Introducción al Routing & MVC

MVC 101

Crear una nueva ASP.NET Core Web Application

1. Abrir Visual Studio 2019
2. Crear una nueva ASP.NET Core application:
3. File -> New -> Project -> ASP.NET Core Web Application (C#)
4. Despues de nombrar el proyecto, seleccionen la versión de .NET Core y ASP.NET Core 2.2 en los dropdowns
5. Seleccionen `Web Application (Model-View-Controller)`, sin autenticación y sin https

Scaffolding-Generar un controlador completo a partir de un nuevo modelo

1. Click derecho en el directorio `Models` y seleccionar `Add`, y luego seleccionar `Class`. Nombren la nueva clase como `Persona.cs` con los siguientes atributos:

```
public class Persona
{
    public int ID { get; set; }
    public string Nombre { get; set; }
    public int Edad { get; set; }
}
```

2. Click derecho en el directorio `Controllers` y seleccionen Add -> Controller -> MVC Controller with views, Using Entity Framework.

3. Seleccionar `Persona` as the **Model Class**.

4. Crear un nuevo **Data context class** dandole click al botón `+`. Lo pueden nombrar `PersonaController`

5. Click **Add** para que genere el control y las vistas de forma automática en base al modelo Persona.

6. Revisar el archivo `appsettings.json` en `ConnectionStrings`, tomen nota del nombre de la base de datos.

7. Desde el `SQL Server Object Explorer`, en `(localdb)`, luego en `Databases`, click derecho a la opción `Add New Database` y le dan el mismo nombre que tomaron nota en el paso anterior.

Correr Migraciones de EF para actualizar la base de datos con el nuevo modelo Person

1. Mostrar la ventana del **Package Manager Console** seleccionando View -> Other Windows -> Package Manager Console
2. Crear una nueva migración para el modelo Person class tipeando en la consola `Add-Migration "Persona class"`
3. Actualice la base de datos utilizando esta migración escribiendo `Update-Database`
4. Ejecute la aplicación y navegue a `/Persona` desde la raíz de su aplicación (e.g. `http://localhost:61496/Persona`)
5. Intenta crear una persona con una edad no válida (e.g. "morada") para observar que la validación de formularios funciona.
6. Cree una persona válida ingresando un nombre y una edad entera.

> **Note:** Una vez termines la actividad el código completo lo subiré a /Labs/Code/Lab2A.

Extra

1. Añadir un Validation Attribute al modelo `Persona` para que la edad quede en un rango entre 1 y 120. *cuando hagan esto, recuerden añadir y correr una nueva migración como en el paso anterior.*
2. Usando Attribute Routing, una ruta adicional, que le permita navegar a la accción `Index` del controlador `PersonaController` usando `http://localhost:[port]/Everyone`.

Scaffolding-Generar un controlador completo a partir de un nuevo modelo

1. Click derecho en el directorio `Models` y seleccionar `Add`, y luego seleccionar `Class`. Nombren la nueva clase como `Producto.cs` con los siguientes atributos:

```
public class Producto
{
    public int ID { get; set; }
    public string Nombre { get; set; }
    public string Descripcion { get; set; }
    public int Stock { get; set; }
}
```

2. Click derecho en el directorio `Controllers` y seleccionen Add -> Controller -> MVC Controller with views, Using Entity Framework.
3. Seleccionar `Producto` as the **Model Class**.
4. Crear un nuevo **Data context class** dandole click al botón `+`.
5. Click **Add** para que genere el control y las vistas de forma automática en base al modelo Producto. üeden llamar al contolador `ProductoController`
6. Revisar el archivo `appsettings.json` en `ConnectionStrings`, tomen nota del nombre de la base de datos.
7. Desde el `SQL Server Object Explorer`, en `(localdb)`, luego en `Databases`, click derecho a la opción `Add New Database` y le dan el mismo nombre que tomaron nota en el paso anterior.

Correr Migraciones de EF para actualizar la base de datos con el nuevo modelo Person

1. Mostrar la ventana del **Package Manager Console** seleccionando View -> Other Windows -> Package Manager Console
2. Crear una nueva migración para el modelo Person class tipeando en la consola `Add-Migration "Producto class"`
3. Actualice la base de datos utilizando esta migración escribiendo `Update-Database`
4. Ejecute la aplicación y navegue a `/Producto` desde la raíz de su aplicación (e.g. `http://localhost:61496/Producto`)
5. Intenta crear un producto con un stock no válido (e.g. "morada") para observar que la validación de formularios funciona.
6. Cree un producto válido.

> **Note:** Una vez termines la actividad completa, ya deberían tener una noción de como funciona el tema del scaffodling en un proyecto MVC con .Net Core.

Agregar un Controlador de Web API

1. Click derecho en el directorio `Controllers` y seleccionen Add -> Controller y seleccionen `API Controller with read/write Actions` y le dan el nombre `ServicioController.cs`
2. Añaden el espacio de nombres `using LabMVC.Models;` al inicio del código del controlador.
3. Reemplacen el método `Get` con la siguiente pieza de código:

```
        // GET: api/Servicio
        [HttpGet]
```

```
        public List<Product> Get()
        {
            var productList = new List<Product>();
            var product1 = new Product
            {
                Name = "Arroz",
                Description = "Grano que se usa bastante en diversos platos"
            };
            var product2 = new Product
            {
                Name = "Mantequilla",
                Description = "Se usa bastante en diversos platos"
            };
            productList.Add(product1);
            productList.Add(product2);
            return productList;
        }
```

4. Corran la aplicación y vayan a la url `http://localhost:puerto/api/servicio`. Como verán la respuesta es transformada a JSON sin añadir código extra.
5. Vamos ir a nuestro controlador `ServicioController.cs` y vamos a reemplazar la cabecera `Route` por la siguiente `[Route("coolapi/products")]`
6. Corran la aplicación y vayan a la url `http://localhost:puerto/coolapi/products` para ver los resultados.

Extra

1. Descargar Postman `https://www.getpostman.com/downloads/`
2. En el archivo `Startup.cs` reemplazen la instrucción de `services.AddMvc()` con la siguiente:

```
services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2).AddXmlSe
rializerFormatters();
```

3. Con dicha instrucción podrán negociar el tipo de respuesta por JSON o por XML
4. Abran Postman y creen una nueva petición a la url `http://localhost:puerto/api/servicio`
5. En la pestaña de `Headers` añaden como KEY `Accept` y como VALUE `application/xml`.
6. Si le dan click al botón azul `Send` verán que la respuesta ahora viene en formato XML

Consumiendo nuestra API recien creada con HttpClient

1. Modificando el controlador de Producto `ProductoController`, vamos a añadir las siguientes instrucciones `using`:

```
using System.Net.Http;
using System.Net.Http.Headers;
```

2. Vamos a añadir el siguiente método, para consumir nuestra API creada en el paso anterior:

```
        public async Task<IActionResult> getProducts()
        {
            client.DefaultRequestHeaders.Accept.Clear();
            client.DefaultRequestHeaders.Accept.Add(
                new MediaTypeWithQualityHeaderValue("application/json"));
            HttpResponseMessage response = await client.GetAsync(new
Uri("http://localhost:57070/coolapi/products"));
            var productList = new List<Product>();
            if (response.IsSuccessStatusCode)
```

```
            {
                productList = await response.Content.ReadAsAsync<List<Product>>();
            }
            return View(productList);
        }
```

3. Fíjense como mapeamos de forma inmediata la respuesta JSON de la web api, a lista de objetos, en la linea `productList = await response.Content.ReadAsAsync<List<Product>>();`
4. Click derecho dentro del método o en `return View` y darle click a `Add View`
5. Dejar todo por defecto y darle click a `Add`. Esto generará la nueva vista para el método `getProducts`.
6. Reemplacen el código de la vista `getProducts`, con el siguiente:

```
@model IEnumerable<LabMVC.Models.Product>

@{
    ViewData["Title"] = "getProducts";
}

<h1>Obtengo productos desde mi API</h1>
<ul>
    @foreach (var cm in Model)
    {
        <li>@cm.Name</li>
    }
</ul>
```

7. Corran la aplicación y vayan a la url `http://localhost:puerto/products/getproducts`.
8. Fíjense en el resultado. La mayoría de las veces que quieran consumir una web api desde otra aplicación, lo van a hacer de forma muy similar a la que vimos.

Usando Javascript por el lado del Cliente

1. Vamos a añadir la librería javascript `Tabulator JS` a nuestra vista recién creada. Esta es una de muchas librerías, que nos vna a permitir crear una grilla en base a un JSON
2. Vamos a modificar la vista `getProducts`, añadiendo el siguiente código al final:

```
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/tabulator/4.4.1/css/tabulator.min.css"
/>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/tabulator/4.4.1/js/tabulator.js">
</script>
<div id="tabla-ejemplo"></div>

<script type="text/javascript">
    var table = new Tabulator("#tabla-ejemplo", {
    height:"311px",
    layout:"fitColumns",
    placeholder:"No Data Set",
    columns:[
        {title:"Nombre", field:"name", sorter:"string", width:200},
        {title:"Descripción", field:"description", sorter:"string"},
    ]
});

var jsonProductos = @Html.Raw(ViewBag.JsonProductos)
table.setData(jsonProductos);
</script>
```

3. Tomen nota de esta línea `var jsonProductos = @Html.Raw(ViewBag.JsonProductos)` donde estamos creando una variable Javascript en base al contenido de un `ViewBag`
4. Ahora vamos a modificar el controlador de Producto `ProductoController`, donde vamos a reemplazar el código del método `getProducts`, con el siguiente:

```
        public async Task<IActionResult> getProducts()
        {
            //client.BaseAddress = new
Uri("http://localhost:puerto/coolapi/products");
            client.DefaultRequestHeaders.Accept.Clear();
            client.DefaultRequestHeaders.Accept.Add(
                new MediaTypeWithQualityHeaderValue("application/json"));
            HttpResponseMessage response = await client.GetAsync(new
Uri("http://localhost:puerto/coolapi/products"));
            var productList = new List<Product>();
            string JsonProductos = "";
            if (response.IsSuccessStatusCode)
            {
                productList = await response.Content.ReadAsAsync<List<Product>>();
                JsonProductos = await response.Content.ReadAsStringAsync();
            }
            ViewBag.JsonProductos = JsonProductos;
            return View(productList);
        }
```

5. Si se fijan, añadimos una variable de tipo string `JsonProductos`, la cual estamos poblando usando el método `ReadAsStringAsync` dentro del `if`, para, antes de retornar el `View`, asignar `JsonProductos` a un ViewBag.
6. Corran la aplicación y vayan a la url `http://localhost:puerto/products/getproducts` para que vean el resultado del código javascript que añadimos en punto 2.

Documentar nuestra API, usando Swagger

1. Click derecho en nuestro proyecto, y seleccionen `Manage Nuget Packages...`
2. En la pestaña `Browse`, busquen el paquete `Swashbuckle.AspNetCore` e instalen la versión `4.0.1`
3. Ahora vamos a hacer 2 modificaciones a nuestro `Startup.cs`, primero en el método `ConfigureServices` vamos a añadir lo siguiente:

```
            services.AddSwaggerGen(c => {
                c.SwaggerDoc("v1", new Swashbuckle.AspNetCore.Swagger.Info
                {
                    Version = "Versión 1 API ANSES",
                    Description = "Documentación autogenerada para el workshop de
.Net Core"
                });

                //Locate the XML file being generated by ASP.NET...
                var xmlFile = $"
{Assembly.GetExecutingAssembly().GetName().Name}.XML";
                var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);

                //... and tell Swagger to use those XML comments.
                c.IncludeXmlComments(xmlPath);
            });
```

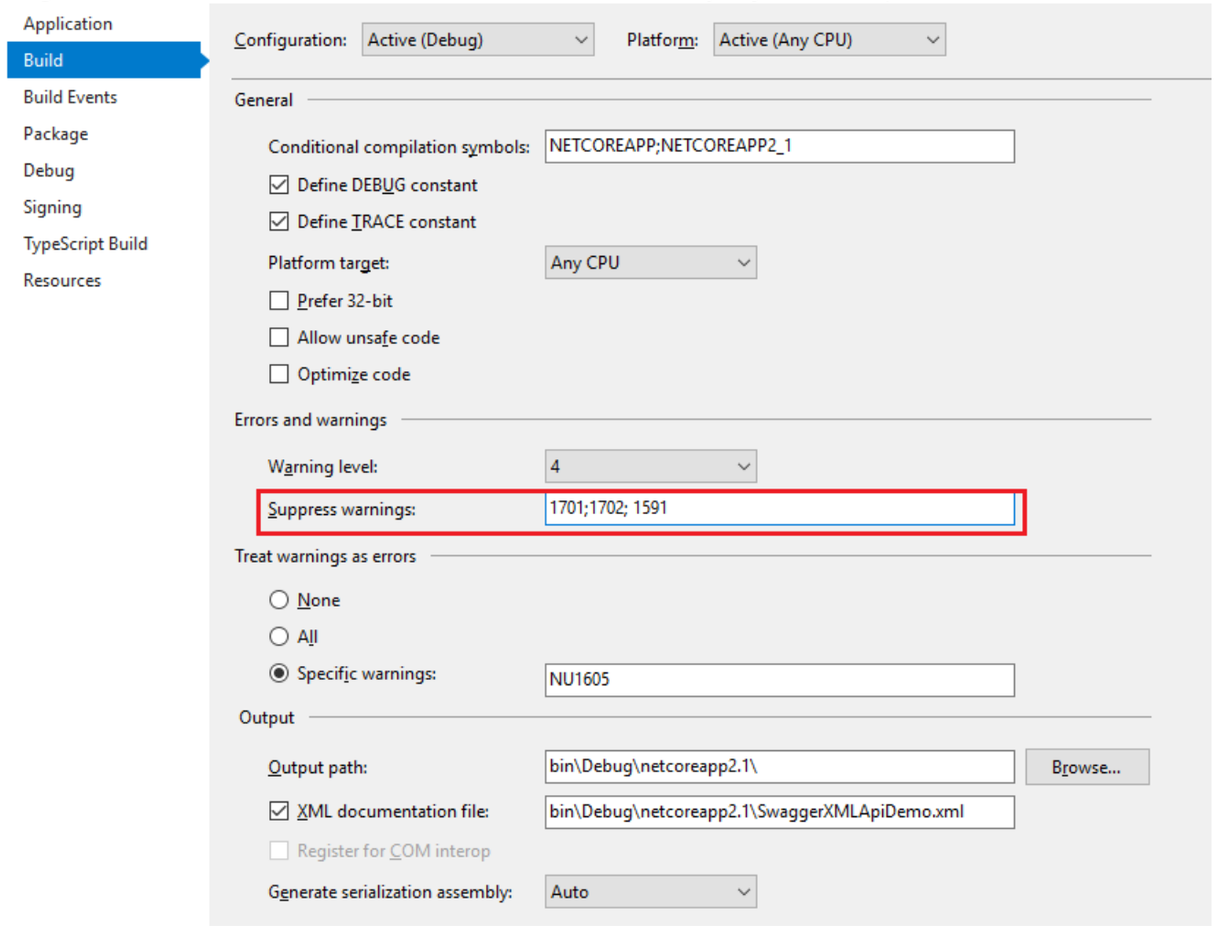
4. Ahora en el método `Configure`, añadimos lo siguiente:

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

5. Ahora añadiremos la siguiente cabecera a los controladores MVC(todos menos a nuestra api de Servicios) `[ApiExplorerSettings(IgnoreApi = true)]`, donde deberían quedar de la forma:

```
    [ApiExplorerSettings(IgnoreApi = true)]
    public class PersonaController : Controller
```

6. Vamos a las propiedades del Proyecto en la opción `Build`. En el textbox de `Supress warnings` añadan la `1591`
7. Luego seleccionen el checkbox de `XML Documentation File` y dejan el valor por defecto.



8. Añadan el siguiente comentario al método `Get` de nuestra api `ServicioController.cs`:

```
        /// <summary>
        /// Endpoint por defecto, retorna lista de productos
        /// </summary>
        /// <returns></returns>
```

7. Corran la aplicación y vayan a la url `http://localhost:puerto/swagger/` para ver el resultado de la actividad.

> **Note:** Felicitaciones por completar el WorkShop de .Net Core. Fue bastante contenido, pero ahora se quedan con las bases para comenzar a trabajar, el resto como siempre es investigación. Les dejo mi correo electrónico, para cualquier duda o consulta, no duden en escribirme robert.rozas@microsoft.com