**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.

**III. Acquisition and Display of Canvas Course/Course Page Data**

The task of acquiring course and page data for display was accomplished through REST calls (see sections I and II).

The first step was to get the courses that the user teaches through the following API endpoint:

<https://centralia.instructure.com/api/v1/courses?enrollment_type=teacher>

This endpoint returns the courses that have the current user marked as “teacher”.

To make this collection in C#, the following method was used:

public static async Task<dynamic> get\_teacher\_courses(string base\_url, string token)

{

return await HttpRequest.get\_get\_response(base\_url, token, "courses?enrollment\_type=teacher");

}

This piece of code takes the base url for the Canvas database and the user’s access token (in this case I used my own) as parameters, and passes them to the GET request method that is defined in my HttpRequest class. The appropriate URL command is also included, and the courses for the instructor are returned as a serializable JSON object. The courses are then each appended as a base node to a TreeView, using FOR loop iteration:

TreeNode t = main\_tree.Nodes.Add(JsonConvert.SerializeObject(my\_courses[i][“name"]).Trim('"'));

The “Trim” method is necessary to eliminate the surrounding quotes from the JSON string subsection.

The next step was to get the pages for each course, if there are any. This was done using the following method:

public static async Task<dynamic> get\_course\_pages(string base\_url, string token, string course\_id)

{

return await HttpRequest.get\_get\_response(base\_url, token, "courses/" + course\_id + "/pages");

}

Like the get\_teacher\_courses() method, this code takes the base URL and the access token along with the query string that is required to get pages for a course, and sends them over to my GET request method. The main difference is that this method sends a variable query string, which is determined by a string parameter which represents the target course ID.

This method is called using the following line of code to retrieve all of the pages from each course:

dynamic pages = await HttpUtilities.get\_course\_pages(BASE\_URL, TEST\_TOKEN,

JsonConvert.SerializeObject(my\_courses[i][“id"]));

The ID of each course is passed as a parameter to the method using a FOR loop, which in this case uses an integer variable “i” as a counter, and a new JSON object is returned which contains all of the pages for the course.

The pages are then added as subnodes to the respective TreeNode t, also using FOR loop iteration:

t.Nodes.Add(JsonConvert.SerializeObject(pages[j][“title"]).Trim('"'));

At this point, the user can click on page nodes in the TreeView and have the text displayed. However, in order to have access to the body text of the page the user wishes to see, we must retrieve the individual page rather than all of the pages at once, using the following method:

public static async Task<dynamic> get\_single\_page(string base\_url, string token, string course\_id, string page\_url)

{

return await HttpRequest.get\_get\_response(base\_url, token, "courses/" + course\_id + "/pages/" + page\_url);

}

This method is almost identical to the get\_course\_pages() method, with the exception of the page\_url parameter, which is used to extract a single page and can be obtained from the multi-page JSON object.

Once the user selects a page node, the correct individual page is retrieved, using a nested FOR loop structure to identify the course and the page:

dynamic page = await HttpUtilities.get\_single\_page(BASE\_URL, TEST\_TOKEN,

JsonConvert.SerializeObject(my\_courses[i]["id"]).Trim('"'),

JsonConvert.SerializeObject(pages[j][“url"]).Trim('"'));

This returns a single page object using the appropriate course ID and page URL.

The content of the page can then be extracted, removing the quotes:

string page\_content = JsonConvert.SerializeObject(page[“body"]).Trim('"');

Finally, we insert the content string into the HtmlEditControl, which is named “new\_hec” in this case:

new\_hec.DocumentHTML = page\_content;

The page content can now be viewed by the user.

**IV. Modification of Canvas Course Page Data**

The following method is used to update the contents of a page:

public static async Task<dynamic> update\_page(string base\_url, string token, string course\_id, string page\_url, string page\_body)

{

return await HttpRequest.get\_put\_response(base\_url, token, "courses/" + course\_id + "/pages/" + page\_url +

"?wiki\_page[body]=" + page\_body);

}

This sends a PUT request to a specific page at a specific course, along with the new content for the page appended to the query string. When the user chooses to “Save”, a nested FOR loop structure is used to find the appropriate course and page, and the above method is called with the required parameters:

dynamic update = HttpUtilities.update\_page(BASE\_URL, TEST\_TOKEN,

JsonConvert.SerializeObject(my\_courses[i]["id"]).Trim('"'),

JsonConvert.SerializeObject(pages[j]["url"]).Trim('"'),

hec.DocumentHTML);

This updates the page the user is editing to match the content of the HtmlEditControl “hec”.

**V. Addition of New Canvas Course Pages**

To create new course pages, the following method is used:

public static async Task<dynamic> create\_page(string base\_url, string token, string course\_id, string page\_name)

{

return await HttpRequest.get\_post\_response(base\_url, token, "courses/" + course\_id + "/pages" +

"?wiki\_page[title]=" + page\_name + "&wiki\_page[published]=true");

}

This method sends the API base URL and user access token, along with the ID of the course to create the page in, and the name of the new page appended to the query string, to the POST method that I have defined in my HttpRequest class. What is returned is merely a response indicating if the POST request was successful.

The method is then called when the user opts to create a new page:

for (int i = 0; i < my\_courses.Count; ++i)

{

if (JsonConvert.SerializeObject(my\_courses[i]["name"]).Trim('"') == current\_class.Text)

{

dynamic new\_page = await HttpUtilities.create\_page(BASE\_URL, TEST\_TOKEN,

JsonConvert.SerializeObject(my\_courses[i]["id"]).Trim('"'), pnf.FileName());

}

}

A FOR loop is used to find the course node that the user has selected (in this case, the current course is represented by “current\_class”). The create\_page() method is then called, passing the appropriate course ID and the name of the new page as parameters.

Here, “pnf” refers to an instance of a second form, called PageNameForm, that is used to collect user input for the desired name of the new page. The second form is simple, and features a textbox for the user to input the new file’s name along with “finish” and “cancel” buttons. When the user presses “finish”, if there is text in the textbox, this text is assigned to the form’s instance variable “file\_name”, which is returned via the PageNameForm.FileName() method and then passed into the create\_page() method in this case.