Introduction pt 1

udemy
.Net Core 2

Web Services interact with other web services and applications.

Loosely Coupled Architecture means that there is little or no dependency. You can replace the client or webservice with little impact.

SOAP is Simple Access Object Protocol. It uses XML to exchange information.

REST is REprisentational State Transfer. REST is a network of web pages and the user navigates through by the use of links. It uses http and is very lightweight.

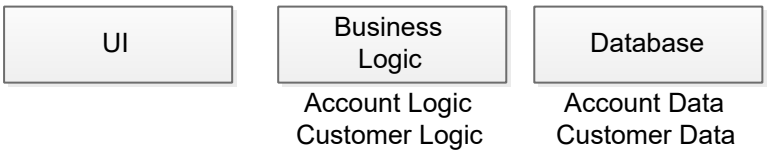
JSON is Java Script Object Notation and it is a subset of Java and Human Readable text.

XML - eXtensible Markup Language - chose JSON over XML whenever possible as it is much easier to read in and convert to objects.

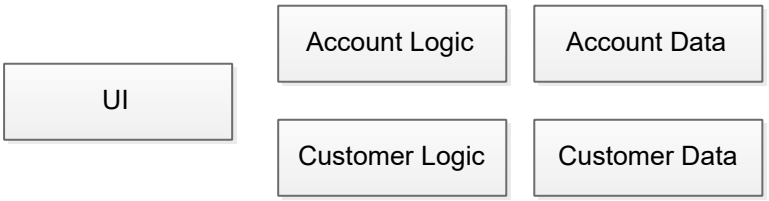
Scaling Up: Upgrading Processors

Scaling Out: Increasing the number of Services

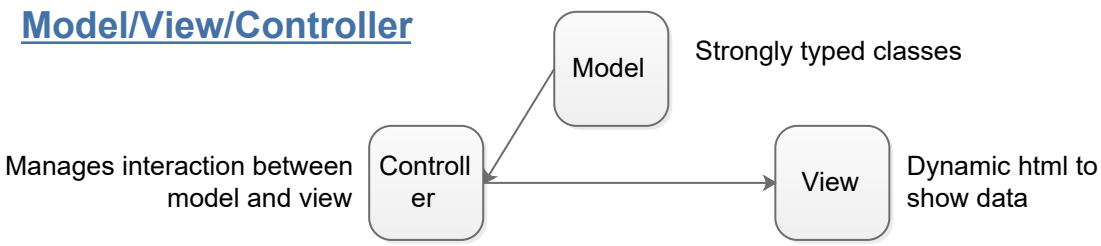
Monolithic Architecture



Microservice Architecture



Model/View/Controller



Introduction pt 2

udemy
.Net Core 2

ASP .NET CORE is a lightweigh framework.
Pattern-based way to build dynamic web-sites.

ASP .NET CORE Elements
/controllers folder - contains classes to handle the initial request.

Program.cs - entry point for the service (like a console app)

Startup.cs - sets up the configuration, dependency injection

appsettings.cs - key/value based config file

web.config - only used for IIS

Program.cs
This has an entry point, like a console app. IWebHost is the main class for the web service.

Startup.cs
WebHostBuilder inside program.cs runs this. It Configures services
Procedure ConfigureServices is used to manage the services list.

Configure is used to manage the http pipeline (bits to go through before it gets to your code and afterwards)

Reverse Proxy
This is where a request comes in from the Internet - the application (usually IIS or Apache etc) forwards it onto the local web service.

Kestrel
This is an application which can be used to manually spin up a .net core web service for testing.

Use .Net Core when
- you need to use containers over different platforms
- you are targetting micro services
- you are using docker containers
- you need high performance - scalable solutions
- you need side-by-side .net versions per application

Introduction pt 3

udemy
.Net Core 2

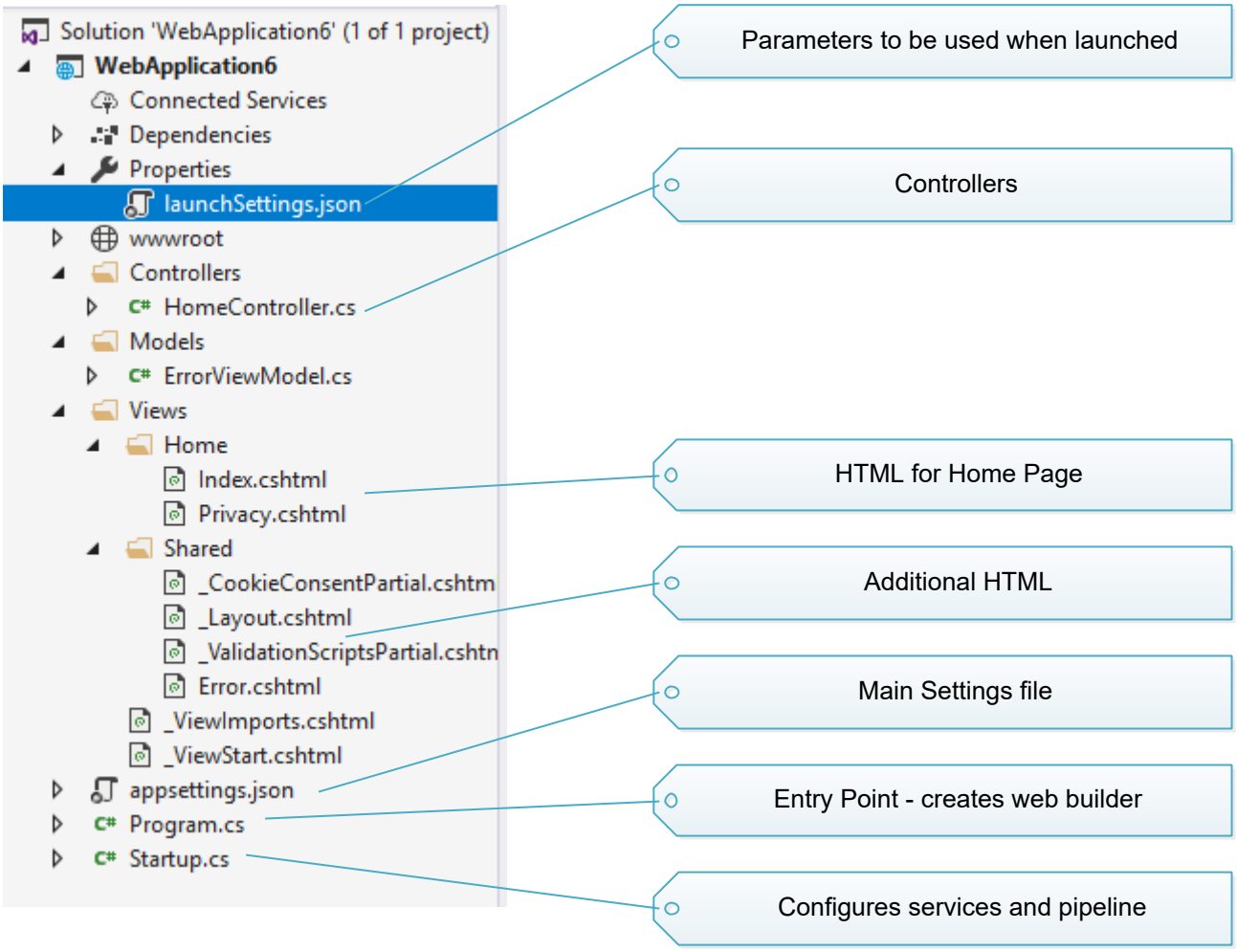
Use .Net Framework when...

- you are currently using framework
- you use third party libraries not available on .net core

MVC App Structure

udemy
.Net Core 2

Create a new ASP.NET Core Web Application ->
Web Application (Model-View-Controller) ->



CRUD Web App pt1

udemy
.Net Core 2

Create Default Project

Here we will be creating a simple Create/Read/Update/Delete web service.

User Interface will have 5 views: Index, Create, Edit, Detail, Delete.

The controller will be a ToDo Controller - a single controller with a few actions.

Database - we will be using an in-memory database.

```
Create a new project: ASP.NET Core Web Application
Application Type: Web Application (Model-View-Controller)

  Install a few NuGet package into Project:DatabaseTest
    - SmartIT.DebugTraceHelper
    - SmartIT.Employee.MockDB
```

Run it - it should open up a browser and show you a nice welcome screen.

CRUD Web App pt2

udemy
.Net Core 2

Add our own controller

```
Select Controllers folder, right click - Add...Controller

  Select MVC Controller with Read/Write Actions
  Give it a name - ensure it ends in Controller
  We're gonna be putting code directly into the controller - you may want
  to put your code in their own classes - see add the helper using statements

    using SmartIT.DebugHelper;
    using SmartIT.Employee.MockDB;
```

Amend the startup to use our own controller...

```
  app.UseMvc(routes =>
  {
    routes.MapRoute(
      name: "default",
      template: "{controller=ToDo}/{action=Index}/{id?}");
  });
```

Now we start adding Views...

CRUD Web App pt3

udemy
.Net Core 2

Create 'Index' View - The List

Here we get the index action to return all rows in the database.

Add a member variable for the todo database...

ToDoController.cs:

```
TodoRepository _todoRepository = new TodoRepository();
```

Amend the 'index' action to return all rows

```
// GET: ToDo
public ActionResult Index()
{
  return View(_todoRepository.GetAll());
}
```

Add our own Model

As you know - in an MVC - there has to be a Model/View/Controller. We already have the Controller, we need to create a Model before we can attach it to a view. We need to use the ToDo class in our MockDB - but it isn't in the drop down yet - so we create a class and derive from it. Right click on the Models folder- Add...Class

This is a bit of a frig - normally you will have your own model and then it will appear normally in the drop down - since we are using a model from SmartIT - we need to create a dummy one derived from it. This is why we have to manually amend the cshtml because the view is created with our dummy model - but we are actually passing the SmartIT model - hey ho.

Enter workaround.cs as a name and create a new class derived from the todoRepository...

```
public class Todo : SmartIT.Employee.MockDB.Todo { }
```

Add our own Index View (list)

Select Views folder, right click - Add...View

Set ViewName to Index
when the Index action is triggered above - it looks for an Index.cshtml in Views/ToDo, then in Views/Shared

Pick 'List' as a template
Pick our newly created ToDo View

CRUD Web App pt4

udemy
.Net Core 2

Customise Index/List View

```
  Views are written in html (or cshtml).
  What we need to do here is add some headings and the table detail.

  Table header
  <thead>
    <tr>
      <th>
        @Html.DisplayNameFor(model => model.Id)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.Name)
      </th>
    </tr>
  </thead>
  Obviously thead is table header, tr is table row, th is table header item



Table detail
  <tbody>
    @foreach (var item in Model)
    {
      <tr>
        <td>
          @Html.DisplayFor(modelItem => item.Id)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
          @Html.ActionLink("Edit", "Edit", new {/*id=item.PrimaryKey*/}) |
          @Html.ActionLink("Details", "Details", new{/*id=item.PrimaryKey*/}) |
          @Html.ActionLink("Delete", "Delete", new{/*id=item.PrimaryKey*/ })
        </td>
      </tr>
    }
  </tbody>
```

Run it - you should get a list of items, with their contents

CRUD Web App pt5

udemy
.Net Core 2

Create 'Create View'

Goto the ToDo controller, select the Create action - right click 'View' of the return(View());, select Add View.

Set ViewName to Create

Pick 'Item' as a template
Pick our newly created ToDo View

You can customise the form Create.cshtml if you like, remove the Id field if it is autogenerated etc, you want to change the model anyway

create.cshtml

```
@model SmartIT.Employee.MockDB.Todo
```

Now we need to do the actual insert...

```
// POST: ToDo/Create
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(IFormCollection collection)
{
    try
    {
        _todoRepository.Add
            (
                new SmartIT.Employee.MockDB.Todo()
                {
                    Name = collection["Name"]
                }
            );

        return RedirectToAction(nameof(Index));
    }
    catch
    {
        return View();
    }
}
```

CRUD Web App pt6

udemy
.Net Core 2

Create an 'Edit View'

```
Goto the ToDo controller, select the Edit action - right click 'View' of the
return(View());, select Add View - type is 'edit'

    Set ViewName to Edit


    Pick 'Item' as a template
    Pick our newly created ToDo View
```

You can customise the form Edit.cshtml if you like, remove the Id field if you don't want it shown etc, you want to change the model anyway

```
  edit.cshtml
    @model SmartIT.Employee.MockDB.Todo
```

Now we need to do the actual edit...
The edit comes from the Index.cshtml...

```
  @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
  this will call the edit action so we need to amend the edit action so that
  it gets the rest of the record...

This happens before the edit form has been shown
        // GET: ToDo/Edit/5
        public ActionResult Edit(int id)
        {
            return View(_todoRepository.FindById(id));
        }


This happens after the edit form has been shown
ToDoController.cs....
        // POST: ToDo/Edit/5
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Edit(int id, IFormCollection collection)
        {
            try
            {
                var findTodo = _todoRepository.FindById(id);
                findTodo.Name = collection["Name"];
                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View();
            }
        }
```

CRUD Web App pt7

udemy
.Net Core 2

Create a 'Detail View'

```
Goto the ToDo controller, select the Detail action - right click 'View' of the
return(View());, select Add View - type is 'edit'

    Set ViewName to Detail


    Pick 'Item' as a template
    Pick our newly created ToDo View
```

You can customise the form Details.cshtml if you like, remove the Id field if you don't want it shown etc, you want to change the model anyway

```
  details.cshtml
    @model SmartIT.Employee.MockDB.Todo
```

Now we need to do the actual edit...
The edit comes from the Details.cshtml...

```
  @Html.ActionLink("Detail", "Detail", new { id=item.Id }) |
  this will call the detail action so we need to amend the detail action
  so that it gets the rest of the record...

This happens before the edit form has been shown
        // GET: ToDo/Details/5
        public ActionResult Details(int id)
        {
            return View(_todoRepository.FindById(id));
        }


This happens after the edit form has been shown
ToDoController.cs....
        To be honest - don't really need to do anything for this...
```

CRUD Web App pt8

udemy
.Net Core 2

Create a 'Delete View'

```
Goto the ToDo controller, select the Delete action - right click 'View' of the
return(View());, select Add View - type is 'delete'

    Set ViewName to Delete


    Pick 'Delete' as a template
    Pick our newly created ToDo View
```

You can customise the form Details.cshtml if you like, remove the Id field if you don't want it shown etc, you want to change the model anyway

```
  details.cshtml
    @model SmartIT.Employee.MockDB.Todo
```

Now we need to do the actual edit...
The edit comes from the Details.cshtml...

```
  @Html.ActionLink("Delete", "Delete", new { id=item.Id }) |
  this will call the delete action so we need to amend the delete action so
  that it gets the rest of the record...

This happens before the edit form has been shown
        // GET: ToDo/Delete/5
        public ActionResult Delete(int id)
        {
            return View(_todoRepository.FindById(id));
        }



This happens after the delete form has been shown
ToDoController.cs....
 // POST: ToDo/Delete/5
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Delete(int id, IFormCollection collection)
        {
            try
            {
                _todoRepository.Delete(_todoRepository.FindById(id));

                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View();
            }
        }
```

Middleware pt1

udemy
.Net Core 2

This is a software component which automatically intercepts the request/ response pipeline.

It can choose whether to continue with the pipeline or nor.

It can perform work before and after your web service code.

These are setup in the config method.

Each stage could stop the pipeline.

Each 'middleware' box is called a request delegate and they are processed in order they are presented. They are returned in reverse order.

Middleware pt2

udemy
.Net Core 2

We will be creating three middleware components: Use with in-line code; Use with custom class; Run with in-line code.

1. Use with in-line code...

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  app.Use
  (
    async (context, next) =>
    {
      await context.Response.WriteAsync("Middleware 1<br/>");
      await next();
    }
  );
```

Fairly straight forward 'await context...' does the work, await next calls the next middleware in the pipeline.

2. Use with custom class...

Middleware2.cs

```
namespace CustomMiddleware
{
    public class Middleware2
    {
        private readonly RequestDelegate _next;

        public Middleware2(RequestDelegate next)
        {
            this._next = next;
        }

        public async Task Invoke(HttpContext context)
        {
            await context.Response.WriteAsync("Middleware 2<br/>");
            await this._next(context);
        }

    }
}
```

again fairly straight forward - the constructor is given the 'next middleware' in the pipeline - we save it - then when it is invoked - we simply call it.
now we patch it into configure - just after the first use...

```
app.UseMiddleware<Middleware2>();
```

3. Run with inline code...

```
app.Run(async (context) =>
{
  await context.Response.WriteAsync("Middleware 3 Request<br/>");
  await context.Response.WriteAsync("Middleware 3 Response<br/>");
});
```

If you want to stop the pipe early (like failing authorisation) then simply do not call "await this._next" - if you did this in MW2 - you would bet MW1->MW2->MW1,

Middleware pt3

udemy
.Net Core 2

The built-in middlewares are: -

Authentication
Provides authentication support. Required before HttpContext.User is required. Terminal for OAuth callbacks.

CORS
Configures Cross Origin Resource Sharing. Required before components that use CORS

Diagnostics
Configures Diagnostics. Required before components that generate errors.

ForwardedHeaders/HttpOverrides
Forwards proxied headers onto current request. Required before components that consume the update fields.

ResponseCaching
Provides support for caching responses. Required before components that require caching.

ResponseCompression
Provides support for compressing responses. Required before components that require compression.

RequestLocalization
Provides localization support. Required before localization sensitive components.

Routing
Defines and constrains request routes. Terminal for matching routes.

Session
Provides support for managing user sessions. Required before components which require sessions.

StaticFiles
Provides support for serving static files and directory browsing. Terminal if request matches files.

URL Rewriting
Provides support for rewriting and redirecting URL requests. Required before components that consume the URL.

WebSockets
Enables the web sockets protocol. Required before components which accept WebSocket requests.

Controller

udemy
.Net Core 2

A controller handes the HTTP Requests and creates HTTP Responses - this is the entry point and exit point for your code. It only gets called once the appropriate middleware components have been called and it is the start of the reponses going back.

The name of the controller always ends in 'Controller' and is inherited from Controller.

A controller can be decorated with the following attributes, the whole controller can be decorated with these - or each individual action can be...

Produces
eg. [Produces ("application/json")]
This specifies the return type.

Route
eg. [Route("api/Employee")]
Determines how this action is called, see Routing page.

Verb decorations (only used for actions)

Http Put
eg. [HTTP PUT]
Specifies that this action is a PUT action.

Http Get
eg. [HTTP GET]
Specifies that this action is a GET action.

Http Delete
eg. [HTTP DELETE]
Specifies that this action is a DELETE action

Convention Based Routing

udemy
.Net Core 2

The purpose of 'routing' is to map an incoming request to an action.

Convention based routing provides a default or base. Basically you setup the details in the 'template' member of MapRoute and it routes to the controller and action from that. It is quite handy if you want all of your controllers to behave in the same manner.

One middleware component which can be used to specify the default controller and default verb is...MVC

```
services.AddMvc();

app.UseMvc
(
  routes =>
  {
    routes.MapRoute
    (
      name: "default",
      template: "{controller=Hi}/{action=Index}/{id?}"
    );
  }
);
```

'default' will be used if no name specified, Controller 'Hi' (ie. HiController) will be used. The action Index will be used if not specified. The Id parameter is optional (as per the question mark)

So ...

```
localhost:51719 becomes local:51719/Hi/Index
```

So here is an example

```
app.UseMvc
(
  routes =>
  {
    routes.MapRoute
    (
      name: "gotoOne",
      template: "one",
      defaults: new
        {
          controller = "Home",
          action = "ViewOne"
        }
    )
  }
);
public class HomeController: Controller
{
  public IACtionResult ViewOne()
  {
     return(Content("/Home/One");
  }
}
```

Attribute Based Routing

udemy
.Net Core 2

This basically overrides the default template - you basically put the routing information on the actual controller or the action itself. If you do put routing on the controller - it ignores the routing in the MapRoute - it does not take both into account. The default routing defined in the MapRoute is used if no routing on the controller or action. The routing information can include the terms [controller] or [action] - or they can just be pure text - you don't usually mix the two though.

You put the routing on the controller in the same way you put routing in MapRoute. This can include 'controller' or 'action' ids.

```
[Route("api/Employees")]
[Route("api/[controller]/[action]/{id?}")]
public class HomeController: Controller
{
  [HttpGet("MyViewOne")]
  public IActionResult ViewOne()
  {
     return(Content("/Home/One");
  }
}
```

So - you can use

```
https://localhost:44379/api/Employees/MyViewOne
  This is a result of using [Route("api/Employees")] and then
    [HttpGet("MyViewOne")]

https://localhost:44379/api/Home/ViewOne
  This is a result of using [Route("api/[controller]/[action]/{id?}")]

https://localhost:44379/api/Home/ViewOne/5
  This is a result of using [Route("api/[controller]/[action]/{id?}")]
```

Note - that you cannot do this...

```
https://localhost:44379/api/Employees/ViewOne
  This is not valid because to use 'ViewOne' you need a routing which
    has [action] in it.  If you just use text in the Controller Route
    then you must use text in the Action Route.
```

Multiple Routes

udemy
.Net Core 2

A Controller can have several routes and if different types and styles - any of which can be used...

```
[Route("Employees"]
[Route("[controller]"]
[Route("api/[controller]/all/[action]"]
[Route("api/[controller]/[action]/{id?}"]
public class HomeController: Controller
{
  [HttpGet("Find")]
  public IActionResult ViewOne()
  {
     return(Content("/Home/One");
  }
}
Example URLs would be:
  to use [Route("Employees"], we use
    http://mymachine/myrest/Employees/Find

  to use [Route("[controller]"], we use
    http://mymachine/myrest/Mine/Find

  to use [Route("api/[controller]/all/[action]"], we use
    http://mymachine/myrest/api/Home/all/Find

  to use [Route("api/[controller]/[action]/{id?}"], we use
    http://mymachine/myrest/api/Home/Find/5
```

Routing Constraints

udemy
.Net Core 2

You can ensure that a parameter conforms to certain rules by applying conditions to the HttpGet attribute

This will ensure that Id is an integer - if not - 404 Not Found will be returned....

```
[HttpGet("{Id:Int}"]
```

This will ensure that Id is in a particular range....

```
[HttpGet("{Id:range(18,40)}"]
```

This will ensure that Name conforms to a RegEx expression....

```
[HttpGet("{Name:regex(^[[a-bA-B0-2]]*)}")]
  ab - ok
  aB - ok
  Ab - ok
  BA - ok
```

Other tokens...

```
  Boolparam:True or param:False
  Datetimeparam: DateTime
  Decimal
  Double
  Float
  Guid
  Maxlength
  Minlength
  Length(min,max)
  Range(min,max)
  Required
  Regex
```

Swagger

udemy
.Net Core 2

Swagger is tool you can use to dynamically document your API.
First you need to add a package 'Swashbuckle.AspNetCore'. Swagger needs unique routing paths - so if you have multiple controllers - you should really use attributes for each controller - like '[Route("api/[controller]")' etc

Add to startup.cs

```
using Swashbuckle.AspNetCore.Swagger;
```

Add this to configureServices (after services.AddMvc)...

```
services.AddSwaggerGe
    (
        c =>
        {
            c.SwaggerD
                (
                    "v1",
                    new Info
                    {
                        Title = "My API",
                        Version = "v1"
                    }
                );
        }
    );
```

Links to the SwaggerEndPoint

Displayed as text on the swagger page
My API v1

Add this to configure

```
app.UseSwagger();
app.UseSwaggerUI
  (
    c=>
      {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
      }
  );
```

Seems to be virtual - does not physically create a file - it's all in the DLL

Appears in the 'spec drop down' top right of swagger page

No run - and use the URL to navigate to what ever is there plus '/swagger'. This should show you the API Calls.

Does not tend to work well with a project based on an empty web service - start with one which has the standard actions and where the controller is derived from controllerbase.

Other uses...

```
[HttpGet]
[Route("~/api/groups/{actionId:guid}/users")]

[SwaggerOperation(Summary = "Get users", OperationId = "get-users",
    Description = "Gets a paged list of all users")]

[SwaggerResponse(StatusCodes.Status200OK, "Ok",
    typeof(PagedResultViewModel<UserViewModel>))]

[SwaggerResponseHeader(StatusCodes.Status200OK,
    HeaderNames.ContentRange, "string",
    "Indicates where in a full body message a partial message belongs")]

public async Task<IActionResult> GetAllUsers(
  [SwaggerParameter("Query used to retrieve users")]
    GetUsersQuery query,

   [SwaggerParameter("Cancel Token")]
    CancellationToken cancellationToken)

{
```

Swagger Additional

udemy
.Net Core 2

You can get swagger to pick up details from the 'comments'.

If you press ///, you get...

```
/// <summary>
///
/// </summary>
/// <returns></returns>
```

If you put something in the summary then it appears in Swagger - but unfortunately you have to do some setting up.

First open up the project (click it in the treeview - the source will appear in the main page) - add the highlighted bit.

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.2</TargetFramework>
  <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
</PropertyGroup>
```

Now when you do a build - you should see an .xml document in the binary directory - in startup.cs - add in the highlighted bit...

```
services.AddSwaggerGen
(
  c =>
  {
    c.SwaggerDoc
    (
      "v1",
      new Info
      {
        Title = "Mine API",
        Version = "v1"
      }
    );
    var fileName =
      this
        .GetType()
        .GetTypeInfo()
        .Module
        .Name
        .Replace(".dll", ".xml")
        .Replace(".exe", ".xml");
    c.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory, fileName));
  }
);
```

Insomnia Testing

udemy
.Net Core 2

Environments and Variables

You can save a whole lot of time by creating environments. Each environment has the same variables - with different values - you can then use these variables in the tests. Add an environment - select the ENVS, select 'manage' - click + to add a new one. Dbl click the name to rename it. Click in the right pane to add the variables in the following format:

```
{
  "host": "localhost",
  ..
}
```

Base Environment - click on this to enter the variables to apply to all envs, then each env can have its own override - this saves you having hundreds of variables if they are all the same.

When you are on the request line - click <ctrl>space to select a variable - you can put variables anywhere in the request - they appear in a coloured block.

Proxies

Useful for working with Oneserve - click 'enable proxy'

Enter the following in the http/https line...

http:<user>:<password>@proxy-c.kcom.com

user is the login without KINGSTON\

Set no_proxy to: localhost, 127.0.0.1

Powershell Testing

udemy
.Net Core 2

Testing GET

```
$Cred = Get-Credential
$Url = "https://localhost:5001/example/MyGet/1"
Invoke-RestMethod -Method 'Get' -Uri $url -Credential $Cred
```

Testing POST

```
$Cred = Get-Credential
$Url = "https://localhost:5001/example/MyGet/1"
$Body = @{ search = "fred=joe" }
Invoke-RestMethod -Method 'Get' -Uri $url -Credential $Cred -Body $Body
```

Further options

API Versioning pt 1

udemy
.Net Core 2

URI/Namespace based versioning

```
namespace SmartIT.  V1Hi .Controllers
{
  public class HiController : Controller
  {
    [Route("~/api/  v1/Hi ")] // Attribute routing
    [HttpGet]
    public IActionResult Get() => Content("Hi Version 1");
  }
}
namespace SmartIT.  V2Hi .Controllers
{
  public class HiController : Controller
  {
    [Route("~/api/  v2/Hi ")]  // Attribute routing
    [HttpGet]
    public IActionResult Get() => Content("Hi Version 2");
  }
}
```

Pros
Quite common
Easy to implement
Easy to consume
Fast Performance
Can be applied to old non-versioned code
No config
Easy to Test

Cons
Disrupts Restful compliance - URI should represent the resource not versions

API Versioning pt 2

using MVC Middleware

udemy
.Net Core 2

Add the middleware...

```
services.AddApiVersioning(
  options =>
  {
    options.ReportApiVersions = true;
    options.AssumeDefaultVersionWhenUnspecified = true;
    options.DefaultApiVersion = new ApiVersion(1, 0);
    //options.ApiVersionReader = new HeaderApiVersionReader("api-version");
  }
);
```

Query String - based

```
[ApiVersion("1.0")]
[Route("api/student")]
public class StudentControllerV1 : Controller
{
  [HttpGet]
  public IActionResult Get() => Content("Version 1");
}

[ApiVersion("2.0")]
[Route("api/student")]
public class StudentControllerV2 : Controller
{
  [HttpGet]
  public IActionResult Get() => Content("Version 2");
}
```

```
http://localhost:56576/api/student                    // uses version 1 (default)
http://localhost:56576/api/student?api-version=1.0    // uses version 1
http://localhost:56576/api/student?api-version=2.0    // uses version 2
```

API Versioning pt 3

using MVC Middleware

udemy
.Net Core 2

Http header/attribute - based

```
[ApiVersion("1.0")]
[ApiVersion("2.0")]
[Route("api/student")]
public class StudentControllerV1 : Controller
{
  [HttpGet]
  public IActionResult Get() => Content("Version 1");

  [HttpGet, MapToApiVersion("2.0")]
  public IActionResult Get() => Content("Version 2");
}
```

In the request - header - set
api-version to 1.0 or 2.0 or leave it out - to default to 1.0

```
http://localhost:56576/api/student                    // uses version 1 - default
http://localhost:56576/api/student?api-version=1.0    // uses version 1
http://localhost:56576/api/student?api-version=2.0    // uses version 2
```

API Versioning pt 4
using MVC Middleware

udemy
.Net Core 2

Convention - based

Configure - middleware

```
services.AddApiVersioning(
  options =>
  {
    options.ReportApiVersions = true;
    options.AssumeDefaultVersionWhenUnspecified = true;
    options.DefaultApiVersion = new ApiVersion(1, 0);
    options.ApiVersionReader = new HeaderApiVersionReader("api-version");
    options.Conventions.Controller<StudentControllerV1>()
      .HasApiVersion(new ApiVersion("1.0")),
    options.Conventions.Controller<StudentControllerV2>()
      .HasApiVersion(new ApiVersion("2.0")),
  }
);
```

```
[Route("api/student")]
public class StudentControllerV1 : Controller
{
  [HttpGet]
  public IActionResult Get() => Content("Version 1");

  [HttpGet, MapToApiVersion("2.0")]
  public IActionResult Get() => Content("Version 2");
}
```

In the request - header - add

```
api-version: 2.0
```

```
http://localhost:56576/api/student                  // uses version 1 - default
http://localhost:56576/api/student?api-version=1.0  // uses version 1
http://localhost:56576/api/student?api-version=2.0  // uses version 2
```

Deprecating a version

```
[Route("api/student")]
public class StudentControllerV1 : Controller
{
  [ApiVersion("1.0",   Deprecated=true  )]
  [HttpGet]
  public IActionResult Get() => Content("Version 1");

  [ApiVersion("2.0", Deprecated=true)]
  [HttpGet]
  public IActionResult Get() => Content("Version 2");
}
```

This does not stop the action being used - it just outputs '-deprecated version 1.0' instead of '-supported version 2.0'

Security pt 1

udemy
.Net Core 2

Authentication - validates the user - making sure they are who they say they are

Authorisation - validations the action - are they allowed to do what they want to do.

Implementing a security scheme
This basically uses middleware to automatically intercept the request and perform the following pseudo logic - automatically
is the user already logged in (does a cookie exist)?
if not - authenticate the user, authorise the action - create a cookie
if the user wants to log out - remove the cookie

configureServices...

```
services.AddAuthentication("SaSecurityScheme")
  .AddCookie
  (
    "SaSecurityScheme",
    options =>
    {
      options.AccessDeniedPath = new PathString("/Security/Access");
      options.LoginPath = new PathString("/Security/Login");
    }
  );
```

This creates a middleware component to catch the request, you need to put it as early as you can in the pipeline.

in configure

```
app.UseAuthentication();
```

This tells MVC to use it.

The middleware automatically handles the authentication if there is a cookie. The following handles the Log in and Log out scenarios.

Security pt 2

udemy
.Net Core 2

in the login method...

```
if(!IsAuthentic(username, password))
  return(....ok....);
```

Quick check to make sure the username and password are valid - if not reject.

Now we do the actually logging in - creating of the cookie

```
// Create claims
List<Claim> claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, "Bob Rich"),
    new Claim(ClaimTypes.Email, inputModel.Username)
};

// Create identity
ClaimsIdentity identity = new ClaimsIdentity(claims, "cookie");

// Create principal
ClaimsPrincipal principal = new ClaimsPrincipal(identity);

// Sign-in - save a cookie for next time
await HttpContext.SignInAsync
(
  scheme: "SaSecurityScheme",
  principal: principal,
  properties: new AuthenticationProperties
  {
    //IsPersistent = true, // for 'remember me' feature
    //ExpiresUtc = DateTime.UtcNow.AddMinutes(15)
  }
);
return Redirect(inputModel.RequestPath ?? "/");

}

public async Task<IActionResult> Logout(string requestPath)
{
  await HttpContext.SignOutAsync( scheme: "SaSecurityScheme");
}
```

Protect the actions you want protecting...

```
[Authorize]
public IActionResult Index()
{
 return View();
}
[Authorize]
public IActionResult Logout()
{
 return .....
}
[AllowAnonymous]
public IActionResult Login()
{
 return ...
}
```

[Authorize] will ensure that the requested user is authenticated and the action is authorised. Logout may want to be protected - to make sure that you can only logout if you are logged in. Also - Login needs to be accessed by non-authenticated users. You can make any action 'Allow Anonymous' - help pages or contact pages for example.

Security pt 3

udemy
.Net Core 2

Claim - determines what the subject is, not what they can do - ie. cooked, cookey@cookey.karoo.co.uk

Policy - this what a definition of what can be done - it is the permission rule, eg. isDeveloper, isAManager.

Claims Check - this is added to a controller or action to state what claim the requester must have to perform this action. This can be simply checking that the claim exists or that it has a specific value.

Adding Policies
Policies (rules) are added in the ConfigureServices when you add authorisations...

```
services.AddAuthorization
(
  options =>
  {
    options.AddPolicy
    (
      "Managers",
      policy => policy.RequireClaim("EmployeeId","10","20","30","33")
    );
    options.AddPolicy
    (
      "Developers",
      policy => policy.RequireClaim("EmployeeId","10","20","40","50")
    );
  }
);
```

So here - if an action has the "Managers" policy on - then the employeeid must be 10, 20, 30 or 33.

If an action has "Developers" policy on - then the employeeid must be 10, 20, 40 or 50. If an action has two policies attached - then the logical operator 'AND' is applied.

```
[Authorize(Policy = "Manager")]
public class ManagerController : Controller
{
  public IActionResult Manager() { return View(); }

  [Authorize(Policy = "CanGiveBonus")]
  public IActionResult GiveBonus() { return View(); }
}
```

So here - 'Manager' is only allowed if the policy "Manager" has been satisfied. The GiveBonus is only allowed if the policies "Manager" AND "CanGiveBonus" have been satisfied.

Security pt 4
custom policy

udemy
.Net Core 2

Writing your own policy is quite simple.

Policy Class - to hold the any relevant info regarding the policy - in our MinimumAgeRequirement it will be the minimum age...

```
namespace SaAuthorizationClaim.Auth
{
  public class MinimumAgeRequirement : IAuthorizationRequirement
  {
    public int MinimumAge { get; set; }
    public MinimumAgeRequirement(int minimumAge)
    {
      MinimumAge = minimumAge;
    }
  }
}
```

Policy Handler Class - to execute the policy logic

```
namespace SaAuthorizationClaim.Auth
{
  public class MinimumAgeRequirementHandler : AuthorizationHandler<MinimumAgeRequirement>
  {
    protected override Task HandleRequirementAsync
    (
      AuthorizationHandlerContext context,
      MinimumAgeRequirement requirement
    )
    {
      if (!context.User.HasClaim
           (c => c.Type == ClaimTypes.DateOfBirth &&
                 c.Issuer == "http://wine.com"))
         return Task.CompletedTask;

      var dateOfBirth = Convert.ToDateTime(
        context.User.FindFirst(
          c =>
            c.Type == ClaimTypes.DateOfBirth &&
            c.Issuer == "http://wine.com").Value);
      int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
      if (calculatedAge >= requirement.MinimumAge)
        context.Succeed(requirement);
      return Task.CompletedTask;
    }
  }
}
```

Create Policy and add Required Claims

```
services.AddAuthorization
(
  options =>
  {
    options.AddPolicy
    (
      "AtLeast21",
      policy => policy.Requirements.Add
      (
        new MinimumAgeRequirement(21)
      )
    );
  }
);
services.AddScoped<IAuthorizationHandler, MinimumAgeRequirementHandler>();
```

Attaching Policy to action

```
[Authorize(Policy = "AtLeast21")]
public class WineController : Controller
{
  public IActionResult Login() => View();
  public IActionResult Logout() => View();
}
```

HTTPS

udemy
.Net Core 2

HTTPS is Hyper Text Transfer Protocol Secure - it is SSL (Secure Socket Layer) over HTTP.

To enable it simply add the following to configureServices...

```
services.AddMvc
(
  options =>
  {
    // HTTPS for IIS Express, you can enable SSL from project
    // Properties > Debug tab by checking the Enable SSL flag.
    options.Filters.Add(new RequireHttpsAttribute());
  }
);
```

And add the following to Configure

```
var options = new RewriteOptions()
  .AddRedirectToHttps(StatusCodes.Status301MovedPermanently, 44377);
app.UseRewriter(options);
```