Build a simple API

Prerequisites

- Download POSTman OR Fiddler

Setting up the MVC API project

1. Use the instructions in Getting Started to setup an Empty Web Application.

2. Add `Microsoft.AspNetCore.Mvc.Core` to `project.json`:

   ```
   "dependencies": {
     "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
     "Microsoft.AspNetCore.Mvc.Core": "1.0.0",
   },
   ```

3. In `Startup.cs` add `services.AddMvcCore()` to `ConfigureServices` and add `app.UseMvc()` to `Configure`:

   ```
   public void ConfigureServices(IServiceCollection services)
   {
       services.AddMvcCore();
   }

   public void Configure(IApplicationBuilder app)
   {
       app.UseMvc();
   }
   ```

4. Create a folder called `Models` and create a class called `Product` in that folder:

   ```
   public class Product
   {
       public int Id { get; set; }
       public string Name { get; set; }
   }
   ```

5. Create a folder called `Controllers` and create a class called `ProductsController` in that folder.

6. Add an attribute route `[Route("/api/[controller]")]` to the `ProductsController` class:

   ```
   [Route("/api/[controller]")]
   public class ProductsController
   {
   }
   ```

7. Add a `Get` method to `ProductsController` that returns a `string` "Hello API World" with an attribute route

   ```
   [Route("/api/[controller]")]
   public class ProductsController
   {
     [HttpGet]
     public string Get() => "Hello World";
   ```

```
}
```

8. Run the application and navigate to `/api/products`, it should return the string "Hello World".

Returning JSON from the controller

1. Add the `Microsoft.AspNetCore.Mvc.Formatters.Json` to `project.json`:

   ```
   "dependencies": {
     "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
     "Microsoft.AspNetCore.Mvc.Core": "1.0.0",
     "Microsoft.AspNetCore.Mvc.Formatters.Json": "1.0.0"
   },
   ```

2. Configure MVC to use the JSON formatter by changing the `ConfigureServices` in `Startup.cs` to use the following:

   ```
   public void ConfigureServices(IServiceCollection services)
   {
       services.AddMvcCore()
           .AddJsonFormatters();
   }
   ```

3. Add a static list of projects to the `ProductsController`:

   ```
   public class ProductsController : ControllerBase
   {
       private static List<Product> _products = new List<Product>(new[] {
           new Product() { Id = 1, Name = "Computer" },
           new Product() { Id = 2, Name = "Radio" },
           new Product() { Id = 3, Name = "Apple" },
       });
       ...
   }
   ```

4. Change the `Get` method in `ProductsController` to return `IEnumerable<Product>` and return the list of products.

   ```
   public IEnumerable<Product> Get()
   {
       return _products;
   }
   ```

5. Run the application and navigate to `/api/products`. You should see a JSON payload of with all of the products.

6. Make a POST request to `/api/products`, you should see a JSON payload with all products.

7. Restrict the `Get` method to respond to GET requests by adding an `[HttpGet]` attribute to the method:

   ```
   [HttpGet]
   public IEnumerable<Product> Get()
   {
       return _products;
   }
   ```

Add a method to Get a single product

1. Add a `Get` method to the `ProductsController` that takes an `int id` parameter and returns `Product`.

   ```
   public Product Get(int id)
   {
       return _products.SingleOrDefault(p => p.Id == id);
   }
   ```

2. Add an `HttpGet` route specifying the `id` as a route parameter:

```
[HttpGet("{id}")]
public Product Get(int id)
{
    return _products.SingleOrDefault(p => p.Id == id);
}
```

3. Run the application, and navigate to `/api/products/1`, you should see a JSON response for the first product.

4. Navigate to `/api/products/25`, it should return a 204 status code.

5. Change the `Get` method in the `ProductsController` to return a 404 if the product search returns null.

   ```
   [HttpGet("{id}")]
   public IActionResult Get(int id)
   {
       var product = _products.SingleOrDefault(p => p.Id == id);

       if (product == null)
       {
           return NotFound();
       }

       return Ok(product);
   }
   ```

6. Run the application and navigate to `/api/products/40` and it should return a 404 status code.

Adding to the list of products

1. Add a `Post` method to `ProductsController` the takes a `Product` as input and adds it to the list of products:

   ```
   public void Post(Product product)
   {
       _products.Add(product);
   }
   ```

2. Add an `[HttpPost]` attribute to the method to constrain it to the POST HTTP verb:

   ```
   [HttpPost]
   public void Post(Product product)
   {
       _products.Add(product);
   }
   ```

3. Add a `[FromBody]` to the `product` argument:

```
[HttpPost]
public void Post([FromBody]Product product)
{
    _products.Add(product);
}
```

4. Run the application and post a JSON payload with the `Content-Type` header `application/json` to `/api/products`:

   ```
   {
     "Id" : "4",
     "Name": "4K Television"
   }
   ```

5. Make a GET request to `/api/products` and the new entity should show up in the list.

6. Change the `Post` method to return an `IActionResult` with a 201 status code and the added `Product`:

   ```
   [HttpPost]
   public IActionResult Post([FromBody]Product product)
   {
       _products.Add(product);
       return CreatedAtAction(nameof(Get), new { id = product.Id }, product);
   }
   ```

7. Add another product to the list by posting to `/api/products`:

   ```
   {
     "Id": "5",
     "Name": "Radio"
   }
   ```

8. It should return a 201 and the `Product` that was added as JSON.

Add model validation

1. Add the `Microsoft.AspNetCore.Mvc.DataAnnotations` package to `project.json`:

   ```
   "dependencies": {
     "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
     "Microsoft.AspNetCore.Mvc.Core": "1.0.0",
     "Microsoft.AspNetCore.Mvc.Formatters.Json": "1.0.0",
     "Microsoft.AspNetCore.Mvc.Formatters.Xml": "1.0.0",
     "Microsoft.AspNetCore.Mvc.DataAnnotations": "1.0.0"
   },
   ```

2. In `Startup.cs` add a call to `AddDataAnnotations()` chained off the `AddMvcCore` method in `ConfigureServices`:

   ```
   public void ConfigureServices(IServiceCollection services)
   {
       services.AddMvcCore()
           .AddJsonFormatters()
           .AddDataAnnotations();
   }
   ```

3. Modify the `Product.cs` file and add a `[Required]` attribute to the name property:

```
public class Product
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }
}
```

4. In `ProductsController.cs` modify the `Post` method and add a `ModelState.IsValid` check. If the model state is not valid, return a 400 response to the client:

```
[HttpPost]
public IActionResult Post([FromBody]Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest();
    }

    _products.Add(product);
    return CreatedAtAction(nameof(Get), new { id = product.Id }, product);
}
```

5. POST an empty JSON payload to `/api/products` and it should return a 400 response:

```
{}
```

6. Modify the `Post` method to return the validation errors to the client by passing the `ModelState` object to the `BadRequest` method:

```
[HttpPost]
public IActionResult Post([FromBody]Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    _products.Add(product);
    return CreatedAtAction(nameof(Get), new { id = product.Id }, product);
}
```

7. POST an empty JSON payload to `/api/products` and it should return a 400 response with the validation errors formatted as JSON.

Adding XML support

1. Add the `Microsoft.AspNetCore.Mvc.Formatters.Xml` package to `project.json`:

```
"dependencies": {
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  "Microsoft.AspNetCore.Mvc.Core": "1.0.0",
  "Microsoft.AspNetCore.Mvc.Formatters.Json": "1.0.0",
  "Microsoft.AspNetCore.Mvc.Formatters.Xml": "1.0.0"
},
```

2. In `Startup.cs` add a call to `AddXmlDataContractSerializerFormatters()` chained off the `AddMvcCore` method in `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvcCore()
        .AddJsonFormatters()
        .AddXmlDataContractSerializerFormatters();
}
```

3. Run the application and make a request to `/api/products` with the accept header `application/xml`. The response should be an XML payload of the products.

Restrict the ProductsController to be JSON only

1. Add a `[Produces("application/json")]` attribute to the `ProductsController` class:

```
[Produces("application/json")]
[Route("/api/[controller]")]
public class ProductsController : ControllerBase
```

Extra

- Add model validation when the product has a missing Name (and return that back to the client)
- Make the JSON properties camel case
- Write a custom output formatter to prints the product name as plain text
- Replace the static list of products with entity framework in memory store
- Replace the static list with entity framework + sqlite