**Canvas Desktop App Documentation**

1. **Logic behind the Rest Calls**

The process of making Rest Calls required just a few steps. First, I had to obtain an API access token from Canvas; I was able to do this from my User Profile Page, which gave me a token that would allow me to make API calls on the behalf of my Canvas account.

The only other thing I needed for this process was the set of URLs at which the API endpoint functions could be accessed. These were easily acquired from the Canvas API Documentation website.

After I had these two items, I was able to access Canvas web resources programmatically. To do this, I had to send an HTTP request to the URL of the API function I wished to call, along with the access token which I had generated using my Canvas account. If the access token was valid, and my account had sufficient permission to access the resources I was requesting access to, then the HTTP request I had sent would return a 200 OK status code, along with the specified resource in the form of a JSON string. I could then parse this string programmatically, turning it into a well-organized hierarchical set of information which I could then utilize for my various purposes.

**II. How the Rest Calls were implemented in C#**

The C# function created to perform GET requests to the Canvas Rest API is shown below:

private async Task<HttpResponseMessage> http\_get(string base\_url, string access\_token, string url\_command)

{

HttpResponseMessage response = null;

HttpClientHandler handler = new HttpClientHandler();

handler.AllowAutoRedirect = false;

using (HttpClient client = new HttpClient(handler, true))

{

client.BaseAddress = new Uri(base\_url);

client.DefaultRequestHeaders.Accept.Clear();

client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", access\_token);

response = await client.GetAsync(url\_command);

}

Console.WriteLine("Response from http\_get: " + response);

return response;

}

This function takes three parameters: the top- and second-level domains of the web service, the account’s access token, and the URL-command, which is appended to the base URL in the HTTP request in order to get the final location of the specific endpoint function. In the body of the function, we first declare an empty variable of type HttpResponseMessage, which will be used to hold the HTTP response we get from the API call.

The function then creates a new HttpClientHandler object, and disables that object’s AllowAutoRedirect property, not allowing it to follow redirection responses. Then, in a using statement to facilitate appropriate cleanup, an HttpClient is created using the new Client Handler. Some operations are then performed on this Client, including setting the BaseAddress to our base URL, clearing the Accept Header, and setting the Authorization Header to include the Access Token set in the parameter list.

Finally, the GetAsync function is called on the HTTP Client, using the specified endpoint URL command, which performs the HTTP Get request and assigns the resulting response to the variable that was created at the beginning of the function. This variable is then returned, and can then be read as a string later in the program using the following code:

string http\_string = await response.Content.ReadAsStringAsync();

This function reads the HTTP response content, and returns its value as a string in JSON format. This string can then be parsed into a usable JSON object using the following JSON conversion utility:

dynamic json = JsonConvert.DeserializeObject(return\_val);

After which specific pieces of the JSON data can be obtained with the complementary function:

string json\_part = JsonConvert.SerializeObject(json[“section”][“subsection”]);

This returns a string from the data specified at subindex “subsection” of index “section” and stores it in the variable json\_part, which can then be used for various purposes.