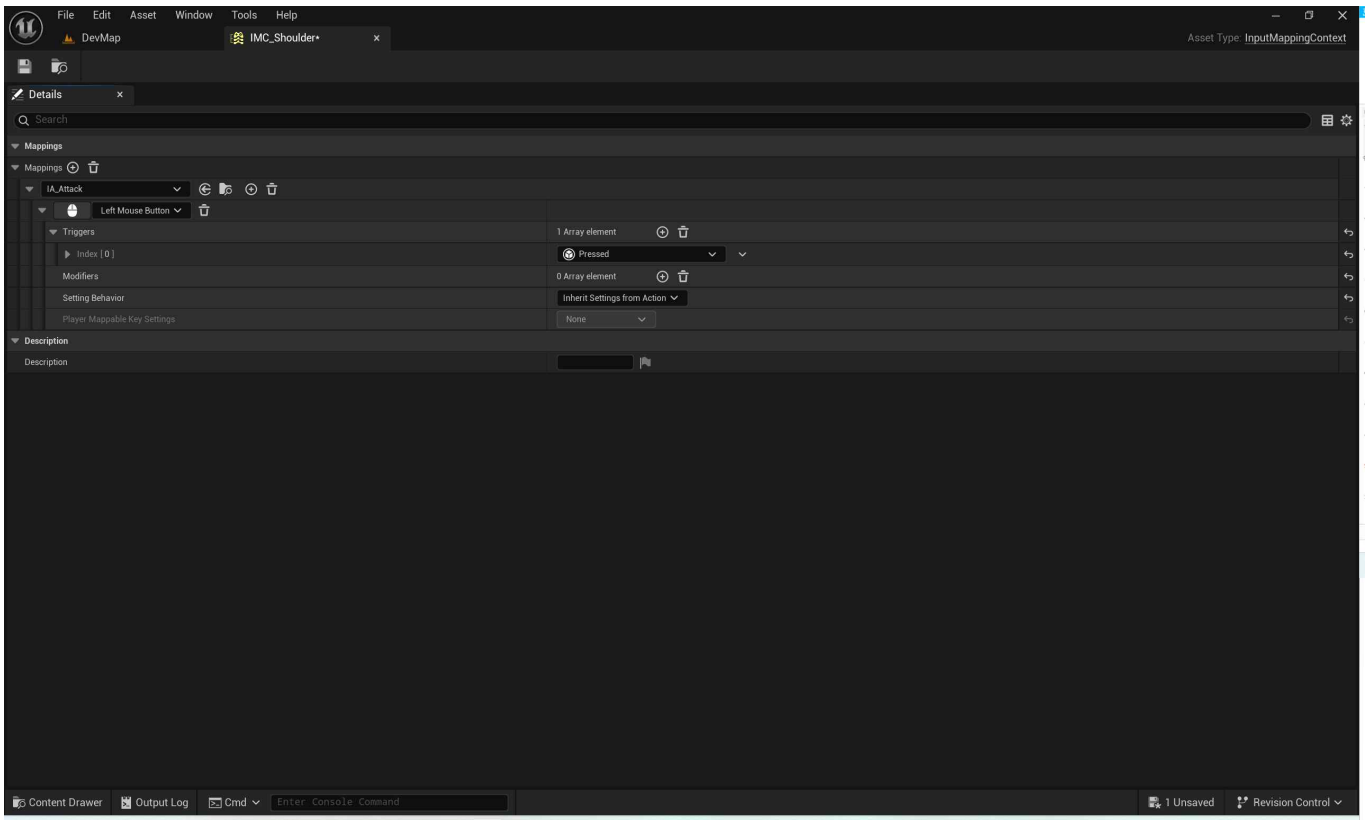


① IA_Attack Input Action

- 마우스 왼쪽 키를 눌렀을 때, 공격 기능을 구현한다고 하면 해당 기능을 처리하기 위해서 필요한 데이터 공간은 눌렀느냐, 눌리지 않았느냐만 판단하는 **Digital(bool)** 타입이면 충분하다.

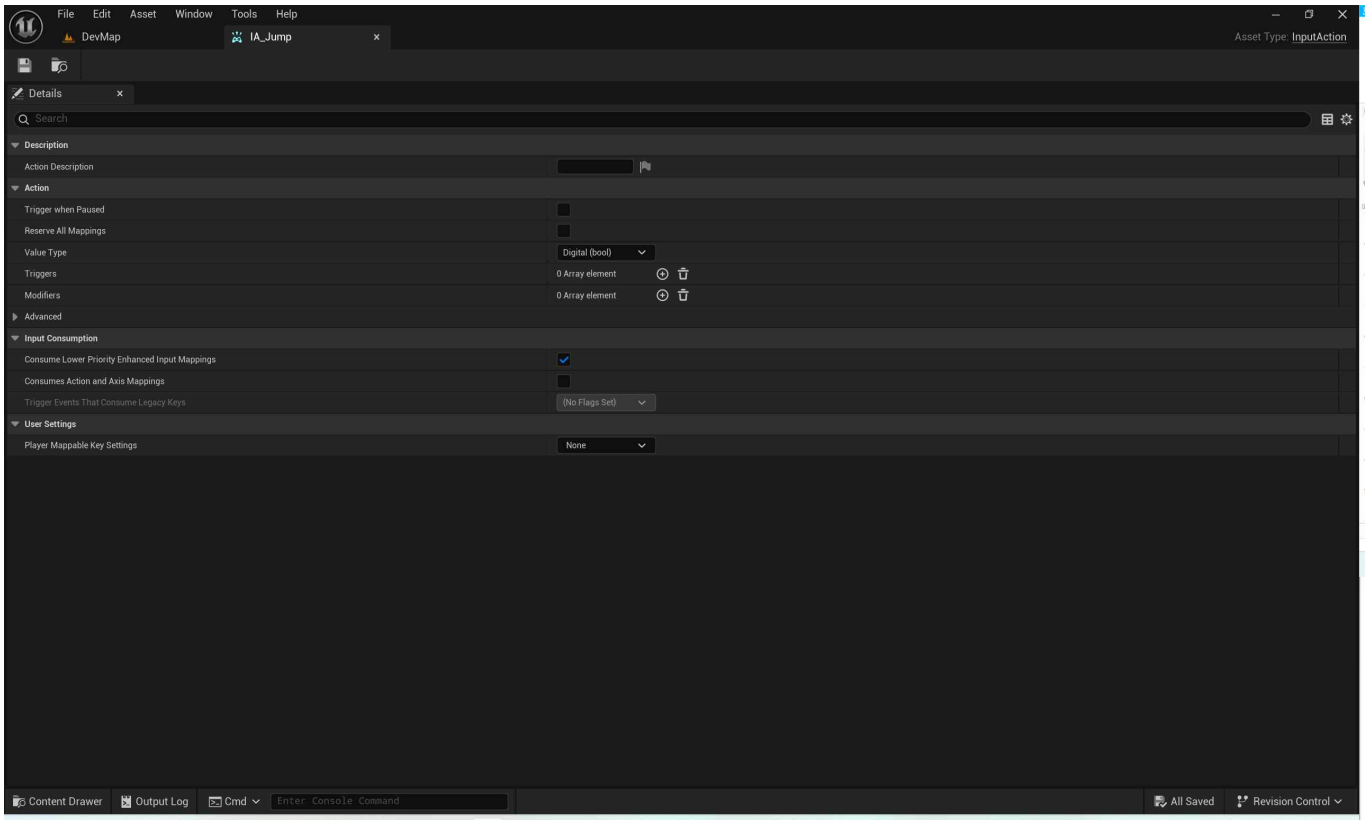


- **IMC_Shoulder**를 열어서 생성한 Input Action과 키를 매핑해보자.
- 먼저, **IA_Attack**를 마우스 왼쪽 버튼에 매핑시키고 **Triggers**의 이벤트 처리를 **Pressed**되도록 하자.

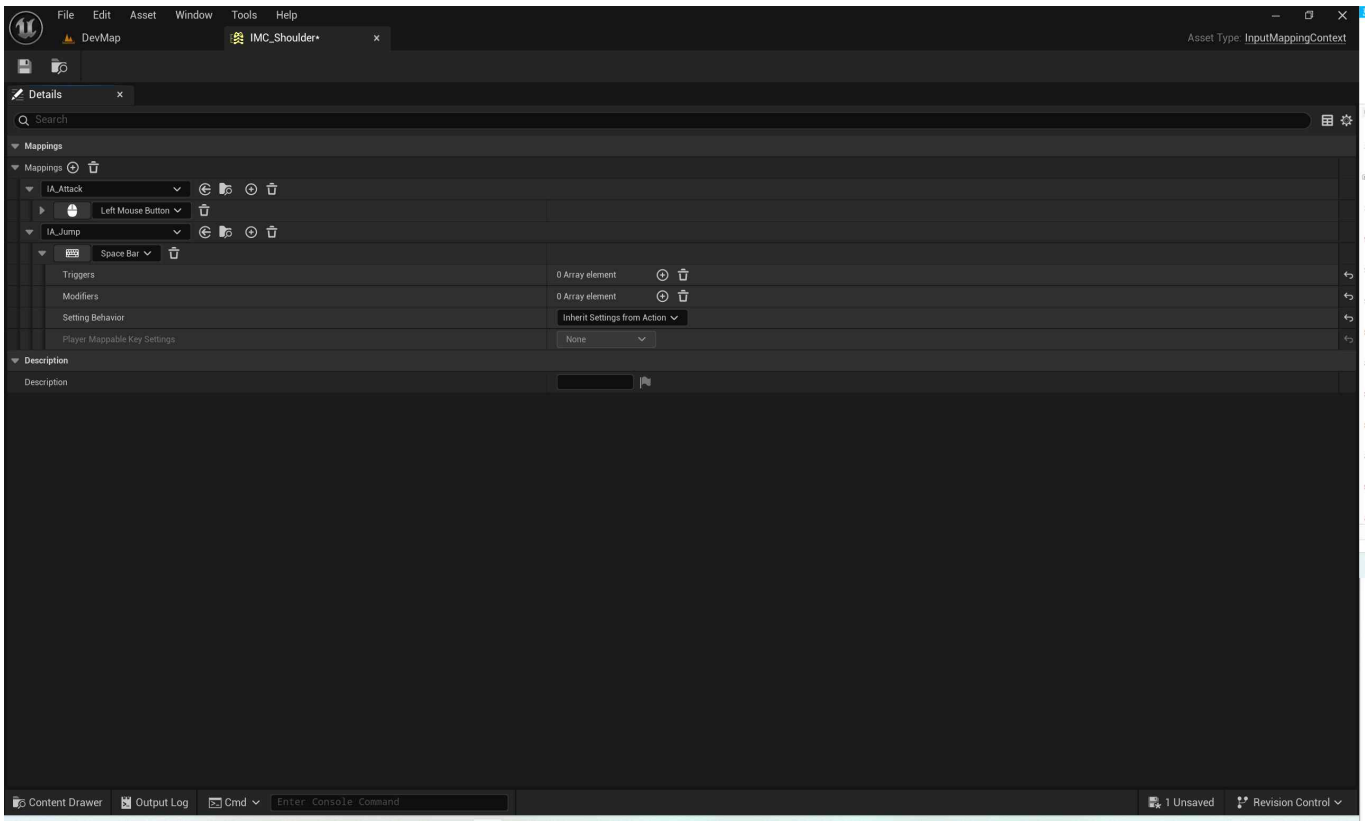


② IA_Jump Input Action

- 먼저, IA_Jump의 Value Type를 생각해보면 SpaceBar를 눌렀을 때, Jump 상태가 되고 누른키가 완료되었을 때, StopJumping을 호출하게 되므로 **Digital(bool)** 타입이면 충분하다.

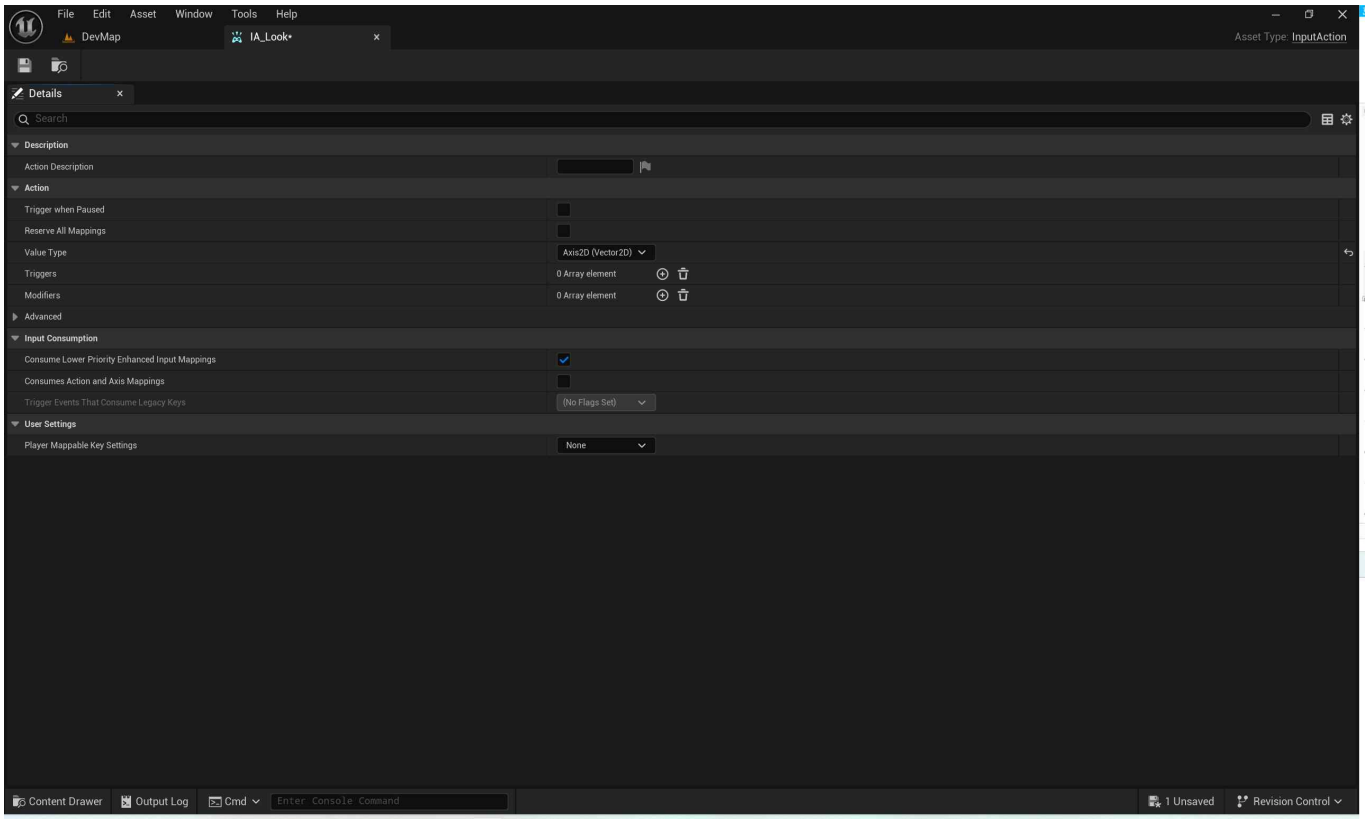


- IMC_Shoulder를 열어서 IA_Jump에 SpaceBar 키를 매핑시키자.
- Trigger 처리는 Start와 Completed를 사용할 예정이라 추가설정은 필요 없다.

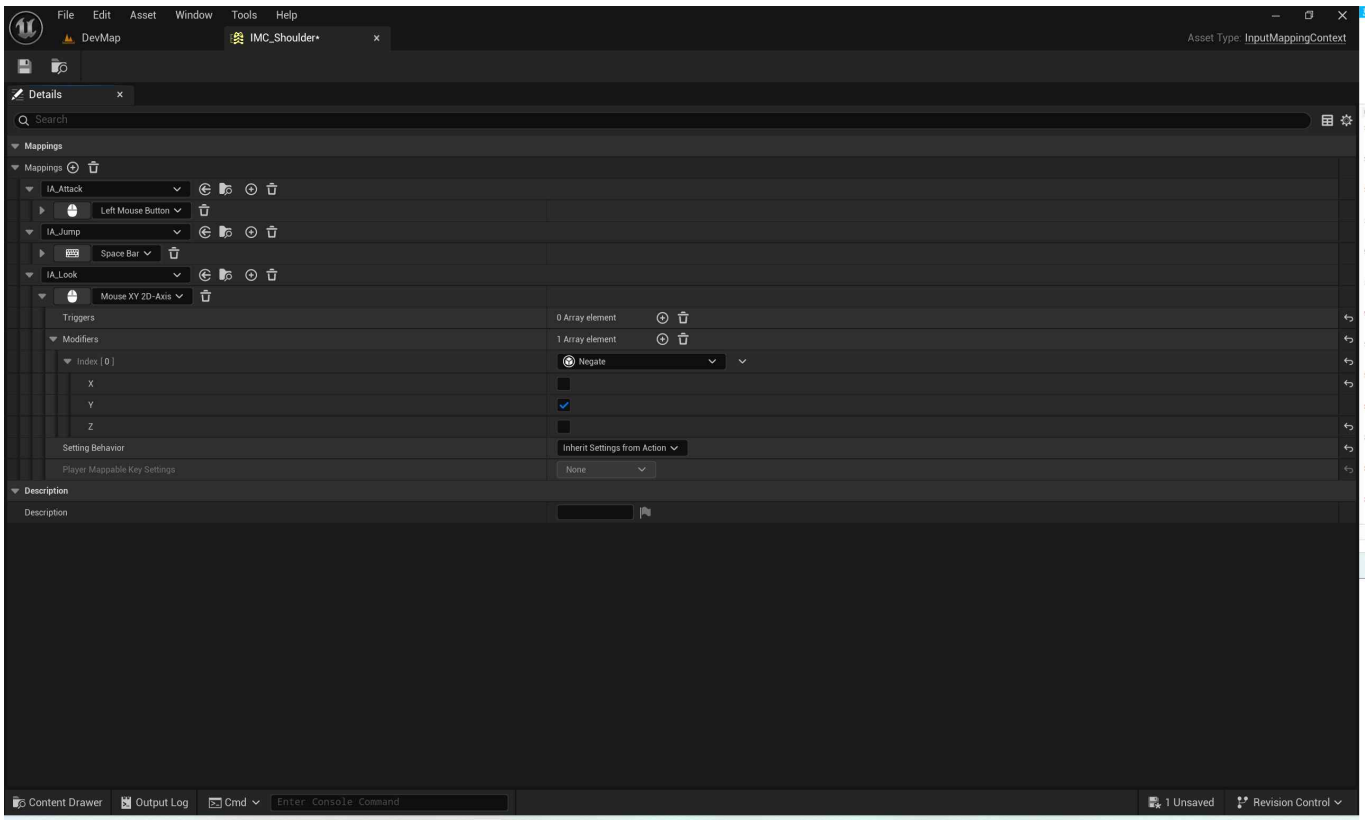


③ IA_Look Input Action

- 마우스를 좌우로 움직일 때, 캐릭터의 회전이 될 수 있도록 구현해보자.
- 마우스의 XY 좌표값이 **Vector2D**에 저장될 수 있도록 Value Type을 설정하자.

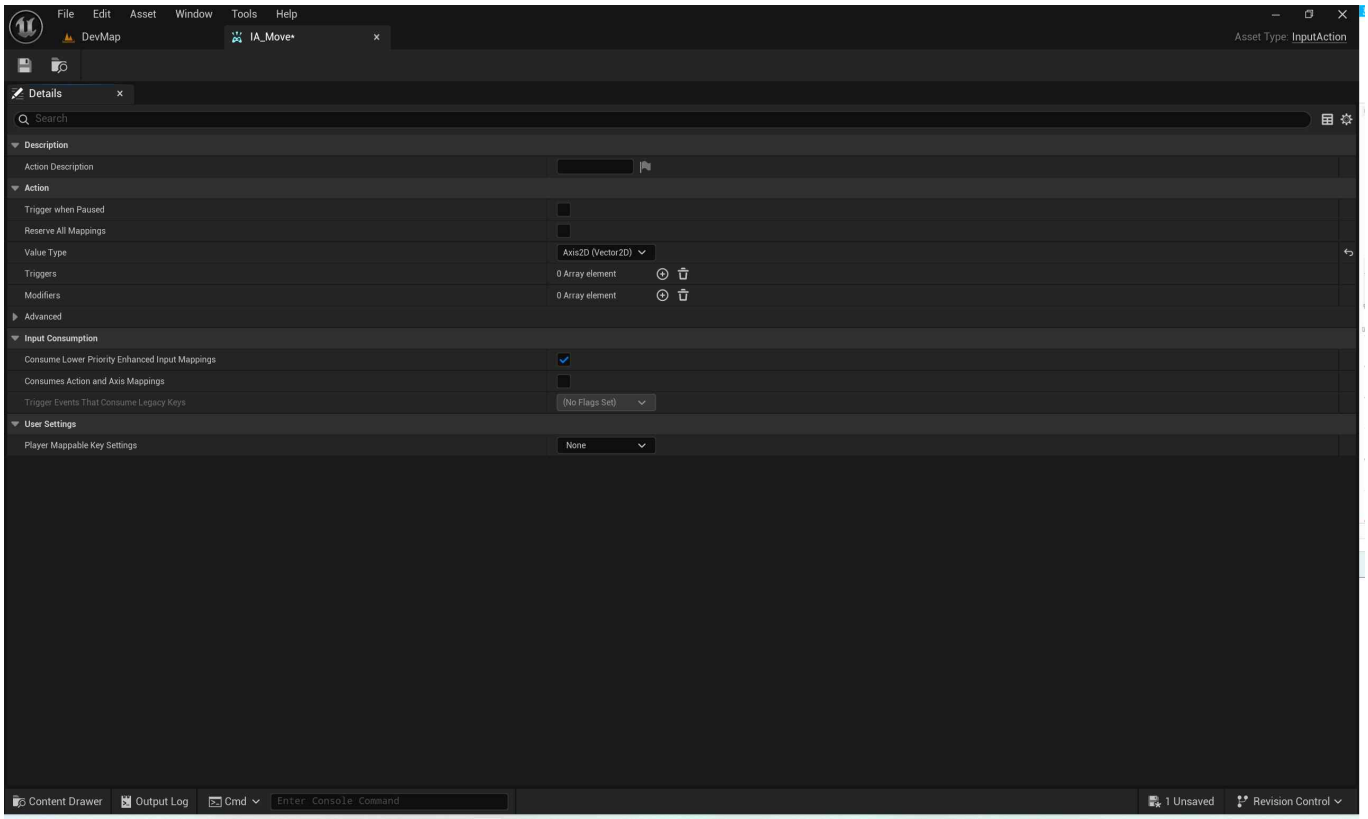


- **마우스의 XY 좌표값** Vector2D에 자동으로 입력되는 키를 매핑하자.
- **Y축**의 값은 마우스 이동과 게임에서 사용하는 좌표축이 반대라서 **Modifiers**설정에 **Negate**처리 하자.

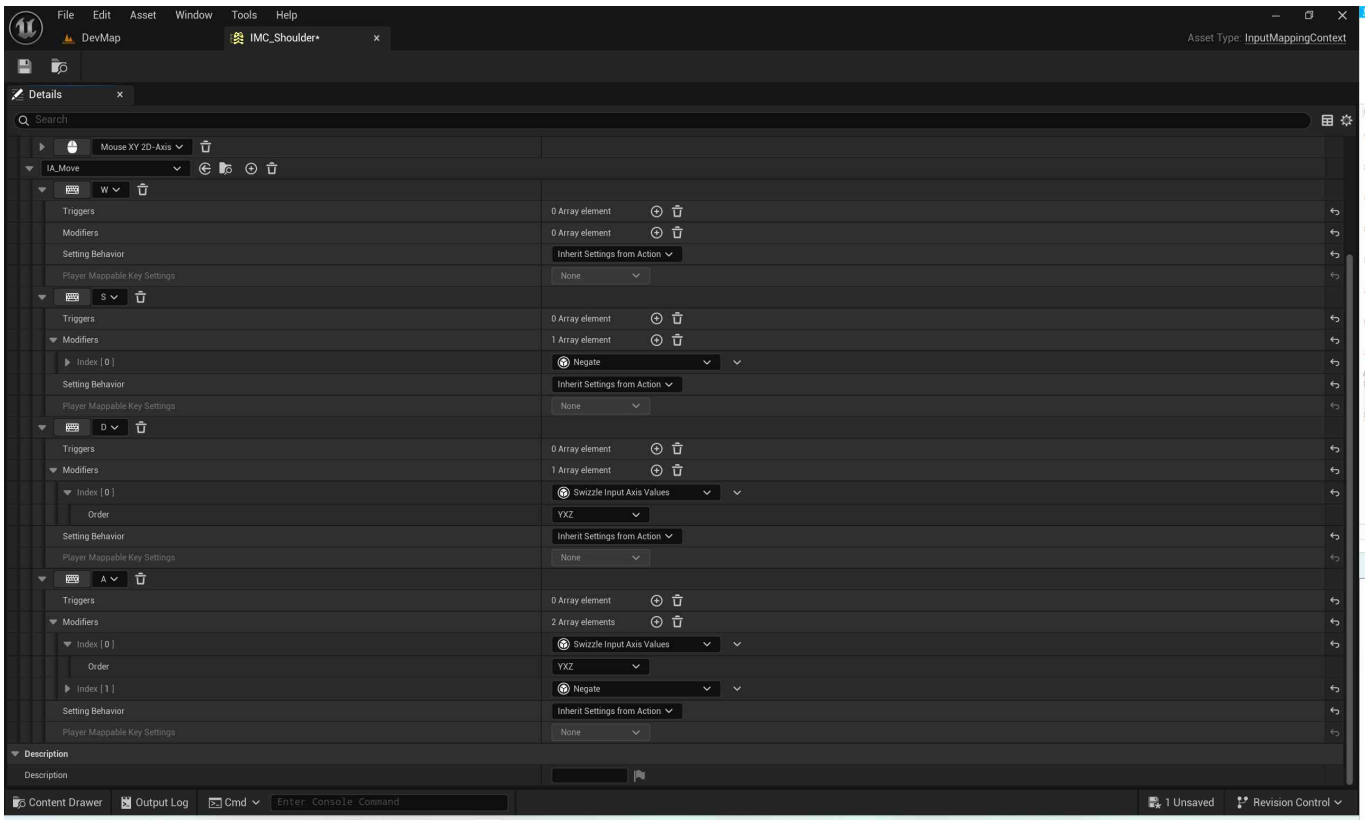


④ IA_Move Input Action

- 먼저, IA_Move의 Value Type를 생각해보면 WSAD키를 누를 때, 해당 방향으로 이동하는 키 값을 저장하는 자료형이 필요하다. 이를 위해서 **Axis2D(Vector2D)**를 사용하는 것이 좋다.



- IMC_Shoulder를 열어서 IA_Move에 4개의 키를 매핑시키자.
- 하지만, 이렇게 4개의 키를 매핑하면 키를 누를때마다 1의 값이 발생한다. 적절히 변형하자.



3) C++에서 구현하기

- **A1Character**에 EnhancedInput을 C++로 구현해보자.

A1PlayerController.h

```
// Fill out your copyright notice in the Description page of Project Settings.


#pragma once


#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "A1Character.generated.h"


struct FInputActionValue;
class UInputMappingContext;
class UInputAction;

UCLASS()
class A1_API AA1Character : public ACharacter
{
	GENERATED_BODY()

public:
	// Sets default values for this character's properties
	AA1Character();

protected:
	// Called when the game starts or when spawned
	virtual void BeginPlay() override;

public:
	// Called every frame
	virtual void Tick(float DeltaTime) override;

	// Called to bind functionality to input
	virtual void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent) override;

protected:
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
	TObjectPtr<class USpringArmComponent> SpringArm;

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
	TObjectPtr<class UCameraComponent> Camera;
```

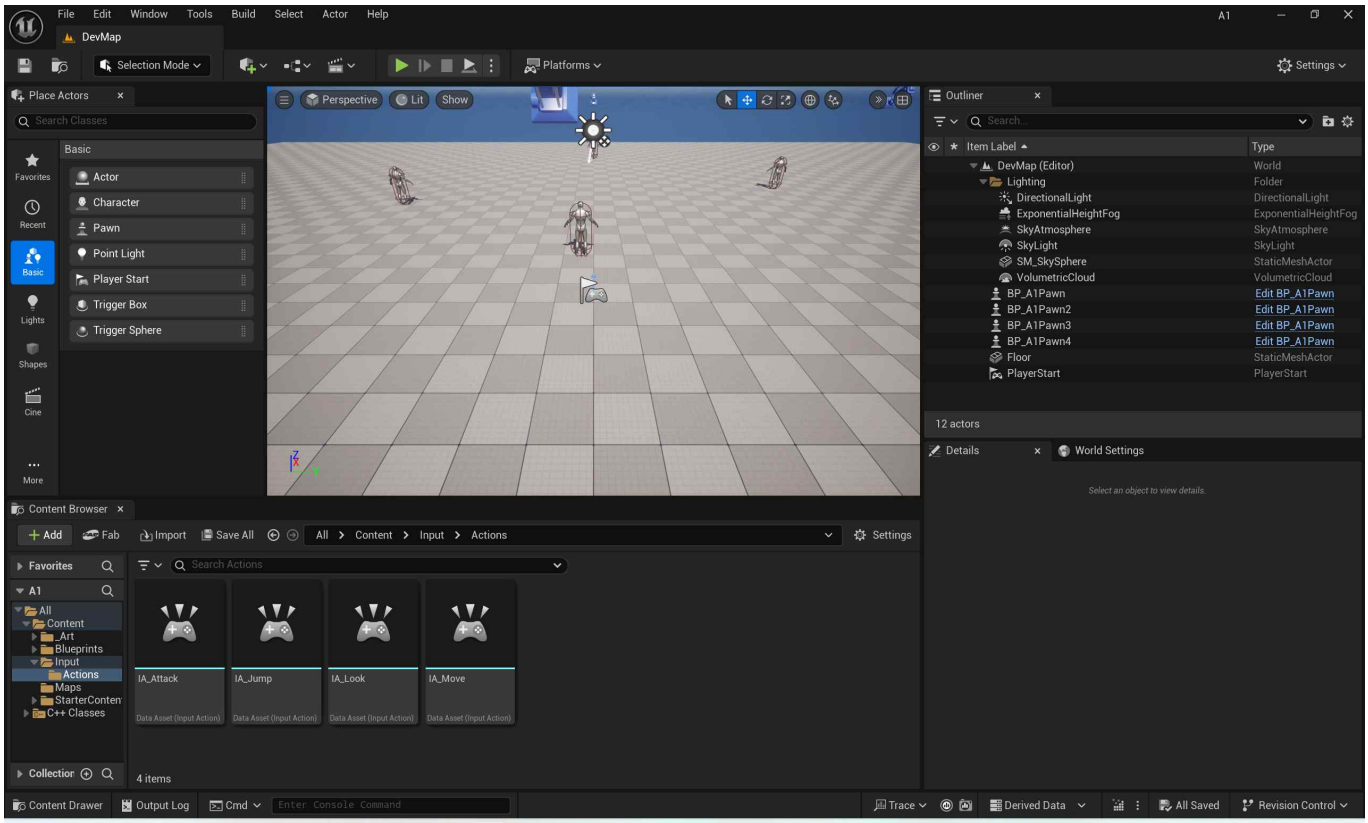
```
#pragma region EnhancedInputSystem
public:
	void Input_Attack(const FInputActionValue& InputValue);
	void Input_Look(const FInputActionValue& InputValue);
	void Input_Move(const FInputActionValue& InputValue);

protected:
	UPROPERTY(EditAnywhere, Category = Input)
	TObjectPtr<UInputMappingContext> IMCShoulder;

	UPROPERTY(EditAnywhere, Category = Input)
	TObjectPtr<UInputAction> AttackAction;

	UPROPERTY(EditAnywhere, Category = Input)
	TObjectPtr<UInputAction> JumpAction;

	UPROPERTY(EditAnywhere, Category = Input)
	TObjectPtr<UInputAction> LookAction;

	UPROPERTY(EditAnywhere, Category = Input)
	TObjectPtr<UInputAction> MoveAction;
#pragma endregion
};
```

A1Character.cpp

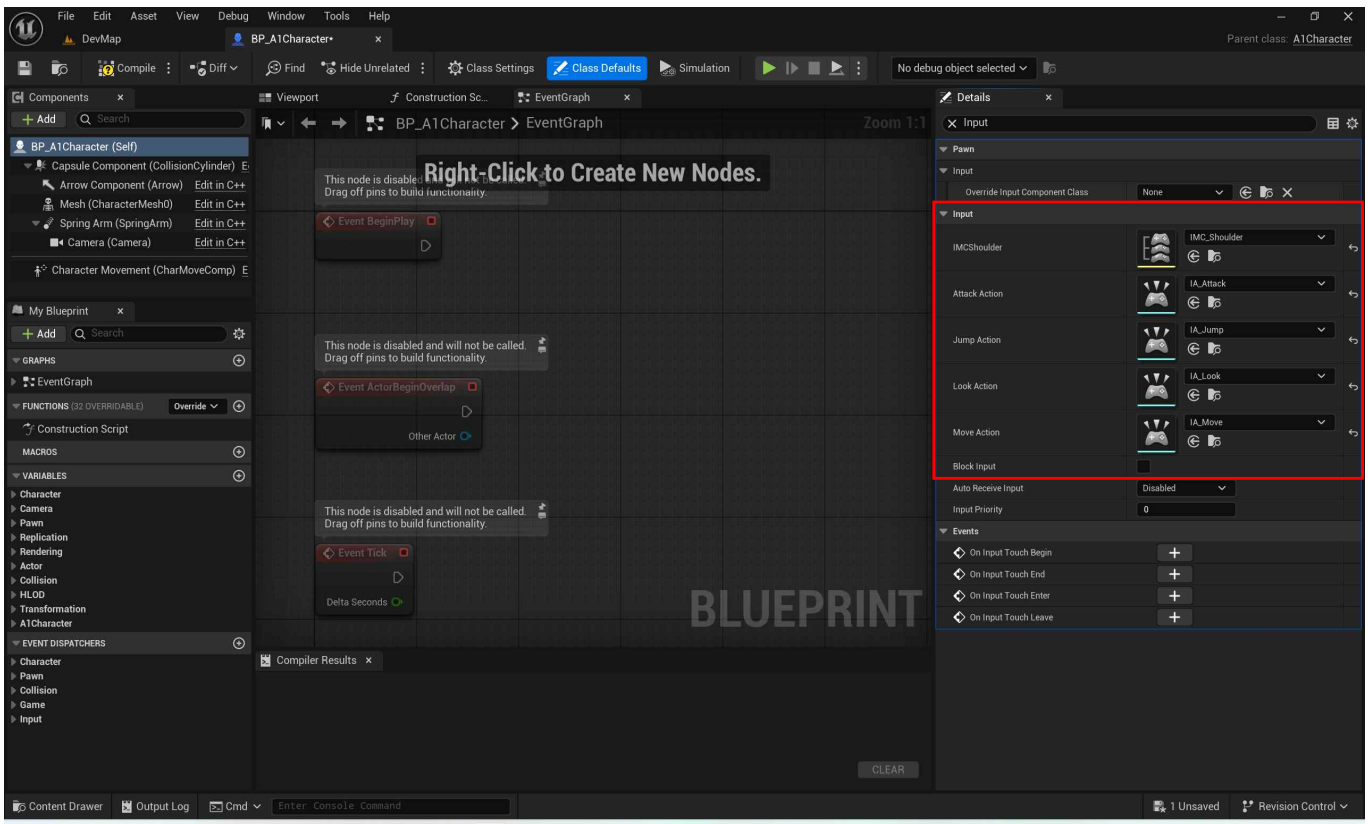
```
// Fill out your copyright notice in the Description page of Project Settings.
#include "Character/A1Character.h"
#include "GameFramework/SpringArmComponent.h"
#include "Camera/CameraComponent.h"
#include <InputActionValue.h>
#include <InputMappingContext.h>
#include <EnhancedInputComponent.h>
#include <EnhancedInputSubsystems.h>

// Sets default values
AA1Character::AA1Character()
{
	// Set this character to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
	PrimaryActorTick.bCanEverTick = true;

	SpringArm = CreateDefaultSubobject<USpringArmComponent>(TEXT("SpringArm"));
	SpringArm->SetupAttachment(GetRootComponent());
	SpringArm->TargetArmLength = 700.0f;
	SpringArm->SetRelativeRotation(FRotator(-30.0f, 0.0f, 0.0f));

	Camera = CreateDefaultSubobject<UCameraComponent>(TEXT("Camera"));
	Camera->SetupAttachment(SpringArm);
}

// Called when the game starts or when spawned
void AA1Character::BeginPlay()
{
	Super::BeginPlay();

	APlayerController* PlayerController = CastChecked<APlayerController>(GetController());
	UEnhancedInputLocalPlayerSubsystem* Subsystem = ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerController->GetLocalPlayer());

	if (Subsystem)
	{
		Subsystem->AddMappingContext(IMCShoulder, 0);
	}
}

// Called every frame
void AA1Character::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);

}
```

```
// Called to bind functionality to input
void AA1Character::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
	Super::SetupPlayerInputComponent(PlayerInputComponent);

	UEnhancedInputComponent* EnhancedInputComponent = CastChecked<UEnhancedInputComponent>(PlayerInputComponent);

	EnhancedInputComponent->BindAction(AttackAction, ETriggerEvent::Triggered, this, &AA1Character::Input_Attack);
	EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Started, this, &ACharacter::Jump);
	EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Completed, this, &ACharacter::StopJumping);
	EnhancedInputComponent->BindAction(LookAction, ETriggerEvent::Triggered, this, &AA1Character::Input_Look);
	EnhancedInputComponent->BindAction(MoveAction, ETriggerEvent::Triggered, this, &AA1Character::Input_Move);

}

void AA1Character::Input_Attack(const FInputActionValue& InputValue)
{
	GEngine->AddOnScreenDebugMessage(-1, 1.0f, FColor::Blue, TEXT("Attack"));
}

void AA1Character::Input_Look(const FInputActionValue& InputValue)
{
	FVector2D LookAxisVector = InputValue.Get<FVector2D>();
	FRotator AddRotation(LookAxisVector.Y, LookAxisVector.X, 0.0f);
	AddActorWorldRotation(AddRotation);
}

void AA1Character::Input_Move(const FInputActionValue& InputValue)
{
	FVector2D MovementVector = InputValue.Get<FVector2D>();

	const FVector ForwardDirection = GetActorForwardVector();
	const FVector RightDirection = GetActorRightVector();

	AddActorWorldOffset(ForwardDirection * MovementVector.X);
	AddActorWorldOffset(RightDirection * MovementVector.Y);
}
```

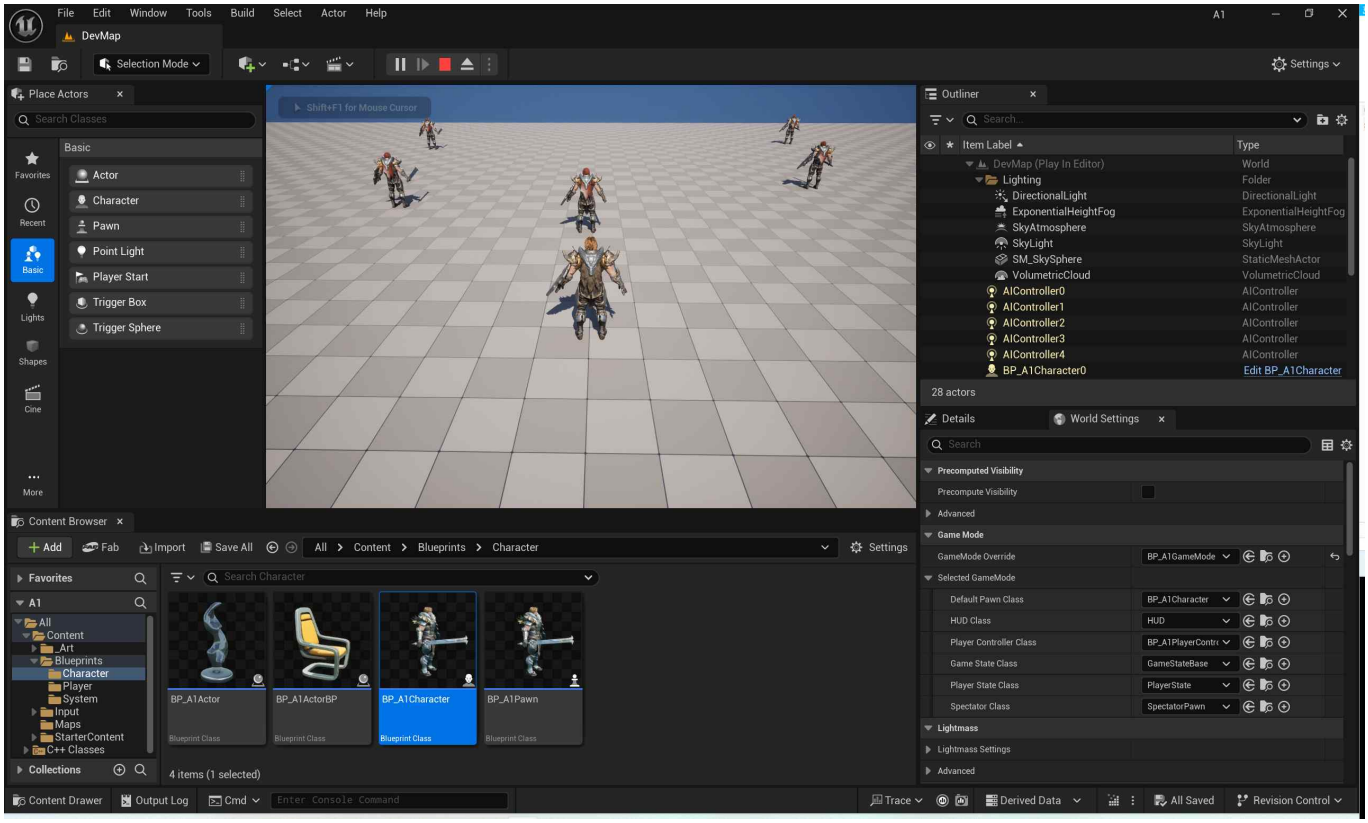
- 빌드 후 언리얼 에디터를 실행해 보자.



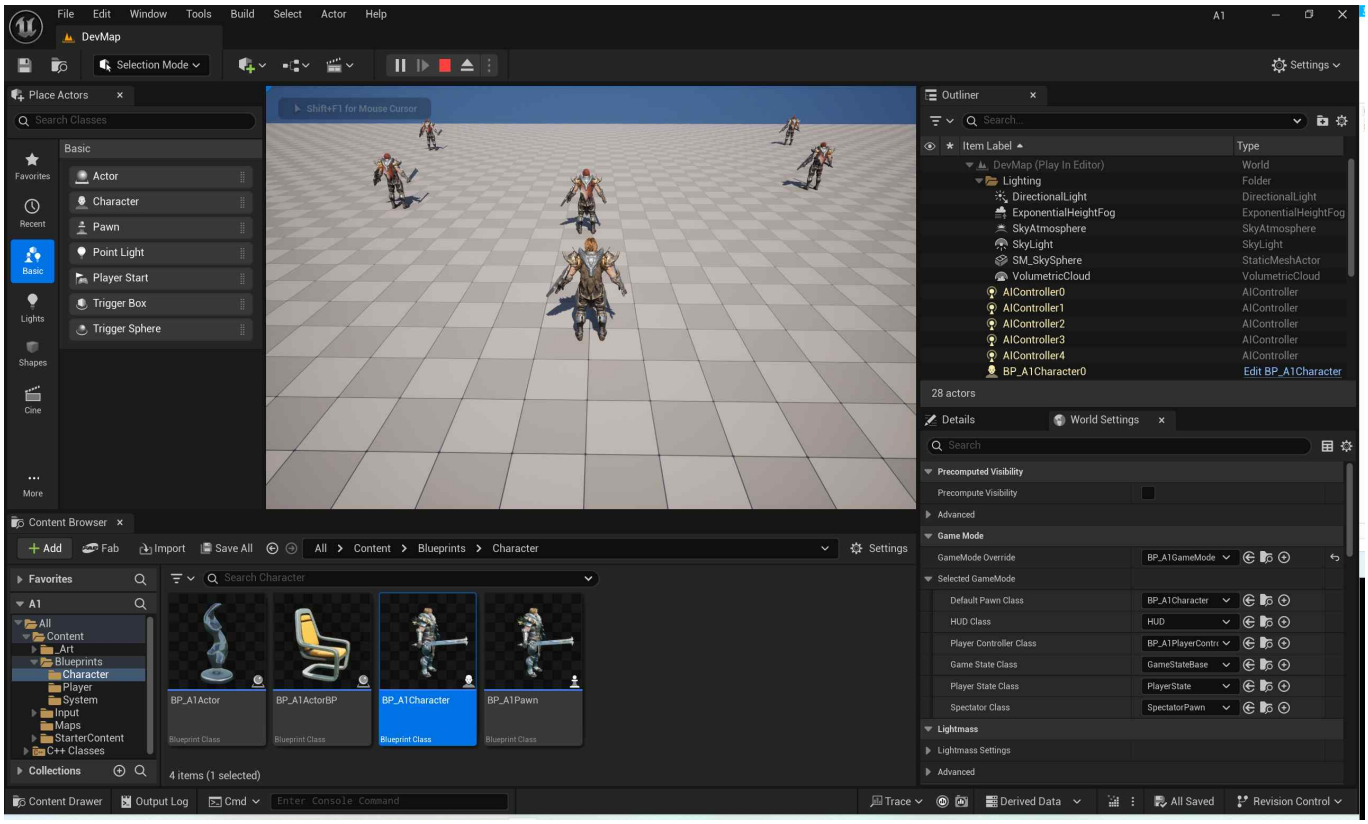
- BP_A1Character를 열어서 InputMappingContext와 InputAction들을 설정하자.
- 카테고리 명인 **Input**을 검색하면 된다.



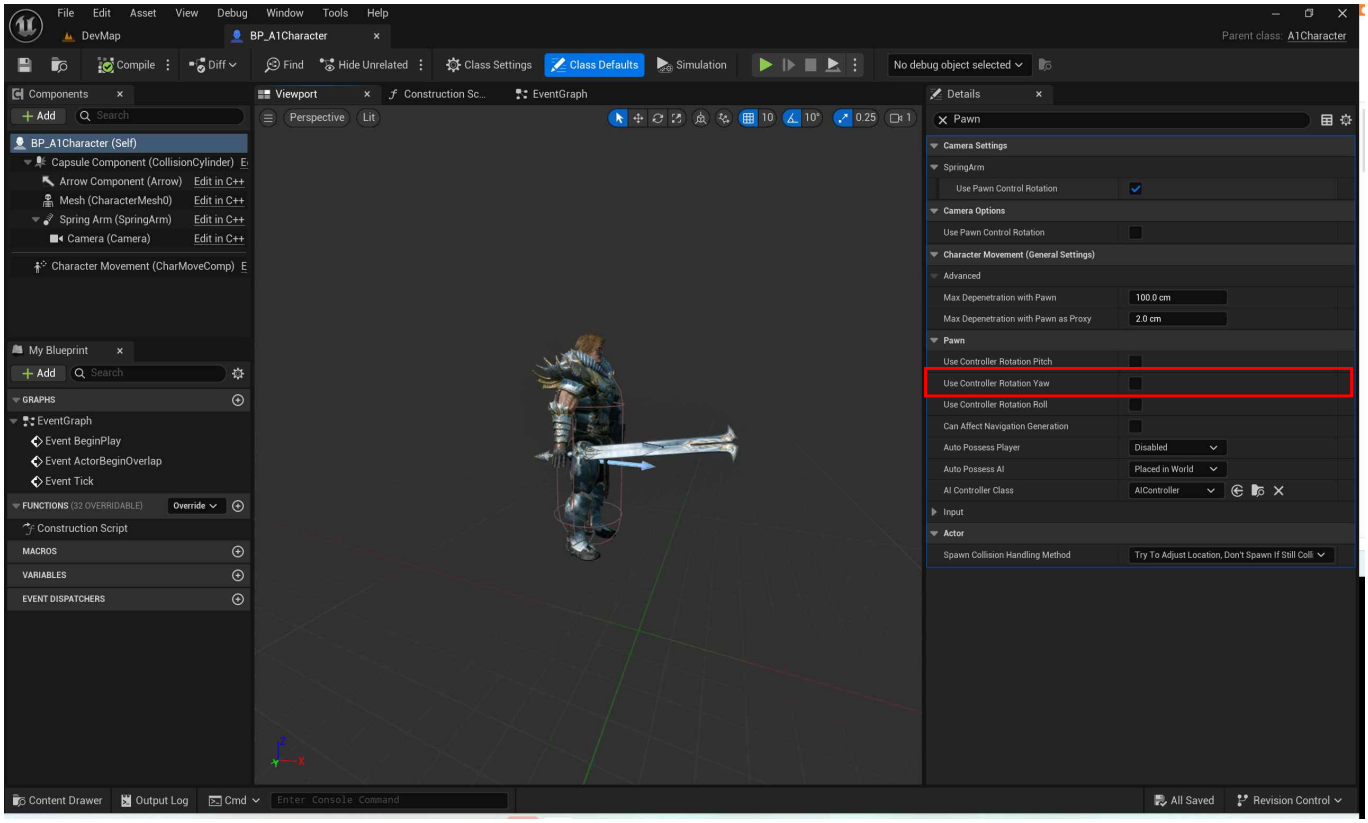
- 이제 게임을 실행해보자.
- 이동이 잘 적용된다.



- 회전도 은 잘되지 않는다.



- BP_A1Character옵션을 하나 변경해보자.
- Pawn을 검색해서 **Use Controller Rotation Yaw**의 옵션을 끄자.



- 다시 게임을 실행하면 캐릭터가 회전한다.