

CREAR UN WEBAPI REST

IMPLEMENTAR UN WEBAPI REST BÁSICA EN .NET

INTRODUCCIÓN



ASP.NET Web API (acrónimo de Application Programming Interface) es un framework para construir servicios basados en HTTP.

ASP.NET Web API Framework no obliga a usar algún estilo de arquitectura específico para crear servicios, aunque es usualmente utilizado para crear servicios RESTful.

Los servicios API pueden ser consumidos por distintos tipos de clientes como navegadores, aplicaciones móviles, aplicaciones de escritorio, así como IoT (Internet of things).

INTRODUCCIÓN



REST es un modelo de arquitectura para crear API y que usa HTTP para comunicarse.

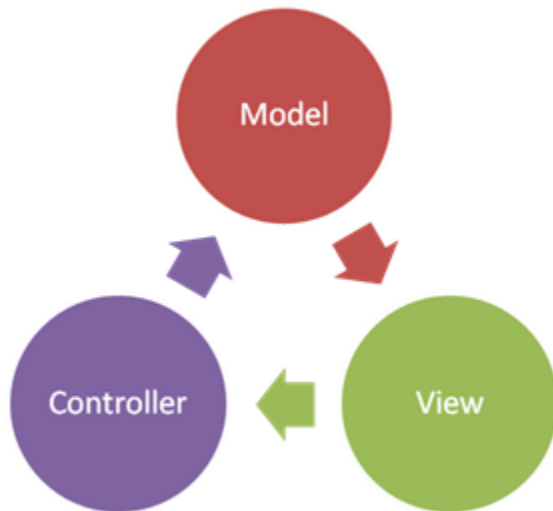
Muchas funcionalidades que puede tener una aplicación se pueden sumar con el acrónimo CRUD (es decir Create, Read, Update y Delete). Hay cuatro métodos HTTP que se corresponden a estas acciones:

- **GET** para recuperar los datos.
- **POST** para agregar un nuevo registro.
- **PUT** para modificar un registro.
- **DELETE** para eliminar el mismo

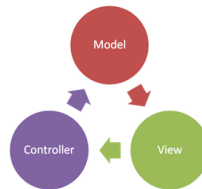
MVC



Antes de comenzar a construir nuestra web api, necesitamos tener cierto entendimiento de que es la arquitectura MVC.



MVC

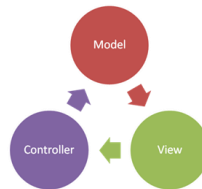


MVC (Modelo-Vista-Controlador) es un patrón de diseño de software comúnmente utilizado para implementar interfaces de usuario, datos y lógica de control.

Hace hincapié en la separación entre la lógica de negocio del software y la visualización.

Esta "separación de preocupaciones" permite una mejor división del trabajo y un mejor mantenimiento.

MVC



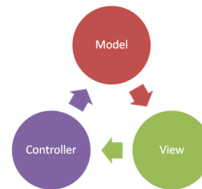
Las tres partes del patrón de diseño de software MVC son:

Modelo: Gestiona los datos y la lógica de negocio.

Vista: Gestiona el diseño y la visualización.

Controlador: Dirige la comunicación/comandos a las partes del modelo y la vista.

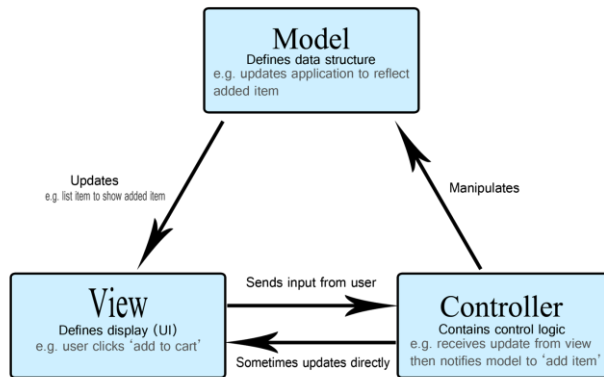
MVC



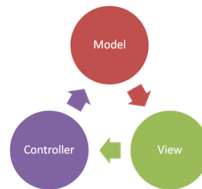
Imaginemos una sencilla aplicación de lista de la compra.

Todo lo que queremos es una lista con el nombre, la cantidad y el precio de cada artículo que tenemos que comprar esta semana.

A continuación describiremos cómo podríamos implementar parte de esta funcionalidad utilizando MVC.



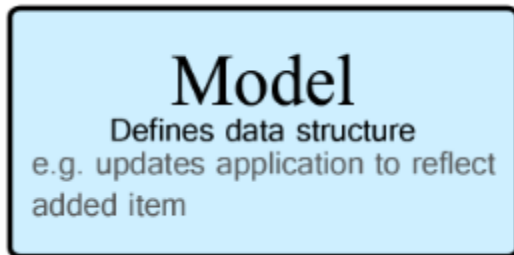
MVC



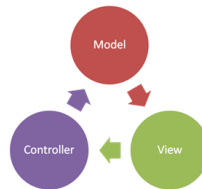
Modelo:

El modelo define los datos que debe contener la aplicación.

En nuestra aplicación de lista de la compra, el modelo especificaría qué datos deberían contener los elementos de la lista -artículo, precio, etc.- y qué elementos de la lista están ya presentes.



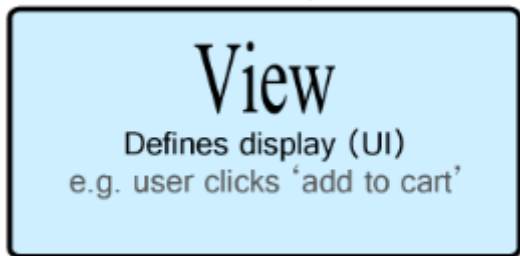
MVC



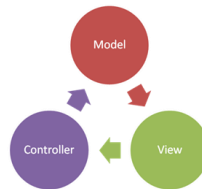
Vista:

La vista define cómo deben mostrarse los datos de la aplicación.

En nuestra aplicación de lista de la compra, la vista definiría cómo se presenta la lista al usuario, y recibiría los datos a mostrar desde el modelo.



MVC



Controlador:

El controlador contiene la lógica que actualiza el modelo y/o la vista en respuesta a las entradas de los usuarios de la aplicación.

La lista de la compra podría tener formularios de entrada y botones que nos permitan añadir o eliminar artículos.

Estas acciones requieren que el modelo se actualice, por lo que la entrada se envía al controlador, que a su vez manipula el modelo según corresponda, que luego envía los datos actualizados a la vista.

Controller

Contains control logic
e.g. receives update from view
then notifies model to 'add item'

ASP.NET WEB API



HTTP no sólo sirve para servir páginas web.

HTTP es también una potente plataforma para construir APIs que exponen servicios y datos.

ASP.NET Web API es un framework para construir APIs web sobre .NET Framework.

ASP.NET WEB API



El Framework Web API de ASP.NET facilita la creación de APIs para aplicaciones móviles, aplicaciones de escritorio, servicios Web, aplicaciones Web y otras aplicaciones.

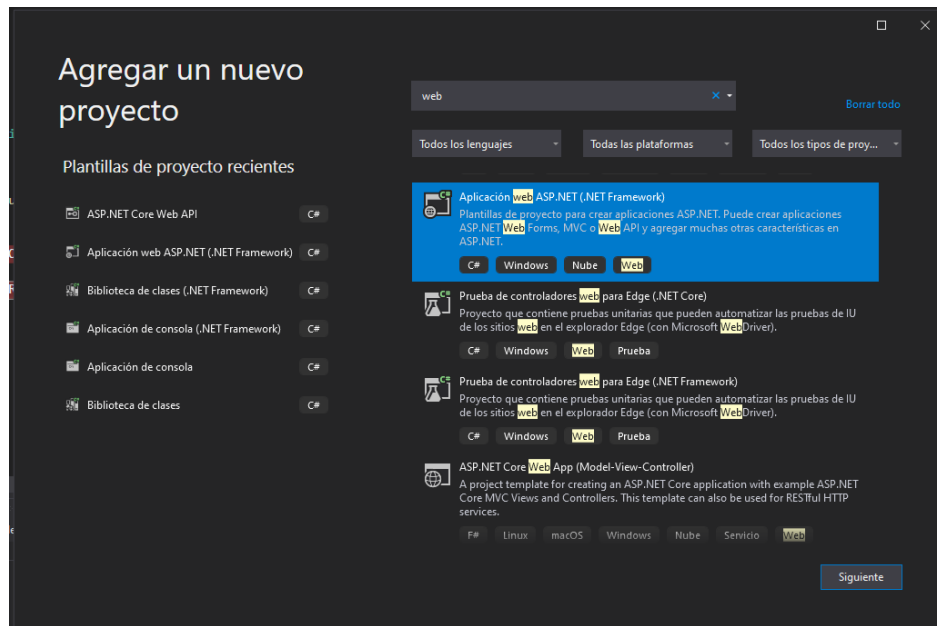
ASP.NET Web API es un Framework que forma parte de ASP.NET MVC y que permite construir APIs habilitadas para REST.

Las APIs habilitadas para REST ayudan a que sistemas externos utilicen la lógica de negocio implementada en una aplicación, incrementando la reutilización de dicha lógica.

IMPLEMENTAR UN WEB API



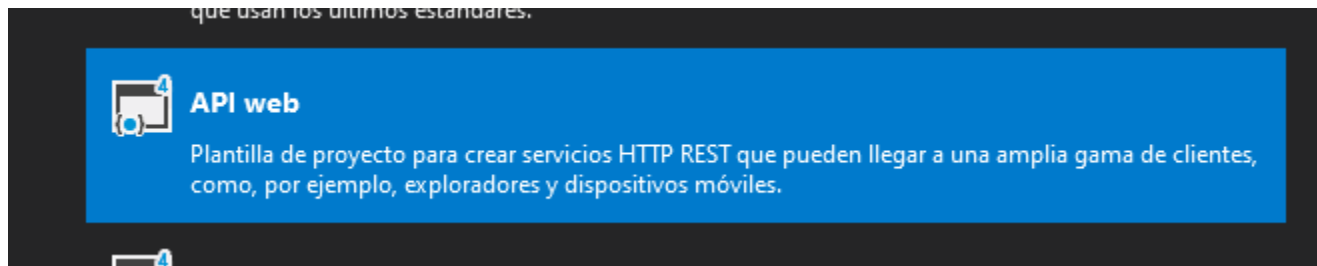
Para crear nuestro web api, el primer paso es crear un nuevo proyecto de aplicación web ASP.NET (Aplicación web ASP.NET (.NET Framework)).



IMPLEMENTAR UN WEB API



Al crear el proyecto podremos elegir una plantilla que nos dará una base para construir nuestra aplicación, en este caso, elegiremos la plantilla “API web”.



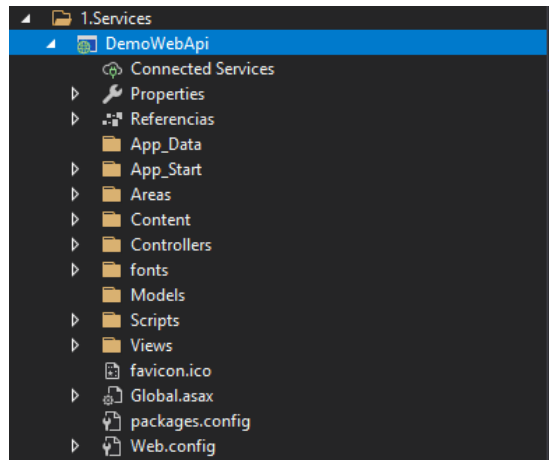
El resto de opciones pueden quedar con el valor por defecto.

IMPLEMENTAR UN WEB API



Como lo dice el nombre, este tipo de proyecto nos permite crear una aplicación web, en este caso, lo usaremos para implementar un web api.

Por lo tanto, existen diversos archivos que vienen incluidos en la plantilla de los cuales podemos prescindir.

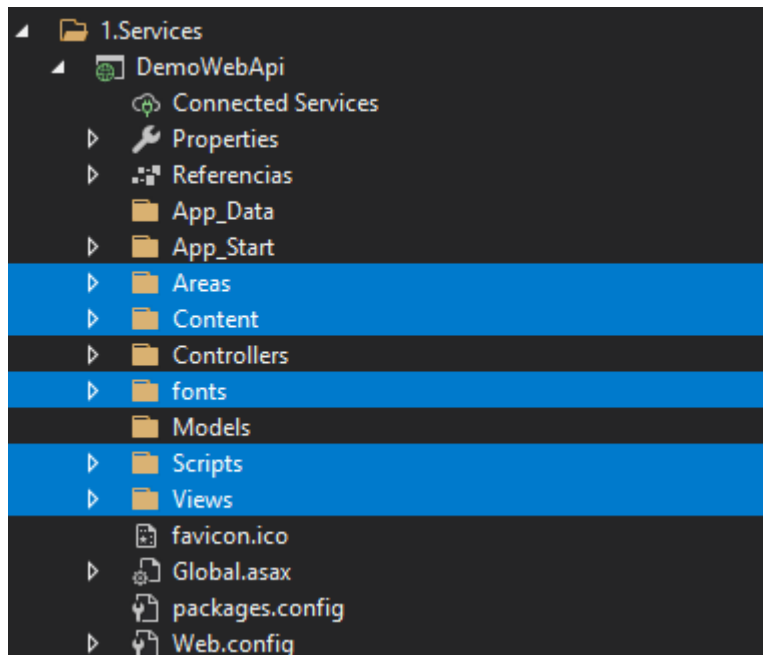


IMPLEMENTAR UN WEB API



Las siguientes carpetas pueden ser eliminadas:

- Views
- Scripts
- fonts
- Content
- Areas

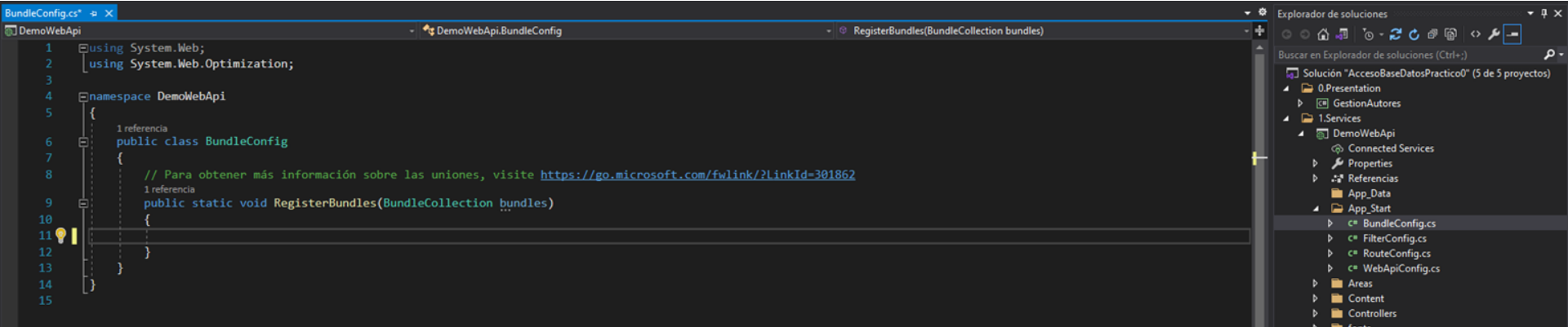


IMPLEMENTAR UN WEB API



Como parte del proceso de depuración del proyecto base, debemos modificar los siguientes archivos:

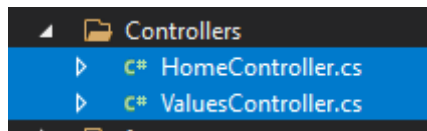
BundleCofig: Remover el contenido del método RegisterBundles



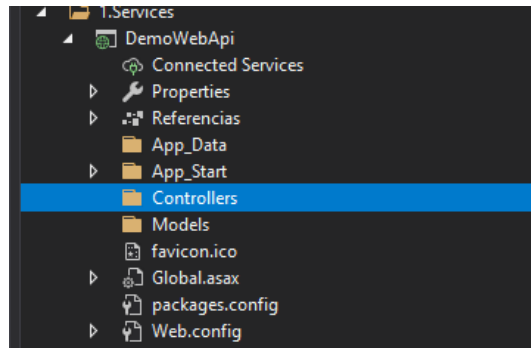
IMPLEMENTAR UN WEB API



Los controladores por defecto pueden ser eliminados, en este ejemplo implementaremos nuestros propios controladores.



Al finalizar nuestro proyecto debe lucir de la siguiente manera:

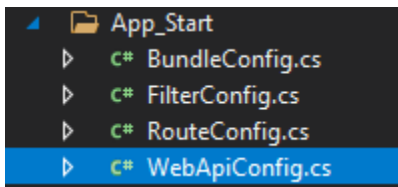


IMPLEMENTAR UN WEB API



Una vez finalizada la depuración de archivos y configuraciones por defecto de nuestro proyecto web, estamos en condiciones de comenzar a implementar nuestro web api.

Lo primero a realizar es configurar el formato de serialización de mensajes que utilizaran nuestros métodos, por defecto, los métodos serializan sus respuestas en XML, para que el formato sea JSON debemos modificar el archivo WebApiConfig.cs ubicado en App_Start.



IMPLEMENTAR UN WEB API



La carpeta App_Start contiene diversos archivos de configuración que son utilizados cuando nuestra aplicación web MVC inicia.

- **BundleConfig**

Configuración de bundles de archivos js y css.

- **FilterConfig**

Configuraciones globales de filtros para las llamadas a los controladores.

- **RouteConfig**

Configuración de rutas de navegación.

- **WebApiConfig**

Configuración de comportamientos del webapi, utilizada cuando se registra el web api.

IMPLEMENTAR UN WEB API



Para configurar el formato de respuesta de mensajes de nuestros métodos y utilizar el formato JSON, debemos remover el formato XML de la configuración por defecto, para eso modificaremos el método Register() en WebApiConfig.cs.

```
1 referencia
public static void Register(HttpConfiguration config)
{
    // Configuración y servicios de API web

    // Rutas de API web
    config.MapHttpAttributeRoutes();

    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    //Remover formato XML de los formatos soportados por el web API
    var appXmlType = config.Formatters.XmlFormatter.SupportedMediaTypes.FirstOrDefault(t => t.MediaType == "application/xml");
    config.Formatters.XmlFormatter.SupportedMediaTypes.Remove(appXmlType);
}
```

```
var appXmlType = config.Formatters.XmlFormatter.SupportedMediaTypes.FirstOrDefault(t=> t.MediaType == "application/xml");
```

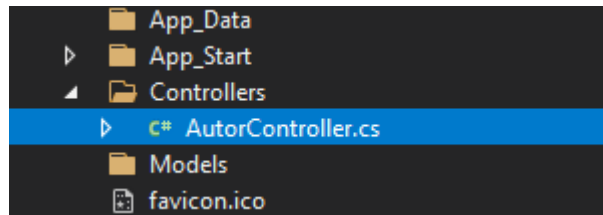
```
config.Formatters.XmlFormatter.SupportedMediaTypes.Remove(appXmlType);
```

IMPLEMENTAR UN WEB API



Ya contamos con el setup inicial para poder comenzar a implementar nuestra web api, para esto comenzaremos por crear un nuevo controlador encargado de coordinar las operaciones realizadas sobre nuestra entidad Autores.

Para esto generaremos la clase AutorController dentro de nuestra carpeta de controladores.



IMPLEMENTAR UN WEB API



Todos los controladores deben terminar por la palabra Controller para que sean reconocidos posteriormente como rutas válidas de nuestra web api.

Una vez creado el controlador de autores “AutorController”, haremos que el mismo herede de la clase “ApiController”.

```
using System.Web.Http;

namespace DemoWebApi.Controllers
{
    0 referencias
    public class AutorController : ApiController
    {
        ...
    }
}
```

IMPLEMENTAR UN WEB API



La clase ApiController define propiedades y métodos de un controlador de una API.

Esta clase define las acciones, respuestas, métodos de validación y redirecciones que pueden ser utilizadas e implementadas en el controlador de una API.

ApiController es parte del ensamblado System.Web.Http de .NET.

Cualquier controlador de nuestra web api debe implementar esta clase base y su nombre debe terminar en “Controller”.

IMPLEMENTAR UN WEB API



Cuando hablamos de web apis y de los métodos HTTP vimos que podemos hacer una relación entre las operaciones CRUD y los métodos HTTP básicos para implementar las mismas.

- **GET** para recuperar los datos.
- **POST** para agregar un nuevo registro.
- **PUT** para modificar un registro.
- **DELETE** para eliminar el mismo.

Veamos como implementar estos métodos...

IMPLEMENTAR UN WEB API



Comenzaremos por crear un método de consulta de autores por ID, para esto crearemos un método del tipo “IHttpActionResult” llamado GetAutorById el cual recibirá el ID del autor por parámetro.

```
0 referencias
public IActionResult GetAutorById(long id)
{
    ...
}
```

Este es un método del tipo GET, esto debemos indicarlo haciendo uso del tag [HttpGet], utilizando este tag, indicamos que la acción GetAutorById solo admite el método HTTP GET.

```
{
    [HttpGet]
    0 referencias
    public IActionResult GetAutorById(long id)
    {
        ...
    }
}
```

IMPLEMENTAR UN WEB API



¿Qué es la interfaz “IHttpActionResult”?

La interfaz IHttpActionResult está contenida en el espacio de nombres System.Web.Http y crea una instancia de HttpResponseMessage de forma asíncrona.

El IHttpActionResult comprende una colección de respuestas personalizadas que podemos utilizar dentro de los métodos de nuestro controlador.

Los métodos de los cuales disponemos son Ok, BadRequest, Exception, Conflict, Redirect, NotFound, and Unauthorized.

IMPLEMENTAR UN WEB API



Las respuestas mencionadas corresponden a distintos códigos HTTP:

- Ok: 200
- BadRequest: 400
- Exception: 500
- Conflict: 409
- Redirect: 302
- NotFound: 404
- Unauthorized: 401

IMPLEMENTAR UN WEB API



Si se desea realizar un manejo manual de los códigos HTTP se puede utilizar la clase “HttpResponseMessage” de manera directa, y devolver los códigos manualmente.

```
[HttpGet]
0 referencias
public HttpResponseMessage GetAutorById(long id)
{
    ...
    return Request.CreateResponse(HttpStatusCode.OK, "resultado JSON");
}
```

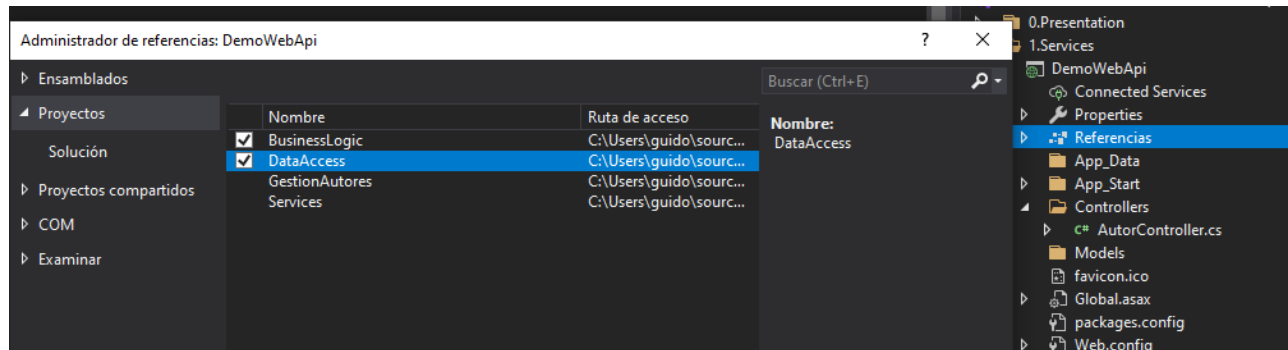
IMPLEMENTAR UN WEB API



El método GetAutorById obtiene el parámetro de id con el cual realizara la búsqueda del autor en nuestra base de datos.

Para poder lograr esto agregaremos la referencia de los proyectos BusinessLogic y DataAccess a nuestra Web Api.

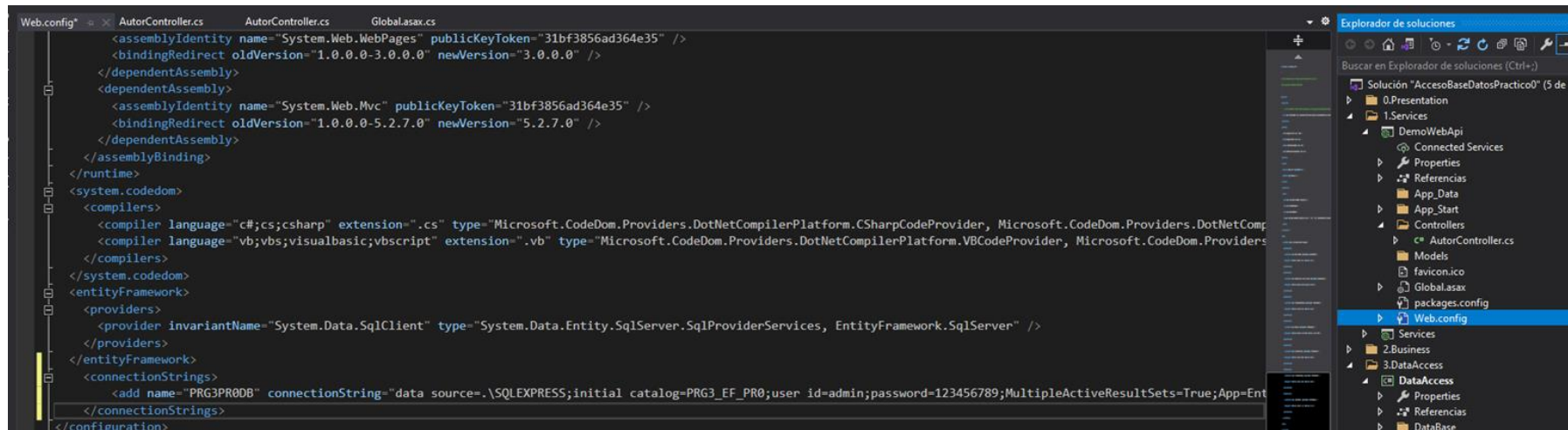
De esta manera tendremos acceso a nuestra unidad de trabajo, repositorios y entidades.



IMPLEMENTAR UN WEB API



Las operaciones contra nuestra base de datos se realizarán en el contexto de ejecución de nuestra API, por lo tanto, necesitamos instalar EntityFramework y agregar la connection string al archivo Web.Config.



IMPLEMENTAR UN WEB API



Ya estamos en condiciones de implementar el metodo de consulta, para esto consultaremos la tabla de Autores haciendo uso del repositorio.

```
[HttpGet]
0 referencias
public IHttpActionResult GetAutorById(long id)
{
    using (var uow = new UnitOfWork())
    {
        var autor = uow.AutorRepository.GetAutorById(id);

        if (autor == null)
            return NotFound();

        return Ok(autor);
    }
}
```


IMPLEMENTAR UN WEB API



Hacemos uso de la unidad de trabajo para acceder al repositorio de autores, AutorRepository, haciendo uso del método de consulta GetAutorById, consultamos nuestra base de datos.

Si el autor no es encontrado, retornamos un código HTTP 404, NotFound.

En caso de encontrar el autor, devolvemos el código 202, OK.

IMPLEMENTAR UN WEB API



Antes de realizar la prueba de nuestro método, debemos realizar un par de configuraciones extra:

1. Debemos configurar la serialización de nuestro web api para ignorar las referencias cíclicas.
2. Debemos configurar nuestro Entity Framework para deshabilitar el lazyLoading.

IMPLEMENTAR UN WEB API



Como ya sabemos, las relaciones de nuestras tablas se ven reflejadas en las clases de nuestro modelo de datos como atributos, estas pueden ser cíclicas dependiendo de la naturaleza de la relación.

Este tipo de relación es problemática al momento de serializar los objetos si no se cuenta con las configuraciones correctas.

Para esto debemos realizar lo siguiente:

IMPLEMENTAR UN WEB API



Modificar el método Register() en la clase WebApiConfig.cs, agregando las siguientes configuraciones de serialización.

```
1 referencia
public static void Register(HttpConfiguration config)
{
    // Configuración y servicios de API web

    // Rutas de API web
    config.MapHttpAttributeRoutes();

    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    //Evito las referencias circulares al trabajar con Entity Framework
    config.Formatters.JsonFormatter.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Serialize;
    config.Formatters.JsonFormatter.SerializerSettings.PreserveReferencesHandling = Newtonsoft.Json.PreserveReferencesHandling.Objects;

    //Remover formato XML de los formatos soportados por el web API
    var appXmlType = config.Formatters.XmlFormatter.SupportedMediaTypes.FirstOrDefault(t => t.MediaType == "application/xml");
    config.Formatters.XmlFormatter.SupportedMediaTypes.Remove(appXmlType);
}
```

```
config.Formatters.JsonFormatter.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Serialize;
```

```
config.Formatters.JsonFormatter.SerializerSettings.PreserveReferencesHandling = Newtonsoft.Json.PreserveReferencesHandling.Objects;
```

IMPLEMENTAR UN WEB API



Entity framework provee numerosas configuraciones para el comportamiento de las consultas que realiza en la base de datos.

El lazy loading consiste en retrasar la carga de los datos relacionados, hasta que se soliciten específicamente. Es lo contrario al eager loading.

Este comportamiento presenta conflictos al serializar objetos retornados por consultas de Entity Framework.

IMPLEMENTAR UN WEB API



Para deshabilitar el lazy loading debemos modificar la configuración de nuestro modelo, agregando lo siguiente en el constructor.

```
public PRG3PR0DB()  
    : base("name=PRG3PR0DB")  
{  
    this.Configuration.LazyLoadingEnabled = false;  
}
```

this.Configuration.LazyLoadingEnabled = false;

IMPLEMENTAR UN WEB API



Para probar nuestro método GET haremos uso del programa PostMan, Postman es una plataforma de API para que los desarrolladores diseñen, construyan, prueben e iteren sus APIs.

Para comenzar a utilizar PostMan debemos descargarlo de la página oficial de la aplicación.

The Postman app

The ever-improving Postman app (a new release every week) gives you a full-featured Postman experience.

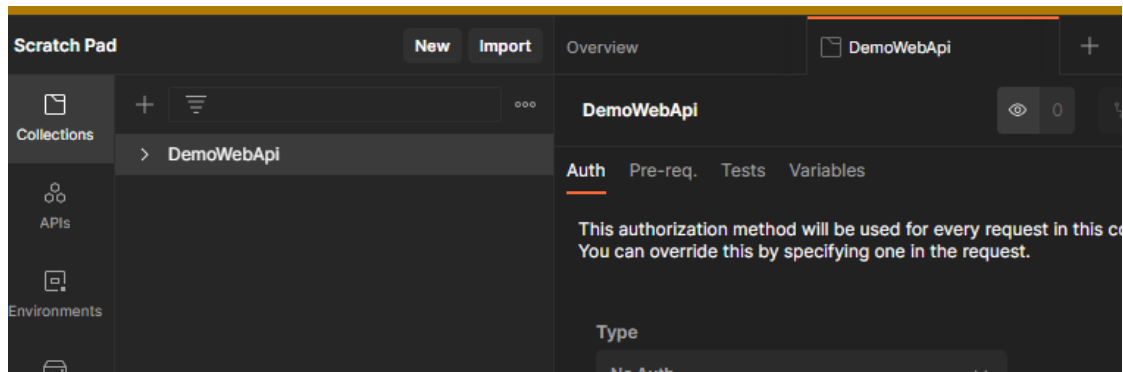
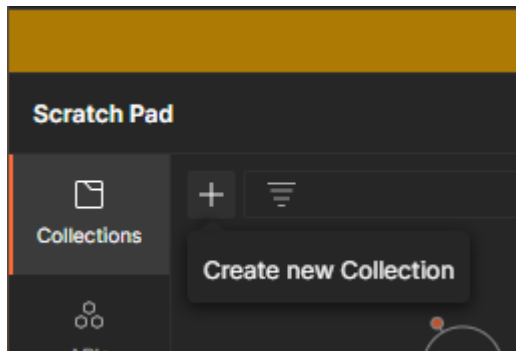
 Windows 64-bit

https://www.postman.com/downloads/?utm_source=postman-home

IMPLEMENTAR UN WEB API



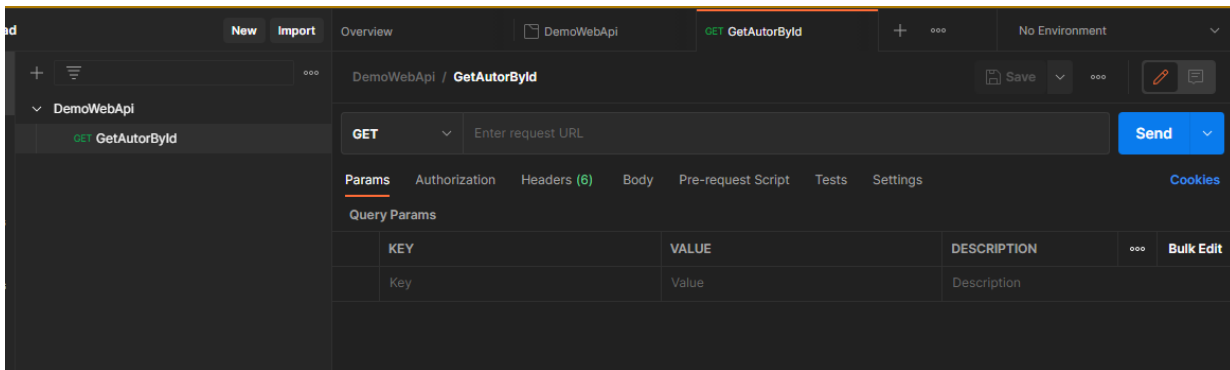
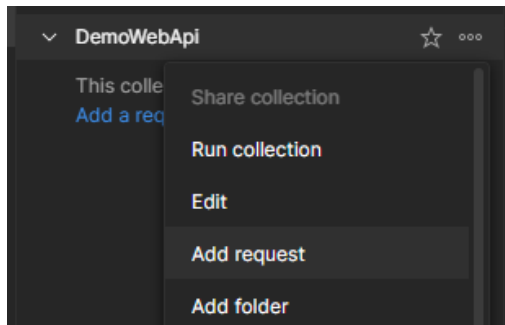
Para comenzar a probar nuestra API debemos crear una nueva colección de requests en PostMan:



IMPLEMENTAR UN WEB API



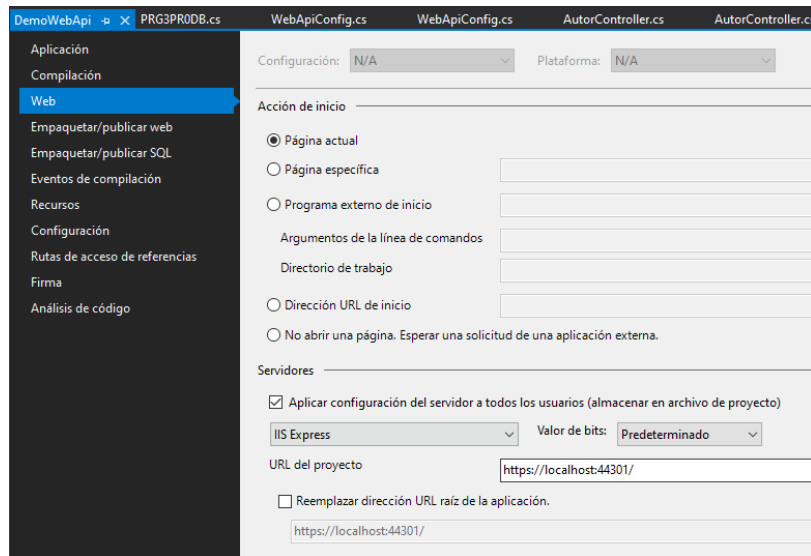
Para probar nuestro método debemos crear una nueva Request GET:



IMPLEMENTAR UN WEB API



Para poder terminar de crear la request a nuestro método GET, debemos saber en qué puerto estará publicada nuestra Web Api, para eso podemos comprobar las propiedades de nuestro proyecto, bajo la sección “Web”, en este caso es el puerto 44301.



IMPLEMENTAR UN WEB API



Para terminar de configurar la request, debemos cargar la url que representa el endpoint de nuestro método, si utilizamos el puerto consultado en las propiedades, y el ruteo por defecto del web api, debería tener el siguiente formato:

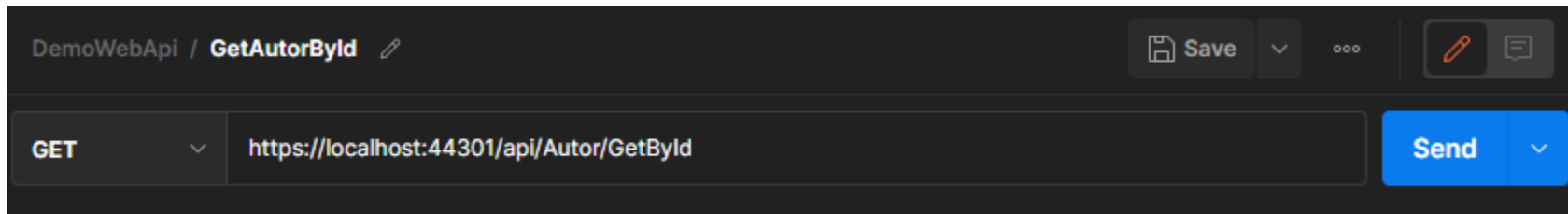
[https://localhost:\[Puerto\]/api/Autor/GetById](https://localhost:[Puerto]/api/Autor/GetById)

`https://localhost:44301/api/Autor/GetById`

IMPLEMENTAR UN WEB API



Podemos avanzar en la configuración de nuestra request, cargando la URL del método de nuestra API.



IMPLEMENTAR UN WEB API



Debemos configurar los parámetros a enviar en nuestra request GET, para esto haremos uso de la sección “Params” dentro de nuestra request, donde agregaremos el parámetro id, el key de este parámetro debe coincidir con el nombre del parámetro del método de nuestra API.

| Params ● Authorization Headers (6) Body Pre-request Script Tests Settings Cookies | | | | |
|---|-----|-------|-------------|-----------|
| Query Params | | | | |
| | KEY | VALUE | DESCRIPTION | ... |
| <input checked="" type="checkbox"/> | id | 1 | | Bulk Edit |

```
[HttpGet]
```

```
0 referencias
```

```
public IActionResult GetAutorById(long id)
```

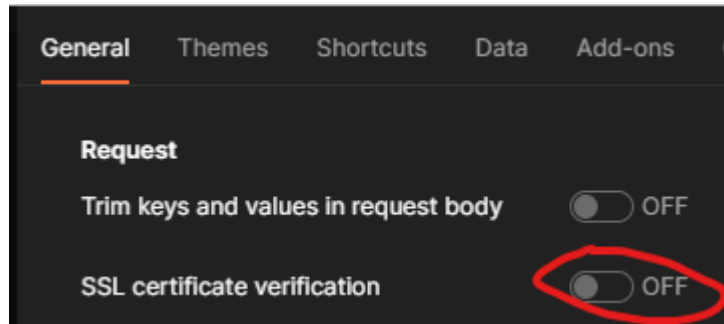
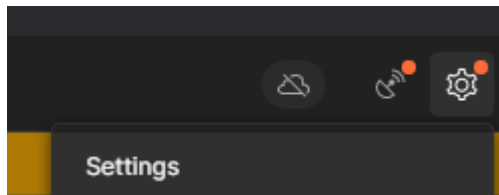
```
{
```

IMPLEMENTAR UN WEB API



El último paso es configurar la validación de certificados SSL en nuestro postman para que podamos utilizar los certificados autofirmados del IIS Express, para esto debemos ir a:

Settings -> General -> Desactivamos la verificación SSL



IMPLEMENTAR UN WEB API

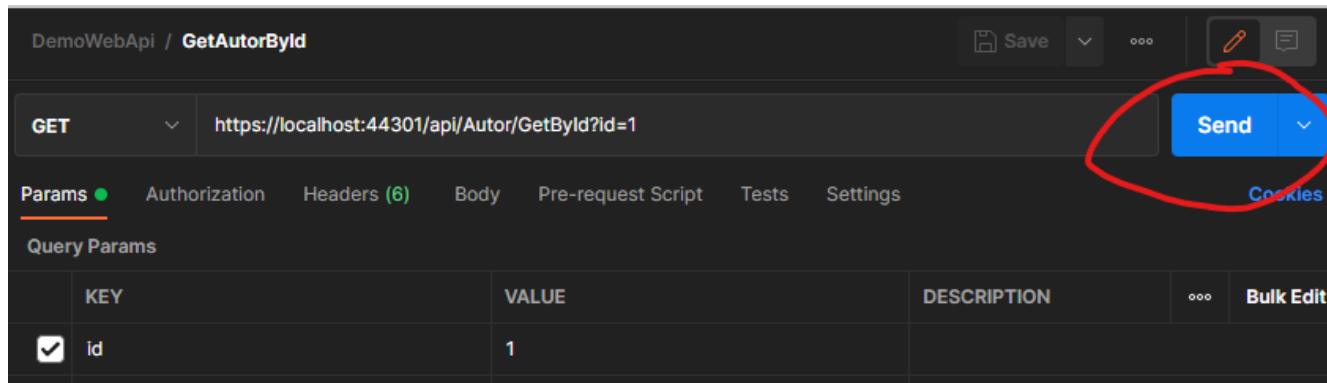


Listo!, ya estamos en condiciones de probar nuestro método.

En primer lugar, debemos iniciar una nueva instancia de nuestra web api:

Click derecho en el proyecto de la web api -> Depurar -> Iniciar nueva instancia.

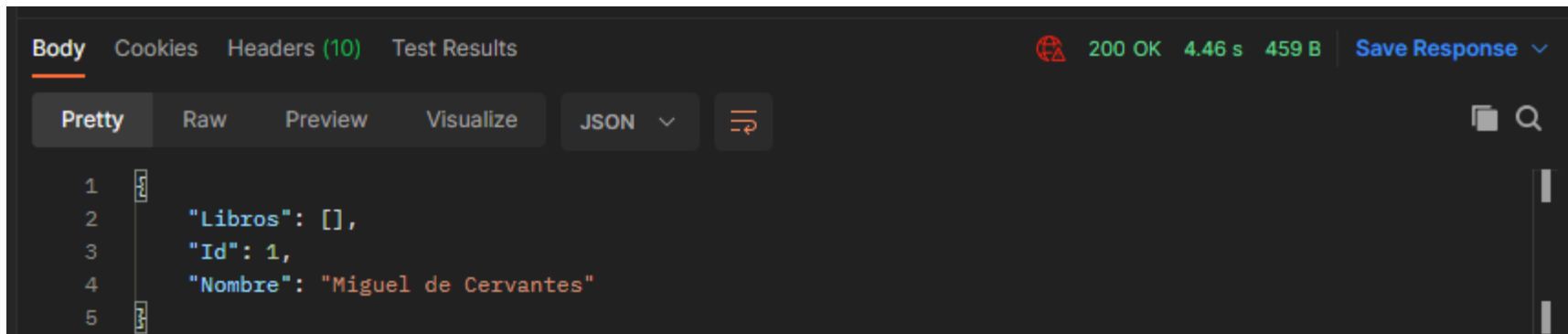
Luego de iniciada la instancia, podemos enviar la request desde Postman:



IMPLEMENTAR UN WEB API



Si todo funciona correctamente, obtendremos la siguiente respuesta:

A screenshot of a web browser's developer console. The 'Body' tab is selected, showing a JSON response. The response status is '200 OK' with a response time of '4.46 s' and a size of '459 B'. The JSON data is displayed in a 'Pretty' format, showing an array of objects. The first object has an 'Id' of 1 and a 'Nombre' of 'Miguel de Cervantes'.

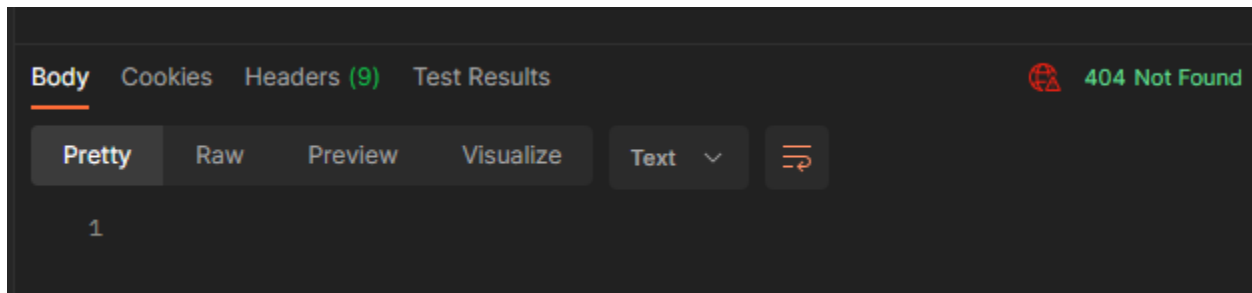
```
1 {  
2   "Libros": [],  
3   "Id": 1,  
4   "Nombre": "Miguel de Cervantes"  
5 }
```

El código de respuesta es 200 (Ok), y obtenemos un json del autor 1.

IMPLEMENTAR UN WEB API



Si modificamos el valor del parámetro Id, colocando un valor de un autor inexistente, el resultado obtenido es el siguiente:



En este caso, el código HTTP obtenido es 404, ya que el autor en cuestión no existe.

IMPLEMENTAR UN WEB API



Ya vimos como implementar un método de consulta GET básico, ahora procederemos a ver como implementar un método POST.

Como ya mencionamos, los métodos suelen ser utilizados para la creación de nuevos recursos.

Para este ejemplo, crearemos un método para crear un nuevo Autor, esto lo lograremos de la siguiente manera:

IMPLEMENTAR UN WEB API



Comenzaremos con crear un método POST llamado AddAutor, en este caso, la acción de crear autores solo estará disponible en un método HTTP POST, por lo tanto, colocaremos el tag [HttpPost].

```
[HttpPost]
0 referencias
public IHttpActionResult AddAutor()
{
    return Ok();
}
```

IMPLEMENTAR UN WEB API



Para crear un autor, es necesario contar con un conjunto de valores que definen al Autor, es aquí donde entran en juego los Modelos del patrón MVC.

En la definición de MVC, decíamos que el modelo define los datos que debe contener la aplicación.

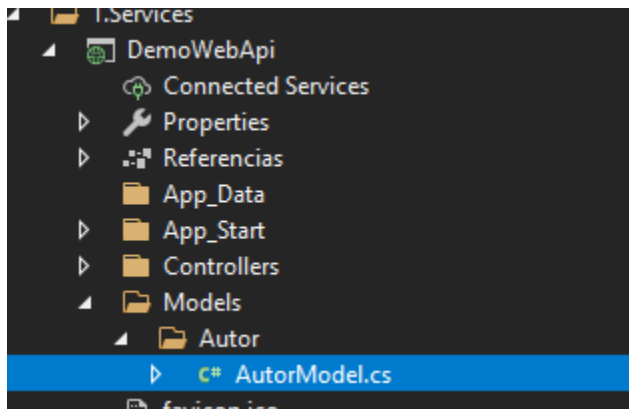
En este caso, nuestra aplicación, necesita un conjunto de datos para generar un Autor, por lo tanto, debemos definir los datos bajo una clase modelo.

Para esto, agregaremos una clase a los modelos de nuestra web api.

IMPLEMENTAR UN WEB API



Crearemos la clase AutorModel la cual estará ubicada en Models -> Autor.



El modelo de autor, contiene los atributos necesarios para definir la entidad autor.

IMPLEMENTAR UN WEB API



La clase AutorModel tendrá los siguientes atributos:

```
0 referencias
public class AutorModel
{
    0 referencias
    public long Id { get; set; }
    0 referencias
    public string Nombre { get; set; }
}
```

Aunque el Id no es necesario para la creación del autor, lo incluiremos como parte del modelo, ya que define la información del Autor.

IMPLEMENTAR UN WEB API



Las clases de modelo no solo son descriptores y contenedores de información, sino que también pueden implementar ciertas validaciones básicas sobre los atributos que las definen.

Estas validaciones pueden ser agregadas a la clase de modelo haciendo uso de un conjunto de tags especiales, donde cada uno representa un tipo de validación.

IMPLEMENTAR UN WEB API



Estas anotaciones son parte del namespace

`System.ComponentModel.DataAnnotations`, algunas de las anotaciones disponibles son:

- Required
- Regular Expression
- Range
- EmailAddress
- DisplayName
- DisplayFormat
- Scaffold
- DataType
- StringLength
- UIHint
- MaxLength
- MinLength

IMPLEMENTAR UN WEB API



Agregaremos algunas anotaciones al modelo de Autor:

```
0 referencias
public class AutorModel
{
    0 referencias
    public long Id { get; set; }

    [Required]
    [MaxLength(50)]
    0 referencias
    public string Nombre { get; set; }
}
```

De esta manera indicamos que el nombre del autor es requerido, y que el largo máximo de este atributo es 50.

IMPLEMENTAR UN WEB API



Volviendo al método POST para crear autores, ahora estamos en condiciones de indicar que el modelo `AutorModel` definirá la información a recibir en la request.

Para esto, indicaremos que se obtendrá una instancia de `AutorModel` del body de la request HTTP.

```
[HttpPost]
0 referencias
public IHttpActionResult AddAutor([FromBody] AutorModel autor)
{
    return Ok();
}
```

IMPLEMENTAR UN WEB API



El tagFromBody indica que se espera recibir información serializada en el cuerpo de nuestra request, esta información será deserealizada como un objeto de la clase AutorModel.

El body de una request http contiene información en bytes, esta es enviada luego de los headers.

Ya contamos con el método de creación de autores AddAutor, además contamos con una clase de modelo que define la información del autor que manejará nuestra api.

IMPLEMENTAR UN WEB API



Para crear nuestro autor, haremos uso de nuestra unidad de trabajo y de los repositorios de Autor.

```
[HttpPost]
0 referencias
public IHttpActionResult AddAutor([FromBody] AutorModel autor)
{
    using (var uow = new UnitOfWork())
    {
        uow.BeginTransaction();

        try
        {
            var autorEntity = new Autores()
            {
                Nombre = autor.Nombre
            };

            uow.AutorRepository.AddAutor(autorEntity);

            uow.SaveChanges();
            uow.Commit();

            return Ok(autorEntity);
        }
        catch (Exception ex)
        {
            uow.Rollback();

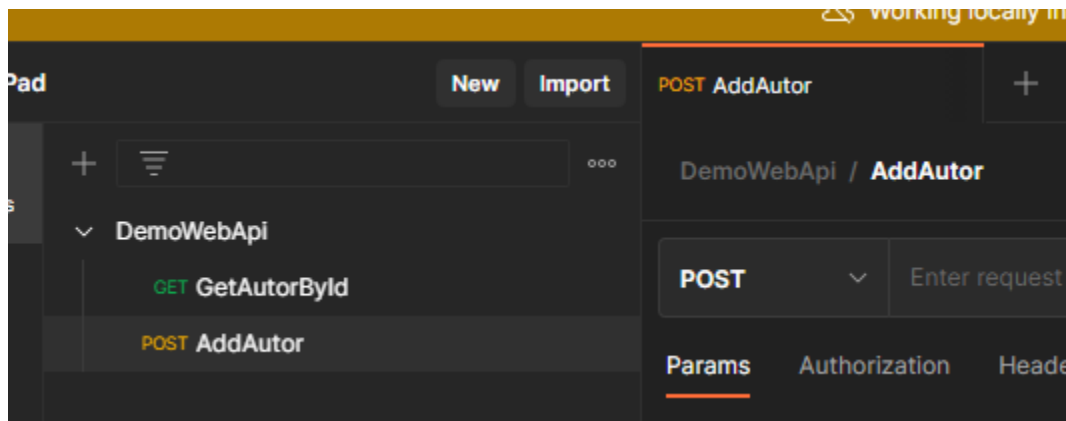
            return InternalServerError(ex);
        }
    }
}
```

IMPLEMENTAR UN WEB API



Luego de creado nuestro método de creación de usuarios POST, estamos en condiciones de realizar las pruebas pertinentes con PostMan.

Para esto generaremos una nueva request, en este caso, una request POST.



IMPLEMENTAR UN WEB API



La url de nuestro método será similar a la que utilizamos para el método GET, en esta caso solo cambiara el identificador del método.

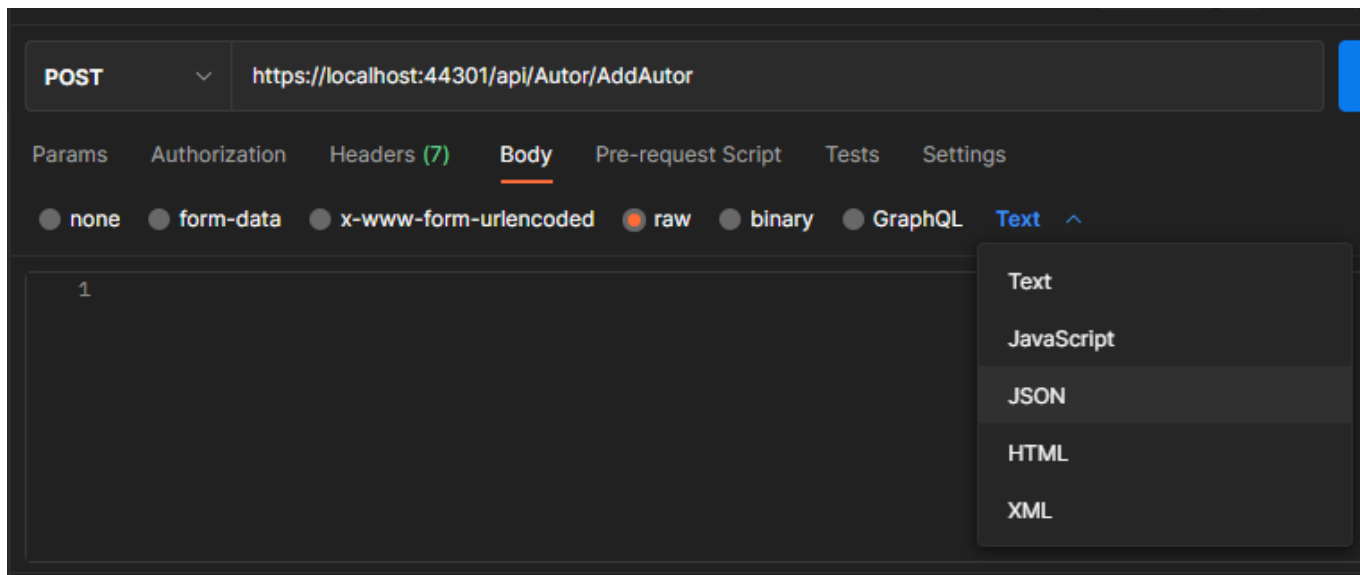
<https://localhost:44301/api/Autor/AddAutor>

Luego de creada la request, y definida la url de la misma, es necesario definir la información que enviaremos en el body.

IMPLEMENTAR UN WEB API



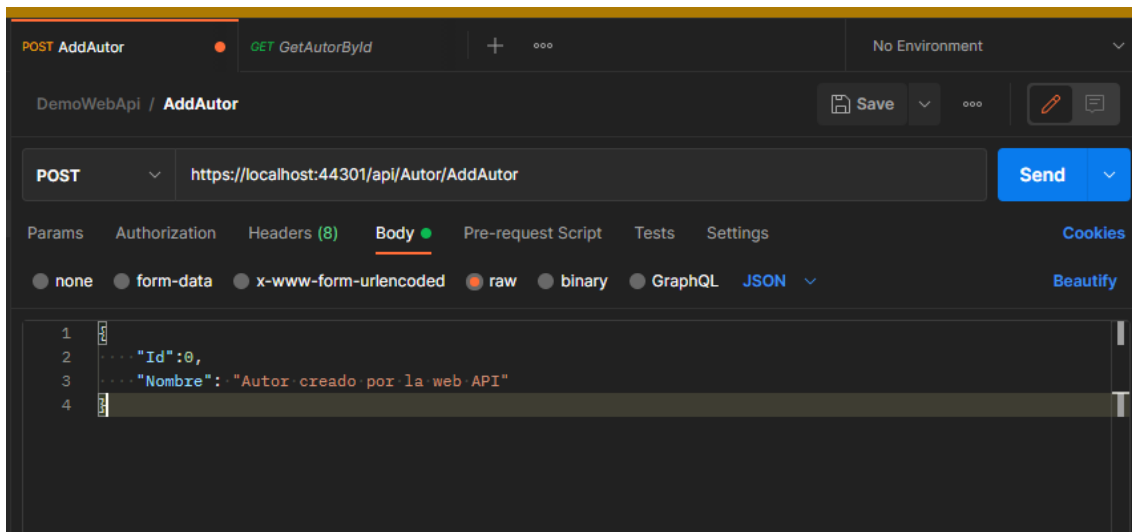
Para esto vamos a la pestaña “Body”, seleccionamos el check “Raw”, en el comboBox que aparece al seleccionar esta opción, elegiremos el formato JSON.



IMPLEMENTAR UN WEB API



Por último, agregaremos un JSON en el body de la request con la información del autor que vamos a crear.



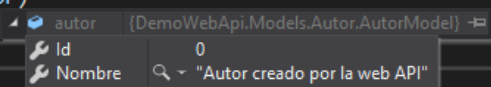
IMPLEMENTAR UN WEB API



Para realizar la prueba, iniciamos una nueva instancia de nuestro web api, y presionamos el botón send en nuestra request POST.

```
[HttpPost]
0 referencias
public IHttpActionResult AddAutor([FromBody] AutorModel autor)
{
    using (var uow = new UnitOfWork())
    {
        uow.BeginTransaction();

        try
        {
            var autorEntity = new Autor();
        }
    }
}
```

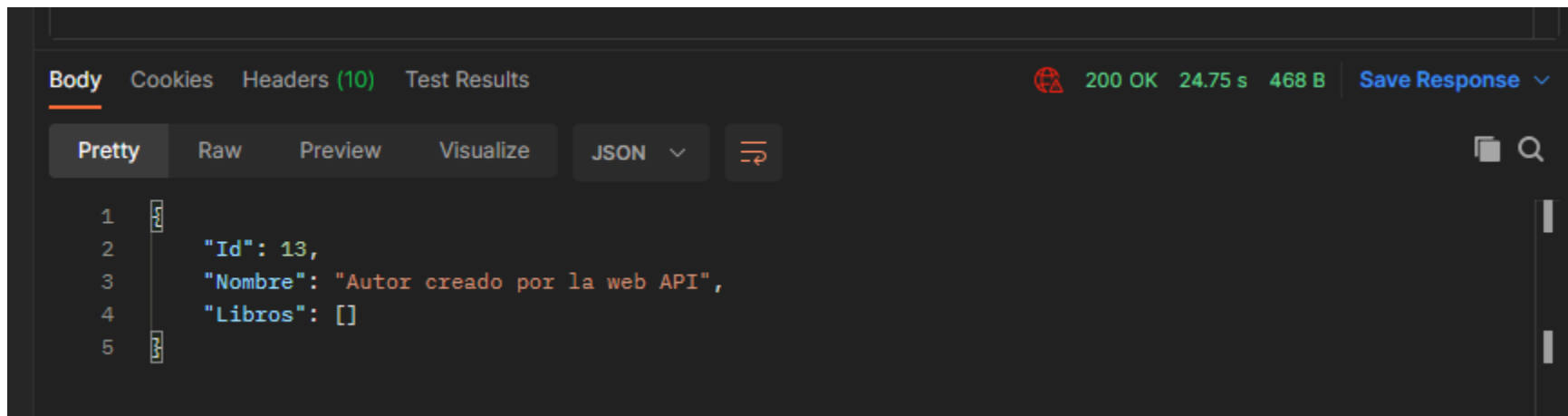
A Visual Studio IntelliSense popup is shown for the 'autor' parameter. It lists the 'Id' property with a value of '0' and the 'Nombre' property with a value of '"Autor creado por la web API"'. The popup is titled 'autor (DemoWebApi.Models.Autor.AutorModel)'.

Podemos observar como el serializador del web api parsea de manera automática la información enviada en nuestro body.

IMPLEMENTAR UN WEB API



Si observamos la respuesta de la request, veremos que obtuvimos un código 200 (OK) y la respuesta contiene el autor creado, incluyendo su id.



IMPLEMENTAR UN WEB API



El siguiente método a implementar es uno de actualización, como vimos anteriormente existen métodos HTTP que se encargan de realizar modificaciones a los recursos, en este caso, implementaremos un método de actualización PUT.

- **PUT** para modificar un registro.

IMPLEMENTAR UN WEB API



Comenzaremos con implementar un método PUT UpdateAutor, el método PUT actualiza el recurso autor, pudiendo actualizar todos los atributos del mismo, es por eso, que definiremos la información que recibe el método haciendo uso del modelo de autor, AutorModel.

```
[HttpPut]
0 referencias
public IHttpActionResult UpdateAutor([FromBody] AutorModel autor)
{
    ...
}
```

En este caso, haremos uso del tag HttpPut.

IMPLEMENTAR UN WEB API



Para actualizar un recurso, primero debemos verificar que el mismo exista, de ser el caso, actualizaremos los distintos atributos del mismo con los obtenidos en el body de la request HTTP.

Como se debe obtener el recurso para poder actualizarlo, en este escenario esperamos que el modelo de autor contenga el ID del mismo, en el método de creación POST esto no era necesario, ya que el recurso aún no existía.

IMPLEMENTAR UN WEB API



En caso de que el autor no existe, devolveremos una respuesta NotFound, la cual hace referencia al código http 404.

En caso de realizar la actualización de manera exitosa podemos retornar un código 200 (OK), en este caso también es posible retornar un código 204 (No Content)

IMPLEMENTAR UN WEB API



```
[HttpPut]
0 referencias
public IHttpActionResult UpdateAutor([FromBody] AutorModel autor)
{
    using (var uow = new UnitOfWork())
    {
        uow.BeginTransaction();

        try
        {
            var autorEntity = uow.AutorRepository.GetAutorById(autor.Id);

            if (autorEntity == null)
                return NotFound();

            autorEntity.Nombre = autor.Nombre;

            uow.SaveChanges();
            uow.Commit();

            return Ok(autorEntity);
        }
        catch (Exception ex)
        {
            uow.Rollback();

            return InternalServerError(ex);
        }
    }
}
```

IMPLEMENTAR UN WEB API



Para realizar las pruebas de nuestro nuevo método PUT, debemos crear una nueva request en PostMan, en este caso del tipo PUT.

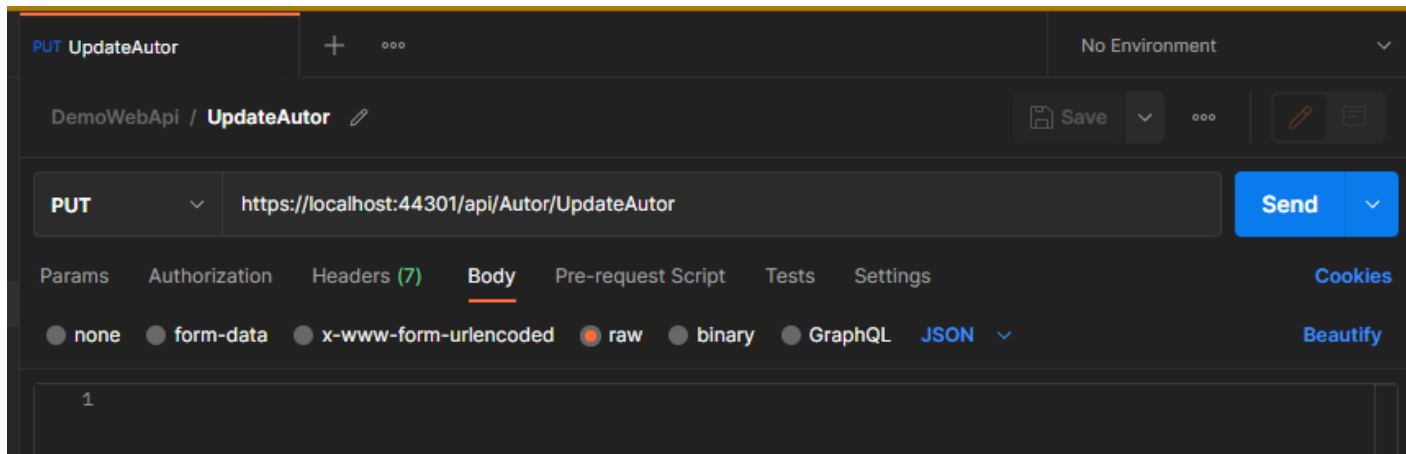
La url será similar a las anteriores, solo que cambiara el método a ejecutar.

<https://localhost:44301/api/Autor/UpdateAutor>

IMPLEMENTAR UN WEB API



Igual que con nuestro método POST, en el metodo PUT debemos cargar la información a transmitir en el body de nuestra request, para eso vamos a la pestaña “Body”, seleccionamos el check “Raw”, en el comboBox que aparece al seleccionar esta opción, elegiremos el formato JSON.



IMPLEMENTAR UN WEB API



En la sección de body, cargaremos un json con el modelo de autor, conteniendo el ID del autor a actualizar, y un nuevo nombre.

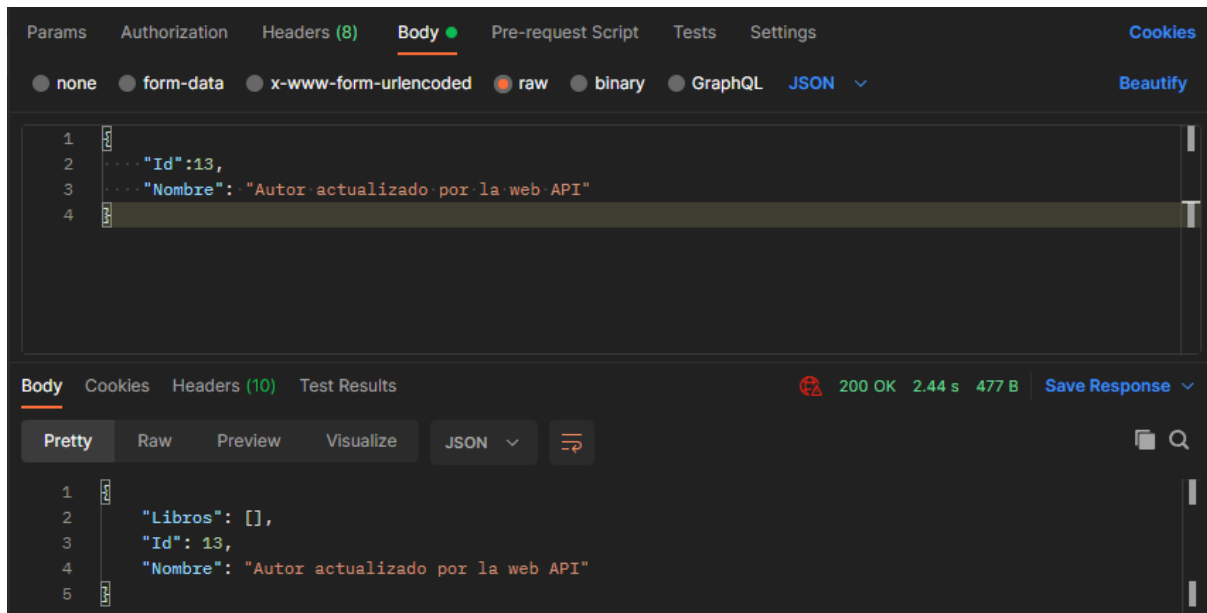
```
● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL
```

```
1  [
2    ... "Id":13,
3    ... "Nombre": "Autor actualizado por la web API"
4  ]
```

IMPLEMENTAR UN WEB API



Para probar nuestro nuevo método, podemos enviar la request, si todo funciona correctamente, obtendremos un código 200 y el autor con los datos actualizados.



IMPLEMENTAR UN WEB API



Ya vimos como implementar métodos GET, para consulta de datos, métodos POST, para la creación de los mismos y métodos PUT para realizar actualización de los recursos.

Para cerrar el conjunto de operaciones CRUD, debemos implementar un método que nos permita eliminar recursos, esto se logra a través del método HTTP DELETE.

- **DELETE** para eliminar un recurso.

IMPLEMENTAR UN WEB API



Comenzaremos implementado un nuevo método RemoveAutor, en este caso haremos uso del tag HttpDelete.

```
[HttpDelete]
0 referencias
public IHttpActionResult RemoveAutor()
{
    ...
}
```

IMPLEMENTAR UN WEB API



Para remover un recurso, suele ser suficiente con contar con el identificador del mismo, por lo tanto, podemos obtener el mismo en los parámetros de la query de nuestra URL. En caso de necesitar más información, se puede obtener la misma del body de la request.

```
[HttpDelete]  
0 referencias  
public IHttpActionResult RemoveAutor(long id)  
{
```

IMPLEMENTAR UN WEB API



Para remover un recurso, primero debemos obtener el mismo y garantizar que está disponible, por lo tanto, si el recurso no existe debemos retornar un código 404.

En caso de contar con el recurso y poder realizar el borrado de manera exitosa podemos retornar un código 200 o 204.

IMPLEMENTAR UN WEB API



```
[HttpDelete]
0 referencias
public IHttpActionResult RemoveAutor(long id)
{
    using (var uow = new UnitOfWork())
    {
        uow.BeginTransaction();

        try
        {
            var autorEntity = uow.AutorRepository.GetAutorById(id);

            if (autorEntity == null)
                return NotFound();

            uow.AutorRepository.RemoveAutor(autorEntity);

            uow.SaveChanges();
            uow.Commit();

            return Ok();
        }
        catch (Exception ex)
        {
            uow.Rollback();

            return InternalServerError(ex);
        }
    }
}
```


IMPLEMENTAR UN WEB API



Tener en cuenta que la forma en que se realiza la implementación de un método DELETE depende el programador, la existencia de un método DELETE no implica necesariamente que el recurso es eliminado completamente.

La implementación de este proceso puede ser un borrado lógico y/o otras modificaciones que inhabilitan el recurso.

IMPLEMENTAR UN WEB API



Para probar nuestro nuevo método, creamos una nueva request en PostMan, en este caso del tipo DELETE.

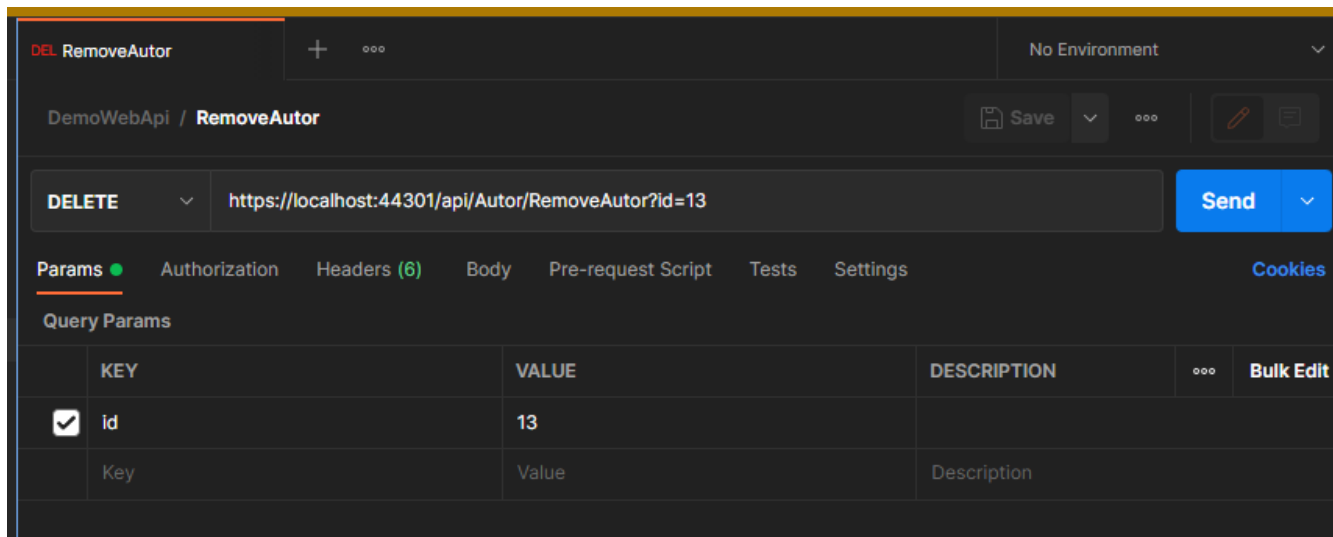
La url será similar a las anteriores, cambiando el método que vamos a ejecutar.

<https://localhost:44301/api/Autor/RemoveAutor>

IMPLEMENTAR UN WEB API



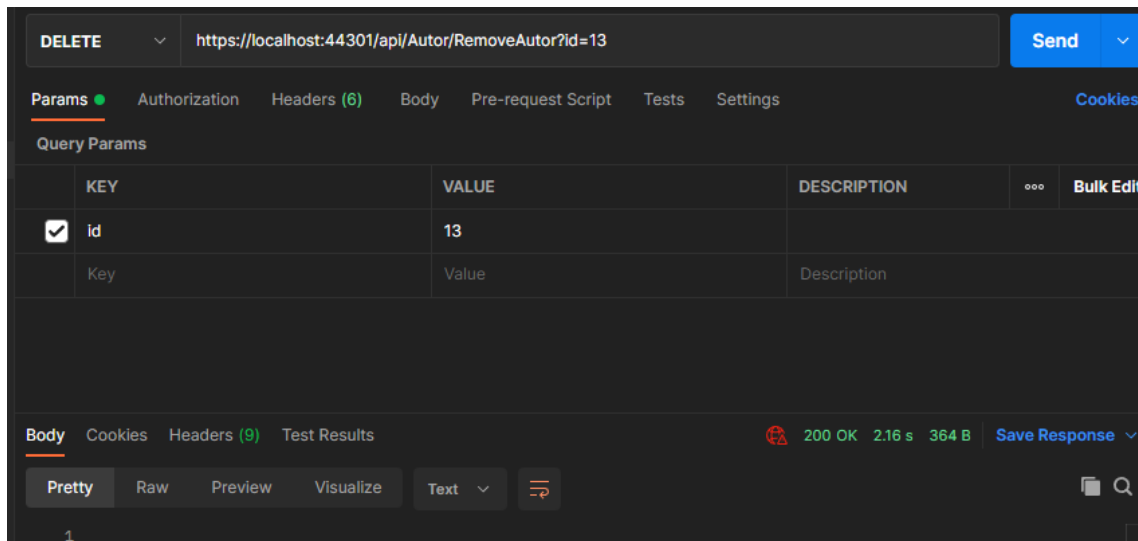
En este caso, nuestro método espera el id del autor como parámetro de la url, por lo tanto, al igual que en el método GET, agregaremos un parámetro cuya key debe coincidir con el nombre del parámetro del método en nuestra api.



IMPLEMENTAR UN WEB API



Para probar nuestro método, podemos enviar la request, si todo funciona de manera correcta, obtendremos un código 200 indicando que el recurso fue removido.

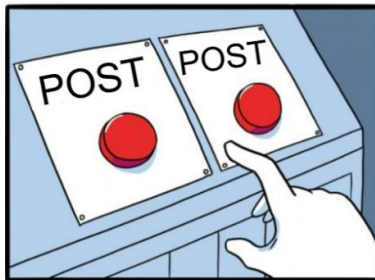


IMPLEMENTAR UN WEB API



De esta manera hemos implementado los cuatro tipos de método HTTP comúnmente utilizados para operaciones CRUD.

Sin embargo, eso no quiere decir que son los únicos existentes, de hecho existen 39 tipos de métodos HTTP.



IMPLEMENTAR UN WEB API



Es recomendable que los métodos de nuestras APIs, así como también nuestras clases, contengan summaries explicando su objetivo, parámetros y valores de retorno.

El tag `<summary>` es uno de los tantos tags XML que visual studio nos permite utilizar para documentar nuestro código.

Visual Studio puede ayudarte a documentar elementos de código como clases y métodos, generando automáticamente la estructura estándar de comentarios de documentación XML.

En tiempo de compilación, puedes generar un archivo XML que contenga los comentarios de la documentación.

IMPLEMENTAR UN WEB API



El archivo XML generado por el compilador puede distribuirse junto con el ensamblado .NET para que Visual Studio y otros IDEs puedan utilizar IntelliSense para mostrar información rápida sobre tipos y miembros.

Además, el archivo XML puede ejecutarse con herramientas como DocFX, Swagger y Sandcastle para generar sitios web de referencia de las APIs.

Para agregar tags de documentación summary a nuestros métodos, solo basta con ingresar /// encima del método, la estructura base del summary sera autocmpletada para que podamos cargarla con la información pertinente.

IMPLEMENTAR UN WEB API



Un <summary> está estructurado de la siguiente manera:

```
/// <summary>
///
/// </summary>
/// <param name="id"></param>
/// <returns></returns>
```

La sección summary debe contener un resumen de lo que nuestro método/clase realiza.

Los parámetros describen los parámetros de entrada de nuestros métodos, tipo y uso.

La sección returns describe cuál es el retorno de nuestro método.

IMPLEMENTAR UN WEB API



```
/// <summary>
/// GetAutorById: Este metodo retorna un autor dado un identificador si este existe
/// </summary>
/// <param name="id">Identificador del autor - tipo long - requerido</param>
/// <returns>
/// Retorna codigo HTTP 200 y un serializado del tipo Autores si el autor existe para el id dado
/// En caso de no existir el autor, retorna codigo HTTP 404
/// </returns>
[HttpGet]
0 referencias
public IActionResult GetAutorById(long id)
{
    using (var uow = new UnitOfWork())
    {
        var autor = uow.AutorRepository.GetAutorById(id);

        if (autor == null)
            return NotFound();

        return Ok(autor);
    }
}
```

FIN