API Clients in C#

One List API

HttpClient to access the API

Creating our one list API client.

We will generate a console application with:

```
dotnet new sdg-console -o OneListClient
```

Create an instance of `HttpClient`

```
var client = new HttpClient();
```

The `HttpClient` is the built-in class we will use to send and receive information to APIs over HTTP.

The `HttpClient` object we create, in the variable named `client`, will have many methods for both sending and receiving data.

Basic GET

The first we will review is a method that makes a `GET` request to a server and returns the `response body` as a `string`.

GetStringAsync

```
var responseAsString = client.GetStringAsync(
  "https://one-list-api.herokuapp.com/items?access_token=sdg-handbook"
);
```

What is Async?

Synchronous

vs

Not-synchronous

await

Add the keyword `await` before the `GetStringAsync`

Any time we ask a method, in this case `Main` to use `await` we need to make the method itself `async` or to have it return a new kind of object called a `Task`.

If you place your cursor on the error and press `Control .` (Windows) or `Command .` (Mac) you will see the *Quick Fix* suggestion `Make method async`. Doing so turns the code into:

```
static async System.Threading.Tasks.Task Main(string[] args)
{
    var client = new HttpClient();

    var responseAsString = await client.GetStringAsync(
      "https://one-list-api.herokuapp.com/items?access_token=sdg-handbook"
    );
```

One last refactor is to take the long `System.Threading.Tasks.Task` and make it just `Task` and then add a `using System.Threading.Tasks;` to our code.

Run the code


```
[
  {
    "id": 1590,
    "user_id": 143,
    "text": "Write some documentation about Insomnia",
    "complete": false,
    "created_at": "2020-04-24T19:32:43.653Z",
    "updated_at": "2020-04-24T19:32:43.653Z"
  },
```

CONGRATULATIONS

You have just written your first API client that accesses data remotely over the Internet!

Processing the data

Notice that this is just one long string of data.

Notice that the JSON data represent an array.

It would be convenient if we could *convert* this JSON string into some objects that `C#` knows how to deal with nicely.

This introduces the idea of **`serialization`**

Serialization / Deserialization

`Serial` - meaning one at a time.

`Deserialization`	Turn a string of characters into more structured data.
`Serialization`	Turn structured data into a string of characters.

Processing the data by defining a class to store the results

The data returned from our `GET` is an array of items.

The objects in that array follow this format:

```
id: int
text: string
complete: bool
created_at: string
updated_at: string
```

Define a C# class to represent this JSON

```
class Item
{
    public int id { get; set; }
    public string text { get; set; }
    public bool complete { get; set; }
    public string created_at { get; set; }
    public string updated_at { get; set; }
}
```

Mapping

Notice that we define the *properties* of our class to have **the same names as the keys of our JSON object**.

This is a **CRITICAL** point because the *deserializer* will use this pattern to know where to put the data.

This process is very similar to the ORM mapping we use with Entity Framework.

Example


```
{
  "id": 6,
  "text": "Finish Assignment",
  "complete": true,
  "created_at": "2016-08-17T20:06:34.874Z",
  "updated_at": "2018-10-02T16:10:59.754Z"
}
```

The deserializer will make a new *instance* of `Item` for us and copy the 6 from the `"id"` JSON key into the `id` property in our object.

Then it will copy the value `"Finish Assignment"` from the `"text"` key in the JSON into the `text` property of our object, and so on.

If we follow this *convention* (pattern) the deserializer will do a lot of work on our behalf without having to write individual statements to tell it how to work.

Using the *deserializer*

In order to use the *deserializer* to convert our response into a `List` of `Item` objects we need to make a modification.

The first is to not retrieve the `GET` result as a `string` but as a `stream`.

A `stream` is a variable that knows how to read data one chunk at a time (often one character at a time).

By using a `stream` the *deserializer* can process data a little at a time in case the input is quite large.

Modify the code

```
var responseAsStream = await client.GetStreamAsync(
  "https://one-list-api.herokuapp.com/items?access_token=sdg-handbook"
);
```

And then we need to supply this `stream` to the *deserializer*

```
var items =
 await JsonSerializer.DeserializeAsync<List<Item>>(
   responseAsStream
);
```

Walk the code

```
//
// Wait for the async
// |
// |                        Use the deserializer
// |                        |
// |                        |                    Deserialize a list of items
// |                        |                    |
// |                        |                    |
// v                        v                    v
   await JsonSerializer.DeserializeAsync<List<Item>>(
     responseAsStream
   );
```

```
// For each item in our deserialized List of Item
foreach (var item in items)
{
    // Output some details on that item
    Console.WriteLine($"The task {item.text}" +
    $" was created on {item.created_at} and " +
    $" has a completion of: {item.complete}");
}
```

Improve

Date is formatted as a long string that is not very user friendly.

We can improve this by changing the data type from `string` to `DateTime` in our `Item` class.

Thus when the *deserializer* is processing those fields it will try to *convert* (the technical term is *coerce*) the string format into a `DateTime`.

Luckily for us that string is in a very specific format called an ISO8601 format and `DateTime` knows how to deal with it.

Complete Status

The `False` is not friendly so let's improve that. We can add another custom property which contains logic for its get implementation:

```
public string CompletedStatus
{
    get
    {
        return complete ? "completed" : "not completed";
    }
}
```

Use C# style property names

```
class Item
{
    [JsonPropertyName("id")]
    public int Id { get; set; }

    [JsonPropertyName("text")]
    public string Text { get; set; }

    [JsonPropertyName("complete")]
    public bool Complete { get; set; }

    [JsonPropertyName("created_at")]
    public DateTime CreatedAt { get; set; }

    [JsonPropertyName("updated_at")]
    public DateTime UpdatedAt { get; set; }
```

Make the output pretty (*OPTIONAL*)

Our output has been boring. Here is one thing we could add to make it more exciting.

```
-------------------------------------------------
| one                                | two  | three |
-------------------------------------------------
| Console Tables                     | 2    | 3     |
-------------------------------------------------
| Can output our data in tables      | Neat | oh    |
-------------------------------------------------
```

ConsoleTables

```
dotnet add package ConsoleTables
```

The documentation for ConsoleTables shows us how to create a table with headers, add rows, and output the table itself.

```
var table = new ConsoleTable(
  "Description",
  "Created At",
  "Completed");

// For each item in our deserialized List of Item
foreach (var item in items)
{
    // Add one row to our table
    table.AddRow(
      item.Text,
      item.CreatedAt,
      item.CompletedStatus);
}

// Write the table
table.Write();
```

Getting the name of the list from the command line.

We've been ignoring a variable that has been present in all of our `C#` applications so far, `args`.

The `string[] args` parameter to `Main` are the command line arguments that appear after our `dotnet run` command.

So we can use this to get the name of the list we want to process.

Getting name of list

Since this is an array we can access the 0th (first) element: `var token=args[0]` and then use string interpolation to generate our URL:

```
var url = $"https://one-list-api.herokuapp.com/items?access_token={token}";
```

What if we don't supply a token?

```
var token = "";

if (args.Length == 0)
{
    Console.Write("What list would you like? ");
    token = Console.ReadLine();
}
else
{
    token = args[0];
}
```

Add a menu

Show all

```
var keepGoing = true;
while (keepGoing)
{
    Console.Clear();
    Console.Write("Get (A)ll todo, or (Q)uit: ");
    var choice = Console.ReadLine().ToUpper();

    switch (choice)
    {
        case "Q":
            keepGoing = false;
            break;

        case "A":
            await ShowAllItems(token);

            Console.WriteLine("Press ENTER to continue");
            Console.ReadLine();
            break;

        default:
            break;
    }
}
```

Fetching a specific todo item

```
case "O":
    Console.Write("Enter the ID of the item to show: ");
    var id = int.Parse(Console.ReadLine());

    await GetOneItem(token, id);

    Console.WriteLine("Press ENTER to continue");
    Console.ReadLine();
    break;
```

Implement GetOneItem

```
static async Task GetOneItem(string token, int id)
{
    var client = new HttpClient();

    // Generate a URL specifically referencing the endpoint for getting a single
    // todo item and provide the id we were supplied
    var url = $"https://one-list-api.herokuapp.com/items/{id}?access_token={token}";

    var responseAsStream = await client.GetStreamAsync(url);

    // Supply that *stream of data* to a Deserialize that will interpret it as a *SINGLE* `Item`
    var item = await JsonSerializer.DeserializeAsync<Item>(responseAsStream);

    var table = new ConsoleTable("ID", "Description", "Created At", "Updated At", "Completed");

    // Add one row to our table
    table.AddRow(item.Id, item.Text, item.CreatedAt, item.UpdatedAt, item.CompletedStatus);

    // Write the table
    table.Write(Format.Minimal);
}
```

What if the user enters an item that doesn't exist?

Wrap code in a `try / catch` to capture exception

```
try
{
  // Code that might THROW an EXCEPTION
}
catch(KindOfException)
{
  // Code to handle if an exception happened
}
```

Handling an item that doesn't exist

```
static async Task GetOneItem(string token, int id)
{
    try
    {
        var client = new HttpClient();

        // Generate a URL specifically referencing the endpoint for getting a single
        // todo item and provide the id we were supplied
        var url = $"https://one-list-api.herokuapp.com/items/{id}?access_token={token}";

        var responseAsStream = await client.GetStreamAsync(url);

        // Supply that *stream of data* to a Deserialize that will interpret it as a *SINGLE* `Item`
        var item = await JsonSerializer.DeserializeAsync<Item>(responseAsStream);

        var table = new ConsoleTable("ID", "Description", "Created At", "Updated At", "Completed");

        // Add one row to our table
        table.AddRow(item.Id, item.Text, item.CreatedAt, item.UpdatedAt, item.CompletedStatus);

        // Write the table
        table.Write(Format.Minimal);
    }
    catch (HttpRequestException)
    {
        Console.WriteLine("I could not find that item!");
    }
}
```

Creating a new element

```
case "C":
    Console.Write("Enter the description of your new todo: ");
    var text = Console.ReadLine();

    var newItem = new Item
    {
        Text = text
    };

    await AddOneItem(token, newItem);

    Console.WriteLine("Press ENTER to continue");
    Console.ReadLine();
    break;
```

Implement the `AddOneItem` method.

```
static async Task AddOneItem(string token, Item newItem)
{
    var client = new HttpClient();

    // Generate a URL specifically referencing the endpoint for adding a todo item
    var url = $"https://one-list-api.herokuapp.com/items?access_token={token}";

    // Take the `newItem` and serialize it into JSON
    var jsonBody = JsonSerializer.Serialize(newItem);

    // We turn this into a StringContent object and indicate we are using JSON
    // by ensuring there is a media type header of `application/json`
    var jsonBodyAsContent = new StringContent(jsonBody);
    jsonBodyAsContent.Headers.ContentType = new MediaTypeHeaderValue("application/json");

    // Send the POST request to the URL and supply the JSON body
    var response = await client.PostAsync(url, jsonBodyAsContent);

    // Get the response as a stream.
    var responseJson = await response.Content.ReadAsStreamAsync();

    // Supply that *stream of data* to a Deserialize that will interpret it as a *SINGLE* `Item`
    var item = await JsonSerializer.DeserializeAsync<Item>(responseJson);

    // Make a table to output our new item.
    var table = new ConsoleTable("ID", "Description", "Created At", "Updated At", "Completed");

    // Add one row to our table
    table.AddRow(item.Id, item.Text, item.CreatedAt, item.UpdatedAt, item.CompletedStatus);

    // Write the table
    table.Write(Format.Minimal);
}
```

Updating an item

```
case "U":
    Console.Write("Enter the ID of the item to update: ");
    var existingId = int.Parse(Console.ReadLine());

    Console.Write("Enter the new description: ");
    var newText = Console.ReadLine();

    Console.Write("Enter yes or no to indicate if the item is complete: ");
    var newComplete = Console.ReadLine().ToLower() == "yes";

    var updatedItem = new Item
    {
        Text = newText,
        Complete = newComplete
    };

    await UpdateOneItem(token, existingId, updatedItem);

    Console.WriteLine("Press ENTER to continue");
    Console.ReadLine();
    break;
```

UpdateOneItem

```
static async Task UpdateOneItem(string token, int id, Item updatedItem)
{
    var client = new HttpClient();

    // Generate a URL specifically referencing the endpoint for getting a single
    // todo item and provide the id we were supplied
    var url = $"https://one-list-api.herokuapp.com/items/{id}?access_token={token}";

    // Take the `newItem` and serialize it into JSON
    var jsonBody = JsonSerializer.Serialize(updatedItem);

    // We turn this into a StringContent object and indicate we are using JSON
    // by ensuring there is a media type header of `application/json`
    var jsonBodyAsContent = new StringContent(jsonBody);
    jsonBodyAsContent.Headers.ContentType = new MediaTypeHeaderValue("application/json");

    // Send the POST request to the URL and supply the JSON body
    var response = await client.PutAsync(url, jsonBodyAsContent);

    // Get the response as a stream.
    var responseJson = await response.Content.ReadAsStreamAsync();

    // Supply that *stream of data* to a Deserialize that will interpret it as a *SINGLE* `Item`
    var item = await JsonSerializer.DeserializeAsync<Item>(responseJson);

    // Make a table to output our new item.
    var table = new ConsoleTable("ID", "Description", "Created At", "Updated At", "Completed");

    // Add one row to our table
    table.AddRow(item.Id, item.Text, item.CreatedAt, item.UpdatedAt, item.CompletedStatus);

    // Write the table
    table.Write(Format.Minimal);
}
```

Delete an item

```
case "D":
    Console.Write("Enter the ID of the item to delete: ");
    var idToDelete = int.Parse(Console.ReadLine());

    await DeleteOneItem(token, idToDelete);

    Console.WriteLine("Press ENTER to continue");
    Console.ReadLine();
    break;
```

Implement DeleteOneItem

```
static async Task DeleteOneItem(string token, int id)
{
    try
    {
        var client = new HttpClient();

        // Generate a URL specifically referencing the endpoint for getting a single
        // todo item and provide the id we were supplied
        var url = $"https://one-list-api.herokuapp.com/items/{id}?access_token={token}";

        await client.DeleteAsync(url);
    }
    catch (HttpRequestException)
    {
        Console.WriteLine("I could not find that item!");
    }
}
```

Complete TODO client app!