Walkthrough 4: Unity Web-Requests (POST and GET)

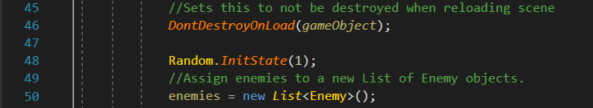
This walkthrough will guide the students through creating web requests through unity. In doing so, students will observe the requirements and properties of http requests used for client to server communication.

Topics covered:

* **GET HTTP Requests**
* **Headers**
* **Body**
* **POST HTTP Requests**
* **JSON Objects & Serialization**
* **Client Side Network Architecture ( modularity of code)**

**Part 1: Game Setup**

1. Open up Unity.
2. Download the Unity Rogue like game zipped on blackboard and unzip to a local directory to work
3. Load the project up and open the **Complete-Game** folder and open the scene located there.
4. Play a round or so of the game to review what we are working with.
5. Open the gamemanager.cs file inside the /Complete-Game/Scripts/ folder of the project and initialize the randoms seed so that everyone is playing the same game.
   * On line 47 add the following **Random.InitState()** code:



1. At the top of the file add a new private string variable named initials and assign it your initials as a string.



1. Find the GameOver function around line 138 and change the game over string to the following:



1. Run and advance to the second level. find a monster, let him kill you, ensure your initials are now displayed in the endgame text.

**Part 2: Sending Results to server**

for this we are going to use UnityWebRequests <https://docs.unity3d.com/2019.1/Documentation/ScriptReference/Networking.UnityWebRequest.Get.html>

*Inside your scripts folder is a Singleton class named NetworkManager.cs.  
This is a basic singleton that we can call to use from the GameManager code.*

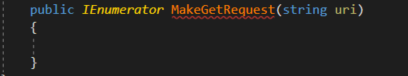
1. The first thing we need to do for our empty singleton is ensure we are using the proper library. At the top of NetworkManager.cs add the following using:



*Our Network Manager is going to need to be able to make both GET and POST HTTP requests.*

*We will begin by implementing a GET request.*

1. To do this, we need to create an **IEnumerator** type function which can execute **asynchronously**. We pass in a variable to the function for the request to call.



1. Inside our MakeGetRequest function, we will actually create the **HTTP** **GET** request.



Here we create an object that will perform our HTTP request for us. The only thing the constructor requires is the URI to communicate with.

1. Next we will send off the GET request through the following function call:

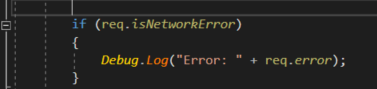


At this point our IEnumerator function sits and waits until it receives a response from sending the get request.

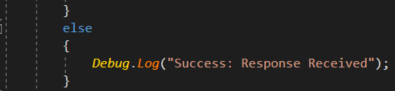
1. We can determine the response code of our request after sending using:



1. Next, when it returns with a response from our yield return, we need to handle the response. We can use **req.isNetworkError** to determine if it had an error connecting to the server or not and **req.error** to find out the error.

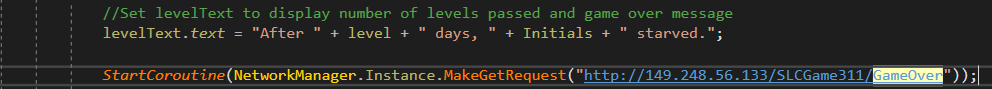


1. For now, we will add an else to this if statement for testing purposes:



1. Let's make a call to the server and see if we can make a successful API call.

inside GameManager.cs, right after the text we modified on line 143, make a call to the following:



1. Run the game, die and check the console output.

* Q: Was it successful?
* Q: what was output in the console?

1. Let's change the API request string to a different URI:



1. Run the game, die and check the console output.

* Q: Was it successful?
* Q: What was the status code?
* Q: What does that indicate?

1. This request has expected header information. It expects a userID passed with the request. To add a header to a unityWebRequest use **req.SetRequestHeader(key, value);.**

Add the following line to your MakeGetRequest function **before the send occurs**:



1. Run the game, die and check the console output.

* Q: Was it successful?

**Part 3: Processing Response Data / JSON deserialization to Object**

*Assuming we did not have an error, we need to actually process a response from the server.*

To be able to accomplish this, we need to add an object to our **webrequest** known as a **DownloadHandlerBuffer**. This object is used to track and process response data.

1. create the handler and assign it to our request (Again, before sending out the request)With this we can now check the response data properly.
2. Below your success debug print inside **MakeGetRequest**, add in a new Debug.Log for:

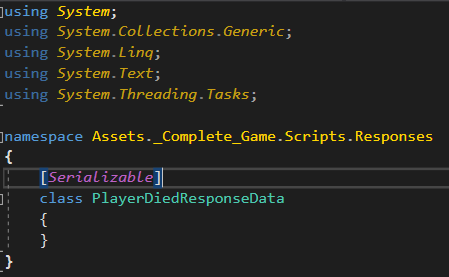


* we should now be able to see the response data from the call.

1. Run the game and lose.
   * Q: What format is the response data in?

We will need to parse this data to be able to use it. To do this we need an object which holds the response data.

1. Inside of the Unity editor, create a New Folder inside the scripts directory called Requests and a second folder called Responses.
2. Inside the Responses folder, in Visual Studio, right click and select Add->NewItem and create a new class named **PlayerDiedResponseData.cs**



1. Add the [Serializable] tag directly above your class definition.
2. Inside the class add a string variable named UserID.

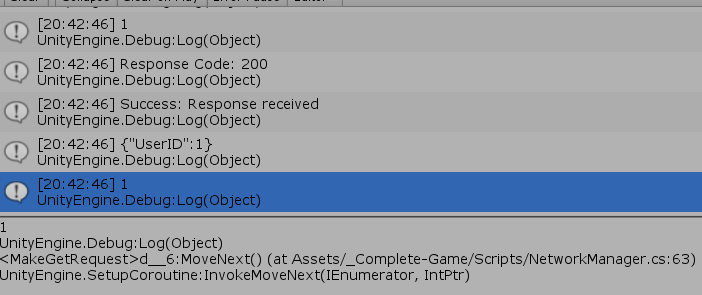
This is case sensitive for the variable name and MUST match the response from the server

1. In the NetworkManager.cs include access to the new file in the top of your file using 
2. In the MakeGetRequest, when we print out the valid response (the else), we need to convert our text object to our actual response object.



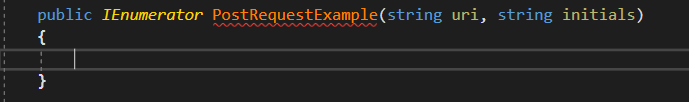
1. We can now print out the userID using a debug log.





**Part 4: Basic Post call**

1. Open up networkManager.cs and create a new function called PostRequestExample() that takes in two strings as parameters (Uri and intials).



1. We will be creating a base request identically to how we created our get requests. Start by adding the following inside our function:



A web requests type is by default a GET request, we didn’t need to set anything to ensure that. With POST we need to change the requests settings.

1. To do this, we need to set the method of the request:



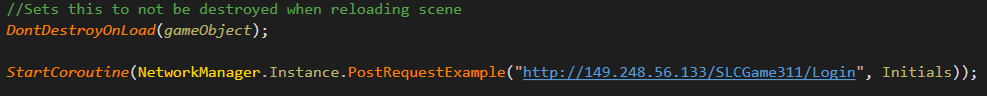
1. We can now send out our request, identically to how we did with our get requests:



1. Last Step is we will print out the response code so we know the state of our call:



1. We are now ready to make our first attempt at calling this POST request. Inside our gameManager.cs Awake() function, at the same time where we initialize our random seed, we will call to login to the server:
   * We will pass into the function /Login as the API to contact and Initials

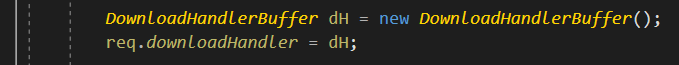


1. Run the game and immediately you should see a response code output to the console immediately
2. What was the response code?
3. What does this indicate?

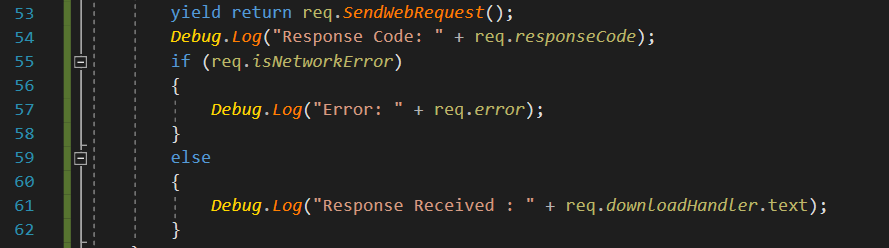
**Part 5: Receiving Response**

Back inside our NetworkManager.cs we need to determine why we got a 400 bad request. To start with let’s get more information from the response of the request.

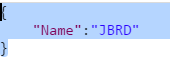
1. Similar to our GET request, we need to add a downloadHandler to our request object to get information about the response.



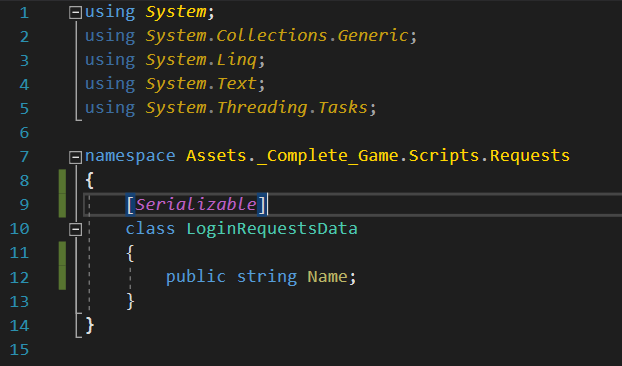
1. Add the following if statement to the bottom of our post request function:



1. Run the game and immediately you should see output in the console again.   
   What does our new response output tell us?
2. This is a post request, and it is expecting a JSON object be passed as the body of the request. The JSON it’s looking for is formatted like so:



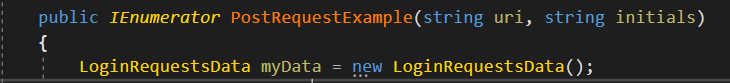
1. Similarly to when we created an object to handle the JSON response from the GET request, we will create a Request object. Inside your previously created Requests folder in visual studio right click and select **Add Class..** name the new class **LoginRequestsData.cs.**
2. Add a public string Name to your new class.
3. Add the serializable tag above your class name



1. We are now ready to use our object. Back inside the NetworkManager.cs add the namespace at the top:



1. Scroll down to your Post function and create a new LoginRequestData object at the very first line of the function:



1. Immediately after this, assign our initials passed into the function to myData.Name.



1. The last part we need to do is convert our object into JSON and attach it to the body of our request. To convert our object to JSON, we will use the following:



1. Our JSON data still isn’t ready yet, we need to convert our string to an array of byte data. To do this we will need to include the following:



1. we can now have access to our string to byte conversion utilities: 

We are now ready to attach our data as the request body. To do this, we need an **UploadHandler** added to our request, similar to the DownloadHandler to process response data.

1. We can now create our **UploadHandlerRaw** object and pass in the bytes of our json data into it’s constructor.
   * The UploadHandlerRaw requires raw byte data be passed and does not have a default contstructor.
   * Assign our upload handler to our request:



1. finally we have to add a header to let the server know that we are transmitting JSON data.



1. Run the game and you should receive a response, what was it?
2. Run the game again, was there a difference in the response the second time?