

Módulo VI: MVC (Modelo Vista Controlador)

AUTOR: GIRIBALDI, Pablo Germán

Módulo VI: MVC (Modelo Vista Controlador)	1
MVC (Modelo-Vista-Controlador)	1
Introducción	1
Modelo	2
Vista	2
Controlador	2
API	3
API REST	4
JSON	4
Crear aplicación ASP.NET API web MVC	4
ORM	8
Entity Framework	8
Creando Modelo de Datos de Entidad	9
Creando un controlador	16
LINQ	19
Códigos de estado HTTP	23
Métodos de petición HTTP	23
POSTMAN	24
Prueba de Web API	26
CRUD Usuarios, registro y login	28
Referencias	36

MVC (Modelo-Vista-Controlador)

Introducción

El presente documento contiene contenido teórico y práctico sobre MVC (Modelo Vista controlador o *Model View Controller* por su nombre en inglés).

MVC es un patrón de arquitectura de software que separa las interfaces de usuario, los datos y la lógica de negocio en distintos componentes (el **modelo**, la **vista** y el **controlador**) definiendo la representación de la información por un lado y la interacción con el usuario por otro lado, facilitando además el trabajo en equipo encapsulando las tareas de desarrollo.

Modelo

El modelo contiene una representación de los datos que gestiona el sistema administrando la entrada, el procesamiento y la salida de los mismos (lo recomendable es que el modelo sea independiente del sistema de almacenamiento).

Éste también define las reglas de negocio llevando un registro de las vistas y los controladores del sistema, además de administrar las credenciales de acceso comunicándose con la *vista* a través del *controlador* al enviarle la información solicitada para que ésta la muestre al usuario.

Vista

La vista es la interfaz de usuario. Se compone de los datos que le envía el modelo para poder mostrar información al usuario y de los mecanismos de interacción con éste.

Controlador

El controlador es el encargado de actualizar el modelo o la vista según las entradas de los usuarios de la aplicación, administrando el flujo de información entre ellos, ya sea respondiendo a los eventos de la vista y realizando las peticiones al modelo o enviando las actualizaciones desde el modelo hacia la vista.

Teniendo en claro los conceptos, podemos describir los pasos de cómo funciona el patrón MVC.

Primero, a través de una interfaz, el usuario envía una petición al controlador, luego el controlador se comunica con el modelo solicitando los datos, luego el modelo devuelve los datos al controlador, este solicita la vista la cual va a ser devuelta al usuario cargando los datos del modelo seleccionado.

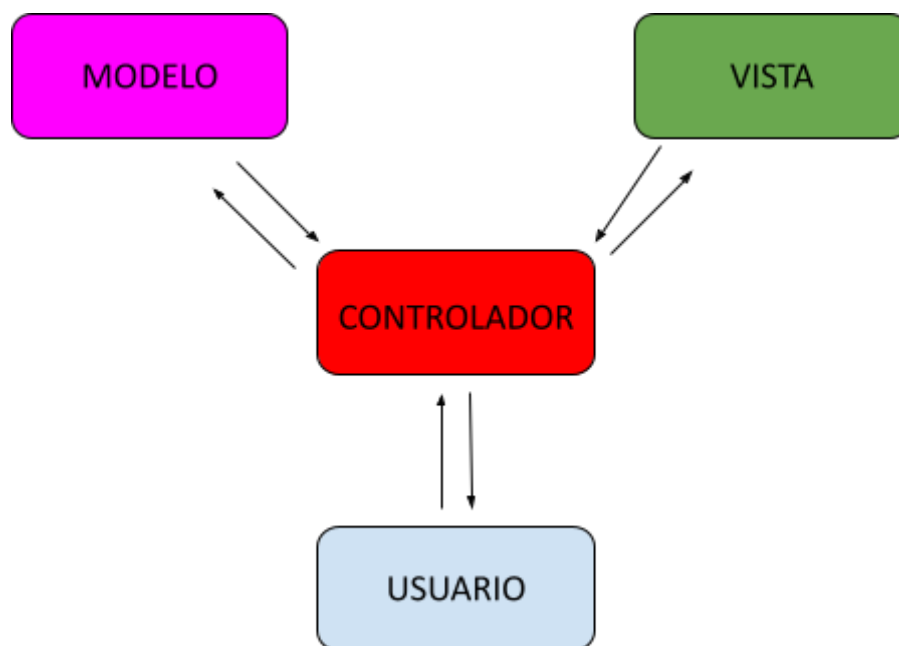


Figura 1: Patrón MVC.

API

Una API (Application Programming Interface) es un conjunto de algoritmos que brindan acceso a distintas funcionalidades de un software. Son utilizadas para brindar información a sistemas informáticos externos al software que tiene la conexión y el acceso a los datos. Se considera como el contrato entre el proveedor de los datos y el usuario.

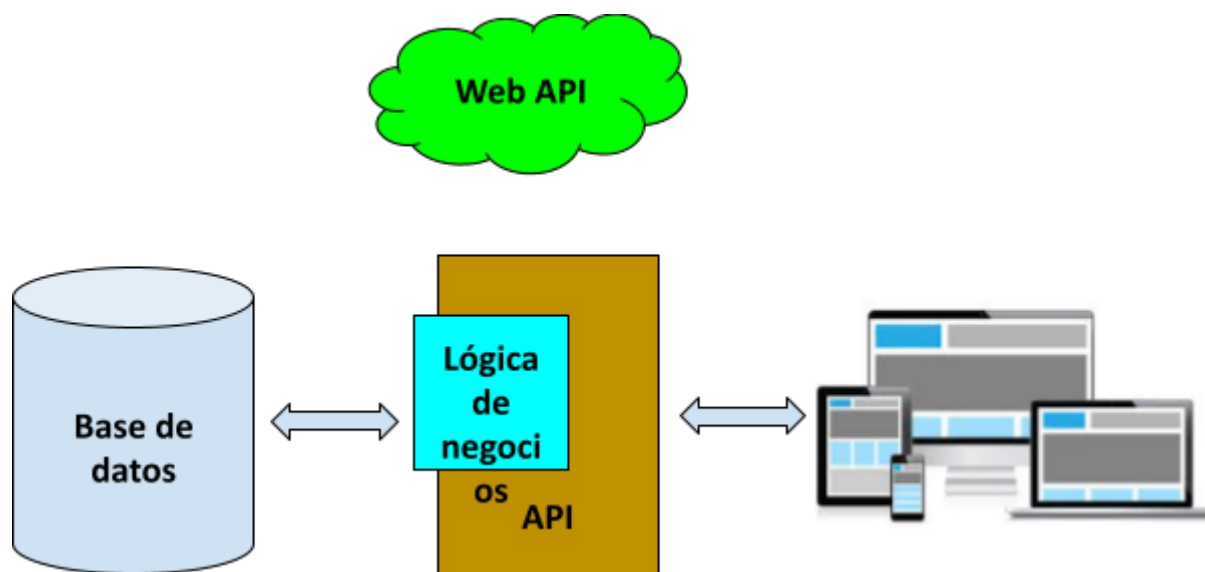


Figura 2: Esquema de una API

API REST

Un servicio REST no es una arquitectura de software, sino que es una interfaz entre sistemas que utilicen el protocolo HTTP para realizar el intercambio de datos en formatos XML y JSON.

JSON

JSON (JavaScript Object Notation) es un formato de intercambio de datos de sencillo entendimiento ya que es posible leer, escribir y analizar los objetos sin mayores complicaciones.

Crear aplicación ASP.NET API web MVC

Para crear nuestro sistema vamos a utilizar el IDE *Visual Studio 2019 Community*, con *MVC 5* y *.NET Framework 4.7.2* y seguimos los siguientes pasos.

Ejecutamos Visual Studio 2019 y hacemos click en “Crear un proyecto”.

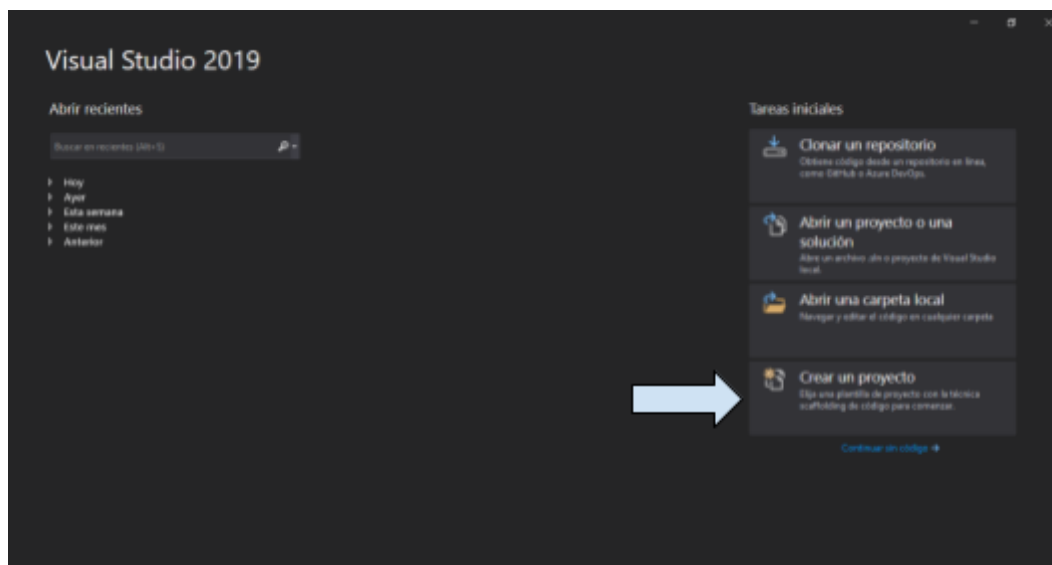


Figura 3: Creación de aplicación

Seleccionamos el lenguaje *C#*, la plataforma *Windows*, el tipo de proyecto *Web* y por último, la opción *Aplicación web ASP.NET (.NET Framework)*, luego hacemos click en *Siguiente*.

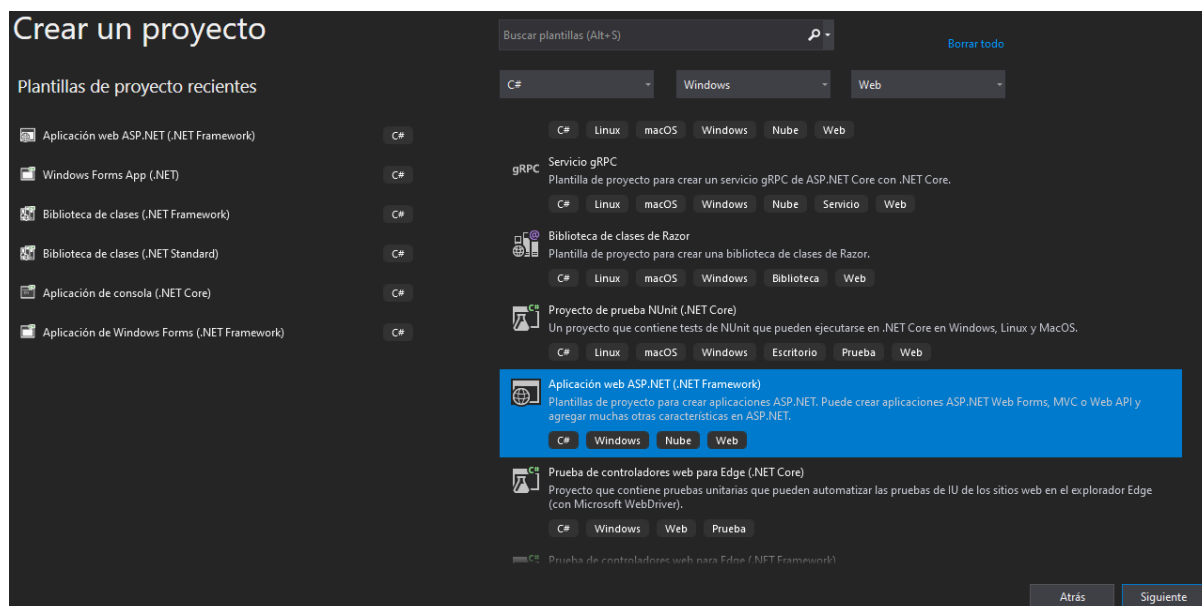


Figura 4: Selección de tipo de proyecto.

Configuramos el *nombre del proyecto*, la *ubicación*, el *nombre de la solución* y la versión del *Framework*, una vez completado este paso, hacemos click en *Crear*.

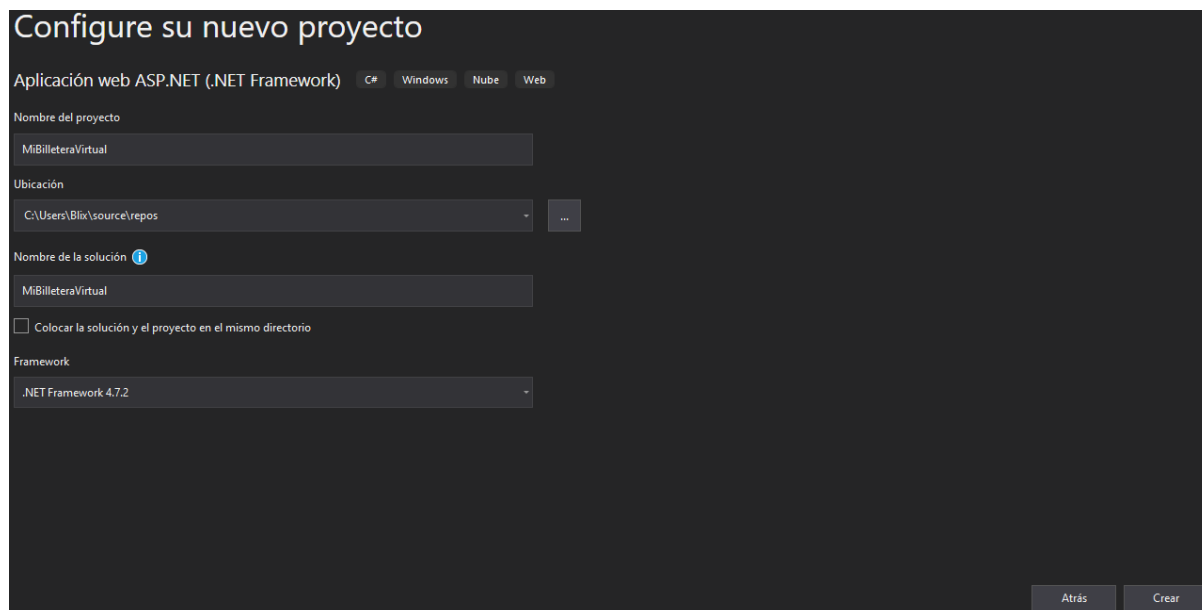


Figura 5: Configuración del proyecto.

Seleccionamos la opción API web y luego hacemos click en *Crear*.

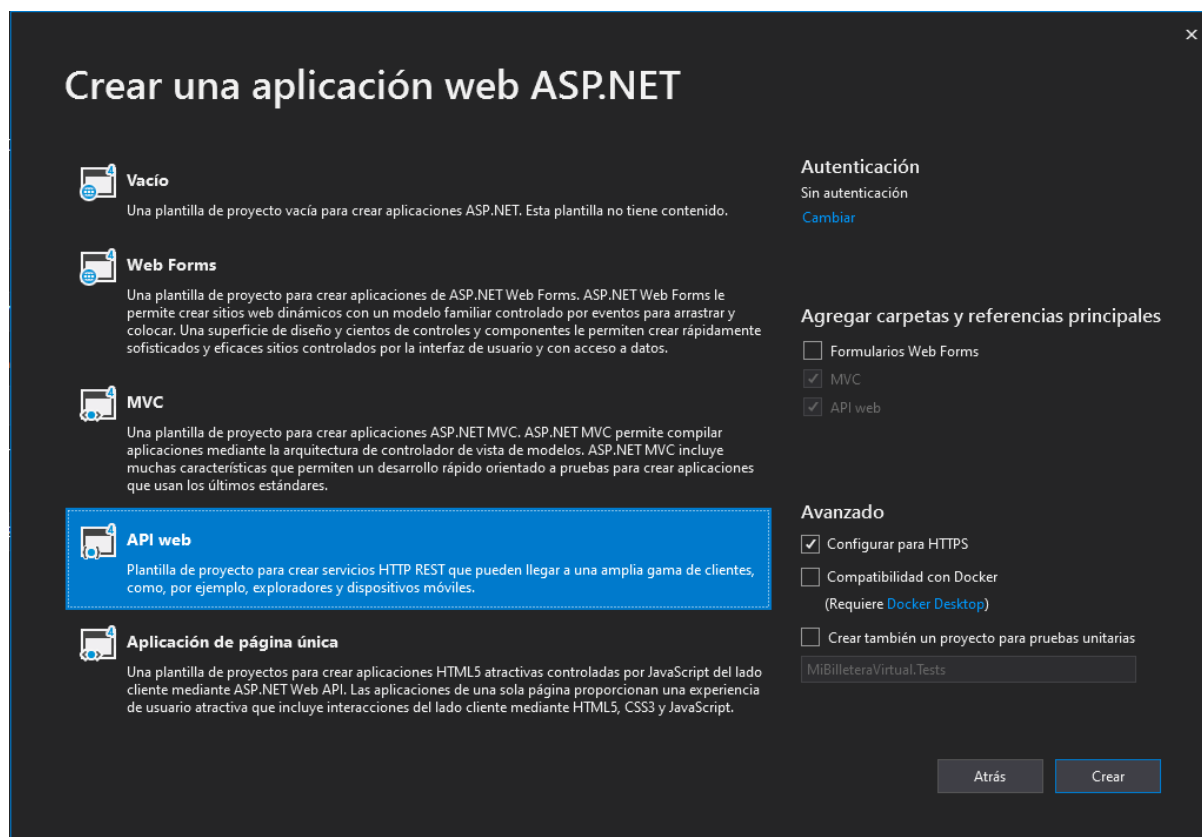


Figura 6: Crear aplicación API web.

Una vez creado el proyecto, debería verse de la siguiente manera.

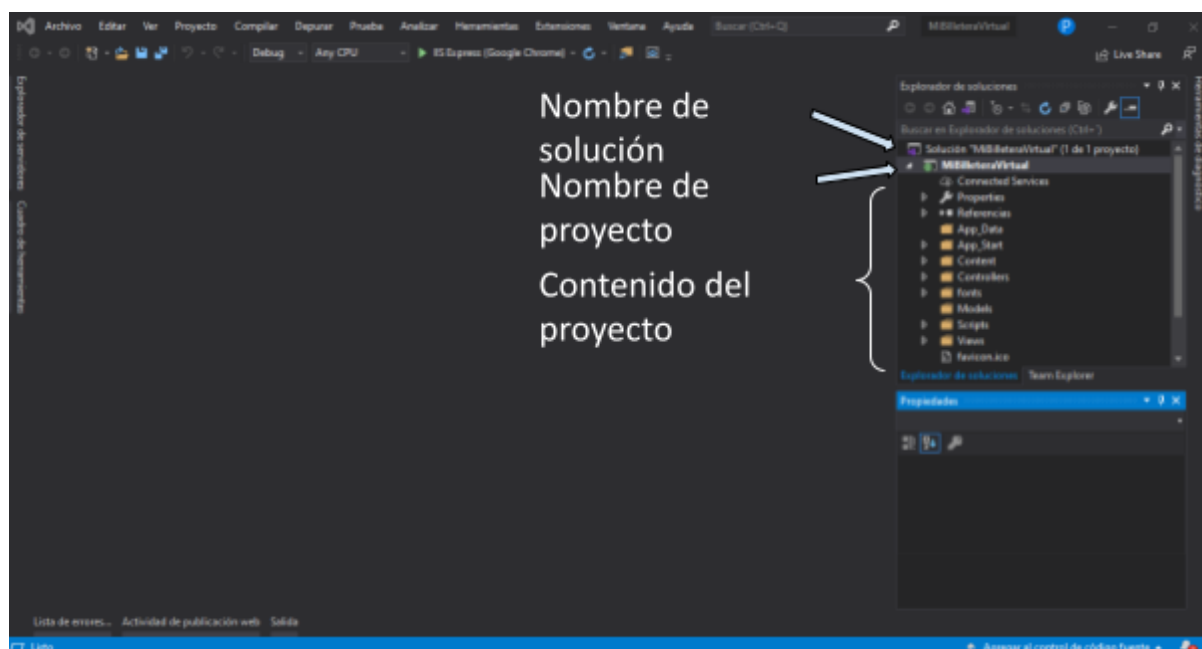


Figura 7: Visualización del proyecto creado.

En el explorador de soluciones podremos observar todos los archivos y carpetas creadas en el proyecto.

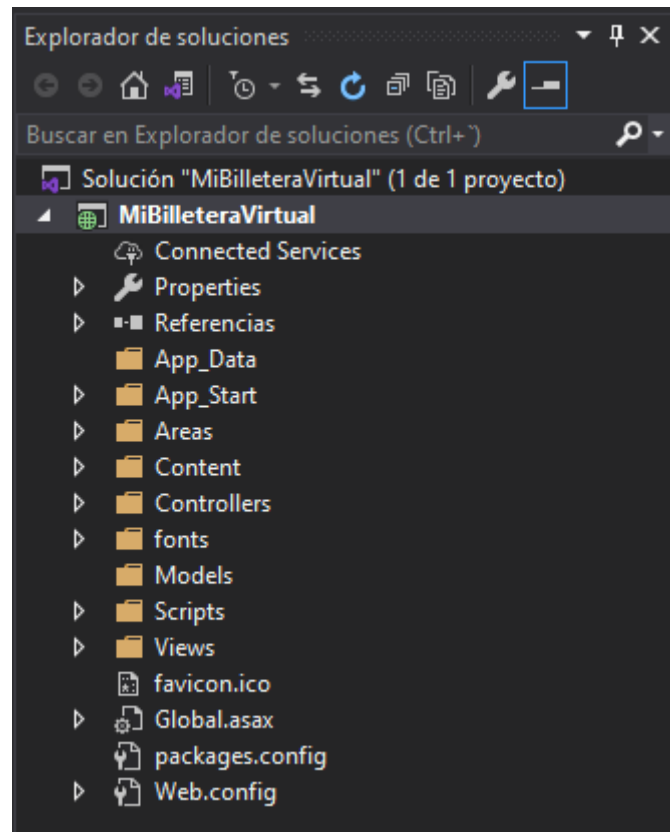


Figura 8: Explorador de soluciones.

Dentro de las carpetas más relevantes, tenemos la carpeta **App_Start**. Ésta contiene las clases de configuración que se ejecutarán cuando inicie la aplicación. Por defecto incluye las clases *BundleConfig*, *FilterConfig* y *RouteConfig*.

La carpeta **Areas** contiene archivos de ayuda y configuración de la API.

La carpeta **Content** contiene archivos estáticos (CSS, imágenes, íconos, etc.). Al crear nuestra aplicación en MVC 5 ya se crean los archivos de Bootstrap para nuestro estilo.

La carpeta **Controllers** contiene archivos de clases para los controladores. El controlador maneja la solicitud de los usuarios y devuelve una respuesta. Todos los nombres de los controladores deben finalizar con la palabra *Controller*. Ej: *AlumnoController*, *ClienteController*, *TransferenciaController*, etc.

La carpeta **Models** contiene archivos de clase de modelos. Normalmente la clase de modelos incluye propiedades públicas que serán utilizadas por la aplicación para almacenar y manipular los datos de la misma.

La carpeta **Views** contiene archivos HTML, los cuales el navegador podrá renderizar para su visualización.

El archivo **Global.asax** nos permite crear código que se ejecuta en respuesta a eventos de nivel de la aplicación.

El archivo **packages.config** administra los archivos de paquete Nuget, para realizar el seguimiento de las versiones que estén instaladas en nuestra aplicación.

El archivo **web.config** contiene las configuraciones necesarias para poder ejecutar la aplicación, ya sean configuraciones de ensamblados, de cadenas de conexión, de versiones de framework, etc.

ORM

Un ORM (por sus siglas en inglés *Object Relational Mapping* o Mapeo Objeto Relacional) nos permite realizar un mapeo entre los objetos de una base de datos virtual creada en nuestra aplicación y los objetos de una base de datos real, siendo el ORM el encargado de ejecutar sobre la base de datos física todas las acciones CRUD (Create, Read, Update, Delete).

Además, la funcionalidad más directa que puede ofrecer un ORM es la de “liberarnos” de escribir o generar las consultas o *queries* en SQL y gestionar la persistencia de datos en nuestra base de datos.

Entity Framework

Teniendo en claro el concepto de ORM y destacando algunas de sus principales ventajas, podremos introducirnos en el concepto de Entity Framework (EF), ya que será el framework que utilizaremos para desarrollar nuestra aplicación.

EF es un framework open source que ofrece Microsoft para un ORM en las aplicaciones .NET. Éste permite a los desarrolladores utilizar los objetos de .NET para convertir sus estructuras de datos en clases, reduciendo de modo considerable la cantidad de código que haría falta para realizar la gestión de los datos en nuestra base de datos.

Dentro de sus principales funcionalidades encontramos las siguientes:

- Nos permite generar un modelado (EDM - *Entity Data Model* o *Modelo de Datos de Entidad*) con entidades de diferentes tipos de datos, los que tendrán sus respectivos get y set, permitiendo utilizar estos modelos para realizar la persistencia y consulta de los datos en la base de datos;
- La utilización de transacciones por defecto al realizar consultas a la base de datos, permitiendo modificar la gestión de transacciones en el caso de ser necesario;
- Memoria caché que previene sobrecargas de la BD si recibiera la misma consulta de manera reiterada;
- La posibilidad de migraciones;
- Ajuste de modelos mediante *Data Annotations* para adaptar las validaciones de acuerdo a las necesidades especificadas.
- Consultas *LINQ* sin necesidad de realizar consultas SQL ya que el gestor de bases de datos se encargará de la traducción.

Creando Modelo de Datos de Entidad

Antes de crear nuestro modelo, crearemos la base de datos a la cual llamaremos **MI_BASE_DE_DATOS**, para eso iremos a SQL Server Management Studio y crearemos la base de datos, luego en una nueva ventana de consulta ejecutaremos el script de creación.

Para poder crear el modelo en nuestra aplicación deberíamos seguir los siguientes pasos: Primero hacemos click derecho en la carpeta *Models*, luego seleccionaremos el submenú *Agregar* y luego hacemos click en *Nuevo elemento*.

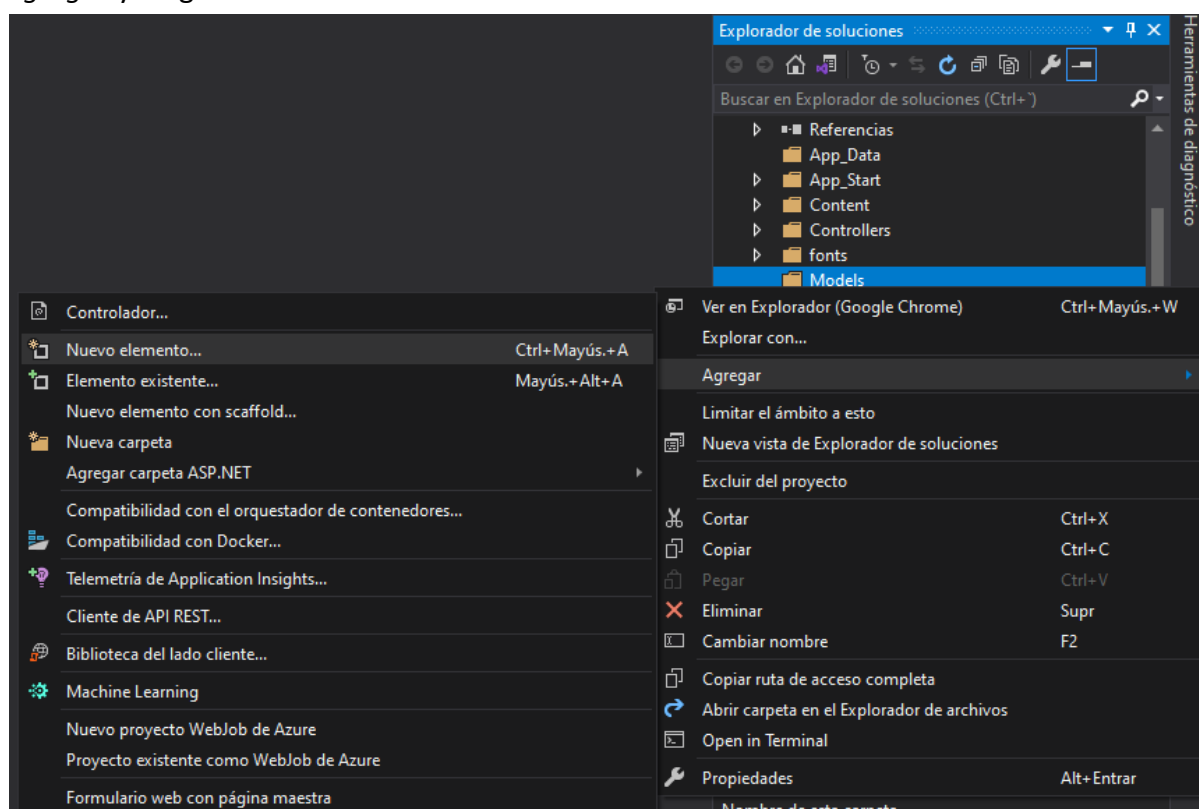


Figura 9: Creación de nuevo modelo

Una vez abierta la ventana modal de elementos seleccionamos el menú *Visual C#*, luego el menú *Datos*, luego la opción *ADO.NET Entity Data Model*, asignamos el nombre de nuestro modelo y por último hacemos click en *Agregar*.

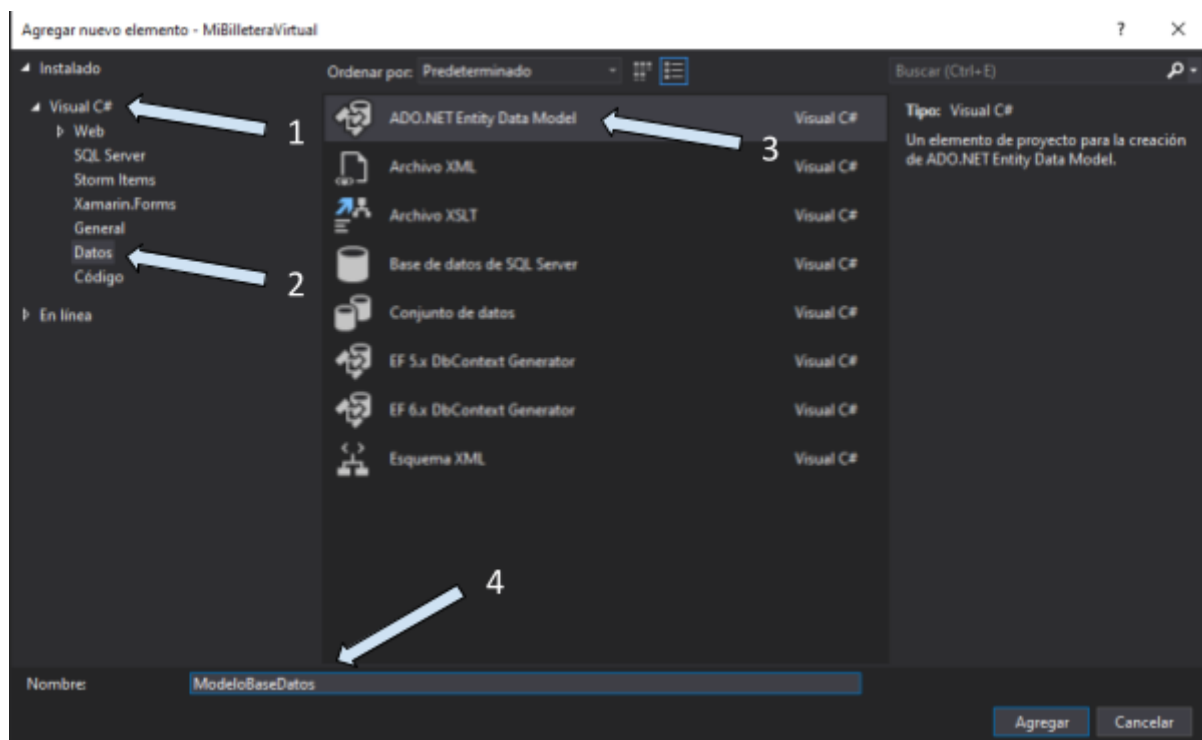


Figura 10: Creación de EDM.

Luego se abrirá el *Asistente para Entity Data Model*, seleccionamos la opción *EF Designer desde base de datos* y hacemos click en *Siguiente*.

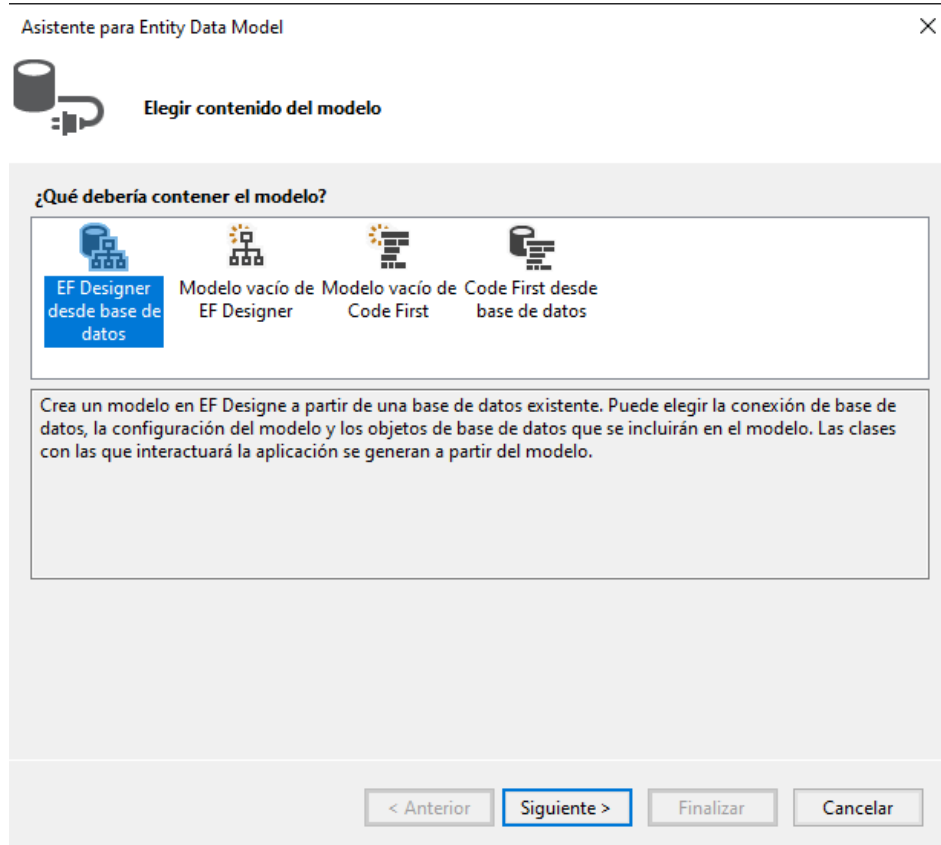
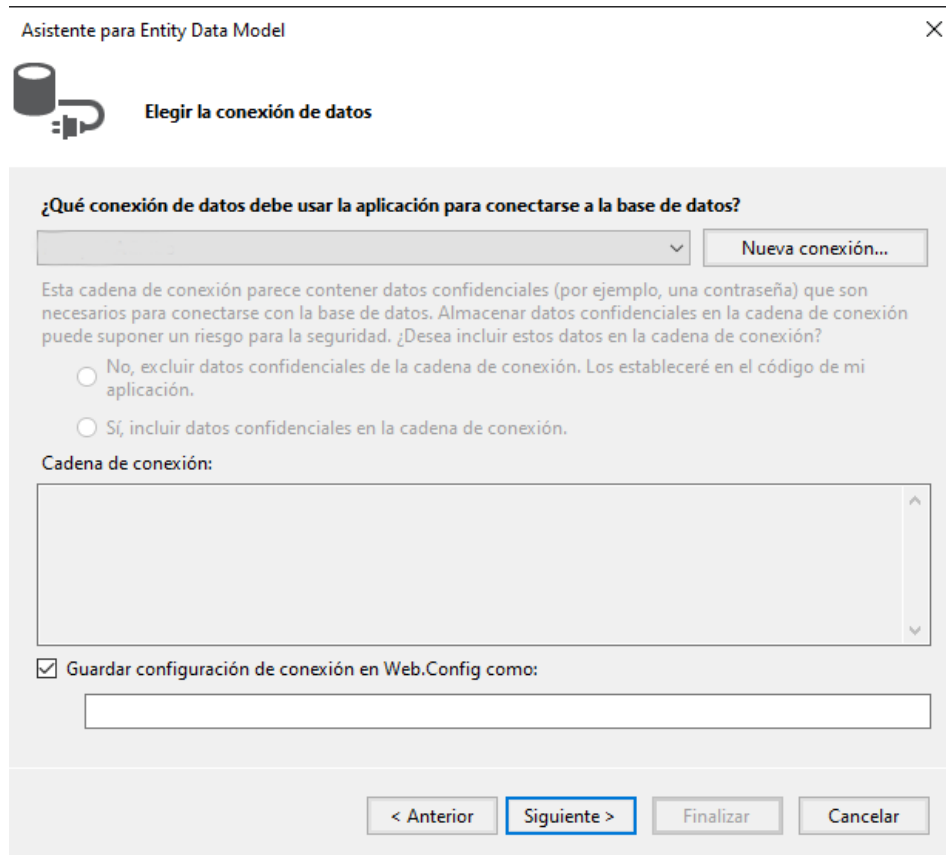



Figura 11: Selección de modelo.

Luego de seleccionar el modelo se abrirá la pantalla de gestión de conexión de datos, hacemos click en el botón *Nueva Conexión*.



Asistente para Entity Data Model

 Elegir la conexión de datos

¿Qué conexión de datos debe usar la aplicación para conectarse a la base de datos?

Nueva conexión...

Esta cadena de conexión parece contener datos confidenciales (por ejemplo, una contraseña) que son necesarios para conectarse con la base de datos. Almacenar datos confidenciales en la cadena de conexión puede suponer un riesgo para la seguridad. ¿Desea incluir estos datos en la cadena de conexión?

☐ No, excluir datos confidenciales de la cadena de conexión. Los estableceré en el código de mi aplicación.

☐ Sí, incluir datos confidenciales en la cadena de conexión.

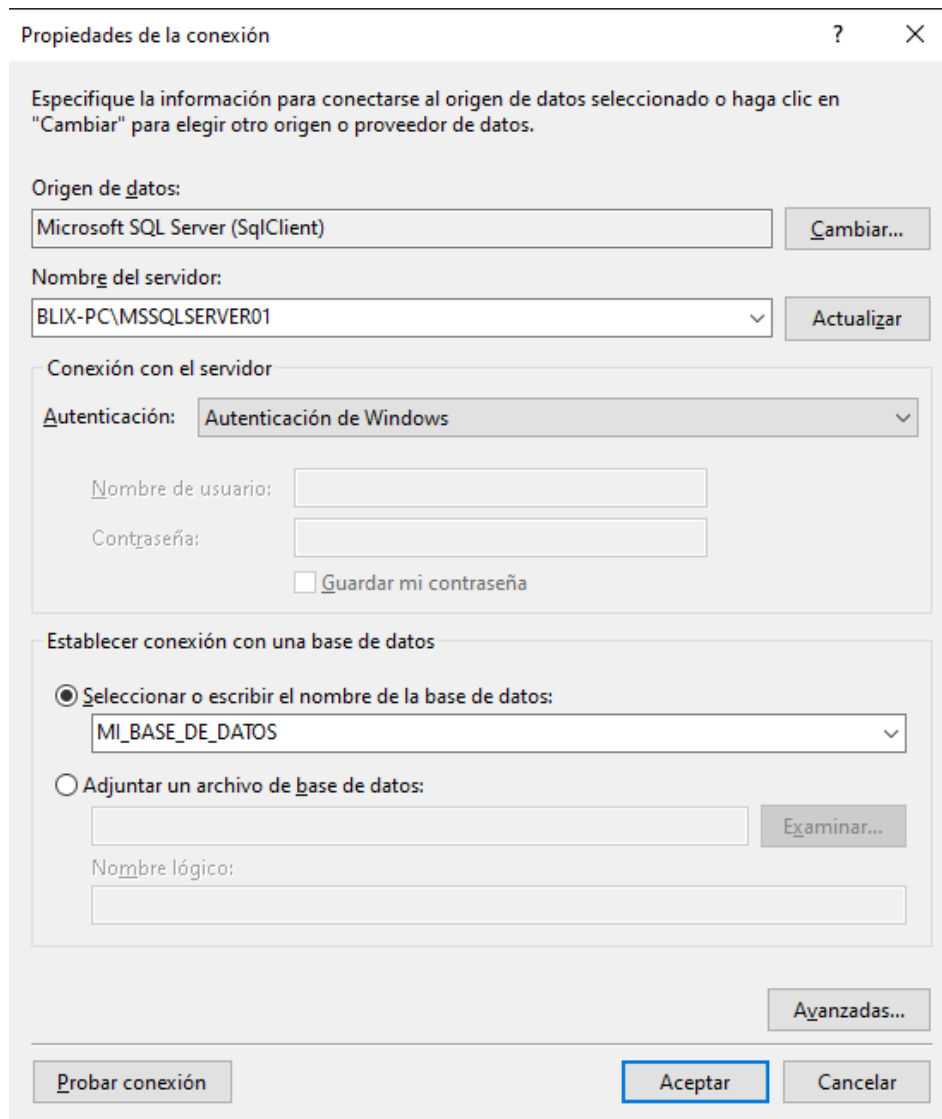
Cadena de conexión:

☒ Guardar configuración de conexión en Web.Config como:

< Anterior **Siguiente >** Finalizar Cancelar

Figura 12: Creación de la cadena de conexión

En la ventana de propiedades de conexión podremos seleccionar nuestro motor de bases de datos con la instancia, el modo de autenticación y la base de datos con la cual mapeamos nuestro modelo.



Propiedades de la conexión

Especifique la información para conectarse al origen de datos seleccionado o haga clic en "Cambiar" para elegir otro origen o proveedor de datos.

Origen de datos:
Microsoft SQL Server (SqlClient) Cambiar...

Nombre del servidor:
BLIX-PC\MSSQLSERVER01 Actualizar

Conexión con el servidor

Autenticación: Autenticación de Windows

Nombre de usuario:

Contraseña:

☐ Guardar mi contraseña

Establecer conexión con una base de datos

☒ Seleccionar o escribir el nombre de la base de datos:
MI_BASE_DE_DATOS

☐ Adjuntar un archivo de base de datos:
 Examinar...

Nombre lógico:


Avanzadas...

Probar conexión Aceptar Cancelar

Figura 13: Propiedades de conexión.

Una vez establecidas todas las propiedades de conexión podremos observar los datos de conexión, como el servidor, la base de datos (La cual se guardará en nuestro archivo Web.Config), la cadena de conexión y podremos asignarle un nombre a nuestro modelo.

Asistente para Entity Data Model

 Elegir la conexión de datos

¿Qué conexión de datos debe usar la aplicación para conectarse a la base de datos?

blíx-pc\mssqlserver01.MI_BASE_DE_DATOS.dbo Nueva conexión...

Esta cadena de conexión parece contener datos confidenciales (por ejemplo, una contraseña) que son necesarios para conectarse con la base de datos. Almacenar datos confidenciales en la cadena de conexión puede suponer un riesgo para la seguridad. ¿Desea incluir estos datos en la cadena de conexión?

☐ No, excluir datos confidenciales de la cadena de conexión. Los estableceré en el código de mi aplicación.

☐ Sí, incluir datos confidenciales en la cadena de conexión.

Cadena de conexión:

```
metadata=res://*/Models.ModeloBaseDatos.csdl|res://*/Models.ModeloBaseDatos.ssdl|
res://*/Models.ModeloBaseDatos.msl;provider=System.Data.SqlClient;provider connection string="data
source=BLIX-PC\MSSQLSERVER01;initial catalog=MI_BASE_DE_DATOS;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

☒ Guardar configuración de conexión en Web.Config como:

MI_BASE_DE_DATOSEntities

< Anterior Siguiente > Finalizar Cancelar

Figura 14: Selección de cadena de conexión creada.

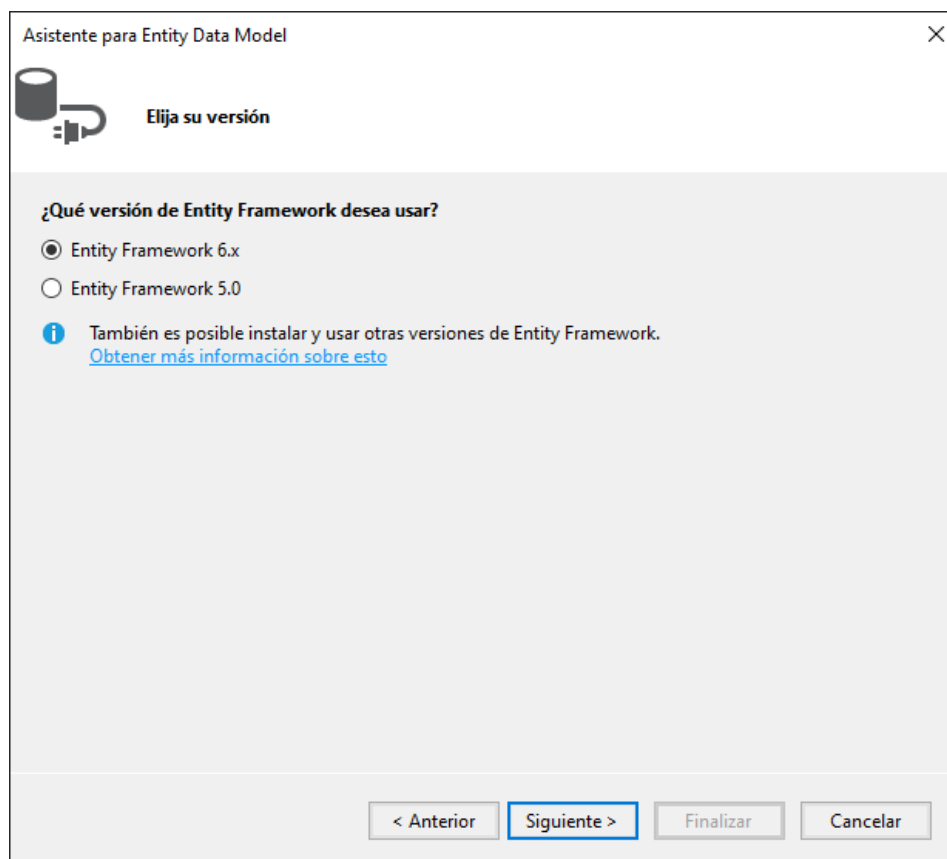


Figura 15: Selección de versión de EF.

Una vez que tengamos la conexión creada y hayamos seleccionado la versión de EF tendremos que seleccionar las tablas que queremos que estén en nuestro EDM.

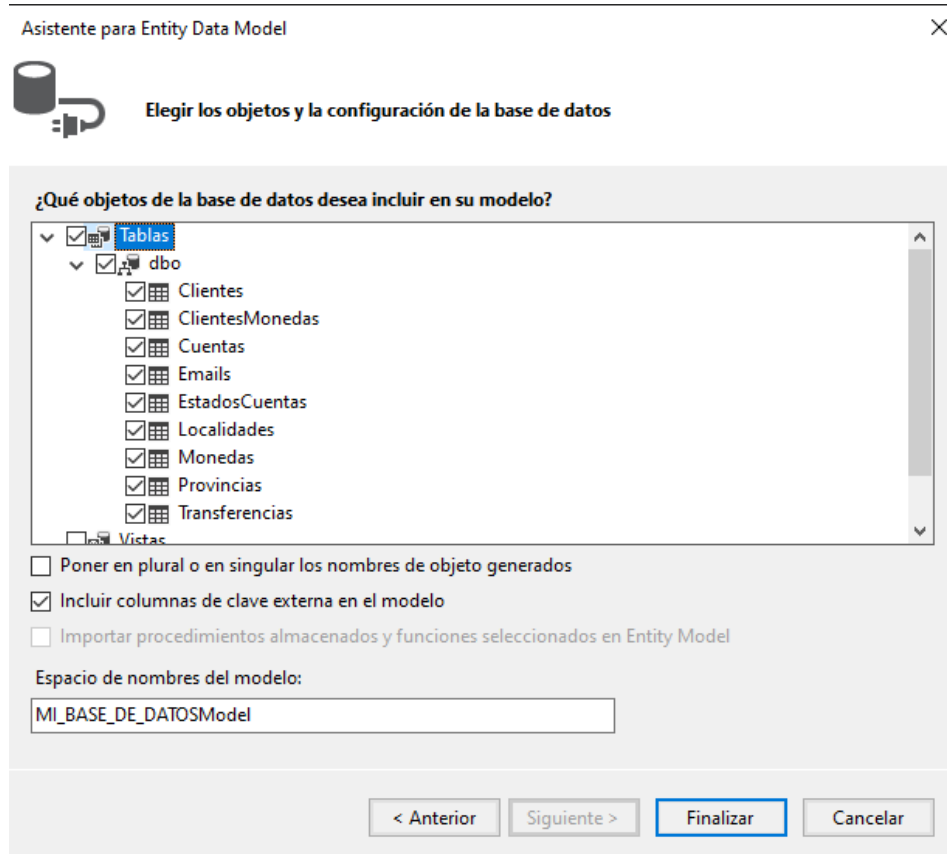


Figura 16: Selección de tablas a agregar en nuestro modelo.

Ya creado nuestro EDM, el diagrama se verá de la siguiente manera.

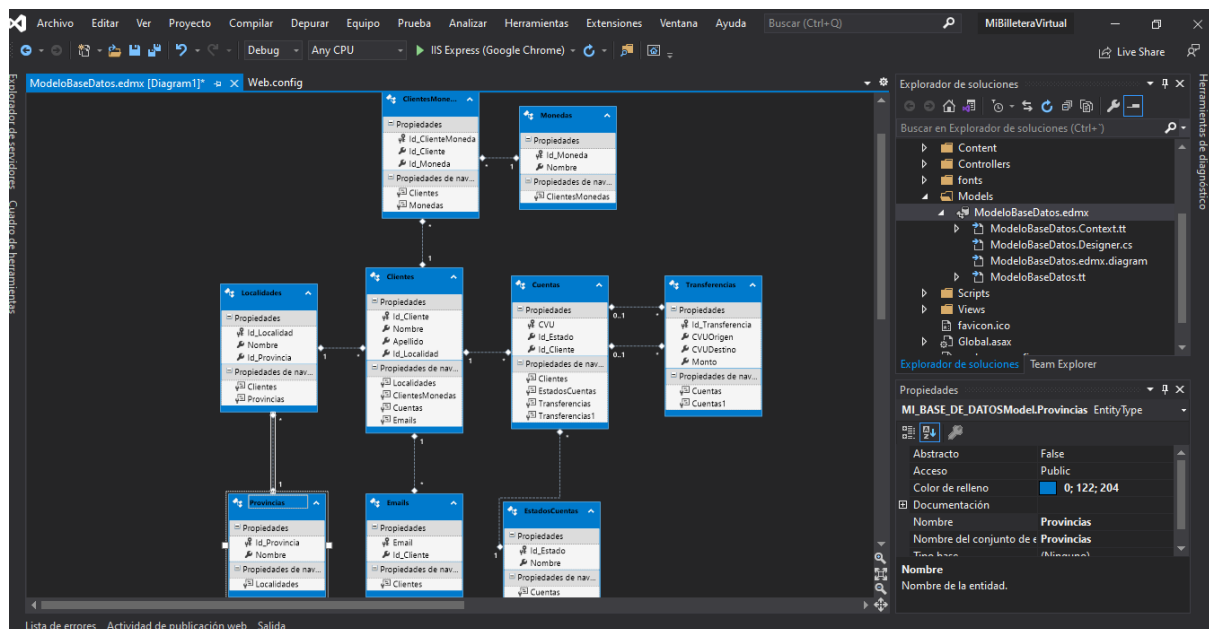


Figura 17: Diagrama de base de datos.

Una particularidad de EF es que si queremos trabajar con alguna tabla de nuestra base de datos que no contenga un ID autoincremental, nos arrojará error. Por ello, para obtener el diagrama como se ve en la imagen anterior, debemos realizar algunos cambios en nuestra base de datos, los cuales son agregar la columna *Id_ClienteMoneda* (PK) de tipo *INT* en la tabla *CientesMonedas* y la columna *Id_Transferencia* (PK) de tipo *INT* en la tabla *Transferencias*.

Creando un controlador

Para agregar un controlador debemos seguir los siguientes pasos:

Primero hacemos click derecho en la carpeta *Controllers*, luego seleccionamos el menú *Agregar* y luego seleccionamos la opción *Controlador*...

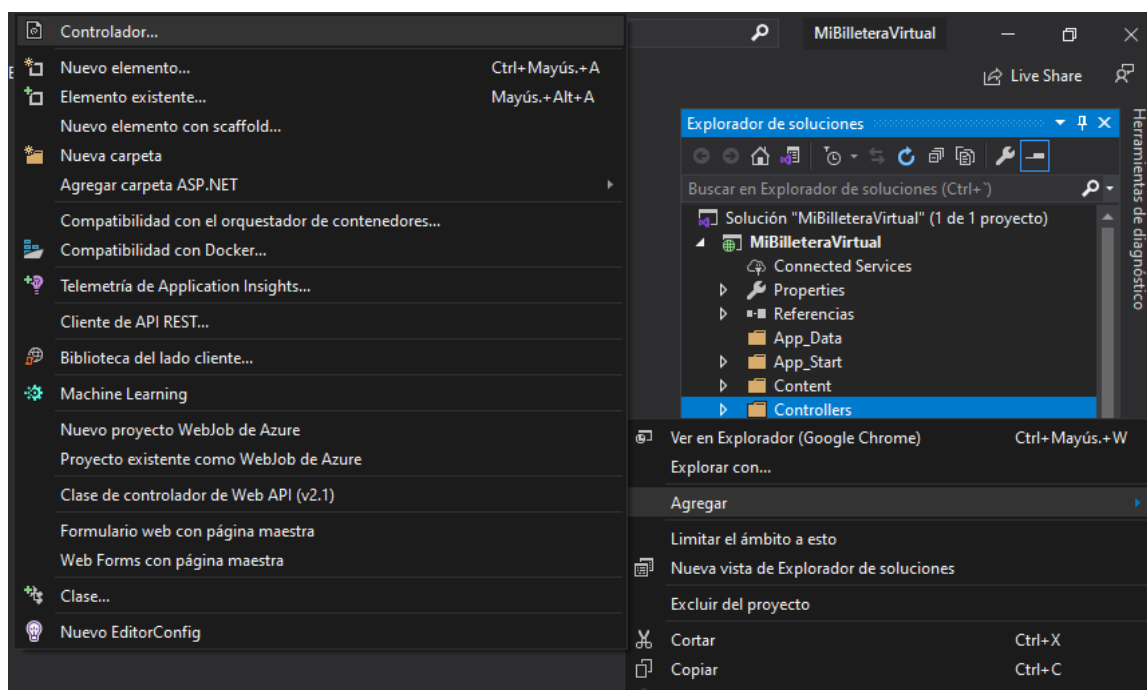


Figura 18: Creación de controlador.

Una vez seleccionada la opción para crear un nuevo controlador, seleccionaremos el menú Web API y luego la opción *Controlador de Web API 2 - en blanco* y hacemos click en *Agregar*.

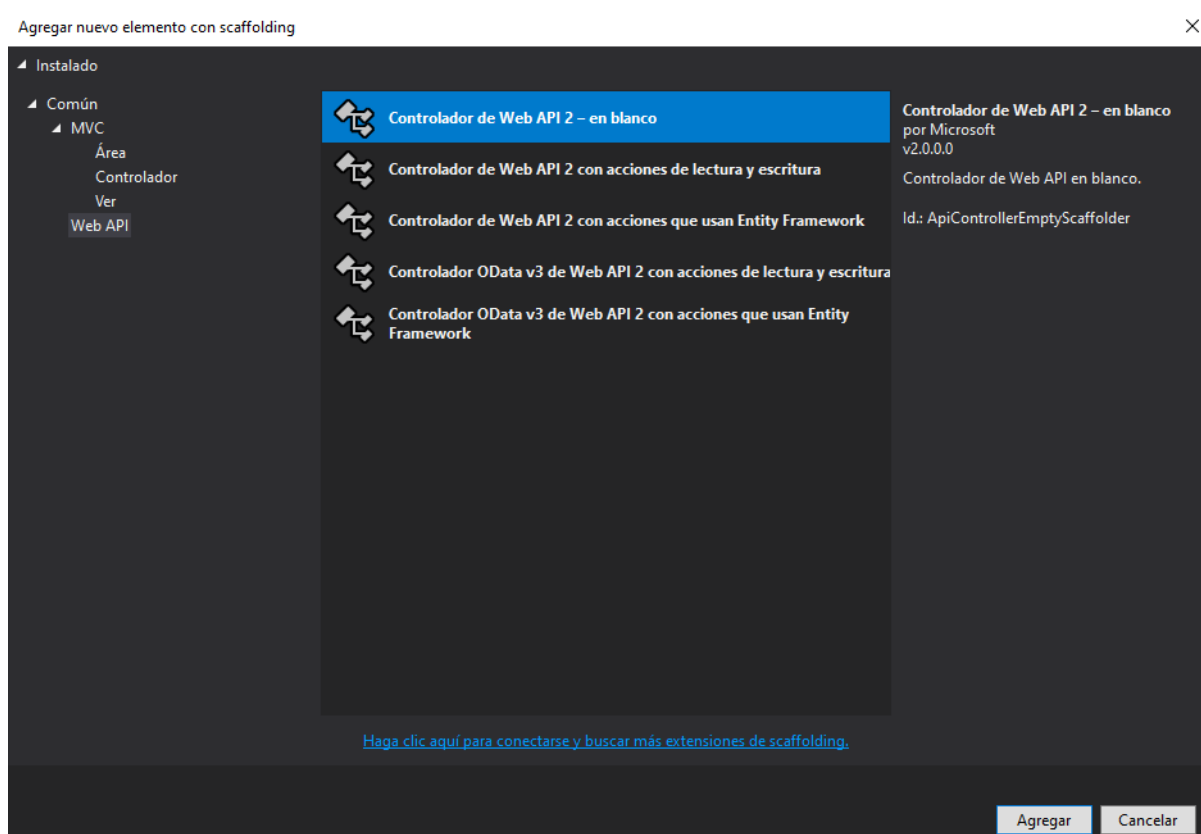


Figura 19: Selección de controlador en blanco.

Una vez seleccionado el tipo de controlador en blanco, le asignaremos un nombre, el cual finalizará con la palabra *Controller*.

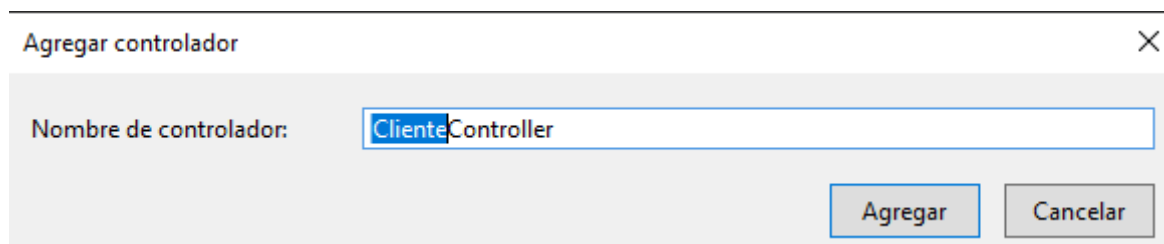


Figura 20: Asignación de nombre del controlador.

El controlador creado recientemente será el encargado de comunicarse con nuestro modelo para realizar las operaciones CRUD en la base de datos. Agregado nuestro primer controlador, deberíamos ver el proyecto de la siguiente manera:

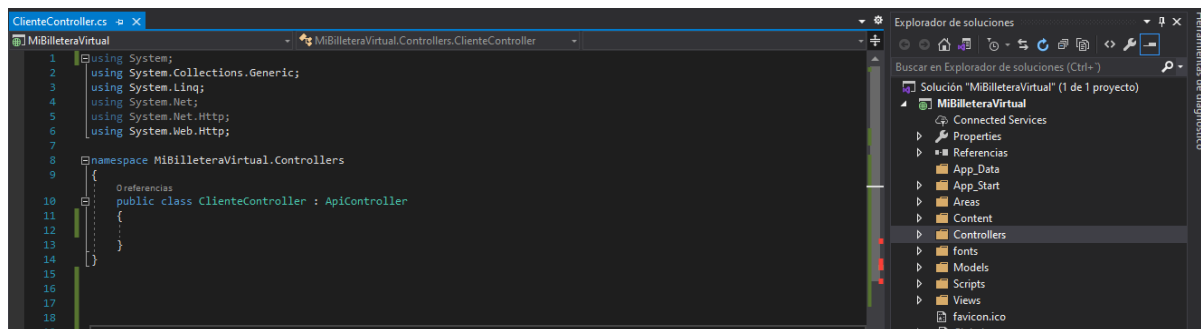


Figura 21: Pantalla de código por defecto al crear el controlador.

Por último, en el archivo *Global.asax* agregaremos dentro del método *Application_Start()* el siguiente código:

```

HttpConfiguration config = GlobalConfiguration.Configuration;
config.Formatters.JsonFormatter
    .SerializerSettings
    .ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore;
    
```

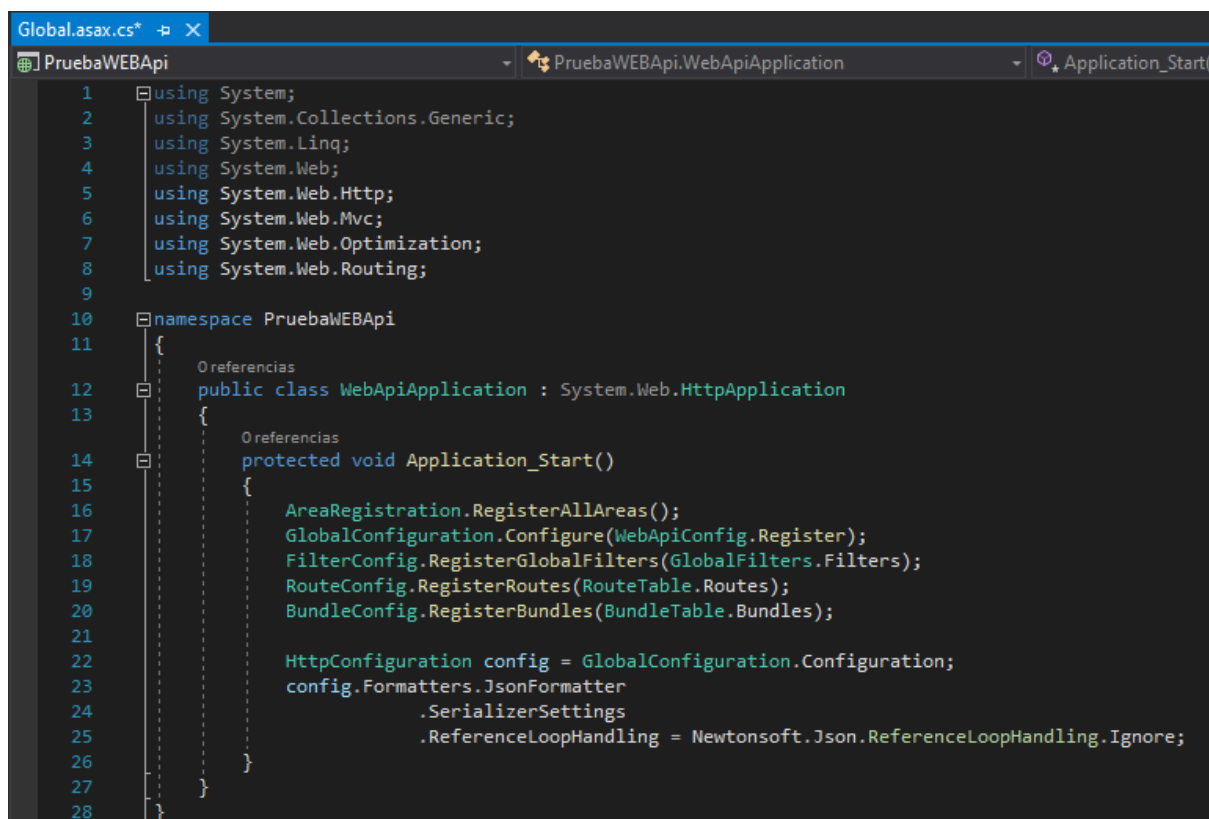


Figura 22: Archivo *Global.asax*

LINQ

LINQ (Language Integrated Query) es una extensión integrada en el lenguaje C# que nos permite manipular con mayor facilidad las colecciones de datos gracias a los métodos de extensión y a las expresiones lambda, reduciendo considerablemente la cantidad de código a desarrollar.

Ya creado nuestro controlador podremos realizar modificaciones en el código para darle funcionalidad.

Para nuestro ejemplo vamos a crear una clase *Cliente* y dos métodos, uno para devolver una lista de nombres de clientes y otro para devolver un objeto *Cliente* filtrando por el *Id_Cliente* que recibirá el método por parámetro.

```
public class Cliente
{
    Oreferencias
    public int Id_Cliente { get; set; }
    Oreferencias
    public string Nombre { get; set; }
    Oreferencias
    public string Apellido { get; set; }
}
```

Figura 23: Creación de clase *Cliente*.

Teniendo creada nuestra clase, podremos continuar con la creación de los métodos.

Comenzaremos primero creando el método que nos retornará una lista de nombres. Para eso debemos especificar primero el modificador de acceso (public, private, protected, internal, protected internal o private protected), luego el tipo de datos que va a retornar (En nuestro ejemplo *List<string>* significa que devolverá una lista con cadenas de caracteres), luego con el nombre de nuestro método y por último el algoritmo que necesitamos finalizando el método con la instrucción *return listaNombres;* (la palabra reservada *return* especifica que está devolviendo un objeto y *listaNombres* es el objeto que devuelve, en nuestro caso se cargó anteriormente con 3 objetos de tipo *Cliente*).

Con **LINQ** no necesitamos recorrer nuestra lista con bucles ni con condicionales, solamente debemos especificar qué atributo necesitamos que devuelva, como se puede ver a continuación.

```
public List<string> ObtenerNombreClientes()
{
    var listaClientes = new List<Cliente>
    {
        new Cliente {Id_Cliente = 1, Nombre = "Pedro",Apellido= "Arano"},
        new Cliente {Id_Cliente = 2, Nombre = "Gladys",Apellido= "Barrera"},
        new Cliente {Id_Cliente = 3, Nombre = "Jorge",Apellido= "Vargas"}
    };

    var listaNombres = listaClientes.Select(x => x.Nombre).ToList();

    return listaNombres;
}
```

Figura 24: Método para retornar una lista de objetos de tipo string.

Continuando con el ejemplo es el momento de crear el método que devuelve un objeto de tipo *Cliente*, nuestra clase creada anteriormente. En este caso en vez de retornar un objeto *List<Cliente>*, retornará un solo ítem de nuestra lista. Para eso primero debemos filtrar con la cláusula *Where* y establecer la condición que se debe cumplir con la expresión lambda *x => x.Id_Cliente == idCliente* (donde *Id_Cliente* es el atributo de nuestra clase y la variable *idCliente* se recibe como parámetro y es de tipo entero). Esto nos daría por resultado una lista de clientes, la cual, para obtener un solo cliente finalizamos con el método *FirstOrDefault()*.

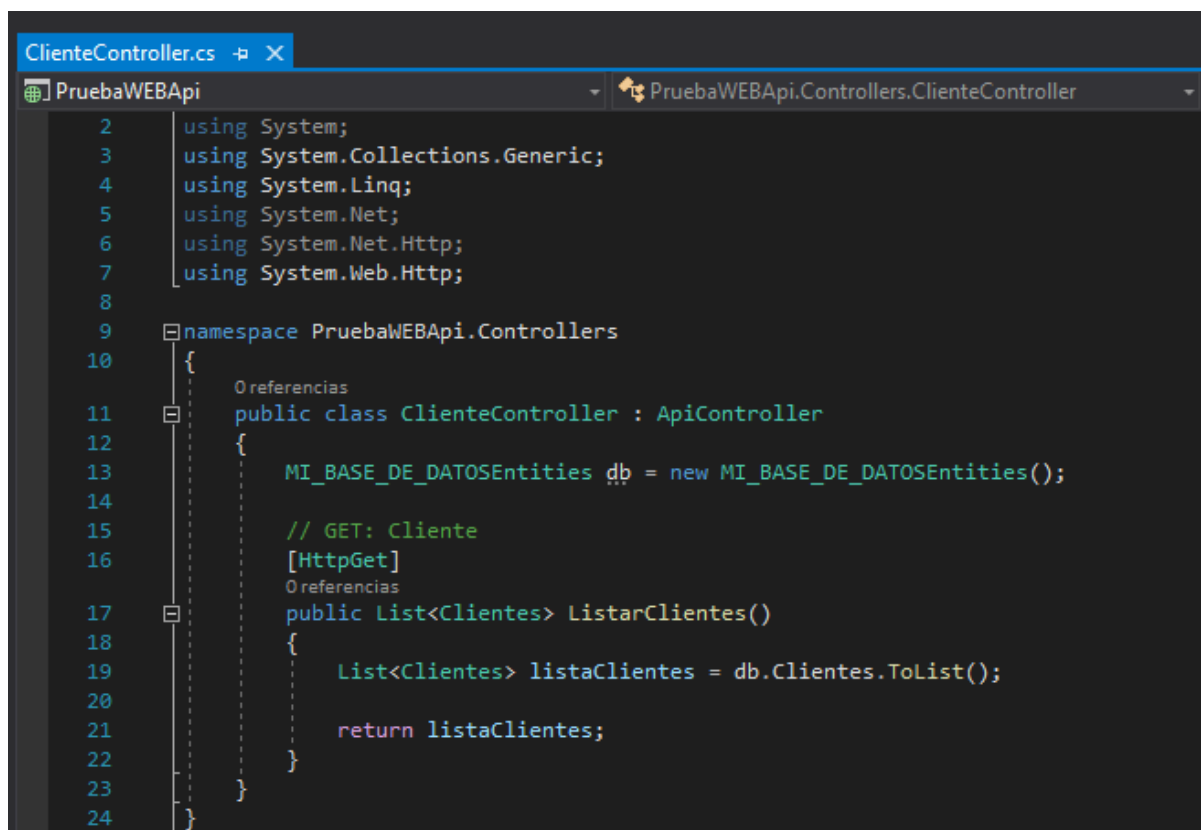
```
public Cliente ObtenerClientePorId(int idCliente)
{
    var listaClientes = new List<Cliente>
    {
        new Cliente {Id_Cliente = 1, Nombre = "Pedro",Apellido= "Arano"},
        new Cliente {Id_Cliente = 2, Nombre = "Gladys",Apellido= "Barrera"},
        new Cliente {Id_Cliente = 3, Nombre = "Jorge",Apellido= "Vargas"}
    };

    var cliente = listaClientes.Where(x => x.Id_Cliente == idCliente).FirstOrDefault();

    return cliente;
}
```

Figura 25: Método para retornar un objeto de tipo *Cliente*.

Si adaptamos el ejemplo de LINQ a nuestra aplicación, podemos crear un método para obtener la lista de clientes y los objetos relacionados a través de las PK y las FK.



```

1  ClienteController.cs
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Net;
6  using System.Net.Http;
7  using System.Web.Http;
8
9  namespace PruebaWEBApi.Controllers
10 {
11     public class ClienteController : ApiController
12     {
13         MI_BASE_DE_DATOSEntities db = new MI_BASE_DE_DATOSEntities();
14
15         // GET: Cliente
16         [HttpGet]
17         public List<Clientes> ListarClientes()
18         {
19             List<Clientes> listaClientes = db.Clientes.ToList();
20
21             return listaClientes;
22         }
23     }
24 }

```

Figura 26: Método que retorna la lista de clientes en nuestra base de datos.

Ya podremos ejecutar nuestra aplicación desde el botón *IIS Express* (Navegador predeterminado) o presionando la tecla F5.

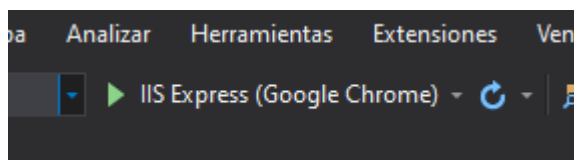


Figura 27: Botón de inicio de nuestra aplicación.

Esto nos abrirá nuestro navegador por defecto y en la barra de direcciones nos mostrará una dirección con un puerto separados por el signo de dos puntos. (<https://localhost:44366> - localhost es la dirección de nuestro equipo y el número 44366 es el puerto).

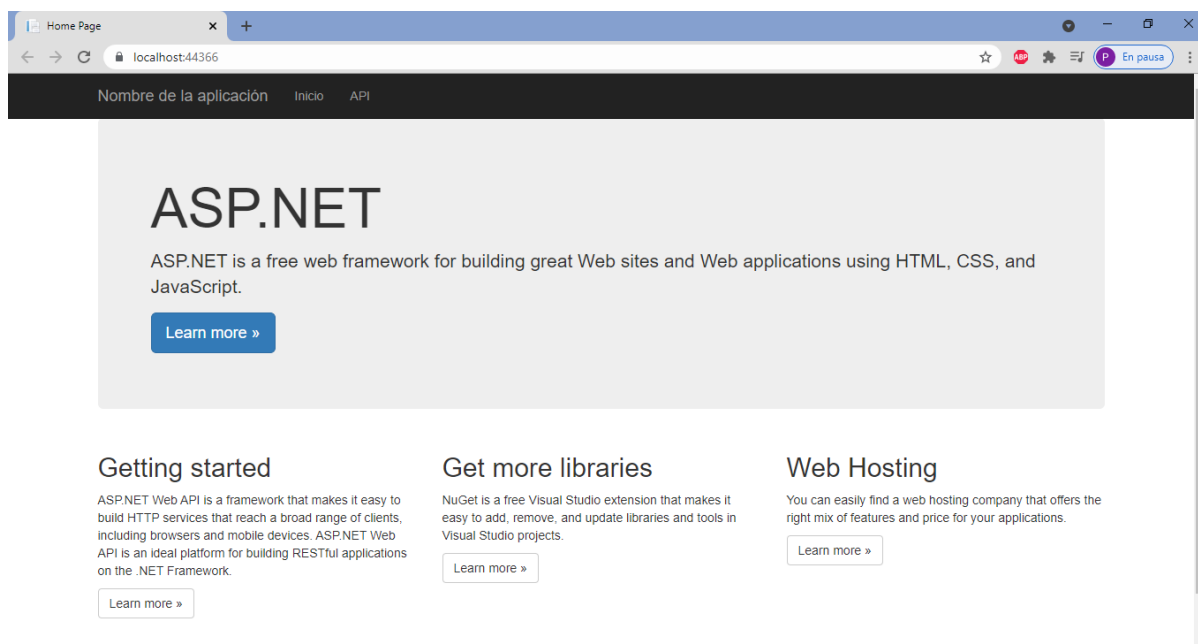


Figura 28: Pantalla de inicio del proyecto.

Cuando hacemos click en el menú *API* veremos los controladores creados y los métodos. En nuestro ejemplo tendremos el controlador *Values* (creado por defecto con la aplicación) y el controlador *Cliente*, cada uno con sus métodos, cada uno antecedido por la palabra *api*.

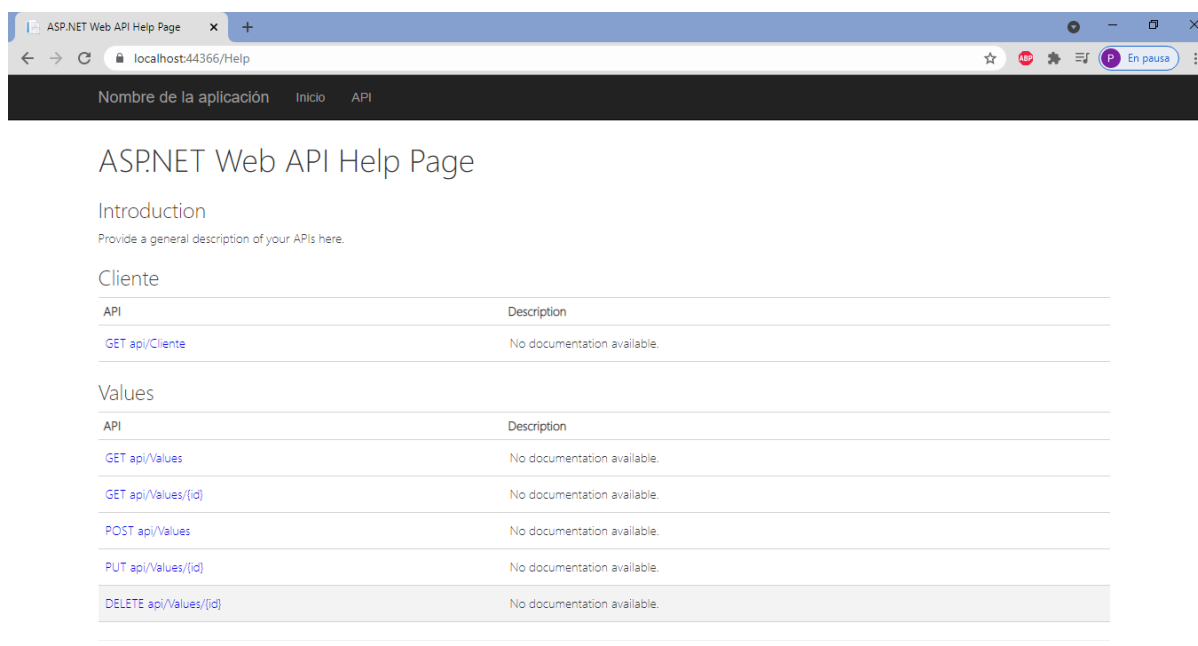


Figura 29: Pantalla de información del proyecto.

Códigos de estado HTTP

Los códigos de estado de respuesta HTTP son indicadores de una solicitud específica, informando si ésta ha sido completada correctamente.

Las respuestas se agrupan en 5 clases

- Respuestas informativas (100–199)
- Respuestas satisfactorias (200–299)
- Redirecciones (300–399)
- Errores de los clientes (400–499)
- Errores de los servidores (500–599)



Figura 30: Códigos de estado HTTP.

Métodos de petición HTTP

Con HTTP se definen distintos métodos de petición (también llamados *HTTP verbs*) para indicar que acción se desea realizar para un determinado recurso. Cada método implementa una semántica distinta, pero tienen similares características compartidas por un grupo de los mismos.

Los métodos HTTP son:

GET

Este método solicita una representación de un recurso específico. Las peticiones que utilizan el método GET sólo recuperan datos.

HEAD

Este método es similar al método GET, la diferencia radica en que el método HEAD pide una respuesta sin el cuerpo de la misma.

POST

Este método envía una entidad a un recurso específico. Este método normalmente se envía con formularios.

PUT

Este método se utiliza por lo general para actualizar el contenido de una entidad.

DELETE

Este método elimina un recurso en específico.

CONNECT

Este método establece la comunicación hacia el servidor, siendo el recurso el que lo identifica.

OPTIONS

Este método se utiliza para describir las opciones disponibles de comunicación en el canal de solicitud y respuesta.

TRACE

Este método ejecuta una prueba de ciclo de respuesta del mensaje a lo largo de la ruta al recurso de destino.

PATCH

Este método se utiliza para realizar modificaciones a un recurso.

POSTMAN

Postman es una herramienta que se utiliza para el testeo de API REST, consumiendo las mismas a través de una URL, las cuales devuelven un objeto serializado (en formato JSON o XML, principalmente). Utilizaremos esta herramienta para probar nuestra API REST.

Podremos descargar POSTMAN desde el siguiente enlace <https://www.postman.com/downloads/> y podremos ver la documentación desde aquí <https://learning.postman.com/>.

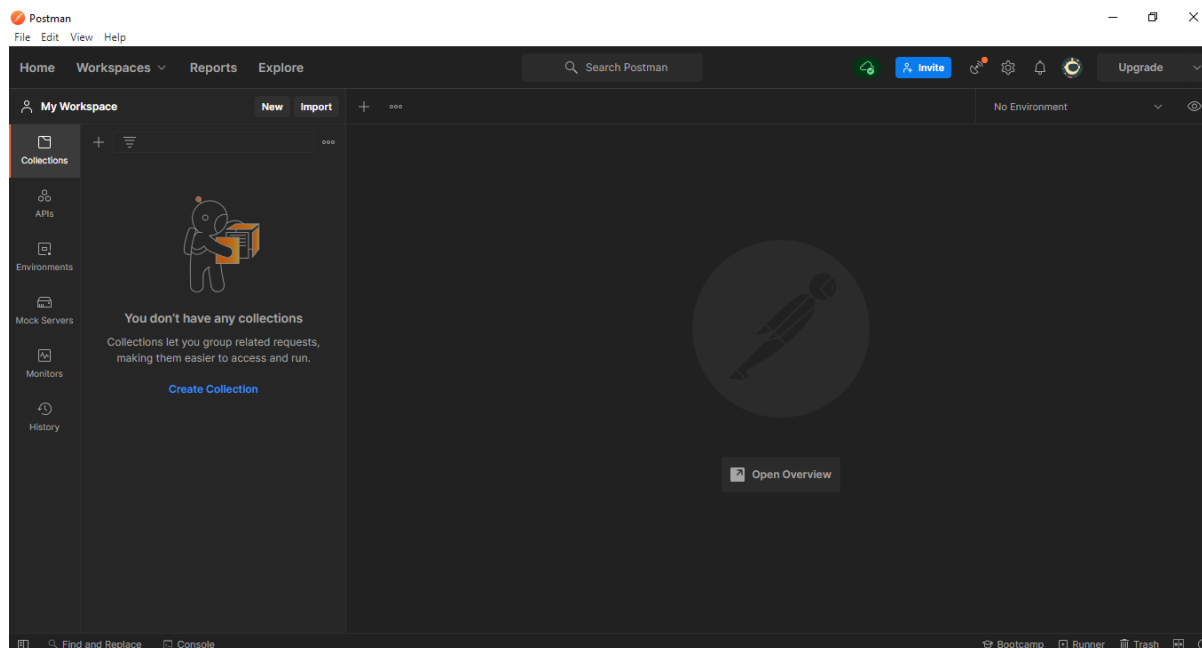


Figura 31: Interfaz de inicio de Postman.

Para probar nuestro Web API con postman debemos tener presente dos puntos fundamentales, primero que nuestra aplicación se esté ejecutando y segundo el nombre del controlador que contiene el método que queremos probar.

Una vez en Postman, hacemos click en el botón + para agregar una pestaña.

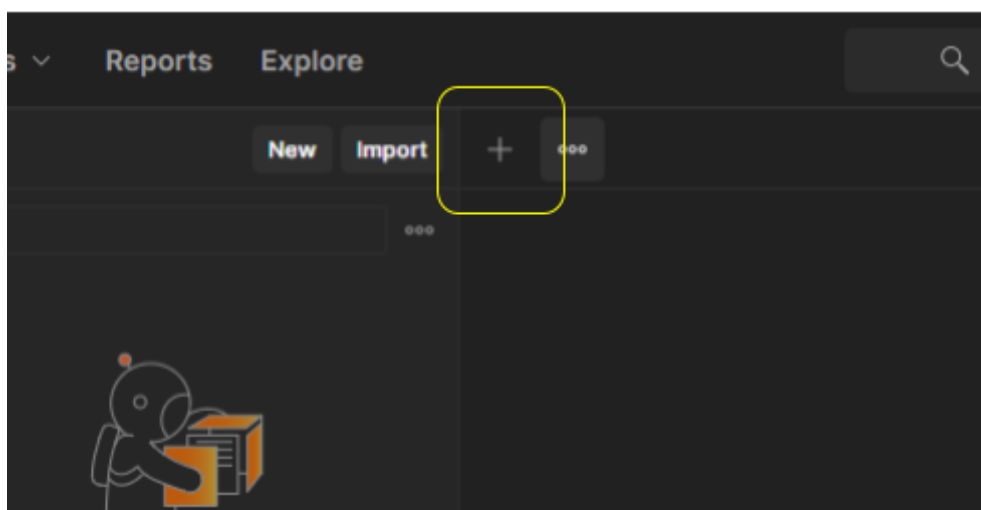


Figura 32: agregar pestaña.

Con la pestaña creada, podremos ver la pantalla la cual nos conectará con nuestra Web API.

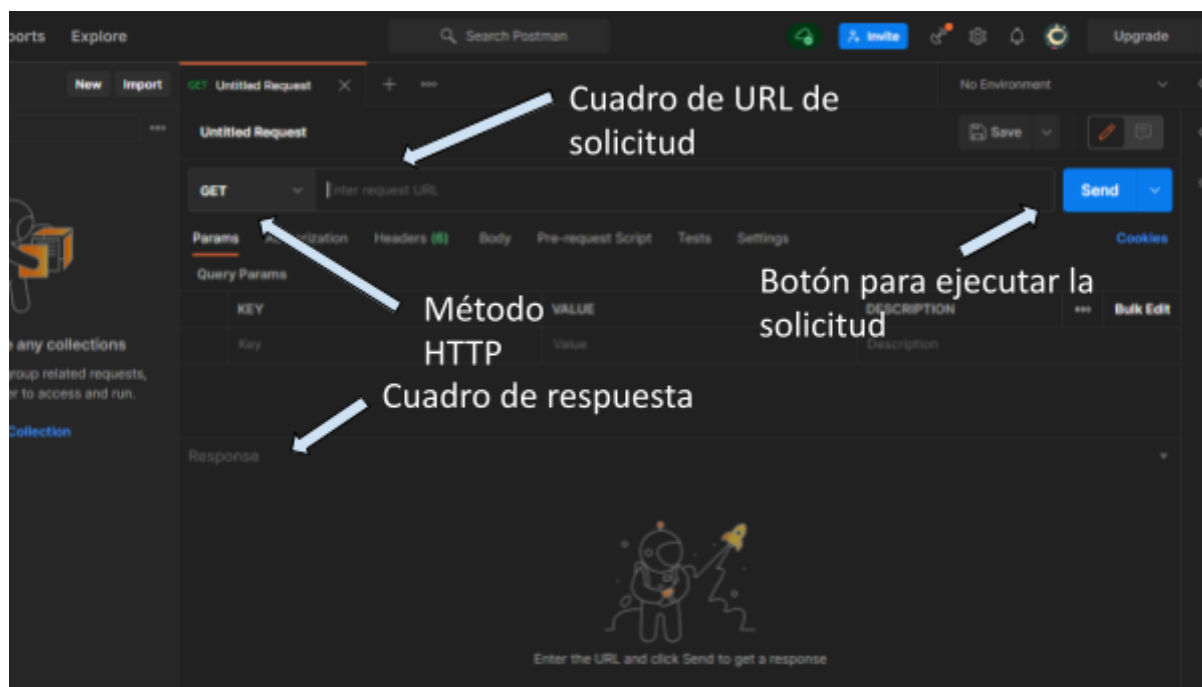


Figura 33: Pestaña para probar la API.

Prueba de Web API

Ya teniendo claros los conceptos y habiendo creado la base de datos con el proyecto, podremos probar la aplicación. Para ello ejecutaremos nuestra aplicación y seleccionaremos el menú *API* y tomaremos nota de nuestra dirección y puerto local y del nombre de nuestro controlador.

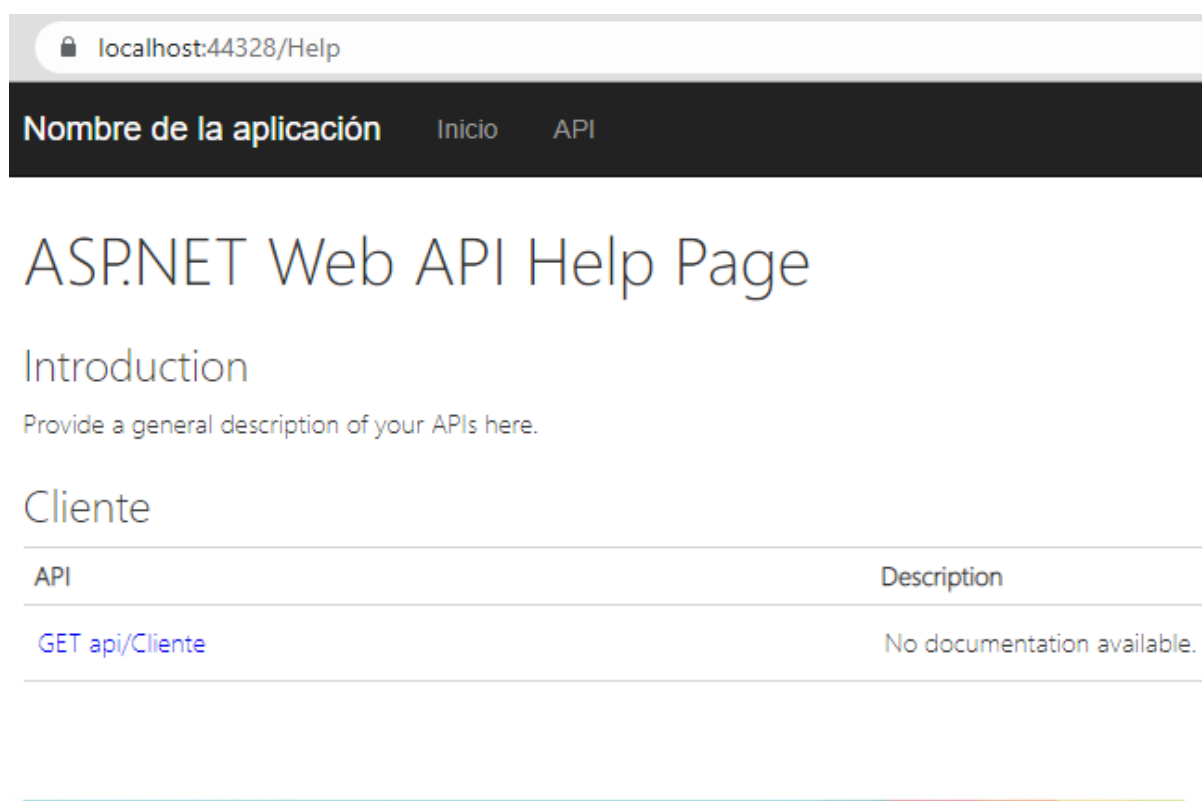


Figura 34: Página de información de nuestra API.

Si combinamos la dirección con el nombre de nuestro controlador, en nuestro ejemplo quedaría de la siguiente manera <https://localhost:44328/api/cliente>. Luego de tomar nota de nuestra URL, ejecutaremos Postman y en cuadro de URL de solicitudes, escribiremos la dirección de la API la cual tomamos nota en el paso anterior. En nuestro ejemplo, el método *ListarClientes()* es de tipo *HttpGet*, lo cual debemos especificar en las opciones de métodos HTTP (GET).

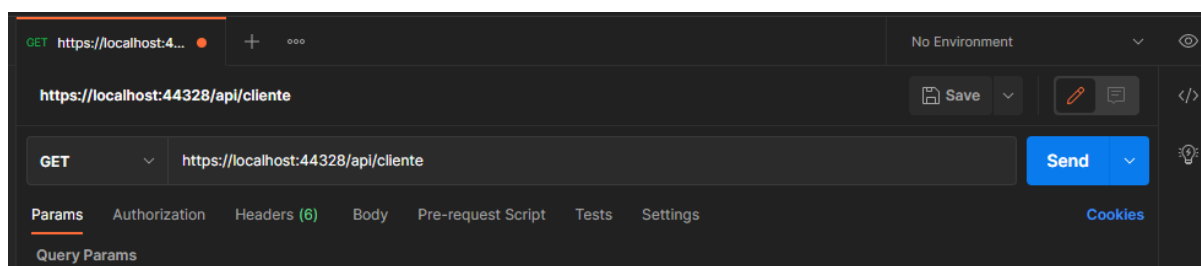


Figura 35: Códigos de estado HTTP.

Una vez completada la URL y seleccionado el método, podremos enviar la solicitud con el botón *Send*. Postman nos mostrará la respuesta en el cuadro *Response* y el estado HTTP con su código.

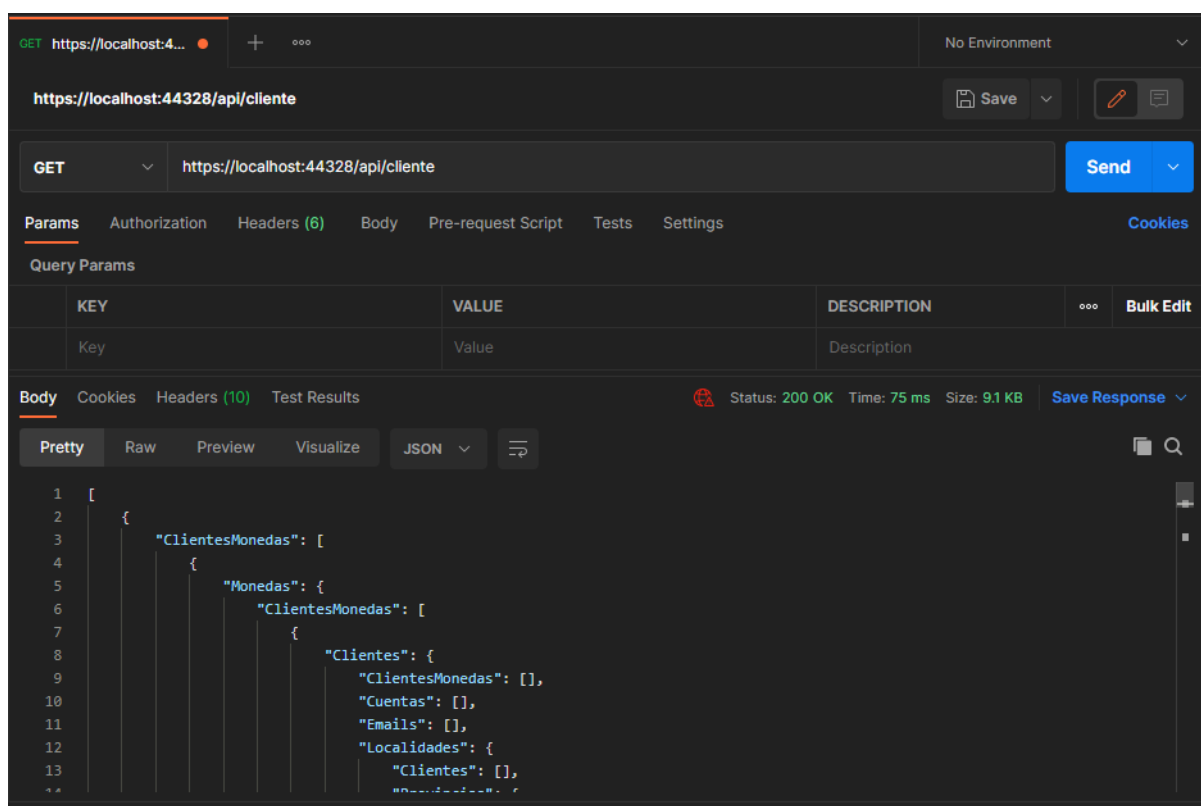


Figura 36: Respuesta de solicitud.

CRUD Usuarios, registro y login

Para completar nuestro ejemplo de CRUD, utilizaremos la entidad Usuario. Para ello, debemos crear la tabla *Usuarios* en nuestra base de datos. Una vez creada la tabla, actualizamos nuestro EDM.

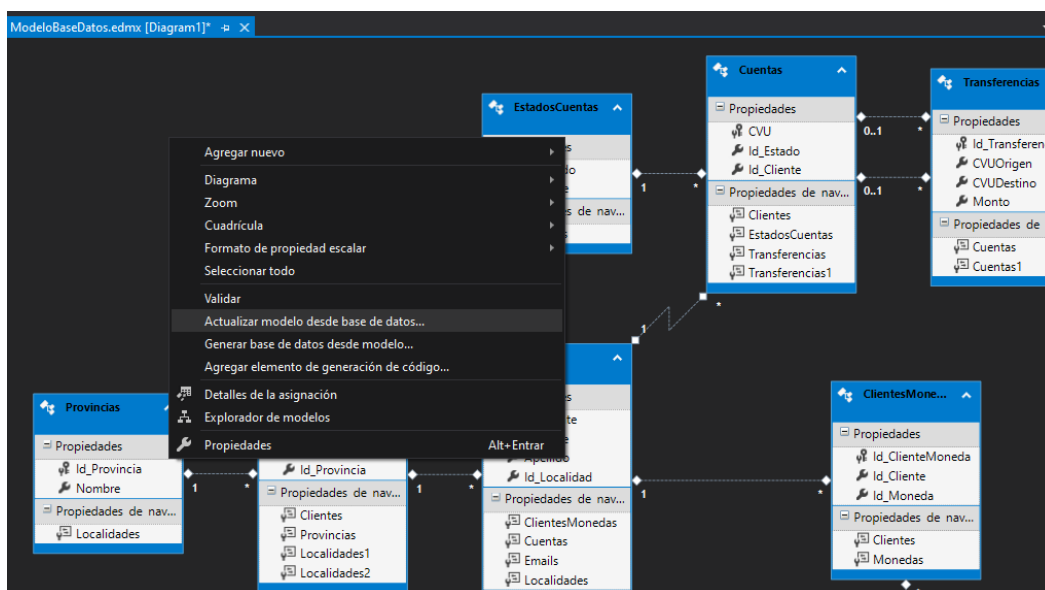


Figura 37: Modelo actualizado.

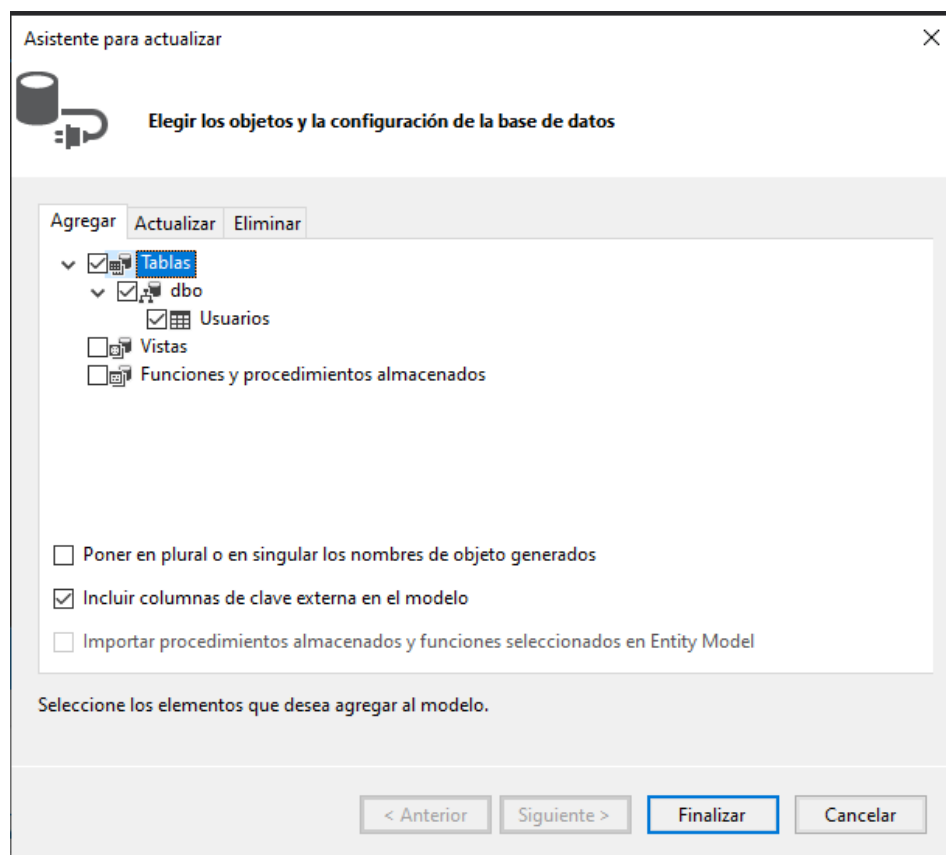


Figura 38: Selección de tablas a agregar al modelo.

Luego de actualizar nuestro EDM agregaremos un controlador llamado *UsuarioController*, esta vez con acciones de lectura y escritura para utilizar el código que nos facilita Visual Studio.

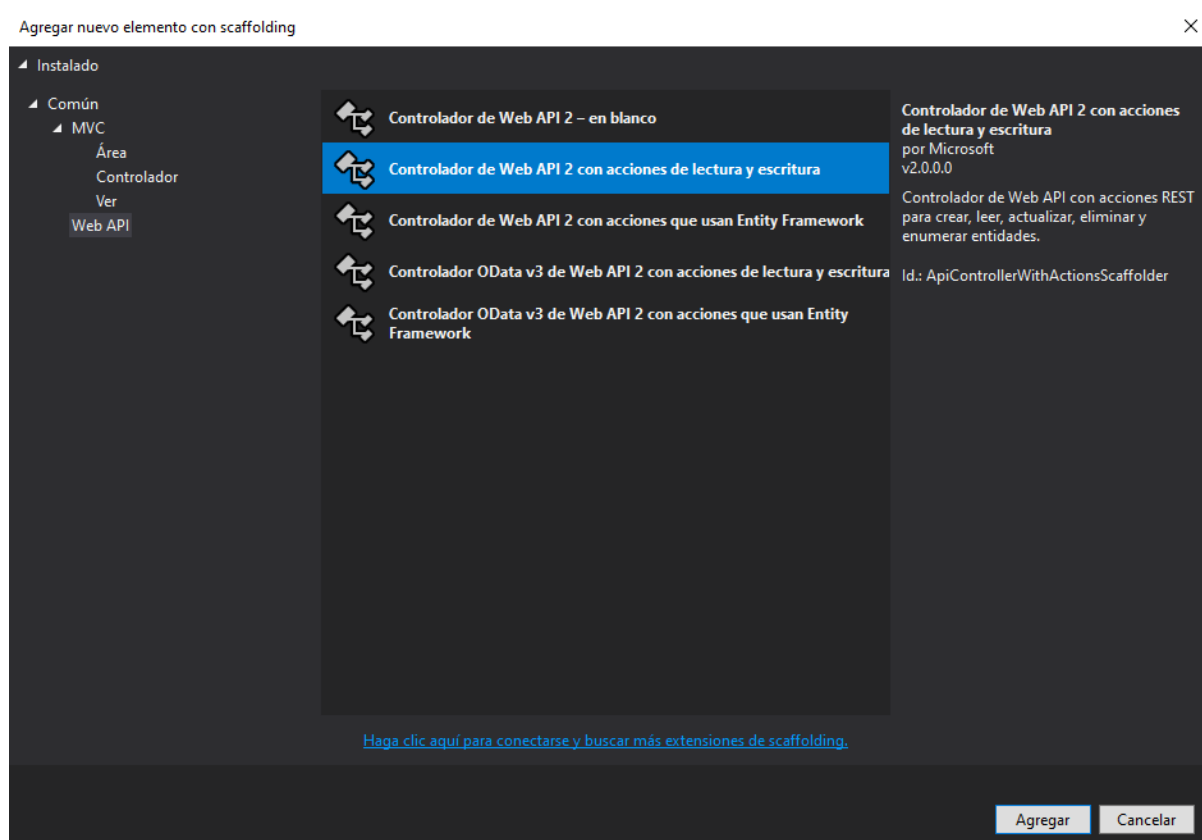


Figura 39: Agregar controlador con acciones de lectura y escritura.

Podremos modificar el código de cada método, para crear los algoritmos que necesitamos. El código creado por defecto por Visual Studio es el siguiente:

```

4  using System.Net;
5  using System.Net.Http;
6  using System.Web.Http;
7
8  namespace MiBilleteraVirtual.Controllers
9  {
10     0 referencias
11     public class UsuarioController : ApiController
12     {
13         // GET: api/User
14         0 referencias
15         public IEnumerable<string> Get()
16         {
17             return new string[] { "value1", "value2" };
18         }
19
20         // GET: api/User/5
21         0 referencias
22         public string Get(int id)
23         {
24             return "value";
25         }
26
27         // POST: api/User
28         0 referencias
29         public void Post([FromBody]string value)
30         {
31         }
32
33         // PUT: api/User/5
34         0 referencias
35         public void Put(int id, [FromBody]string value)
36         {
37         }
38
39         // DELETE: api/User/5
40         0 referencias
41         public void Delete(int id)
42         {
43         }
44     }
45 }

```

Figura 40: Código por defecto al crear controlador con acción de lectura y escritura.

Para obtener todos los usuarios modificamos el método GET, el cual nos retornará una lista de usuarios, sin condiciones (podremos ponerle las condiciones que necesitemos).

```

public class UsuarioController : ApiController
{
    MI_BASE_DE_DATOSEntities db = new MI_BASE_DE_DATOSEntities();

    // GET: api/Usuario
    0 referencias
    public IEnumerable<Usuarios> Get()
    {
        List<Usuarios> listaUsuarios = db.Usuarios.ToList();
        return listaUsuarios;
    }
}

```

Figura 41: Método para obtener la lista de usuarios.

Para probar nuestro método, ejecutamos nuestra aplicación, seleccionamos el menú *API* y tomamos nota de los métodos que se han creado. Sin detener la aplicación nos dirigimos a Postman seleccionamos el tipo de método GET, e introducimos en la URL de petición la dirección de la API (En nuestro ejemplo <https://localhost:44328/api/usuario>)

Nombre de la aplicación Inicio API	
ASP.NET Web API Help Page	
Introduction	
Provide a general description of your APIs here.	
Usuario	
API	Description
GET api/Usuario/GetLogin	No documentation available.
GET api/Usuario	No documentation available.
GET api/Usuario/{id}	No documentation available.
POST api/Usuario	No documentation available.
PUT api/Usuario/{id}	No documentation available.
DELETE api/Usuario/{id}	No documentation available.

Figura 42: Listado de métodos que tiene el controlador Usuario.

https://localhost:44328/api/usuario				Save	
GET	https://localhost:44328/api/usuario			Send	
Params	Authorization	Headers (6)	Body	Pre-request Script	Tests Settings Cookies
Query Params					
	KEY	VALUE	DESCRIPTION	...	Bulk Edit
	Key	Value	Description		

Figura 43: Prueba de método GET. Devuelve el listado de usuarios.

Para realizar el *Login* de un usuario, crearemos un método de tipo *Get* y le asignaremos la acción `[Route("api/Usuario/GetLogin")]`. Esto lo asignamos antes de nuestro método ya que es una sobrecarga del primer método *Get* y no podemos tener dos nombres idénticos (sobrecarga) de un método.

En este caso creamos una clase llamada *Login* que tiene dos propiedades (*email* y *password*). Ésta clase la utilizaremos para recibir como parámetro del método *Login*, la cual,

la palabra *[FromBody]* antecede el nombre de la clase Login, ya que le estamos informando al método que los parámetros vienen en el cuerpo del *request*. Una vez recibidos los parámetros, podremos filtrar los resultados (en nuestro caso, el usuario y la contraseña para comprobar que exista el usuario).

```
1 referencia
public class Login
{
    1 referencia
    public string email { get; set; }
    1 referencia
    public string password { get; set; }
}

[Route("api/Usuario/GetLogin")]
0 referencias
public Usuarios Get([FromBody] Login datosEntrada)
{
    Usuarios oUsuario = db.Usuarios.Where(a => a.Email == datosEntrada.email
    && a.Password == datosEntrada.password).FirstOrDefault();

    return oUsuario;
}
```

Figura 44: Clase y método para obtener el usuario que coincida el email y el password.

Para probar nuestro método enviando parámetros en el cuerpo del request, debemos seleccionar *Body*, luego *raw* y por último seleccionamos la opción *JSON*, ya que los parámetros los enviaremos con el formato *JSON*.

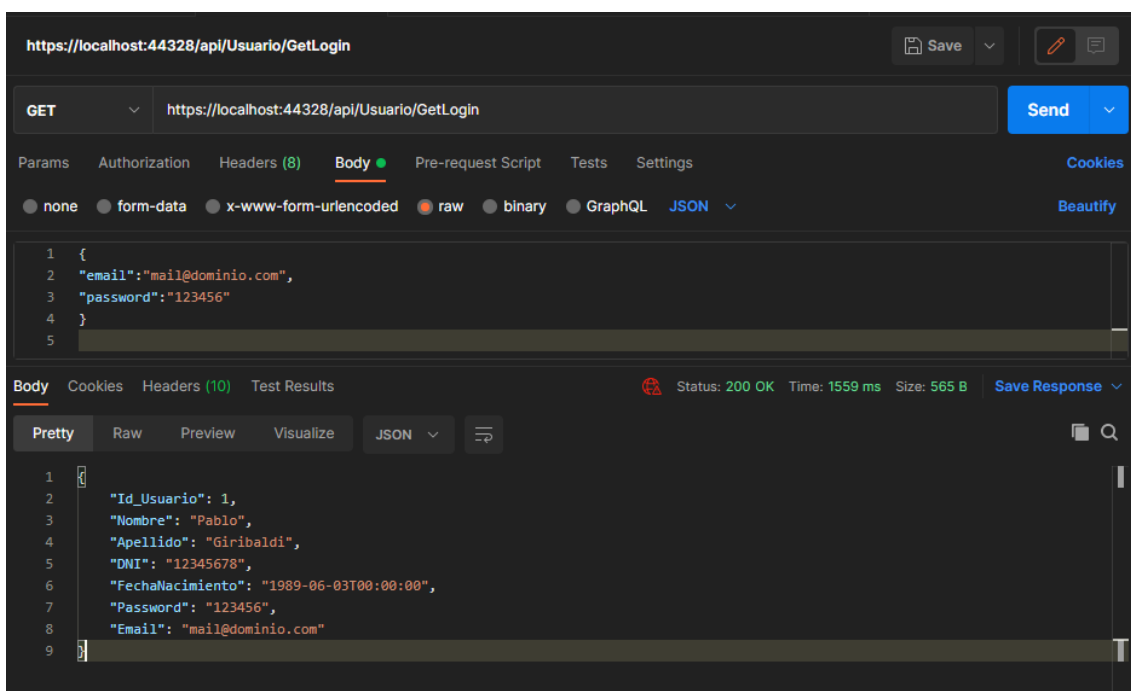


Figura 45: Prueba de método GET. Devuelve el usuario que coincida el mail y el password.

Para nuestro método CREATE agregaremos una validación (lo recomendable es agregar en todos los métodos la misma validación) del estado del modelo luego de enlazar los datos. Al igual que en el método de Login, recibimos los parámetros en el cuerpo del *request*, con dos principales diferencias, la primera es que ahora recibiremos un objeto de tipo *Usuarios* (la entidad que se agregó recientemente en nuestro EDM) y la segunda que el tipo de método es *POST* ya que lo que queremos hacer, es enviar datos para su inserción.

```
// POST: api/Usuario
[HttpPost]
// Referencias
public void Post([FromBody] Usuarios oUsuario)
{
    if (!ModelState.IsValid)
    {
        return;
    }

    db.Usuarios.Add(oUsuario);
    db.SaveChanges();
}
```

Figura 46: Método de inserción.

Para probar nuestro método en Postman, debemos cambiar el método de petición a *POST* y enviaremos el objeto *Usuario* que recibirá el método por parámetros de la siguiente manera.

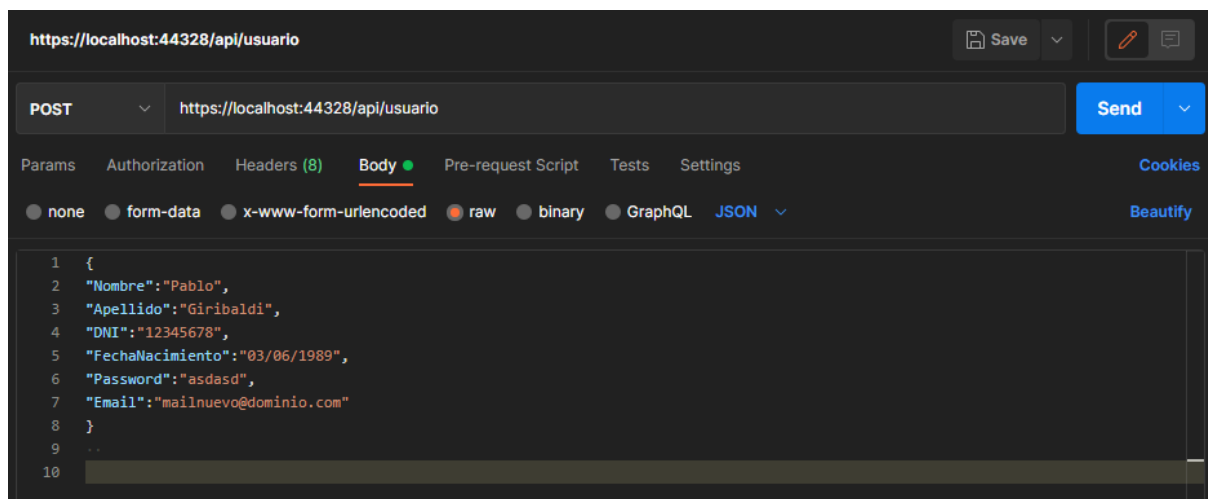


Figura 47: Prueba de método POST. Inserta un nuevo registro.

Para la actualización, debemos tener en cuenta dos puntos fundamentales, que registro deseamos modificar y los datos que se podrán modificar. Por eso enviamos dos parámetros, un ID que será el que nos indicará cual modificaremos y el objeto con los datos de la entidad *Usuario*. Con el ID buscamos el registro y lo asignamos en la variable *oUsuarioAModificar*, y con los datos enviados “pisamos” los datos viejos.

```
// PUT: api/Usuario/5
[HttpPut]
0 referencias
public void Put(int id, [FromBody]Usuarios oUsuarioNuevo)
{
    Usuarios oUsuarioAModificar = db.Usuarios.Where(a => a.Id_Usuario == id).FirstOrDefault();
    oUsuarioAModificar.Nombre = oUsuarioNuevo.Nombre;
    oUsuarioAModificar.Apellido = oUsuarioNuevo.Apellido;
    oUsuarioAModificar.DNI = oUsuarioNuevo.DNI;
    oUsuarioAModificar.Email = oUsuarioNuevo.Email;
    oUsuarioAModificar.Password = oUsuarioNuevo.Password;
    oUsuarioAModificar.FechaNacimiento = oUsuarioNuevo.FechaNacimiento;

    db.Usuarios.Attach(db.Usuarios.Single(a => a.Id_Usuario == id));
    db.Entry(oUsuarioAModificar).State = System.Data.Entity.EntityState.Modified;
    db.SaveChanges();
}
```

Figura 48: Método para modificar un registro.

Para probar nuestro método, debemos asignar el método de petición en PUT y enviaremos el ID por URL, mientras mandamos el objeto *Usuario* con formato *JSON*.

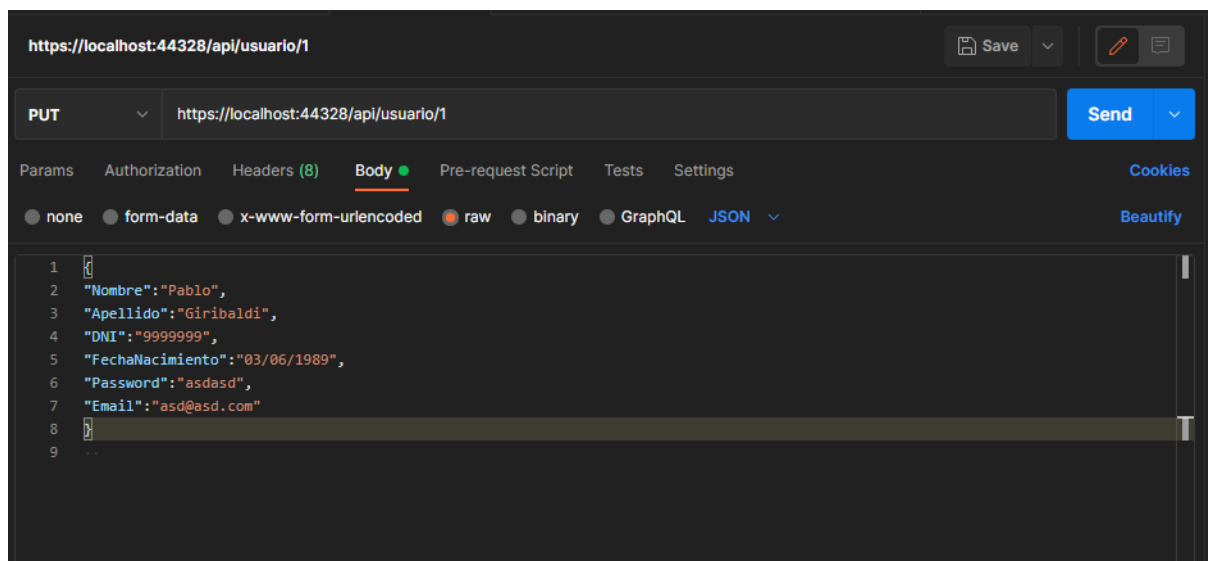


Figura 49: Prueba de método PUT. Modifica un registro particular.

Para eliminar el registro, recibiremos por parámetro el ID del usuario, el cual utilizaremos para obtener el registro que vamos a eliminar.

```
// DELETE: api/Usuario/5
0 referencias
public void Delete(int id)
{
    if (!ModelState.IsValid)
    {
        return;
    }
    Usuarios oUsuario = db.Usuarios.Where(a => a.Id_Usuario == id).FirstOrDefault();
    db.Usuarios.Remove(oUsuario);
    db.SaveChanges();
}
```

Figura 50: Método para eliminar un registro.

Para probar nuestro método, debemos setear nuestro método de petición en DELETE y enviaremos el ID por URL como se muestra a continuación.

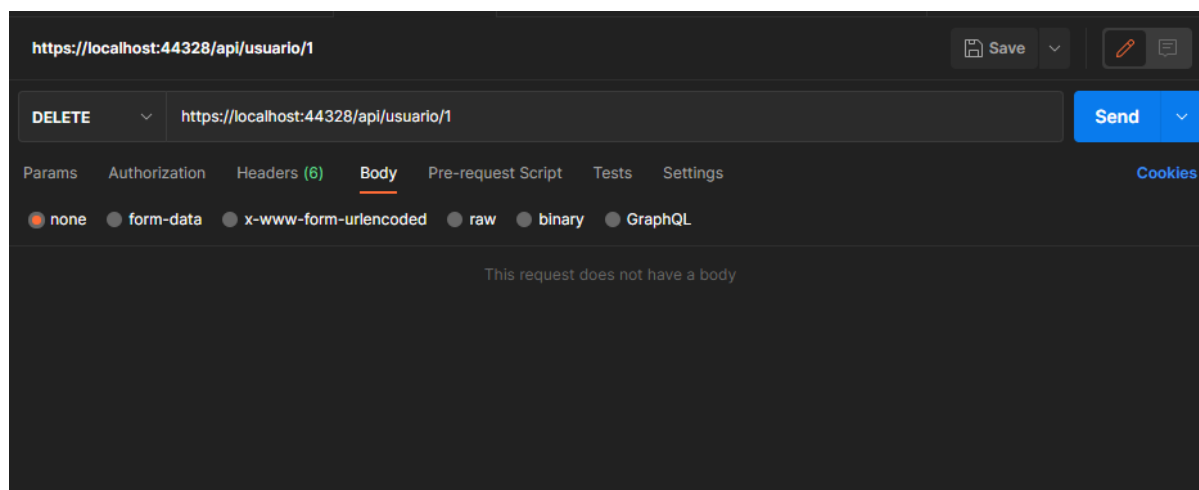


Figura 51: Prueba de método DELETE. Elimina un registro específico.

Referencias

Postman: <https://www.postman.com/postman/>

Stack overflow: <https://es.stackoverflow.com/>

Microsoft docs: <https://docs.microsoft.com/es-es/>

W3Schools: <https://www.w3schools.com/>

MDN Web Docs - Mozilla: <https://developer.mozilla.org/>