Dist. Sys. ASP.NET Core Web API

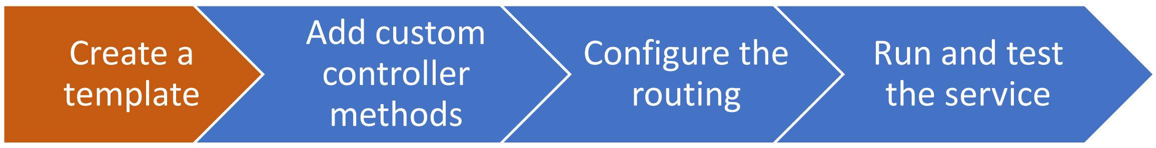
In this lab you'll be looking at ASP.NET Core Web API, which is very much related to your ACW!

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 lectures 8 and 9.

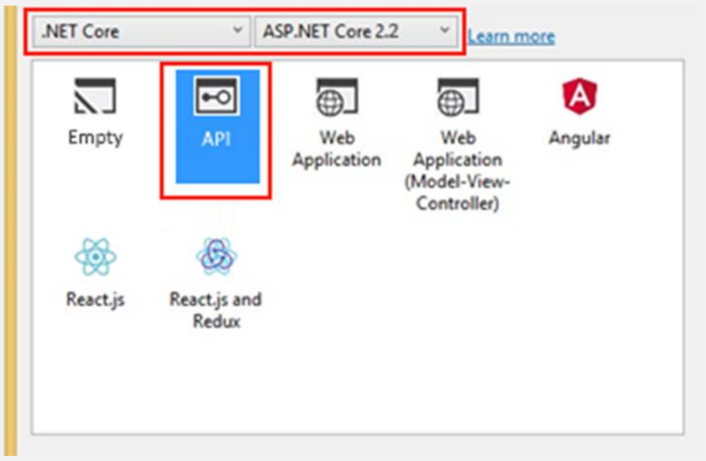
We will be following this process to start:



Open Visual Studio and create a new C# ASP.NET Core Web Application. Choose a save location in your folder on the C: drive 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.**

In the box that opens, ensure ASP.NET Core 2.2 is selected, click 'API' and then uncheck 'Configure for HTTPS' if it is an option:


.NET Core | ASP.NET Core 2.2

Click 'OK' 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 it Controller it has made for you by default. Now right-click the 'Controllers' folder in the Solution Explorer Window and select Add->Controller. Select 'API Copntroller with Read/Write Actions' and press Add.

Change the Controller name field to 'TranslateController' 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.

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

Next, then, comes 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. Notice how the Post and Put methods have parameters which use `[FromBody]` attributes too? Hopefully those are pretty self-explanatory. Consider the HTTP request packet; it is made from 2 parts, just like the SOAP envelope: the HEADER and BODY. This attribute allows ASP.NET to identify that it should expect a string `value` in the Body of the request.

What about the other parameters that don't have an explicit attribute identifying where they will come from? Well, the automatically generated comments give us some hints about these.

```
// GET: api/Translate/5
public string Get(int id)
{
    return "value";
}
```

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

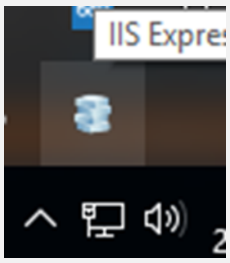
Have a read through the MSDN information on parameter binding to see your options here: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-2.2#sources

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

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.

Press F5 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 but that's only because you havent yet pointed your browser to a valid URI.

Your browser should have taken you to the address: http://localhost:22288/ although the port number may be different for you. If it doesn't work, whilst the application is running open your notification panel (bottom right of the task bar) and look for the IIS symbol:



Right click this icon then hover over you application name (under View Sites) and select the correct address under Browse Applications.

Remember: this will only be possible whilst your application is running!

Add the path `/api/Translate/5` to the end of the URI in your address bar. It should now look something like this: `localhost:22288/api/Translate/5`

Press enter to make the browser send the request. You should now see either some XML :

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/
">Your number plus 100 is 105</string>
```

Or simply the string `"Your number plus 100 is 105"`

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

Navigate to `localhost:22288/api/Translate/`

Note the different output.

Close the browser (which should close the application too – if it doesn't just stop it in VS) and go back to the Controller.

If you've followed the above instructions you should get a response back from the server but it doesn't look the same, even though you just didn't pass a number. Look back at your TranslateController.cs file. Can you identify what has happened?

Hopefully you can see that the controller has selected the *other* Get method as the most appropriate method for your request. But we only changed a parameter! What's going on?

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

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…

First, let's inspect the Controller.

Note how we have two attributes decorating the controller class: `[ApiController]` and `[Route("api/[controller]")]`

`[ApiController]` tells the framework that this is a controller and so contains endpoints.

`[Route("api/[controller]")]` indicates the path that should be used to get to these endpoints. Given this path, the framework will route requests to your controller. Note how [controller] is in square brackets – this allows us to be generic. Seeing as this is decorating the TranslateController, the route path would be 'api/translate' at runtime.

Now return to the `public string Get(int id)` method. See that it, too, has an attribute.

```
[HttpGet("{id}", Name = "get")]
```

This tells the framework that when there is a parameter in the path, it should route to this endpoint rather than the other 'get' method above that doesn't expect a parameter.

Restart the application and navigate to `localhost:22288/api/Translate/hello`

What does your application send back?

Web API is clever. It has identified that your method expects an integer, so when you give it a string it sends an error in response.

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) method, add the new method below:

```
[HttpGet("{input}", Name = "get")]
public string Get(string input)
{
    return "You sent the string " + input;
}
```

Restart the application and navigate to `localhost:22288/api/Translate/hello`

Read the response that is returned. What has happened?

ASP.NET doesn't like 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)` change the attribute to `[HttpGet("{id:int}", Name = "get")]`

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 `localhost:22288/api/Translate/hello`

Read the response that is returned. What has happened?

Now the framework is complaining that you have two Get endpoints with the same name but different parameter constraints. Again, this makes routing difficult.

We want to differentiate our endpoint names:

Above `Get(int id)` change the attribute to `[HttpGet("{id:int}", Name = "getInt")]`

Above `Get(string id)` change to `[HttpGet("{id}", Name = "getString")]`

Restart the application and navigate to `localhost:22288/api/Translate/hello`

Now navigate to `localhost:22288/api/Translate/5`

Both endpoints should now work correctly.

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}", Name = "getName")]
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 `Get(string input)` to `GetString(string input)`

Change the other `Get(string name)` to `GetName(string name)`

Visual Studio is happy.

Restart the application and navigate to `localhost:22288/api/Translate/hello`

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.

Using Actions

At the moment we have been extracting a parameter from the path and using this to identify the endpoint method that should be called.

This assumption isn't always useful, and it is often better to be more explicit with our URL to prevent any issues differentiating different endpoints.

Change `[Route("api/[controller]")]` to `[Route("api/[controller]/[action]")]`

This updates our routing to pay attention to the action name.

Restart the application and navigate to `localhost:22288/api/Translate/hello`

What has happened? It's no longer working! Correct. Now we care about our action name, we need to add this to our URL.

navigate to `localhost:22288/api/Translate/`**`GetString`**`/hello`

navigate to `localhost:22288/api/Translate/`**`GetName`**`/myName`

navigate to `localhost:22288/api/Translate/`**`Get`**`/5`

It's worth knowing that ASP.NET will uses the method name as the action name if we have not specified an action name in an attribute. This is not the same as defining name in the `[HttpGet]` attribute.

As a test, navigate to `localhost:22288/api/Translate/`**`GetInt`**`/5`

It fails because there is no Action named GetInt

Before the Get(int) method add the attribute: `[ActionName("GetInt")]`

Navigate to `localhost:22288/api/Translate/`**`GetInt`**`/5`

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.

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 an 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.

Before you continue, make sure you understand about routing, and read through your options for routing in ASP.NET Core, here: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-2.2

Ok! 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 that the Post method gets its parameter from the body of the HTTP request using the `[FromBody]` attribute. Let's use the similar `[FromQuery]` attribute to explicitly tell our API that we are expecting a parameter in the URI.

Replace the attribute `[HttpGet("{name}", Name = "getName")]` with just `[HttpGet]`

and change your `GetName(string name)` method signature to
`GetName([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
`localhost:22288/api/Translate/getname/myName`
then to `localhost:22288/api/Translate/getname`**`?name=myName`**

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.

You have now explicitly defined a route for each of these get methods. This explicit routing prevents Web API from having to do any guesswork – but could be more work to maintain in the long run. As a developer, you'll need to choose what is the best routing option for your use case.

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 doesn't seem to be information on using data from the Header. This is correct. In order to get information from the header in Web API we have to use `this.Request.Headers` which gets us a collection of all the headers in the request. Then we can do our own filtering to find out exactly which headers exist and parse them if we wish.*

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.

You should be able to use PostMan on the lab computers and it will be very useful for the coursework so you should learn how to get it interacting with your API.

Of course, the other option is building a client to test your API requests. 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:

```
using System;
using System.Threading.Tasks;
using System.Net.Http;


namespace ConsoleClient
{
    class Program
    {
        static HttpClient client = new HttpClient();

        static void Main(string[] args)
        {
            RunAsync().Wait();
            Console.ReadKey();
        }

        static async Task RunAsync()
        {
            client.BaseAddress = new Uri("http://localhost:22288/");

            try
            {
                Task<string> task = GetStringAsync("/api/Translate/GetString/hello");
                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)
        {
            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 starts, 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://docs.microsoft.com/en-us/aspnet/web-api/overview/advanced/calling-a-web-api-from-a-net-client

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

Final Tasks

1. Modify your Get(string input) method to return an `ActionResult` type. With an ActionResult 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

2. Add to your client to use all of your API endpoints.

3. Using your Pig Latin translation code from previous labs, alter your GetString method to translate to Pig Latin. Return a 400 error (Bad Request) if only one word is submitted. Hint: You may have to consider using a Post request or look up how spaces are managed in a URI.

4. Spend some time adding in extra Post and Get functionality to your server. Test it out by modifying and using your client. 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. Test this against your Pig Latin translator and new Post/Get functionality.