Build an API in ASP.net

Preperations

For this exercise we are going to load our data from a local file, so we need to set up our application so we read static files.

1: Create a new application

Create a new dotnet application using the webapi template:

If you are using VSCODE

```
dotnet new webapi -o fullapidemo
```

Open the project in VSCode.

If you are using Visual Studio

Open Visual Studio and create a new project.

Call it **fullapidemo** and select the API template.

2: Add the middleware function

In your **Startup.cs** add the this line at the bottom:

`app.UseStaticFiles();` right under the `app.UseAuthorization();` line.

These methods are middleware, which are functions that add extra functionality.

3: Add the folder wwwroot for static files

Next we need to create a folder called `wwwroot` in our root directory and in there we create a file called **data.json**

In there type in this code:

```
[
  {
    "Name": "Jeff Kranenburg",
    "Email": "jeff.kranenburg@gmail.com"
  },
  {
    "Name": "Marisa Robertson",
    "Email": "marisa.robertson@apple.com"
  },
  {
    "Name": "John Appelseed",
    "Email": "john.appleseed@apple.com"
```

```
  }
]
```

4: Create a model class

Create a file called **Logins.cs** and add the same name space as your other files + .Models at the end. For example **Fullapidemo.Models**

Add the following content to this class:

```
public class Login
{
    public string Name { get; set; }
    public string Email { get; set; }

    public Login(string name, string email)
    {
        Name = name;
        Email = email;
    }
}
```

5a: Modifying the Controller

```
// Need to setup the list of the models
static List<Login> logins = new List<Login>();

// IHostingEnvironment allows us to access local paths
private readonly IHostingEnvironment _hostingEnvironment;

// Constructor to initialise the hostingEnvironment
public ValuesController(IHostingEnvironment hostingEnvironment)
{
    _hostingEnvironment = hostingEnvironment;
}
```

5b: Get Method for all data

For each of the methods we need to link to the data.json file. The `path` variable links to this using the hostingEnvironment object.

For the get read methods we need to use the `StreamReader` object to read the contents of the file.

We can store all of the information in a string variable, that I have called `json` below.

`logins` is the static list that we created at the top.

```
[HttpGet]
public ActionResult<IEnumerable<Login>> Get()
{
    string path= ($"{_hostingEnvironment.WebRootPath}/data.json");

    using (StreamReader r = new StreamReader(path))
    {
        string json = r.ReadToEnd();
        logins = JsonConvert.DeserializeObject<List<Login>>(json);
        return logins;
    }
}
```

5c: Get Method for a single ID

This is very much the same as above, the return statement only returns a single item.

```
[HttpGet("{id}")]
public ActionResult<Login> Get(int id)
{
    string path= ($"{_hostingEnvironment.WebRootPath}/data.json");

    using (StreamReader r = new StreamReader(path))
    {
        string json = r.ReadToEnd();
        logins = JsonConvert.DeserializeObject<List<Login>>(json);
        return logins[id];
    }
}
```

5d: Post new Data

To add data to the list, you need to load the json file into the path variable.

The parameter passed into the method comes in when a user fills in a from or it is that triggers this method. That data is then added to the List called logins.

Now that we have changed the logins list, we need to serialise this datat back into json and then write the new data to our **data.json** file

```
public IActionResult Post(Login login)
{
    string path= ($"{_hostingEnvironment.WebRootPath}/data.json");

    logins.Add(login);

    string stringify = JsonConvert.SerializeObject(logins,
Formatting.Indented);
```

```
    using (StreamWriter sw = new StreamWriter(path))
    {
        sw.WriteLine(stringify);
    }

    return Content("Content Added Successfully");
}
```

5e: Update some data

The PUT option, is used to update a current record.

Again we are setting the path variable.

Next we need to need to select the index of the list we want to update and assign the new login object to that index, so it overwrites that information only.

Then we serialise the data and write it back to our **data.json** file

```
public IActionResult Put(int id, Login login)
{
    string path= ($"{_hostingEnvironment.WebRootPath}/data.json");

    logins[id] = login;

    string stringify = JsonConvert.SerializeObject(logins,
Formatting.Indented);

    using (StreamWriter sw = new StreamWriter(path))
    {
        sw.WriteLine(stringify);
    }

    return Content("Content Updated Successfully");
}
```

5f: Delete a record

To remove a record from the list, we only need the ID.

So once we have that, we need to read the data from the json file and then remove the item that is related to the ID that got passed in.

Once the item has been removed the list needs to be serialised again and the new content written to the file.

```
[HttpDelete("{id}")]
public IActionResult Delete(int id)
{
```

```
    string path= ($"{_hostingEnvironment.WebRootPath}/data.json");

    using (StreamReader r = new StreamReader(path))
    {
        string json = r.ReadToEnd();
        List<Login> items = JsonConvert.DeserializeObject<List<Login>>
(json);
        items.RemoveAt(id);

        string stringify = JsonConvert.SerializeObject(items,
Formatting.Indented);

        using (StreamWriter sw = new StreamWriter(path))
        {
            sw.WriteLine(stringify);
        }
    }

    return Content("Content Removed Successfully");
}
```