

# **Tecnologías de Desarrollo de Software IDE**

**Implementación de un cliente WinForm  
que consuma una API Rest**

---

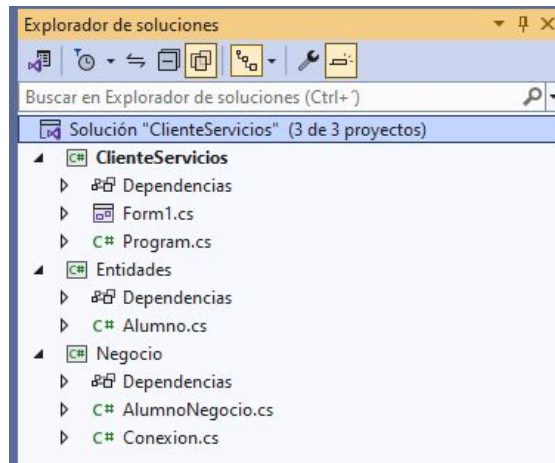
# Implementación de un cliente WinForm que consume una API Rest

Vamos a crear una nueva solución que consume la API Rest que creamos en la presentación anterior.

La misma va a contar con 3 proyectos:

- ClienteServicios va a ser el que tiene nuestro cliente WinForm. Va a tener dependencias a los proyectos Entidades y Negocio.
- Entidades: es igual a la que generamos en la solución anterior.
- Negocio: es el proyecto que va a consultar a los servicios. Va a tener una dependencia al proyecto entidades.

Comencemos!



# Implementación de un cliente WinForm que consume una API Rest

Creemos (o copiamos) el proyecto Entidades (del tipo Biblioteca de Clases). La clase Alumnos, dentro del proyecto Entidades es igual a la que teníamos en la solución anterior:

```
1  using System.ComponentModel.DataAnnotations;
2
3  namespace Entidades
4  {
5      11 referencias
6      public class Alumno
7      {
8          2 referencias
9          [Key]
10         public String DNI { get; set; }
11         0 referencias
12         public String ApellidoNombre { get; set; }
13         0 referencias
14         public String Email { get; set; }
15         0 referencias
16         public DateTime FechaNacimiento { get; set; }
17         0 referencias
18         public decimal NotaPromedio { get; set; }
19     }
20 }
```

# Implementación de un cliente WinForm que consuma una API Rest

Con esto ya listo vamos a crear el proyecto `Negocio`. El mismo va a ser una Biblioteca de clases. Como es el proyecto que va a consultar los servicios tenemos que agregarle una librería con NuGet:







- [Microsoft.AspNet.WebApi.Client](#)

Examinar


Instalado




Actualizaciones

Microsoft.AspNet.WebApi.Client x ↕ ☐ Incluir versión preliminar


	<b>Microsoft.AspNet.WebApi.Client</b>  por Microsoft, <b>458M</b> descargas This package adds support for formatting and content negotiation to System.Net.Http.	5.2.9
	<b>Microsoft.AspNet.WebApi</b>  por Microsoft, <b>147M</b> descargas This package contains everything you need to host ASP.NET Web API on IIS.	5.2.9
	<b>Microsoft.AspNet.WebApi.WebHost</b>  por Microsoft, <b>167M</b> descargas This package contains everything you need to host ASP.NET Web API on IIS.	5.2.9


Administrador de paquetes NuGet: ClienteServicios

Origen del paquete: nuget.org 

 **Microsoft.AspNet.WebApi.Client**  

Versión: Versión estable más reciente 5.2.9 Instalar

 La asignación del origen del paquete está desactivada. [Configurar](#)

 Opciones

Descripción

# Implementación de un cliente WinForm que consume una API Rest

Lo primero que vamos a hacer es crear una nueva clase: Conexión. Esta clase será la que generará el cliente con la configuración inicial. Después, iremos usándolo para llamar a los diferentes servicios usando los métodos que nos provee.

Este cliente va a ser de tipo `HttpClient`. En este código podemos ver que estamos usando el patrón Singleton para crear una única instancia del mismo y trabajar siempre con un único objeto.

```
1  using System.Net.Http.Headers;
2  namespace Negocio
3  {
4
5      9 referencias
6      public sealed class Conexion
7      {
8
9          1 referencia
10         private Conexion() { }
11         private static Conexion? instancia;
12         private HttpClient _Cliente = new HttpClient();
13
14         5 referencias
15         public HttpClient Cliente
16         {
17             get { return _Cliente; }
18         }
19
20         5 referencias
21         public static Conexion Instancia { get
22             { if (instancia == null)
23                 {
24                     instancia = new Conexion();
25                     instancia._Cliente.DefaultRequestHeaders.Accept.Clear();
26                     instancia._Cliente.DefaultRequestHeaders.Accept.Add(
27                         new MediaTypeWithQualityHeaderValue("application/json"));
28                 }
29             return instancia;
30         }
31     }
32 }
```

# Implementación de un cliente WinForm que consume una API Rest

Ahora crearemos una clase llamada `AlumnoNegocio`. En la misma vamos a crear nuestro primer método: `GetAll` para recuperar todos los Alumnos de la Base de Datos.

Este método (`GetAll`) va a ser un método asíncrono ya que vamos a estar consumiendo un servicio. Para esto usamos las palabras claves `async/await` y el método va a retornar un `Task` con la lista de `Alumno` recuperada.

```
1  using Entidades;
2  using Newtonsoft.Json;
3  namespace Negocio
4  {
5      4 referencias
6      public class AlumnoNegocio
7      {
8          1 referencia
9          public async static Task<IEnumerable<Alumno>> GetAll()
10         {
11             var response = await Conexion.Instancia.Cliente.GetStringAsync("https://localhost:7011/api/Alumno/");
12             var data = JsonConvert.DeserializeObject<List<Alumno>>(response);
13             return data;
14         }
15     }
```



# Implementación de un cliente WinForm que consume una API Rest

Como podemos ver estamos obteniendo un Cliente (que es un elemento static), llamando un método: GetStringAsync al cual le tenemos que pasar la URL en la que está escuchando nuestro servicio. Recordemos que en el json de configuración de nuestro otra Solución teníamos:

Acá podemos ver que el puerto 7011 es donde está escuchando nuestro servicio.

```
"profiles": {  
  "Ejemplo": {  
    "commandName": "Project",  
    "dotnetRunMessages": true,  
    "launchBrowser": true,  
    "launchUrl": "swagger",  
    "applicationUrl": "https://localhost:7011;http://localhost:5013",  
    "environmentVariables": {  
      "ASPNETCORE_ENVIRONMENT": "Development"  
    }  
  },  
}
```

# Implementación de un cliente WinForm que consume una API Rest

---

Además, en la configuración de nuestro servicio GetAll, la forma de invocarlo era a través de la ruta Alumno:

```
[HttpGet(Name = "Alumno")]  
0 referencias  
public ActionResult<IEnumerable<Alumno>> GetAll()  
{  
    return _context.Alumnos.ToList();  
}
```



# Implementación de un cliente WinForm que consume una API Rest

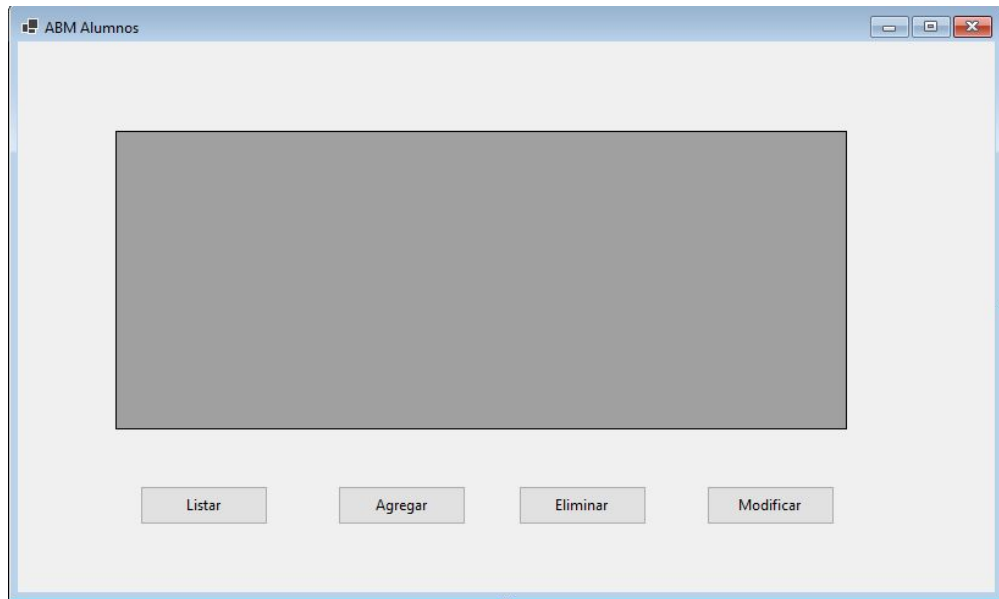
Luego, lo que tenemos es transformar ese Json que obtenemos como respuesta en objetos. Esto lo hacemos con la clase `JsonConvert` y el método `DeserializeObject`.

```
1  using Entidades;
2  using Newtonsoft.Json;
3  namespace Negocio
4  {
5      4 referencias
5      public class AlumnoNegocio
6      {
7          1 referencia
7          public async static Task<IEnumerable<Alumno>> GetAll()
8          {
9              var response = await Conexion.Instancia.Cliente.GetStringAsync("https://localhost:7011/api/Alumno/");
10             var data = JsonConvert.DeserializeObject<List<Alumno>>(response);
11             return data;
12         }
13     }
14 }
15 }
```

# Implementación de un cliente WinForm que consume una API Rest

Con esto ya listo. Vamos al proyecto ClienteServicios (de tipo Aplicación de Escritorio). En el mismo vamos a crear un Form con estos elementos:

- un DataGridView;
- cuatro Botones (button1, button2, button3, button4 respectivamente).



# Implementación de un cliente WinForm que consume una API Rest

Ahora vamos a generar el evento click del botón Listar. El código que vamos a tener es el siguiente:

1. creamos un método `cargarTabla`. Este método va a llamar al método `GetAll` de `AlumnoNegocio` y nos retornará la colección obtenida.
2. Desde el oyente del evento click vamos a llamar a este método creando un `Task`, iniciándolo y, una vez que sea recibido se cargue en el `DataSource` del `DataGridView`.

Observación: Como el `cargarTabla` llama a un método asíncrono no queremos quedar bloqueados esperando. Por esto es que se marca como `async` al oyente y usamos `await` para la carga del `DataSource`.

```
1 using Negocio;
2 using Entidades;
3
4 namespace ClienteServicios
5 {
6     3 referencias
7     public partial class Form1 : Form
8     {
9         private Task<IEnumerable<Alumno>>? l;
10
11         1 referencia
12         public Form1()
13         {
14             InitializeComponent();
15
16         1 referencia
17         public IEnumerable<Alumno> cargarTabla()
18         {
19             l = AlumnoNegocio.GetAll();
20             return l.Result;
21         }
22
23         1 referencia
24         private async void button1_Click(object sender, EventArgs e)
25         {
26             Task<IEnumerable<Alumno>> task = new Task<IEnumerable<Alumno>>(cargarTabla);
27             task.Start();
28             dataGridView1.DataSource = await task;
29         }
30     }
31 }
```



# Implementación de un cliente WinForm que consuma una API Rest

---

Una vez hecho esto podemos probar el funcionamiento de nuestra aplicación. Para esto tenemos que abrir la solución Servicios que hicimos previamente y la ejecutamos. Va a quedar corriendo.

Ahora sí, ejecutamos la aplicación WinForm.

Se va a abrir la ventana y, al presionar el botón Listar debería cargarse la grilla.



# Implementación de un cliente WinForm que consuma una API Rest

---

Lo que vamos a hacer ahora es agregar la funcionalidad a los oyentes de los clicks de los botones y, generaremos los métodos correspondientes en la capa de Negocio. Vamos a hacer el Eliminar.

El servicio Delete estaba implementado de la siguiente manera:

```
[HttpDelete("{DNI}")]  
0 referencias  
public ActionResult<Alumno> Delete(String DNI)
```

O sea, tenemos que pasar el DNI (clave primaria) del Alumno para poder eliminarlo.



# Implementación de un cliente WinForm que consume una API Rest

En la clase Negocio, agregamos:

1 referencia

```
public async static Task<Boolean> Delete(Alumno alumno)
{
    var response = await Conexion.Instancia.Cliente.DeleteAsync("https://localhost:7011/api/Alumno/" + alumno.DNI);
    return response.IsSuccessStatusCode;
}
```

El método que usamos con el Cliente es DeleteAsync y, a nuestra ruta le pasamos el DNI que recibimos desde la capa de Presentación.



# Implementación de un cliente WinForm que consume una API Rest

El oyente del evento va a ser:

```
private async void button3_Click(object sender, EventArgs e)
{
    int filaSeleccionada = dataGridView1.SelectedRows[0].Index;
    await AlumnoNegocio.Delete(1.Result.ToList()[filaSeleccionada]);
    button1_Click(sender, e);
}
```

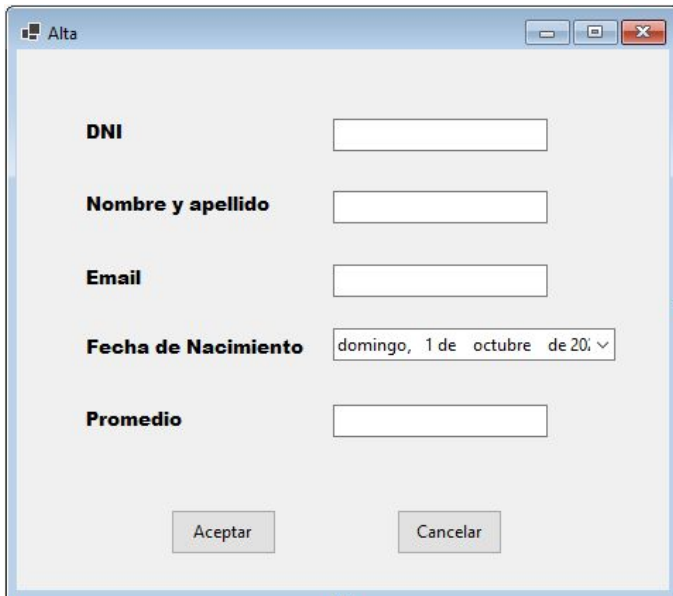
Tenemos que seleccionar la fila del Alumno que se quiere eliminar y, al presionar el botón Eliminar:

1. Recuperamos la fila elegida de la lista de Alumnos
2. Invocamos al método Delete de AlumnoNegocio pasándole el Alumno que se encuentra en esa posición.
3. Volvemos a llamar al oyente del botón Listar para que recargue el DataGridView.

# Implementación de un cliente WinForm que consuma una API Rest

Una vez hecho esto, vamos a ir por el Alta y la Modificación. Generaremos otro WinForm (que llamaremos Alta) y cuyo diseño debería ser similar al siguiente:

Ahora vamos a programar el oyente del botón Aceptar.



The screenshot shows a Windows Form titled "Alta". It contains five input fields with labels to their left: "DNI", "Nombre y apellido", "Email", "Fecha de Nacimiento", and "Promedio". The "Fecha de Nacimiento" field is a date picker showing "domingo, 1 de octubre de 20". At the bottom of the form are two buttons: "Aceptar" and "Cancelar".





# Implementación de un cliente WinForm que consume una API Rest

El servicio Create estaba implementado de la siguiente manera usando Post y recibiendo el alumno que se quería crear:

```
[HttpPost]  
0 referencias  
public ActionResult<Alumno> Create(Alumno alumno)
```

En la clase Negocio, agregamos:

```
public async static Task<Boolean> Add(Alumno alumno)  
{  
    var response = await Conexion.Instancia.Cliente.PostAsJsonAsync("https://localhost:7011/api/Alumno/", alumno);  
    return response.IsSuccessStatusCode;  
}
```

donde usamos el método PostAsJsonAsync para pasarle el alumno.

# Implementación de un cliente WinForm que consume una API Rest

Volviendo a nuestro WinForm (Alta), el evento del botón Aceptar quedaría así:

Se crea un Alumno y se inicializa con los valores de los TextBox y del DateTimePicker.

Luego de eso se llama al método Add de AlumnoNegocio.

Mientras que el de Cancelar simplemente destruye la ventana.

```
1 using Entidades;
2 using Negocio;
3
4 namespace ClienteServicios
5 {
6     3 referencias
6     public partial class Alta : Form
7     {
8         1 referencia
8         public Alta()
9         {
10             InitializeComponent();
11         }
12
13         1 referencia
13         private async void button1_Click(object sender, EventArgs e)
14         {
15             Alumno a = new Alumno();
16             a.DNI = textBox1.Text;
17             a.ApellidoNombre = textBox2.Text;
18             a.Email = textBox3.Text;
19             a.FechaNacimiento = dateTimePicker1.Value;
20             a.NotaPromedio = Convert.ToDecimal(textBox4.Text);
21             await AlumnoNegocio.Add(a);
22             Dispose();
23         }
24
25
26         1 referencia
26         private void button2_Click(object sender, EventArgs e)
27         {
28             Dispose();
29         }
30     }
31 }
```

# Implementación de un cliente WinForm que consume una API Rest

---

Finalmente, en nuestro Form principal, el oyente del botón Alta sería:

```
private void button2_Click(object sender, EventArgs e)
{
    new Alta().ShowDialog();
    button1_Click(sender, e);
}
```

Invoco al formulario Alta y, cuando vuelvo, ejecuto el botón Listar para recargar el DataGridView.



# Implementación de un cliente WinForm que consume una API Rest

Finalmente, vamos a crear el oyente del botón Modificar. Para esto, lo que vamos a hacer es pasarle al formulario Alta el Alumno que se quiere modificar (que fue seleccionado):

```
1 referencia
private void button4_Click(object sender, EventArgs e)
{
    int filaSeleccionada = dataGridView1.SelectedRows[0].Index;
    new Alta(1.Result.ToList()[filaSeleccionada]).ShowDialog();
    button1_Click(sender, e);
}
```

Luego de que se ejecute el formulario se llama al oyente del botón Listar para recargar el DataGridView.

# Implementación de un cliente WinForm que consume una API Rest

Agregamos un constructor en el formulario Alta que reciba el alumno que se quiere modificar. Lo cargue en los TextBox y el DateTimePicker y modifique el texto del Botón (Modificar): Finalmente, vamos a crear el oyente del botón Modificar.

```
1 referencia
public Alta(Alumno alumnoAModificar)
{
    InitializeComponent();
    button1.Text = "Modificar";
    textBox1.Text = alumnoAModificar.DNI;
    textBox1.Enabled = false;
    textBox2.Text = alumnoAModificar.ApellidoNombre;
    textBox3.Text = alumnoAModificar.Email;
    dateTimePicker1.Value = alumnoAModificar.FechaNacimiento;
    textBox4.Text = Convert.ToString(alumnoAModificar.NotaPromedio);
}
```



# Implementación de un cliente WinForm que consume una API Rest

El servicio Update estaba implementado de la siguiente manera:

```
[HttpPut("{DNI}")]  
0 referencias  
public ActionResult Update(string DNI, Alumno alumno)
```

O sea, tenemos que pasar el DNI (clave primaria) del Alumno y el nuevo Alumno para poder actualizarlo utilizando la acción Put.

En la clase Negocio, agregamos:

```
public async static Task<Boolean> Update(Alumno alumno)  
{  
    var response = await Conexion.Instancia.Cliente.PutAsJsonAsync("https://localhost:7011/api/Alumno/" + alumno.DNI, alumno);  
    return response.IsSuccessStatusCode;  
}
```

donde usamos el método PutAsJsonAsync para pasarle el alumno y la ruta incluye el DNI del alumno que se quiere modificar.

# Implementación de un cliente WinForm que consume una API Rest

Finalmente, modificamos el oyente del botón del formulario de Alta preguntando si es un Modificar o un Alta. En un caso llamamos al método Update de AlumnoNegocio y en el otro, como estábamos haciendo, al Add.

```
private async void button1_Click(object sender, EventArgs e)
{
    Alumno a = new Alumno();
    a.DNI = textBox1.Text;
    a.ApellidoNombre = textBox2.Text;
    a.Email = textBox3.Text;
    a.FechaNacimiento = dateTimePicker1.Value;
    a.NotaPromedio = Convert.ToDecimal(textBox4.Text);
    if (button1.Text == "Modificar")
    {
        await AlumnoNegocio.Update(a);
    }
    else { await AlumnoNegocio.Add(a); }
    Dispose();
}
```

# Implementación de un cliente WinForm que consuma una API Rest

Un pequeño cambio que podemos hacer es definir una constante string que tenga la URL del servicio y la pasamos en cada uno de los métodos de la capa Negocio:

```
static readonly string defaultURL = "https://localhost:7011/api/Alumno/";  
1 referencia  
public async static Task<IEnumerable<Alumno>> GetAll()  
{  
    var response = await Conexion.Instancia.Cliente.GetStringAsync(defaultURL);  
    var data = JsonConvert.DeserializeObject<List<Alumno>>(response);  
    return data;  
}  
0 referencias  
public async static Task<Alumno> GetOne(string DNI)  
{  
    var response = await Conexion.Instancia.Cliente.GetStringAsync(defaultURL + DNI);  
    var data = JsonConvert.DeserializeObject<Alumno>(response);  
    return data;  
}  
1 referencia  
public async static Task<Boolean> Add(Alumno alumno)  
{  
    var response = await Conexion.Instancia.Cliente.PostAsJsonAsync(defaultURL, alumno);  
    return response.IsSuccessStatusCode;  
}  
1 referencia  
public async static Task<Boolean> Update(Alumno alumno)  
{  
    var response = await Conexion.Instancia.Cliente.PutAsJsonAsync(defaultURL + alumno.DNI, alumno);  
    return response.IsSuccessStatusCode;  
}  
1 referencia  
public async static Task<Boolean> Delete(Alumno alumno)  
{  
    var response = await Conexion.Instancia.Cliente.DeleteAsync(defaultURL + alumno.DNI);  
    return response.IsSuccessStatusCode;  
}
```





# Implementación de un cliente WinForm que consume una API Rest

---

Con esto ya tendríamos un cliente WinForm funcional que consume los servicios que creamos en la presentación anterior.

Quedaría hacer el control de excepciones para evitar que la aplicación de escritorio se rompa. También podríamos recuperar los retornos de la capa de negocio para indicar si el Alumno fue modificado, creado, eliminado o actualizado satisfactoriamente.

En este caso, el cliente y los servicios corren en soluciones separadas. Podríamos hacer correr todo desde una única solución. ¿Se dan cuenta cómo?