Dist. Sys. ASP.NET Core Web API

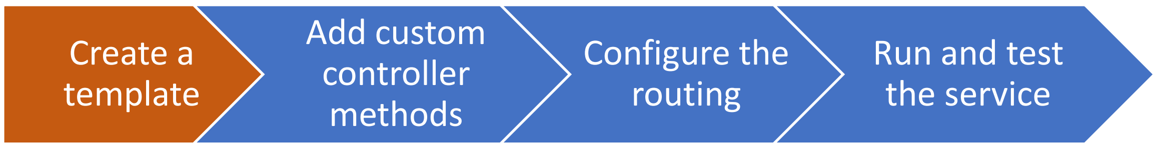
In this lab you'll be looking at ASP.NET Web API, which is very much related to your ACW!

If you have prior experience in Web API, use this lab time to refresh your knowledge and experiment with combining Entity Framework (see previous labs) and Web API.

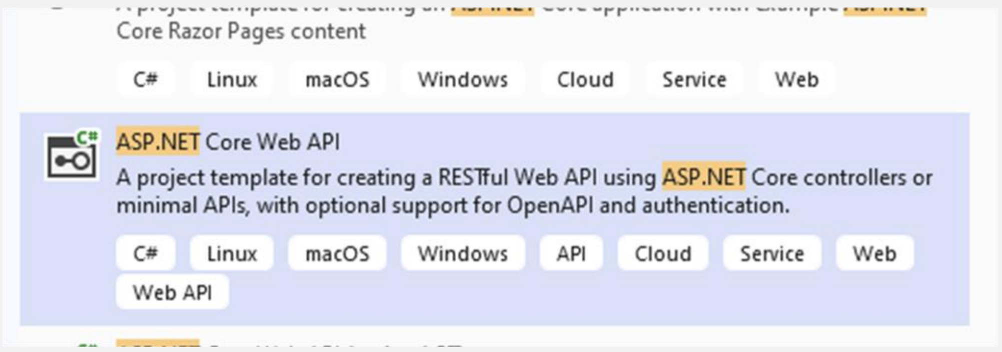
Web API is a RESTful service that accepts HTTP methods like GET, POST, DELETE, etc. and returns a response, based on methods that you write for specific situations.

Before you attempt this lab you should refresh your memory about RESTful web services and Web API by reading through the associated lectures.

We will be following this process to start:



Open Visual Studio and create a new C# ASP.NET Core Web API project.



Core Razor Pages content

C# Linux macOS Windows Cloud Service Web

ASP.NET Core Web API

A project template for creating a RESTful Web API using ASP.NET Core controllers or minimal APIs, with optional support for OpenAPI and authentication.

C# Linux macOS Windows API Cloud Service Web Web API

Name your Project MyFirstAPI and choose a save location which is not on a networked drive or in your OneDrive folder, to ensure it runs.

You can usually find your folder in C:/Users/<ID>, where <ID> is your 6 digit user login. **If you save to C, remember to save your file elsewhere once you finish the lab as the C: drive is not shared across computers and is regularly wiped. Strongly consider using source control!**

Choose .NET 8.0 LTS, and then uncheck both 'Configure for HTTPS' and 'Enable OpenAPI support' (these are useful options to explore, but add extra complexity for now).

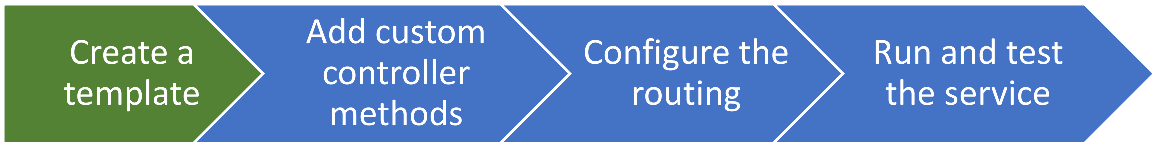
Click 'Create' and Visual Studio will generate you an empty Web API project with all the files, folders and references you need to get started.

Once it has finished generating, open the 'Controllers' folder in Solution Explorer and delete any Controller it has made for you by default. Now right-click the 'Controllers' folder in the Solution Explorer Window and select Add->Controller. In the tree on the left, choose API and then select 'API Controller with Read/Write Actions' and press Add.

Name the Controller 'DataController' and press Add.

Now open the file and take a look at the methods that have been created for you.

So we have our template! Easy. Let's mark than one off as done.



Next come the controller methods. We have already got some in front of us and (hopefully) they already make some sense to you.

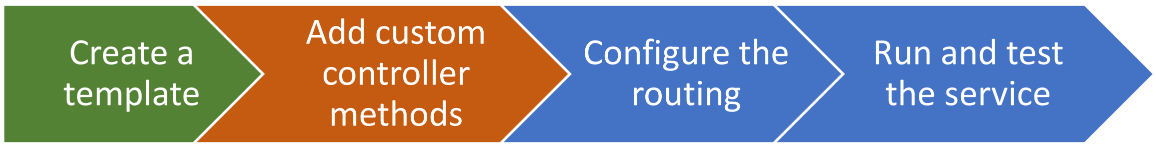
Look at the method names: Get, Post, Put, Delete. These are all HTTP request methods that we've discussed in the lectures. Return to the lecture on WebAPI to understand where each method parameter comes from. Consider the HTTP request packet; it is made from 2 parts, just like the most application protocol envelopes: the HEADER and BODY.

These parameters don't have an explicit attribute identifying where they will come from so the framework is using its default behaviours. Useful but also can lead to unexpected behaviours if the default isn't what you want. The automatically generated comments give us some hints about what is going on here:

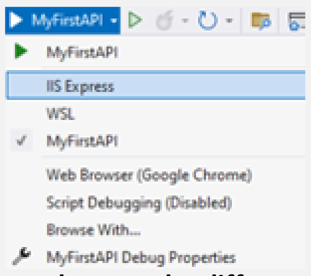
```
// GET api/<DataController>/5
[HttpGet("{id}")]
public string Get(int id)
{
    return "value";
}
```

The comment above this method identifies that this method can be accessed simply through the URI api/Data/5. The value 5 is actually an integer that is mapped from the URI as a parameter. We looked at this in the lectures.

Have a read through the MSDN information on parameter binding to see your options here: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding



Modify the Get method above to calculate id += 100, then return the string "Your number plus 100 is <x>" where <x> is the number you calculated.


MyFirstAPI
MyFirstAPI
IIS Express
WSL
MyFirstAPI
Web Browser (Google Chrome)
Script Debugging (Disabled)
Browse With...
MyFirstAPI Debug Properties

Click the run dropdown and choose IIS Express to start your service. IIS should kick in to host your web service and a browser should open. After a bit of loading it'll normally show you a big error.

Your will probably have taken you to the address: http://localhost:58789/weatherforecast **although the port number may be different for you.**

If it doesn't work, open up Properties->launchSettings.json and find the iisExpress applicationUrl. Paste this into a browser address bar.

Remove `weatherforecast` (if it is there) and add the path `api/Data/5` to the end of the URI in your address bar. It should now look something like this:
http://localhost:58789/api/Data/5

Press enter to make the browser send the request.

Excellent. That should now work.

You may have noticed that there were two GET methods in the controller. Why would we ever want that?

Often, a RESTful GET method allows us to either get a single piece of data or get all of the data. When we specify the data we want (in this case by supplying an id of 5) we only get that single piece of data, otherwise we'll get everything. Remember that REST is just an architectural style so this is by no means a rule. Let's make some changes to our code so we get this type of behaviour.

Add this collection to your DataController class:

```
string[] myData = new string[] { "zero", "one", "two", "three", "four", "five" };
```

Now, modify your `Get(int id)` method to return the value in the index position from the myData array.

Finally, modify the `Get()` method to return the whole myData array.

Start up the application again and navigate (remember your port might be different) to:

http://localhost:58789/api/Data/5 You should now see "five"

navigate to: http://localhost:58789/api/Data You should now see the whole array

Congratulations, you have successfully created and used a RESTful service!

Do you recognise that data format that the array is returned as?

These days it is extremely likely to be JSON, particularly as JSON is the default, but the framework can do the content type negotiation for you if the browser includes a header that says it will only accept XML and you set up the framework to do XML formatting.

Learn more about response formatting here: https://learn.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting

We have done something bad though haven't we? We've strongly coupled our business logic and our data access. Definitely, we should have known better. Let's quickly rectify this…

Add a new folder to your project called DataAccess.
Add a new class to this folder called MyDataCRUD

Remove the myData member from your DataController and create a private string[] property called MyData inside MyDataCRUD.

Now add **C**reate, **R**ead, **U**pdate and **D**elete methods.

For our purposes, the Create method should set MyData to the same string array as before: `new string[] { "zero", "one", "two", "three", "four", "five" };`

There should be two Read methods (one which returns the whole array, and one which returns the string at a given index position. The Update method takes an int index and a string to update the string at the given index and the Delete method replaces the string at a given index with the value `"Deleted"`

Example:

```csharp
public class MyDataCRUD
{
    private string[] MyData { get; set; }

    public void Create()
    {
        MyData = new string[] { "zero", "one", "two", "three", "four", "five" };
    }

    public string[] Read()
    {
        return MyData;
    }
    public string Read(int index)
    {
        return MyData[index];
    }

    public void Update(int index, string data)
    {
        MyData[index] = data;
    }

    public void Delete(int index)
    {
        MyData[index] = "Deleted";
    }
}
```

Our data access class gives us the ability to modify the Create, Read, Update and Delete behaviours whenever we like, without having to modify anything in the controller. Typically, you won't want to store ANY data in the application – use a database instead. Here, we can easily swap out the hard-coded data for database calls and we don't have to modify the controller at all!

This is great for fast prototyping and testing where we don't want to set up a database until later.

First we will need to be able to use this Data Access class. We could create a new object of the type each and every time we get a request but that would be a bit wasteful… and, of course, it would mean that any changes to our data via Create, Update or Delete would not persist between requests.

We're going to use the singleton pattern that we explored in the lectures. This will allow us to always get the same data object returned, no matter whether it has been created before or not. There is a big caveat though – the way we will implement this solution is NOT thread safe. This means that the data access could easily go wrong. We'll explore this in a later lecture. When you modify your data access layer, to a thread-safe version that uses a database to store data, you'll probably want your data access layer to use a transient* service lifetime instead of a singleton. Nevertheless, for now, a singleton works **because** we are storing the data in a single object and we always want that same object returned!

We'll do some dependency injection* to set that up. Dependency injection asks the framework to provide us with the object(s) we want when it creates the instance of the controller that our request will be handled by.

Don't worry, it sounds worse than it is, and dependency injection is extremely useful (and used in your coursework).

* Yes, there's more reading:
https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection#service-lifetimes

Let’s dependency inject the singleton…

Open `Program.cs` and add the line below ***before*** `var app = builder.Build();`

```
builder.Services.AddSingleton<MyFirstAPI.DataAccess.MyDataCRUD>();
```

This adds a singleton of the MyDataCRUD type to the Service Container that will be automatically injected by the framework whenever it is a parameter of a constructor.

Return to your controller and add a readonly local variable to the class of type MyDataCrud and called _myDataCRUDAccess. Now add a constructor to the controller class that requires a MyDataCRUD parameter and assign its value to _myDataCRUDAccess.

```
readonly MyDataCRUD _myDataCRUDAccess;

public DataController(MyDataCRUD myDataCRUDAccess)
{
    _myDataCRUDAccess = myDataCRUDAccess;
}
```

That wasn’t so bad was it?

Now update your Get() and Get(int id) methods to use _myDataCRUDAccess.

```
[HttpGet]
public IEnumerable<string> Get()
{
    return _myDataCRUDAccess.Read();
}

[HttpGet("{id}")]
public string Get(int id)
{
    return _myDataCRUDAccess.Read(id);
}
```

Start up the application again and navigate to:

http://localhost:58789/api/Data/5

You should be looking at an exception.

Ah, of course, we forgot to add a call to actually CREATE our data. Let’s do it now.

Create a new GET method with no parameters inside your controller.

It can’t be called Get because there is already a method called Get with no parameters, and it wouldn’t make a great deal of sense to call it Get anyway because we aren’t going to get anything. So this time we’ll call the method CreateData()…

```
[HttpGet]
public void CreateData()
{
     _myDataCRUDAccess.Create()
}
```

Start up the application again and navigate to:

http://localhost:58789/api/Data/CreateData

Hmm. A wild error has appeared.

Give the error a read. It's the framework doing its absolute best to help you out without giving anyone else too much information if this error happened to make it into production code.

The key information for us (except for the traceId, which would be useful if we were logging our errors) is the part that reads:

```
"id":["The value 'CreateData' is not valid."]
```

The question is: why not?!

Well, think about what we've been doing until now….

http://localhost:58789/api/Data/5 has been routing us to the `Get(int id)` method so it stands to reason that http://localhost:58789/api/Data/CreateData would route us there too, because when the ROUTING is calculated, the framework only knows that it is expecting a GET request on the data controller with some additional parameter at the end. It's only when it actually routes the data there that it finds out that it can't parse "CreateData" into an integer.

Create a template
Add custom controller methods
Configure the routing
Run and test the service

Notice that, above the DataController declaration is the attribute

```
[Route("api/[controller]")]
```

Well you've just discovered routing. Unfortunately Web API isn't magic, and still relies on you telling it what you want. Actually this is a good thing as it gives lots more flexibility. However, we'll need to understand how to configure our routing and that's not as simple as it could be because we have a few options available to us…

Here's the rub. The framework needs to know how to route that URL to your specific controller and method. It even needs to know where the id parameter is supposed to come from if you want it to be sent as part of the URI path. By default, WebAPI leaves this up to you and we have some options on how to establish the routing. See the resource here:

https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing

We are going to continue to use routing as Microsoft have done in their template.

This class-level attribute tells the framework how to route EVERYTHING in the class… except where it is insufficient. For example, in `Get(int id)` the [HttpGet] attribute actually extends the routing behaviour to include the id parameter: `[HttpGet("{id}")]`

This tells the routing middleware that if it discovers the additional id parameter on the end of a path, it should route the request to this GET request rather than the one without the parameter specified.

We'll do a similar thing for our `CreateData()` method, except this time we are specifying that the routing middleware needs to pay attention to the name of the method. Just as our Class is the Controller, our Method is the Action.

Modify the [HttpGet] attribute above `public void CreateData()` to:

```
[HttpGet("[action]")]
```

Start up the application again and navigate to: http://localhost:58789/api/Data/CreateData (it returns void so you'll get a blank browser page). Then navigate to http://localhost:58789/api/Data/5

We have actually just been a bit naughty. I know!

Our CreateData method is now forcing what was a lovely RESTful API to behave more like RPC (Remote Procedure Call). CreateData is a behaviour – it is not resource (data) focused.

As long as we are aware that we are breaking the rules a bit, that's fine. REST is not a specification or protocol, just architectural guidance. This is "acceptable use" for our experimentation purposes.

We want another Get method that accepts a single parameter. This time though we want to take a string in as the parameter.

Underneath your Get(int id) method, add the new method below:

```
[HttpGet("{id}")]
public string Get(string id)
{
    return "Data not found.";
}
```

Restart the application and navigate to http://localhost:58789/api/Data/Five

Read the response that is returned. What has happened?

ASP.NET can't manage the fact that you now have two Get endpoints with the same routing. The key here is to recognise that everything in a URL path is a string, so it doesn't want to differentiate between your two endpoints based on parameter type alone.

We want to explicitly state that ANYTHING that can be cast to an integer should go to one method and everything else should go to the other:

Above `Get(int id)` modify the the attribute to `[HttpGet("{id:int}")]`

We are now explicitly adding a constraint, identifying that the method requires an int. The other doesn't need to be updated because a path is a string by default!

Restart the application and navigate to http://localhost:58789/api/Data/Five

Great. This works!

Now let's add another get method that returns a different string.

We want another Get method that accepts a single parameter. Again we want to take a string in as the parameter.

Underneath your Get(string) method, add the new method below:

```
[HttpGet("{name}")]
public string Get(string name)
{
    return "Your Name is " + name;
}
```

Of course…. This is still c#. We aren't allowed to do this because two methods can't have the same extension.

ASP.NET used to allow routing based on method name, where the method name matched the HTTP verb that you wanted to use (e.g. Get, Post, etc.) but this is no longer the case. This means that there is no need to call your get methods Get() – all we need to do is make sure that we use the correct attribute to identify the HTTP verb that we want to accept. In this case we are already identifying that we want to handle Get requests through the use of `[HttpGet]`

We want to differentiate our methods:

Change

```
[HttpGet("{name}")]
public string Get(string name)
```

to:

```
[HttpGet("{name}")]
public string Name(string name)
```

Visual Studio is happy.

Restart the application and navigate to http://localhost:58789/api/Data/John

What has happened?

Of course, there is no way for routing to differentiate between our methods so it doesn't know which to choose. We need to be more explicit to allow this type of functionality, and this is where we can use actions without breaking RESTful architecture.

Alter the HttpGet attribute above the Name method to include routing using both an action and name parameter .

navigate to http://localhost:58789/api/Data/Name/John

If you've set up the routing correctly, this should now work. Notice how this is still RESTful.

As a test, navigate to http://localhost:58789/api/Data/Create

It fails because there is no Action named Create

Before the `CreateData()` method add the attribute:`[ActionName("Create")]`

Navigate to http://localhost:58789/api/Data/Create

This works because you have explicitly told the framework the name of this action so the route matching process doesn't need to start looking at method names to make a match.

Note the new behaviour at: http://localhost:58789/api/Data/CreateData

Can you work out why this is happening now?

We've got loads of options on how to accomplish routing in ASP.NET; mostly it comes down to what is most useful to you as a web developer. You can pick and choose how you develop your application, depending on what you want it to do and how you want it to do it.

We know how to manage custom routes. How about this though…

Rather than requiring Web API to map parameters for us, we can be more explicit about what we are sending. We've seen in the lectures that a Post method gets its parameter from the body of the HTTP request using the `[FromBody]` attribute.

You can see this is already used in the POST and PUT default methods.

Let's use the similar `[FromQuery]` attribute to explicitly tell our API that we are expecting a parameter in the URI.

Replace the **`HttpGet`** attribute for the Name method with just `[HttpGet("[action]")]`

and change your `Name(string name)` method signature to `Name([FromQuery]string name)`.

This is a new situation. We can no longer rely on our routing to do any work for us. Instead, we have to be more detailed in our URI composition.

From the lecture slides, recall the composite parts of a URI. We need to make use of the query string. This uses the postfix '?' and works in this form:
`http://domain.com/path/?paramName=value&secondParamName=secondValue...`

Try it out.
Restart the application and navigate to
http://localhost:58789/api/Data/Name?name=John

We can similarly use the `[FromBody]` attribute to get values from the body of the HTTP request but this is harder to test using only a browser as usually they give little-to-no control to the user over the body of a request.

Create a template
Add custom controller methods
Configure the routing
Run and test the service

Testing the service is not as simple as opening a web browser in the majority of cases. As discussed earlier, this is because using a browser to submit a URI causes the browser to format a GET request for you and gives you no control over the request type or the header/body.

NOTE: You may have identified that we've talked about getting values/data from the URI and Body, but there is another option that may be useful [FromHeader]. However, usually we would grab the header values in middleware or a filter.

If we wanted to grab headers without injecting them as parameters we could also access them from within the method using `Request.Headers` *to get a dictionary of headers. This is especially important as headers aren't always required to use an endpoint, but may change how the endpoint functions.*

You won't be able to easily test other request types using a browser (e.g. POST, PUT, DELETE…) but you may be able to use applications like PostMan which allow you to create a request in its entirety.

PostMan will be very useful for the coursework so you should now take some time to learn how to get it interacting with your API.

Here are some resources. It should already be installed on the lab machines.
https://learning.postman.com/docs/getting-started/introduction/

You can sign in to PostMan using the instructions in appendix A.

Query your existing endpoints using PostMan.

Of course, we might also build a client to use our API. We're going to build a very simple client now…

Add a new Console Application project to your solution and replace the file contents with the following code (ensure you change the port number as appropriate):

```
HttpClient client = new HttpClient();

RunAsync(client).Wait();
Console.ReadKey();

static async Task RunAsync(HttpClient client)
{
    client.BaseAddress = new Uri("http://localhost:58789/api/");

    try
    {
        Task<string> task = GetStringAsync("Data/Name?name=John", client);
        if (await Task.WhenAny(task, Task.Delay(20000)) == task)
        {
            Console.WriteLine(task.Result);
        }
        else
        {
            Console.WriteLine("Request Timed Out");
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.GetBaseException().Message);
    }
}

static async Task<string> GetStringAsync(string path, HttpClient client)
{
    string responsestring = "";
    HttpResponseMessage response = await client.GetAsync(path);
    responsestring = await response.Content.ReadAsStringAsync();
    return responsestring;
}
```

Set both of your projects to start on StartUp and **Press F5 or click Start** to run your client and server.

After an initial delay as your server and client start, the new client should receive and display the response from your server: "You sent the string hello". Congratulations, you have a test client.

Have a read through this article on making an HTTP Client (it'll likely be useful for your coursework!):

https://learn.microsoft.com/en-us/dotnet/csharp/tutorials/console-webapiclient

Create a template
Add custom controller methods
Configure the routing
Run and test the service

Final Tasks

1. Add to your client so it can use all of your API endpoints. Ensure all of your CRUD behaviours can be used in the server.

2. Modify your methods to return the `IActionResult` type. With an IActionResult you can return a lot more information, like a specific server response code (e.g. 400 – Bad Request). Research how to do this and implement a few server responses with specific codes.
   See https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

3. Add a new server action method to translate to Pig Latin. Return a 400 error (Bad Request) if only one word is submitted. Hint: You may want to consider using a Post request and putting the data in the body for long messages.

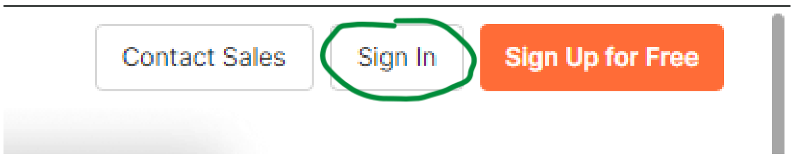
4. Experiment with sending data via the Header, Query and Body. Send multiple header and query parameters. Send JSON in the body. The more you can experiment and find out now, the easier your coursework should be for you!

5. Convert your client so that you can give it commands and messages to be sent by interacting with the Console. Test this against your Pig Latin translator and existing functionality.

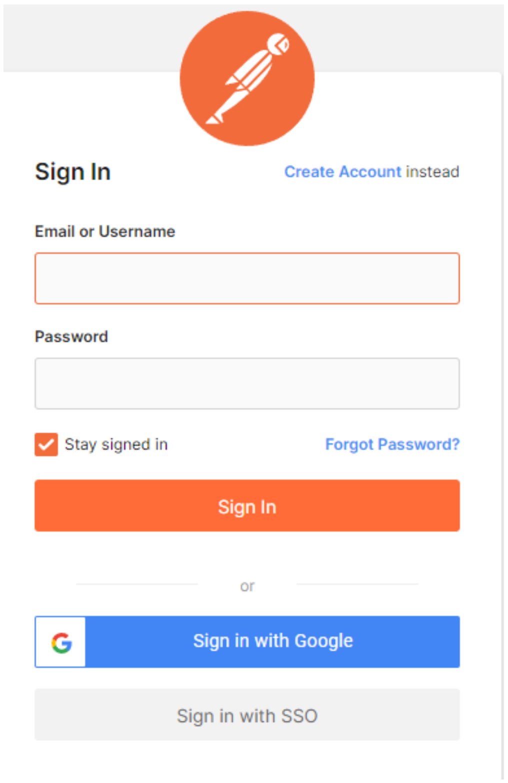
Appendix A

Setting Up a Free Unlimited Student PostMan account

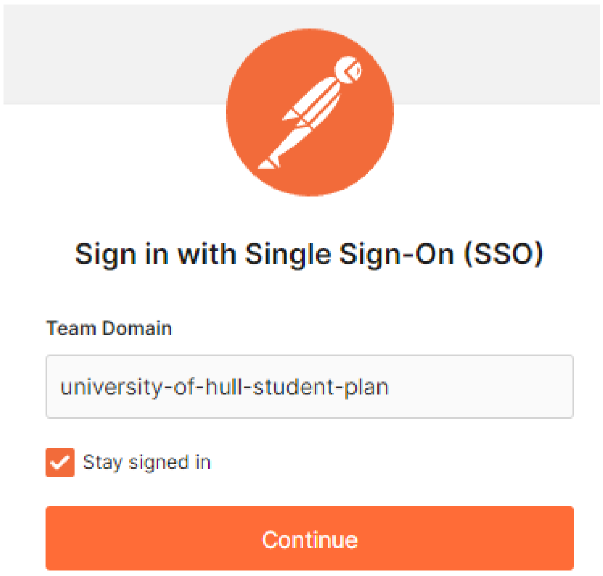
Visit https://getpostman.com and select Sign In from the top right of the screen.



Choose ‘Sign in with SSO’ at the bottom of the box on the sign-in page.

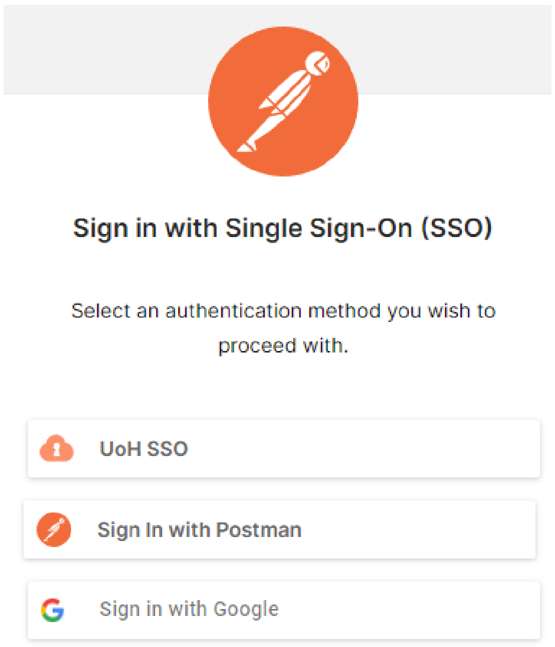


In the Team Domain textbox, copy/paste or type: ‘university-of-hull-student-plan’ and press Continue.


Sign in with Single Sign-On (SSO)

Team Domain: university-of-hull-student-plan

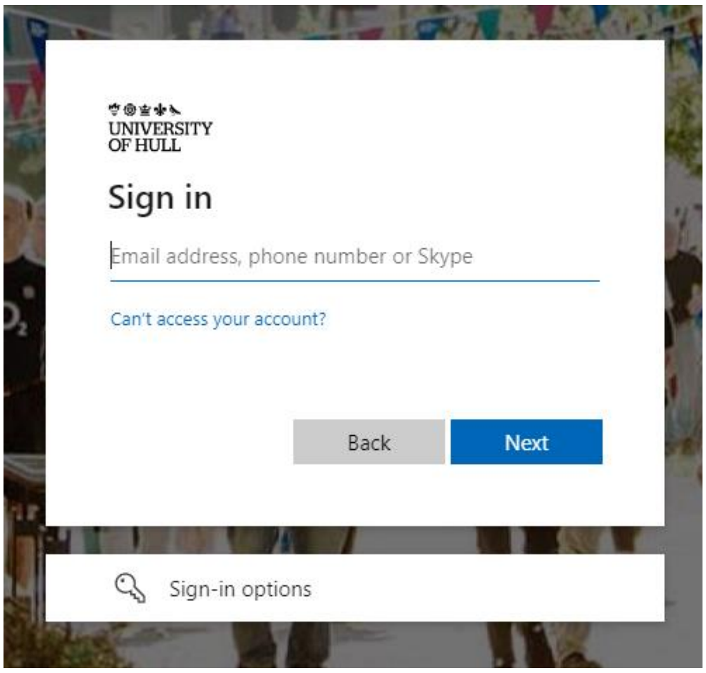
On the next screen, click UoH SSO.


Sign in with Single Sign-On (SSO)
Select an authentication method you wish to proceed with.
UoH SSO
Sign In with Postman
Sign in with Google

You should be directed to the Microsoft sign in page.

The URL for the login page should start with https://login.microsoftonline.com/

Log in with your University of Hull credentials, as you would normally do to log in to any University of Hull account or service. If you are on a Campus PC, you may not need to need to log in as you are logged in to the machine. If you are on your own machine or off Campus you may need to log in and you may need to perform 2FA.


UNIVERSITY OF HULL
Sign in
Email address, phone number or Skype
Can't access your account?
Back
Next
Sign-in options

When logged in, click 'My Workspace' to create your own Collection.

Once you have a Collection, you can add and send Requests to/from your server, create tests, etc.