UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN II

Unidad Temática N°5:
Web API CRUD

Material Teórico
1° Año – 2° Cuatrimestre

Web API CRUD

Introducción

En esta unidad vamos a ver cómo consumir desde una aplicación de escritorio (sobre .NET Framework) una API Rest desarrollada en C#.

Como sabemos, proporcionar una API Rest es una forma común de comunicación con aplicaciones Web. Por lo tanto, es frecuente que tengamos que interactuar con ellas desde nuestra aplicación.

Afortunadamente, comunicarnos con una API Rest desde una aplicación .Net es sencillo con la clase *HttpClient* que proporciona un mecanismo para enviar solicitudes HTTP y recibir respuestas HTTP de un recurso identificado por URI.

El objetivo de esta unidad es integrar los contenidos de las cuatro unidades anteriores mediante la implementación de aplicaciones de escritorio que soporten el uso de APIs Web para el procesamiento de datos y lógica de negocios.

CLIENTE HTTP

Objeto HttpClient

Como se mencionó en la introducción, HttpClient es una clase del paquete `System.Net.Http` que permite enviar solicitudes HTTP y recibir respuestas HTTP de un recurso identificado por un URI. EL siguiente código muestra un ejemplo sencillo de una aplicación de consola haciendo un pedido HTTP hacia el buscador *Google* e imprime el código de estado obtenido (si es 200 Google está activo):

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

namespace HttpClientStatus
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using(var client = new HttpClient()){
            var result = await client.GetAsync("https://www.google.com/");
              Console.WriteLine(result.StatusCode);
          }
        }
    }
}
```

Algunas consideraciones sintácticas para tener en cuenta:

- Las variables locales se pueden declarar sin dar un tipo explícito. La palabra clave **var** indica al compilador que infiera el tipo de expresión a partir de la expresión del lado derecho de una instrucción de inicialización.
- La sentencia **using** proporciona una sintaxis que asegura el uso correcto de los objetos *IDisposable.* La clase HttpClient implementa la interfaz IDisposable, lo que implica que nuestro objeto implementa el comportamiento Dispose() que permite liberar los recursos *administrados y no administrados* que usa el objeto en las solicitudes al servidor. Cuando se generan los pedidos HTTP se utilizan ciertos recursos relacionados con las conexiones TCP subyacentes (sockets) que tienen lugar en el servidor. Estos recursos puede que continúen abiertos aún cuando del lado del cliente ya no son necesarios. Esta forma de utilizar la creación del objeto HttpClient dentro del bloque using no nos garantiza siempre que el número de sockets que usa nuestra aplicación para las peticiones es el adecuado. Por lo que es buena práctica implementar el

patrón **Singleton** que nos permita tener un único objeto HttpClient en toda la aplicación.

- Otro aspecto para tener en cuenta es la **naturaliza asíncrona** de los pedidos HTTP. Cuando realizamos un pedido HTTP la respuesta no es inmediata y se procesa según los tiempos de respuesta del servidor. Por lo que es necesario utilizar el operador de espera await. Este operador suspende la evaluación del método asincrónico adjunto hasta que se complete la operación asincrónica representada por su operando. Cuando se completa la operación asincrónica, el operador await devuelve el resultado de la operación, si lo hubiera. Esto último implica que para poder usar un método asíncrono es necesario hacerlo desde un bloque marcado como **asíncrono**, por lo que se agrega al método *main* las palabras reservadas: **async Task** para indicarle al compilador esta situación.

GET con HttpClient

El método GET solicita una representación del recurso especificado. Para solicitar un recurso por método GET existen dos comportamientos distintos ofrecidos por HttpClient: GetAsync() y GetStringAsync(). El primero permite obtener información completa de la respuesta: contenido, código de estado, cabeceras, etc. Por su parte GetStringAsync(), es muy utilizado pero solo retorna el cuerpo de la respuesta en formato cadena.

Para analizar las diferencias vamos a utilizar la API Web desarrollada en la unidad anterior desde una aplicación de consola.

- Usando **GetAsync()**

```
class Program
    {
        static async Task Main(string[] args)
        {
            string url = "https://localhost:5001/api/Docentes";
            using (var client = new HttpClient())
            {
                var result = await client.GetAsync(url);
                var content = await
                result.Content.ReadAsStringAsync();
                Console.WriteLine(content);

            }
        }

    }
```

Notar que el puerto utilizado para la url es 5001. Para validar dicho valor revisar el archivo launchSettings.json dentro de la carpeta properties del proyecto Web API. Notar que tanto la llamada al método GetAsync(url) como al método ReadAsStringAsync() son asíncronas. Esta última permite obtener el cuerpo de la respuesta como una cadena. Mediante GetAsync() es posible acceder al contenido de la respuesta, al código de estado, cabeceras y demás datos del procesamiento de la petición HTTP.

- Usando **GetStringAsync()**

```
class Program
    {
        static async Task Main(string[] args)
        {
            string url = "https://localhost:5001/api/Docentes";
            using (var client = new HttpClient())
            {
                var content = await client.GetStringAsync(url);

                 Console.WriteLine (content);

            }
         }
    }
```

El GetStringAsync() envía una solicitud GET a la URI especificada y devuelve directamente el cuerpo de la respuesta como una cadena en una operación asíncrona.

Las dos formas anteriores permiten realizar un pedido GET y obtener la respuesta como una cadena. Si bien esto es muy útil, lo más común es obtener de la respuesta un objeto o colección de objetos recibidos en formato JSON. Para ello vamos a utilizar una librería NewtonSoft.Json, que se puede descargar desde *el Administrador de paquetes NuGet* como se muestra en la siguiente imagen:

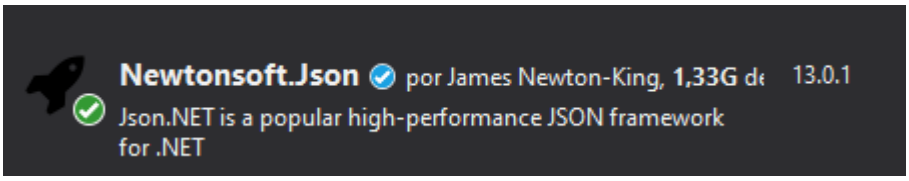

Imagen 1: Elaboración propia

Esta librería permite **serializar**, convertir en un objeto en una cadena en formato JSON y viceversa, a partir de una cadena en formato JSON construir un objeto o colección de objetos recibidos desde el servidor (**deserializar**). La clase JsonConvert tiene dos comportamientos estáticos: `SerializeObject` y

`DeserializeObject` que implementan los procesos mencionados. A continuación el código muestra cómo obtener una lista de objetos Docente desde el endpoint /api/docentes de la API desarrollada en la unidad anterior:

```
class Program
    {
        static async Task Main(string[] args)
        {
            string url = "https://localhost:5001/api/Docentes";
            using (var client = new HttpClient())
            {
                var result = await client.GetAsync(url);
                var content = await    result.Content.ReadAsStringAsync();
                List<Docente> lst =
                JsonConvert.DeserializeObject<List<Docente>>(content);
            }
         }
       }
```

Notar que el método **`DeserializeObject`** permite indicar mediate Generics<> el tipo de objeto a deserializar.

POST con HttpClient

El método HTTP POST envía datos al servidor. El tipo de cuerpo de la solicitud se indica en el encabezado Content-Type. El siguiente ejemplo envía un objeto docente a la API mediante el objeto HttpClient:

```
class Program
    {
        static async Task Main(string[] args)
        {
            string url = "https://localhost:5001/api/Docentes";
            using (var client = new HttpClient())
            {
              var json = JsonConvert.SerializeObject(new Docente() { Id =
              2, Nombre = "Carlos", Apellido = "Prueba" });
                result = await client.PostAsync(url, new
                StringContent(json, Encoding.UTF8, "application/json"));
            }
         }
       }
```

Tener en cuenta las siguientes consideraciones:

- JsonConvert.SerializeObject recibe un objeto y retorna una representación cadena en formato JSON de este.
- client.PostAsync() recibe dos parámetros: la url y un objeto que represente el contenido de la solicitud. Es posible enviar para este segundo parámetro cualquier clase concreta que herede de

HttpContent. Un objeto StringContent permite enviar como parte de la solicitud un objeto cadena con un sistema de codificación y tipos de contenido definidos.

Manejo de códigos de estado

Como mencionamos antes, si utilizamos GetAsync tenemos toda la información de la respuesta HTTP obtenida. Una propiedad muy útil es: IsSuccessStatusCode, que devuelve un valor booleano indicando si la respuesta HTTP ha sido exitosa. El siguiente código muestra cómo recuperar el cuerpo de la respuesta solo si el pedido ha sido exitoso:

```
class Program
    {
        static async Task Main(string[] args)
        {
            string url = "https://localhost:5001/api/Docentes";
            using (var client = new HttpClient())
            {
                var result = await client.GetAsync(url);
                if(result.IsSuccessStatusCode){
                    var content = await
                    result.Content.ReadAsStringAsync();
                    Console.WriteLine(content);
                }else{
                    Console.WriteLine("Ha ocurrido un error!");

                }

            }
        }

    }
```

Ahora bien, si se necesita implementar un mecanismo diferenciado de los posibles códigos de estado obtenidos es posible utilizar una estructura **switch** sobre la propiedad StatusCode, tal como se muestra en el siguiente código:

```
class Program
    {
        static async Task Main(string[] args)
        {
            string url = "https://localhost:5001/api/Docentes";
            using (var client = new HttpClient())
            {
                var result = await client.GetAsync(url);
                switch(result.StatusCode){
                    case HttpStatusCode.OK:
                        break;
                    case HttpStatusCode.NotFound:
                        break;
                    case HttpStatusCode.BadRequest:
```

```
                    break;

                //y así por cada valor definido en la enumeración
                //HttpStatusCode

            }

        }
    }
}
```

APLICACIÓN CLIENTE-SERVIDOR

Para integrar los contenidos de las todas unidades anteriores y cumplimentar con el objetivo general de la asignatura vamos a refactorizar nuestro caso de estudio pensando en una aplicación Cliente-Servidor donde ciertas ventanas cliente consumirán servicios de una Web API (servidor) que exponga las operaciones CRUD de la relación Maestro-Detalle del dominio. Para ello vamos a desarrollar una solución sobre la plataforma .NET que incluya los siguientes proyectos:

- Una biblioteca de código que represente una abstracción del negocio con la capa de acceso a datos incluida
- Una Web API que permita exponer dicha lógica de negocio
- Una aplicación de escritorio que permita consumir la API e implementar las funcionalidades necesarias para que un usuario pueda gestionar la relación Maestro-Detalle del dominio.

Caso de estudio: Carpintería cliente-servidor.

Convertir la aplicación de nuestro caso de estudio en una implementación cliente-servidor supone refactorizar la solución desarrollada en las unidades anteriores de tal manera que una aplicación cliente (proyecto Window Forms) consuma los servicios expuestos en un servidor (proyecto Web API). Toda la lógica de negocio quedará desarrollada en una librería de código (archivo .dll) que será utilizada por la Web API como una dependencia de proyecto.

Como las entidades de negocio son comunes tanto para el cliente como para el servidor, una vez creados los tres proyectos, será necesario definir correctamente las dependencias de compilación. Dado el siguiente esquema de solución (sugerido):

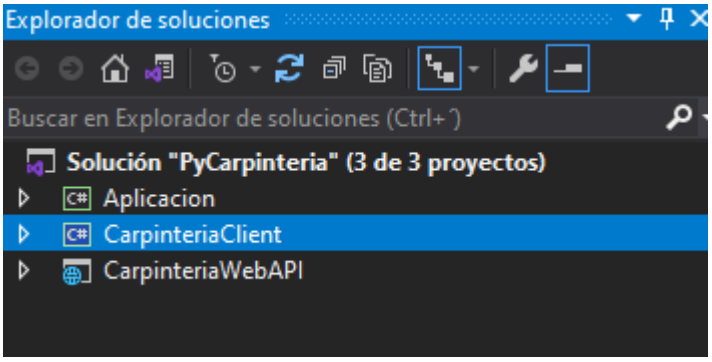

Imagen 2: Elaboración propia

es necesario definir las dependencias de cada proyecto. Para desplegar el proyecto y en el ítem **Dependencias** hacer click derecho y seleccionar la opción: *Agregar referencia del proyecto…*tal como se muestra en la siguiente imagen:

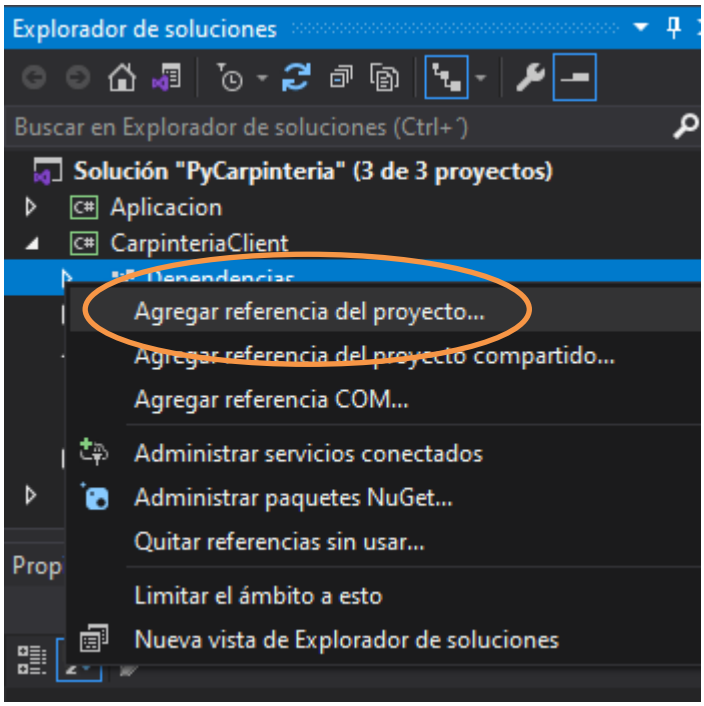

Imagen 3: Elaboración propia

Es nuestro caso tanto el proyecto CarpinteriaClient como CarpinteriaWebAPI tendrán como referencia al proyecto Aplicación. Como se menciona en el párrafo anterior esto es sugerido, el alumno puede optar por otro esquema de dependencias según considere necesario en función la solución planteada.

Consumir servicios desde un objeto Form

Para poder consumir los servicios expuestos en el proyecto CarpinteriaWebAPI es necesario utilizar el objeto HttpClient dentro los manejadores de eventos de las pantallas. Como se mencionó toda la aplicación cliente debería utilizar un único objeto HttpClient para evitar el uso innecesario de recursos en el servidor. Si recordamos lo estudiado en la unidad 2, un patrón creacional muy utilizado en el desarrollo de software es el *Singleton*, cuyo objetivo es delegar a una clase la creación de una única instancia propia. El siguiente código muestra un ejemplo de implementación de este patrón que permite generar peticiones GET y POST asíncronas que serán llamadas desde los eventos de formulario:

```
class ClientSingleton
{
    private static ClientSingleton instancia;
    private HttpClient client;

    private ClientSingleton()
    {
        client = new HttpClient();
    }


    public static ClientSingleton GetInstance()
    {
        if (instancia == null)
            instancia = new ClientSingleton();
        return instancia;
    }

    public async Task<string> GetAsync(string url)
    {
        var result = await client.GetAsync(url);
        var content = "";
        if (result.IsSuccessStatusCode)
            content = await result.Content.ReadAsStringAsync();
        return content;
    }

    public async Task<string> PostAsync(string url, string data)
    {
         StringContent content = new StringContent(data, Encoding.UTF8,
         "application/json");
        var result = await client.PostAsync(url, content);

        var response = "";
        if (result.IsSuccessStatusCode)
            response = await result.Content.ReadAsStringAsync();
        return response;
    }
   }
```

Algunas consideraciones:

- A modo de ejemplo, se implementan solo los métodos Get y Post. El resto de los verbos quedan a cargo de los estudiantes.
- Por convención, los nombres de los métodos se su fijan con **asinc**, para indicar la naturaleza asíncrona de su ejecución.
- Notar que tanto la llamada a `client.GetAsync()` como `result.Content.ReadAsStringAsync()` están precedidos por el operador **await**.
- Ambos métodos retornan como cadena la respuesta obtenida desde la Web API.

A modo de ejemplo, el código a continuación muestra como utilizar esta clase desde el formulario de alta de presupuestos diseñado previamente. Primero se obtiene el listado de productos que permite llenar el combo de la interfaz, el segundo permite enviar los datos completo de un presupuesto para su registración en la capa de persistencia.

- Desde el evento Load del formulario se llama a un método auxiliar llamado: `CargarComboAsync()`. Al ser un método asíncrono la llamada se precede del operador **await**, por lo que el método manejador del evento Load se indica como **asinc**.

```
private async void Frm_Alta_Presupuesto_Load(object sender,
EventArgs e)
        {
            if (modo.Equals(Accion.CREATE))
            {
                await CargarComboAsync();
                ConsultarUltimoPresupuesto();
                txtFecha.Text = DateTime.Now.ToString("dd/MM/yyyy");
                txtDto.Text = "0";
                txtCliente.Text = "CONSUMIDOR FINAL";
            }

        }

 private async Task CargarComboAsync()
        {
            string url = "https://localhost:44312/api/Productos";
            var data = await
           ClientSingleton.GetInstance().GetAsync(url);
            List<Producto> lst =
           JsonConvert.DeserializeObject<List<Producto>>(data);
            cboProducto.DataSource = lst;
            cboProducto.ValueMember = "IdProducto";
            cboProducto.DisplayMember = "Nombre";

        }
```

Notar que:

- Se llama al endpoint **`"https://localhost:44312/api/Productos"`**
- La llamada a **`ClientSingleton.GetInstance()`** permite obtener la única instancia de ClienteSingleton que usa el objeto HttpClient para hacer el pedido GET.
- La respuesta JSON se deserializa mediante **`JsonConvert.DeserializeObject<List<Producto>>(data)`**

- Desde el evento Click del botón Aceptar se llama al método auxiliar `GrabarPresupuestoAsync(oPresupuesto)`. Al igual que en el método anterior la llamada es asíncrona por lo que el método handler de evento se marca como asíncrono. El código se muestra a continuación:

```
private async Task<bool> GrabarPresupuestoAsync(Presupuesto
presupuesto)
        {
            string url =
"https://localhost:44312/api/Presupuestos";
            string presupuestoJson =
JsonConvert.SerializeObject(oPresupuesto);

            var result = await
ClientSingleton.GetInstance().PostAsync(url, presupuestoJson);
            return result.Equals("true");
        }
```

Notar que:

- Se llama al endpoint **`https://localhost:44312/api/Presupuestos`**
- **`JsonConvert.SerializeObject(oPresupuesto)`** permite serializar el objeto **oPresupuesto** para poder enviar como parte del cuerpo de la solicitud POST
- La respuesta obtenida como resultado del método **`PostAsync(url, presupuestoJson)`** se compara con la cadena “true” para obtener un valor booleano que permita validar la confirmación exitosa de la registración del presupuesto.

BIBLIOGRAFÍA

Microsoft. HttpClient Class. Recuperado de: https://docs.microsoft.com/

API Rest en C#. Recuperado de: https://www.netmentor.es/entrada/api-rest-csharp

Async and Await In C#. Recuperado de: https://www.c-sharpcorner.com/article/async-and-await-in-c-sharp/

Cómo manejar asincronismo en .NET. Recuperado de https://www.campusmvp.es/