COMP280 – Networking	Workshop 5
HTTP Clients in Unreal	

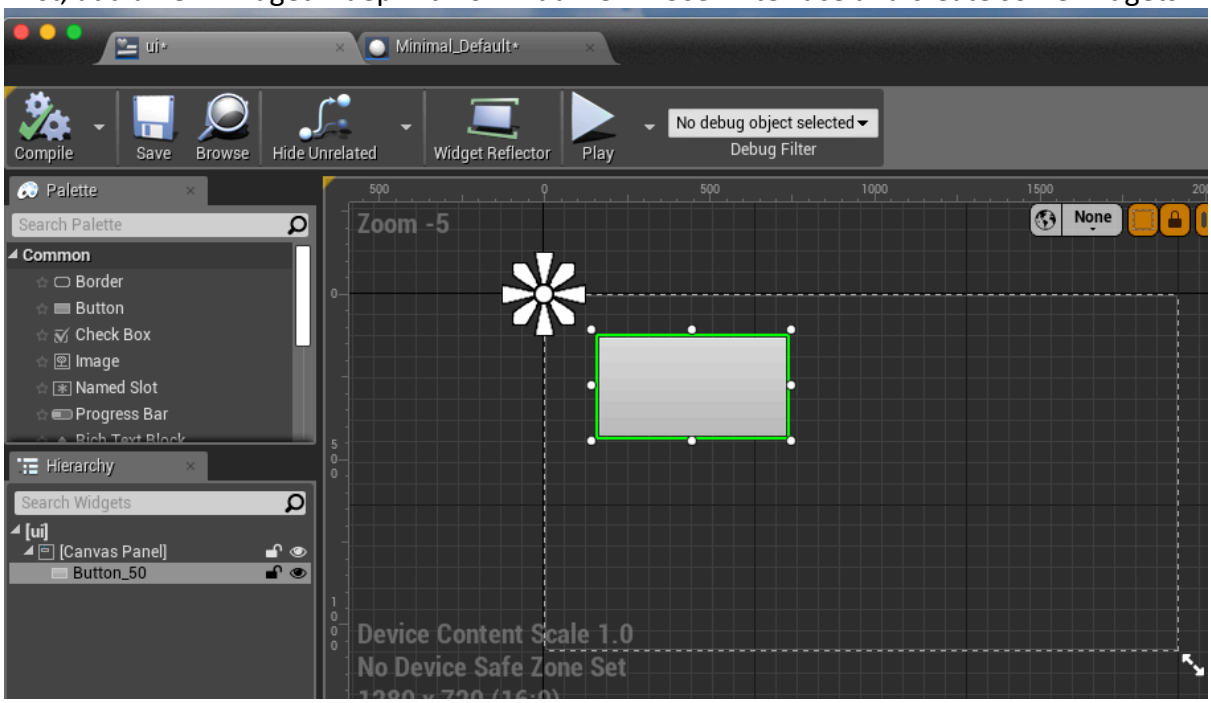
Introduction

The goal of this workshop is to add an HTTP Client to a simple Unreal project. To do this, we will start from a 'simple' Blueprint UI application and C++ and HTTP functionalities.

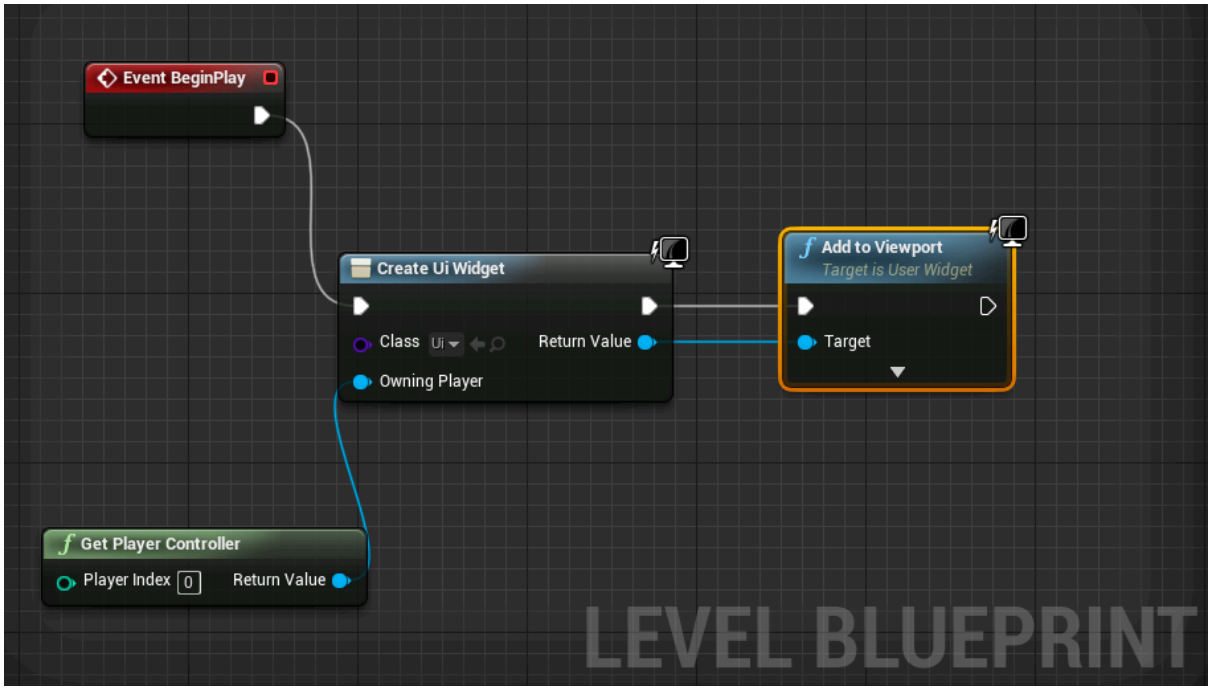
Creating a Blueprint-based UI application in Unreal

As a starting point, let's create a new Blueprint-based UE4 project. From Brian's previous workshops, we are aware that UI can be created in a wysiwyg style using the UI designer to layout the look and feel of the UI, whilst the widget blueprint can be used to implement functionality. Likewise, the level blueprint is used to link the UI to both the player and the display viewport (so we can actually see what we've made).

First, add a new Widget Blueprint from Add New->User Interface and create some widgets.



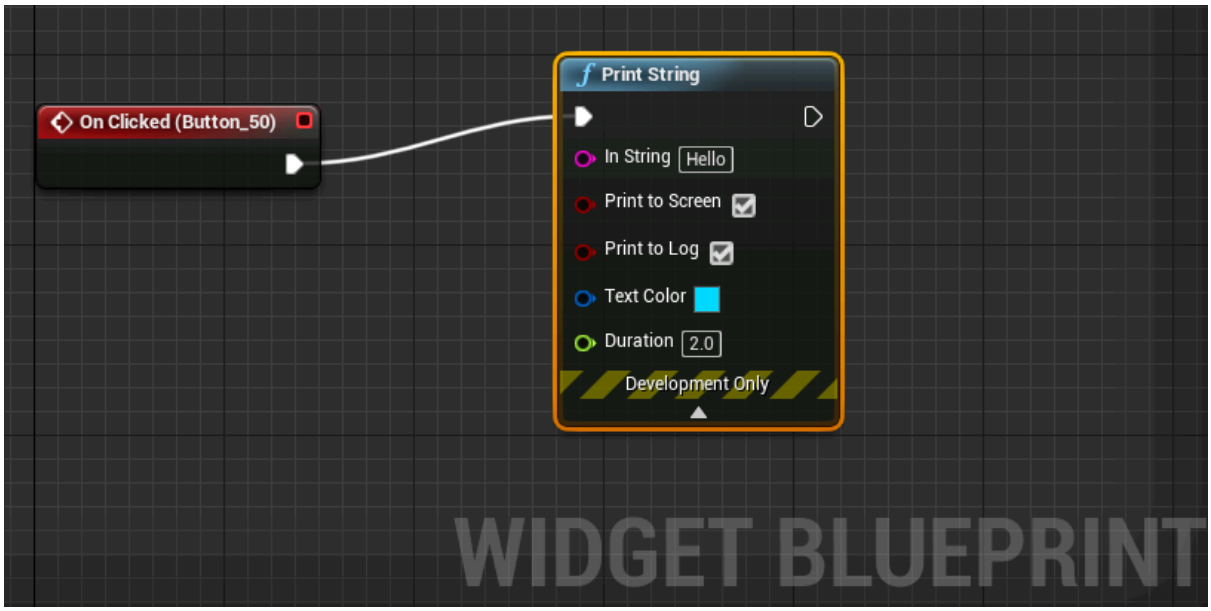
Next, open the level blueprint (blueprints icon) & link the UI blueprint to the application. This needs to be driven by the Event BeginPlay which will call CreateWidget for the UI class we have just made. CreateWidget requires a player controller (to control it) and its output will be used as an input to AdtoViewport.



For more info: https://docs.unrealengine.com/en-US/Engine/UMG/HowTo/CreatingWidgets/index.html

To make the buttons actually do something, add an 'OnClicked' event to a button. This will bring up the widget blueprint and we can add functionality. It's worth noting that you can rename all the widgets to more useful names that button_100.

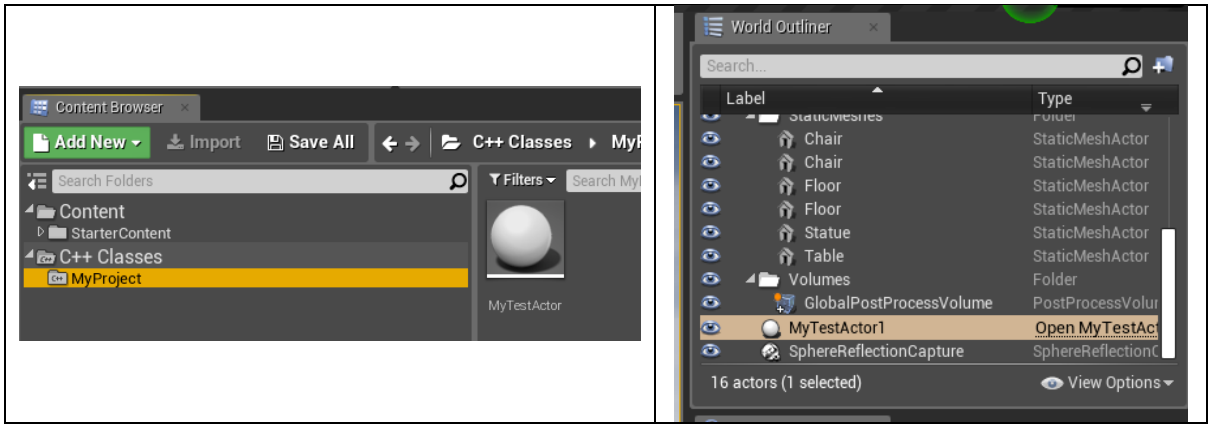
We can use PrintString to display debug messages in the game, like so, and link it to the button's OnClicked event so that every time the button is pressed, it will display a message.



https://docs.unrealengine.com/en-US/Engine/Blueprints/BP_HowTo/Debugging/index.html

Adding C++ Actors

Moving beyond pure blueprint functionality, we can add classes of functionality in C++. To do this, we can add a new C++ class and base it off the Actor class. Unreal will build the new class which will take some time, but once it has finished, you will be able to add your actor into the WorldOutliner.



To be able to reference the class in blueprints, we need to set some attributes in the C-code, the UCLASS definition needs to include 'Blueprintable' as a term. Re-compile the UE4 project and it will possible to reference the class in the widget blueprint.

MyTestActor.h

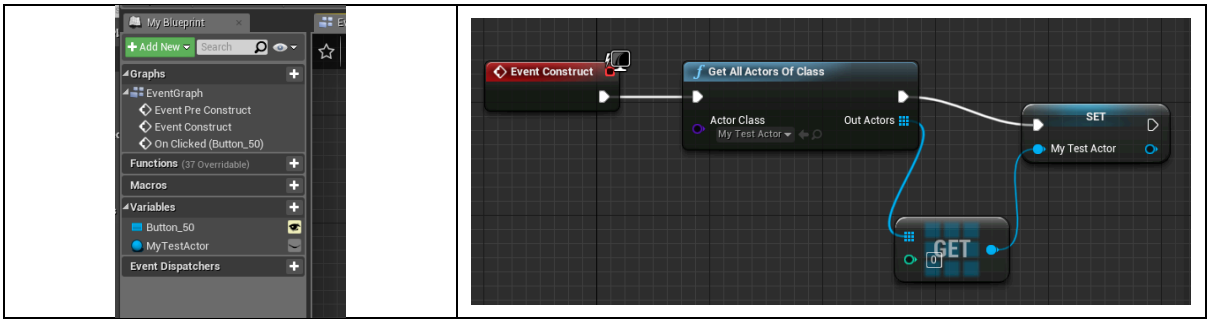
```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "MyTestActor.generated.h"

UCLASS(Blueprintable)
class MYPROJECT_API AMyTestActor : public AActor
{
    GENERATED_BODY()
```

In Variables part of the widget blueprint, add a new variable to hold the actor instance. We can get the actor getting all the actors of that class when the UI starts up and assigning the first one to the variable we have created, like so:



When you create the variable, set it to an Actor type (blue circle).

Adding C++ Actor functions

Now we have a c++ class running in UE4, we can add some functionality to it and call it from the widget blueprint. To do this, we need to create some functions in the Actor class and expose them to blueprints in a similar manner to the class itself:

.h	.cpp

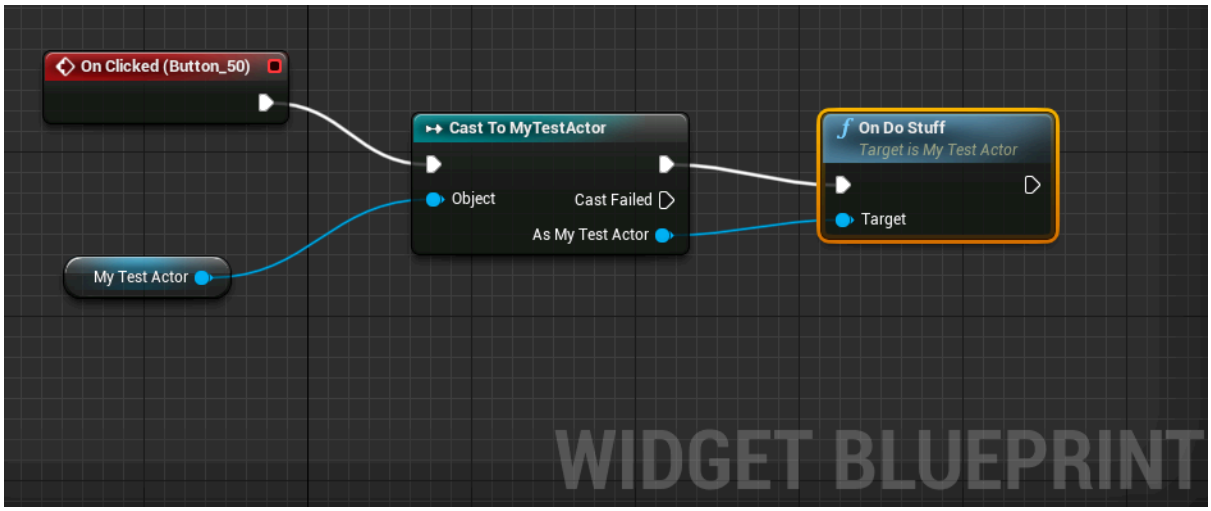
`UFUNCTION(BlueprintCallable, Category="MyTestActor")`
`void OnDoStuff();`
	`void AMyTestActor::OnDoStuff()`
	`{`
	`  GEngine->AddOnScreenDebugMessage(-1,100.0f, FColor::Red, TEXT("MyTestActor::Hello"));`
	`}`

Unlike C# and other languages, C++ requires header (.h) and implementation (.cpp) versions of classes. Putting implementation details in a header will eventually cause circular dependences to stop the compilation form succeeding.

Once this class compiles, you can go into the widget blueprint and use the functions. The Category string in the header file allows you to group exposed methods.

https://wiki.unrealengine.com/Blueprints,_Creating_C%2B%2B_Functions_as_new_Blueprint_Nodes

In the Widget blueprint, turn off 'context sensitivity' to highlight either your function or class, then hook up the OnDoStuff function to the OnClicked event for the button, like so.



When you run the application, it will call the debug print function from the class.

Adding HTTP functionality

To add HTTP functionality, Epic has provided a short article (https://wiki.unrealengine.com/Http-requests) that outlines how to extend your applications.

Part I – Add HTTP and JSON functionality to the project

This is done through the project's build.cs settings to add http, json and json utility components:

```
using UnrealBuildTool;

public class test00 : ModuleRules
{
    public test00(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject",
            "Engine", "InputCore" });
        PrivateDependencyModuleNames.AddRange(new string[] { "Http", "Json",
            "JsonUtilities" });
```

Part II – Create an Actor class

This uses the boilerplate code from the UE example. Like the C# example before, C++ requires us to make explicit structures that will hold JSON data:

```
USTRUCT()
struct FRequest_ScoreDetails {
    GENERATED_BODY()
    UPROPERTY() FString name;
    UPROPERTY() int score;

    FRequest_ScoreDetails() {}
};

USTRUCT()
struct FResponse_ScoreDetails {
    GENERATED_BODY()
    UPROPERTY() TArray< FRequest_ScoreDetails> details;

    FResponse_ScoreDetails() {}
};
```

Much of the UE4 example class is boilerplate and needs to be kept, if not well understood. From our Python server example, the HTTPClient class needs to POST new high scores to the server and GET the highscore table. In UE4, this uses a framework of calling GetRequest() or PostRequest() and passing a function that is called on request completion.

In the case of adding a score to the server:

```
void AMyHTTPRequest::OnAddScoreButtonPress()
{
    GEngine->AddOnScreenDebugMessage(-1, 100.0f, FColor::Blue, TEXT("OnAddScoreButtonPress"));

    FString ContentJsonString;
    FRequest_ScoreDetails score;
    score.name = "aaa";
    score.score = 100;

    GetJsonStringFromStruct<FRequest_ScoreDetails>(score, ContentJsonString);

    TSharedRef<IHttpRequest> Request = PostRequest("add_score", ContentJsonString);
    Request->OnProcessRequestComplete().BindUObject(this, &AMyHTTPRequest::PostScoreDetails_Response);
    Send(Request);
}

void AMyHTTPRequest::PostScoreDetails_Response(FHttpRequestPtr Request, FHttpResponsePtr Response, bool bWasSuccessful)
{
    if (!ResponseIsValid(Response, bWasSuccessful))
    {
        GEngine->AddOnScreenDebugMessage(-1, 100.0f, FColor::Red, TEXT("PostScoreDetails_Response-FAIL"));

        return;
    }

    GEngine->AddOnScreenDebugMessage(-1, 100.0f, FColor::White, TEXT("PostScoreDetails_Response-SUCCESS"));

}
```

And in the case of getting scores from the server

```
void AMyHTTPRequest::OnGetScoresButtonPress()
{
    GEngine->AddOnScreenDebugMessage(-1, 100.0f, FColor::Green, TEXT("OnGetScoresButtonPress"));

    TSharedRef<IHttpRequest> Request = GetRequest("get_scores");
    Request->OnProcessRequestComplete().BindUObject(this, &AMyHTTPRequest::GetScores_Response);
    Send(Request);
}

void AMyHTTPRequest::GetScores_Response(FHttpRequestPtr Request, FHttpResponsePtr Response, bool bWasSuccessful)
{
    if (!ResponseIsValid(Response, bWasSuccessful))
    {
        GEngine->AddOnScreenDebugMessage(-1, 100.0f, FColor::Red, TEXT("GetScores_Response-FAIL"));

        return;
    }

    GEngine->AddOnScreenDebugMessage(-1, 100.0f, FColor::White, TEXT("GetScores_Response-SUCCESS"));

    FString rawJson = Response->GetContentAsString();

    GEngine->AddOnScreenDebugMessage(-1, 100.0f, FColor::White, rawJson);

    FResponse_ScoreDetails result;
    FJsonObjectConverter::JsonObjectStringToUStruct<FResponse_ScoreDetails >(Response->GetContentAsString(), &result, 0, 0);

    for (int32 Index = 0; Index != result.details.Num(); ++Index)
    {
        FString JoinedStr;
        JoinedStr += FString::FromInt(Index);
        JoinedStr += TEXT(" ");
        JoinedStr += result.details[Index].name;
        JoinedStr += TEXT(" ");
        JoinedStr += FString::FromInt(result.details[Index].score);

        GEngine->AddOnScreenDebugMessage(-1, 100.0f, FColor::Yellow, JoinedStr);
    }

}
```

For the testbed, these are called from UI blueprints.