¿Qué veremos en este módulo?

En el módulo anterior hemos creado nuestra primera aplicación ASP.NET MVC utilizando la plantilla por defecto, y realizando pequeños cambios sobre ella. Esto nos ha permitido obtener una visión panorámica del funcionamiento de una solución basada en el framework, y conocer los componentes implicados en el ciclo de vida de las peticiones:

- *Las rutas*, que indican el componente que procesará cada petición en función de la dirección del recurso solicitado.
- *Los controladores y las acciones*, donde se implementan el proceso de las mismas.
- *Las vistas*, que definen el interfaz de usuario enviado al cliente como respuesta a una petición.

Ahora vamos a ampliar las funcionalidades de esta aplicación, creando la estructura desde cero y paso a paso: el modelo, reglas en la tabla de rutas, controladores y las vistas que necesitemos.

Lo haremos, además, utilizando dos procedimientos distintos; en el material de lectura recorreremos las rutas, el controlador y vistas, deteniéndonos en cada uno de ellos para implementar los componentes correspondientes a las tres funcionalidades al mismo tiempo. En el audiovisual, en cambio, modificaremos la perspectiva, e iremos creando una por una las funcionalidades completas, desde la ruta hasta la vista, obteniendo así un resultado más incremental.

¿Pasamos a la acción?

Declaración de intenciones

A lo largo de este módulo desarrollaremos los componentes necesarios para añadir nuevas funcionalidades a nuestra aplicación. Sin embargo, es lógico que previamente definamos con exactitud el resultado funcional que pretendemos obtener.

Vamos a ampliar la aplicación añadiéndole un motor simple de visualización de artículos, algo que podría ser la base para la creación del front-end de un sistema de publicación de blogs. Las operaciones que permitiremos son las siguientes:

- Visualizar las 10 últimas entradas publicadas
- Visualizar la relación de artículos publicados un mes y año concreto
- Visualizar un artículo determinado

Asimismo, con objeto de mejorar el posicionamiento en buscadores, vamos a aprovechar el sistema de routing, ofreciendo acceso a las funcionalidades descritas con URLs muy amigables, tal y como se refleja en la siguiente tabla:

Patrón de URL	Ejemplos de petición	Acción a realizar
{controller}/{action}/{id} (Ruta por defecto)	/blog	Muestra el listado con un resumen de las 10 últimas entradas publicadas, ordenadas cronológicamente.
blog/archive/{year}/{month}	/blog/archive/2005/12 /blog/archive/2006/8	Muestra el resumen de las entradas publicadas en el mes y año indicado en los parámetros de ruta.
blog/{code}	/blog/welcome-to-aspnet-mvc	Muestra el artículo cuyo código coincida con el especificado en la ruta.

Cada artículo, o post, constará de las propiedades citadas a continuación:

- El título del post, por ejemplo "Bienvenidos a ASP.NET MVC".
- Código único del post, una representación del título apta para ser utilizada como parte de la URL, por ejemplo "Bienvenidos-a-ASPNET-MVC".
- El contenido textual del artículo.

- La fecha de publicación.
- El autor.

Comenzaremos definiendo los componentes del Modelo de nuestro sistema.

Creación del Modelo

En este capítulo crearemos el Modelo de nuestra aplicación. Recordemos que sus componentes representan los datos del dominio, son responsables de aportar la lógica de negocio y deben aportar los mecanismos de persistencia del sistema.

Por simplificar el código y nos desviarnos mucho de nuestros objetivos, almacenaremos los posts de nuestro blog en una base de datos LocalDb, y utilizaremos *Entity Framework* para el mapeo E/R y la gestión de los datos. Adicionalmente, crearemos un componente de gestión de más alto nivel que incluya la lógica de negocio y aísle al resto de componentes del sistema de persistencia elegido.

> Podemos implementar el Modelo de las aplicaciones orientadas a datos usando cualquier tecnología: Entity Framework, NHibernate, micro-ORMs, ADO.NET clásico, etc. ASP.NET MVC no establece ningún requisito ni directriz al respecto.

Dado que el objetivo de este curso no es profundizar en Entity Framework, lo utilizaremos de la forma más simple y directa posible: **el enfoque denominado "code-first"**. Como su nombre sugiere, este enfoque permite que nos centremos en la definición de las entidades de datos desde el código fuente, obviando todos los detalles relativos a la persistencia. Es decir, definiremos las entidades de datos que vamos a necesitar, y, con muy poca información adicional, **Entity Framework se encargará de crear la base de datos por nosotros** y gestionar automáticamente la persistencia.

Así, los artilugios que necesitamos añadir a nuestro proyecto relativos a Entity Framework son:

- Las entidades de datos que necesitamos. En nuestro caso, sólo necesitaremos de momento la entidad "Post".
- El contexto de datos de Entity Framework, que es la clase que nos abstrae del almacén de persistencia subyacente y nos ofrece mecanismos para obtener y almacenar entidades en él de forma muy sencilla y transparente.
- Para facilitarnos la vida durante el desarrollo, crearemos también una clase de inicialización de datos. Veremos más adelante lo que es.

Creación de las entidades

Ya hemos comentado que utilizando el enfoque "code-first" de Entity Framework debemos centrarnos en el código de las entidades de datos que modelan el dominio de nuestra aplicación. Normalmente se trata de clases POCO (*Plain Old CLR Objects*, u objetos planos), como la siguiente, que cumple los requisitos expresados en el capítulo anterior:

```
public class Post
{
   public int PostId { get; set; }
   public string Title { get; set; }
   public string Code { get; set; }
   public string Text { get; set; }
   public DateTime Date { get; set; }
   public string Author { get; set; }
}
```

Atendiendo a las convenciones de ubicación de archivos propuesta por ASP.NET MVC, un buen sitio para esta clase sería la carpeta /Models del proyecto.

El contexto de datos de Entify framework

Muy básicamente, el contexto de datos es la clase que nos permitirá acceder fácilmente al almacén de persistencia para leer o grabar entidades. Un contexto de datos simple para nuestro proyecto podría ser el siguiente:

```
public class BlogContext: DbContext
{
   public BlogContext(): base("DefaultConnection") { }

   public DbSet<Post> Posts { get; set; }
}
```

De esta forma tan sencilla estamos creando un contexto de datos propio (heredando de `DbContext`, una clase propia de Entity Framework) que contiene un conjunto de objetos de tipo `Post` (`DbSet<Post>`) al que accederemos utilizando la propiedad llamada `Posts` del contexto. Los `DbSet` actúan de forma parecida a otras colecciones en memoria de .NET: podemos consultarlas mediante sentencias LINQ y manipular su contenido

usando métodos como `Add()` o `Remove()`. Aunque en este caso nuestro dominio es muy simple y sólo necesitamos un conjunto de datos, lo normal es que en el contexto de datos de una aplicación tengamos definidos varios `DbSet`, uno por cada tipo de entidad a la que necesitaremos acceder.

El constructor utilizado indica que la cadena de conexión a la base de datos a utilizar se encuentra definida en el web.config bajo la denominación "DefaultConnection". Esta cadena de conexión viene de serie en proyectos MVC, y apunta la instancia de LocalDb de la máquina local, pero podríamos modificarla fácilmente en el archivo de configuración, o bien indicar otro valor (o incluso directamente otra cadena de conexión) en el constructor del contexto de datos.

Inicializadores y datos de prueba

La primera vez que intentemos acceder al contenido de un `DbSet` del contexto de datos, ya sea para consultarlo o modificarlo, Entify Framework comprobará si existe la base de datos que actuará como almacén para estos datos, y si es necesario la creará de forma automática atendiendo a la estructura de datos que le hemos ordenado gestionar. En el ejemplo anterior, EF es lo suficientemente inteligente como para determinar que necesita crear una tabla para almacenar entidades de tipo `Post` , e incluso los campos (tipos incluidos) que debe introducir en ella para que se puedan almacenar entidades de este tipo.

Sin embargo, si la base de datos existía, Entity Framework comprueba si su estructura coincide con la que sería necesaria atendiendo a la estructura del contexto de datos y entidades, fallando estrepitosamente en tiempo de ejecución si esto no ocurre.

Para evitar estos problemas y dar un poco de flexibilidad a este proceso, en Entity Framework existen los **inicializadores**, que son clases mediante las cuales podemos indicar qué queremos que haga Entity Framework durante su inicialización, y podemos aprovechar para añadir datos de prueba, algo que resulta bastante útil durante el desarrollo. El siguiente código muestra un inicializador personalizado que indica que la base de datos debe ser eliminada y creada de nuevo cuando cambie la estructura de las entidades o el contexto, y aporta datos de inicialización para que la base de datos aparezca ya con alguna información:

```
public class BlogInitializer :
DropCreateDatabaseIfModelChanges<BlogContext>
{
   protected override void Seed(BlogContext context)
   {
```

```
        context.Posts.Add(new Post() {
            Author = "José M. Aguilar",
            Title = "Hello, world!",
            Code = "hello-world",
            Date = new DateTime(2005, 8, 12),
            Text = "Hi, everybody, this is my first blog post!"
        });
        context.Posts.Add(new Post() {
            Author = "José M. Aguilar",
            Title = "Second post",
            Code = "second-post",
            Date = new DateTime(2005, 8, 22),
            Text = "Well, it's time to start writing... :)"
        });
        context.Posts.Add(new Post() {
            Author = "José M. Aguilar",
            Title = "Today is my birthday",
            Code = "today-is-my-birthday",
            Date = new DateTime(2005, 9, 1),
            Text = "Today is my birthday! I accept gifts ;)"
        });

        // Other seed data

        base.Seed(context);
    }
}
```

La clase `DropCreateDatabaseIfModelChanges` es uno de los inicializadores que acompañan de serie a Entity Framework, y cuya intencionalidad queda totalmente clara simplemente viendo su nombre. Existen otros, como `DropCreateDatabaseAlways` o `CreateDatabaseIfNotExists` (el usado por defecto por EF) que también podríamos haber empleado, heredando de ellos, si estas estrategias fueran más apropiadas para nuestras necesidades.

En cualquier caso, para que un inicializador se tenga en cuenta, **debemos añadir el siguiente código para que se ejecute durante el arranque del sistema**. Un buen sitio podría ser en el evento `Application_Start` en el archivo Global.asax.cs

```
public class MvcApplication : System.Web.HttpApplication
{
   protected void Application_Start()
   {
      Database.SetInitializer(new BlogInitializer()); // Set
default initializer
      AreaRegistration.RegisterAllAreas();
      FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
      RouteConfig.RegisterRoutes(RouteTable.Routes);
      BundleConfig.RegisterBundles(BundleTable.Bundles);
   }
}
```

Componente de gestión

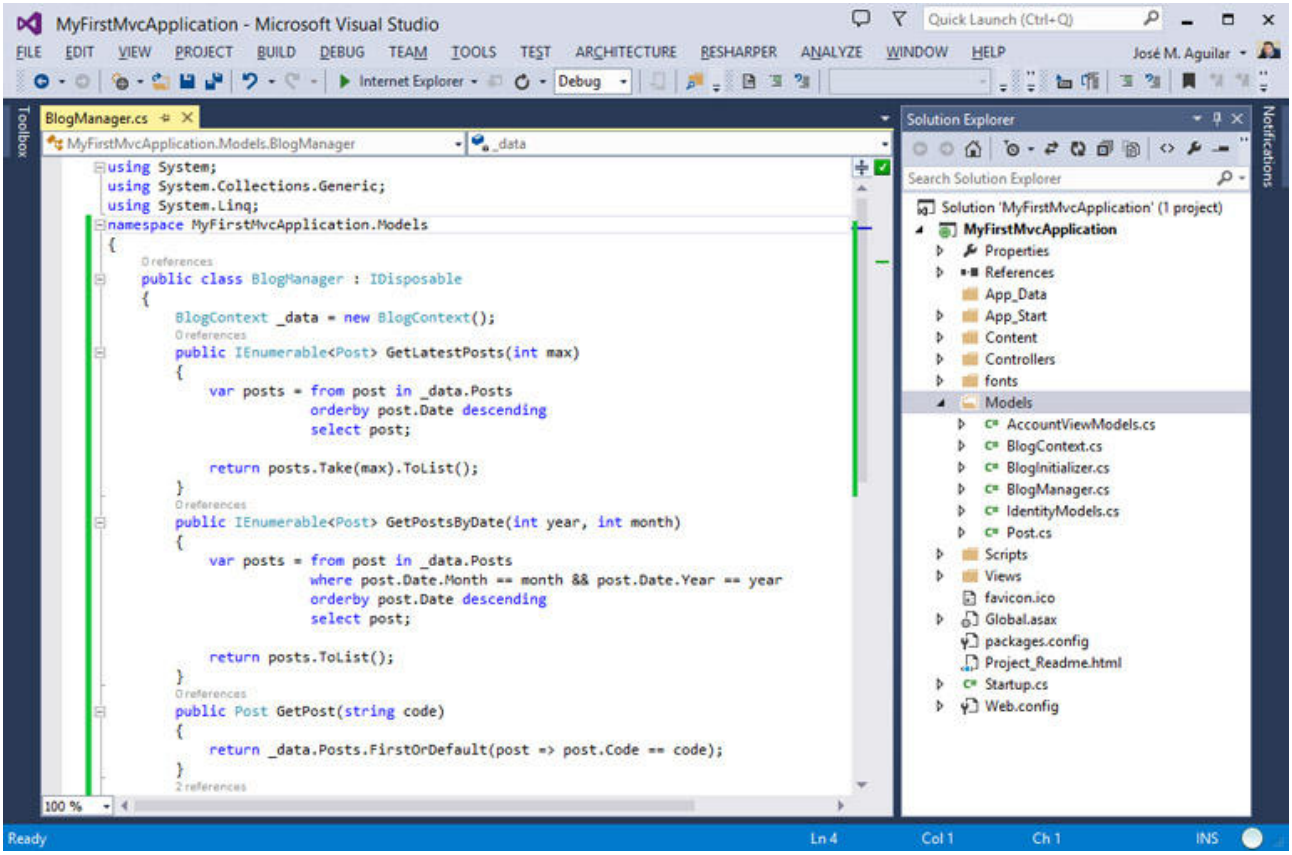
Ya tenemos resuelta la infraestructura de acceso a datos. Ahora crearemos el componente del Modelo que aportará las operaciones de alto nivel que podremos utilizar desde el resto del sistema, y que vienen definidas por los requisitos que hemos especificado anteriormente. Para ello, crearemos, de nuevo dentro de la carpeta /Models, la clase `BlogManager`.

Como se puede observar, instanciamos a nivel de clase el modelo de datos de Entity Framework, e implementamos los tres únicos métodos necesarios; el primero de ellos para obtener los últimos posts, el segundo para obtenerlos según el mes y año de publicación, y el tercero para obtener un post dado su código. La implementación del interfaz `IDisposable` permitirá la liberación de los recursos usados por el contexto (por ejemplo, internamente utiliza una conexión física a la base de datos).

```
public class BlogManager: IDisposable
{
   BlogContext _data = new BlogContext();

   public IEnumerable<Post> GetLatestPosts(int max)
   {
      var posts = from post in _data.Posts
                  orderby post.Date descending
                  select post;
      return posts.Take(max).ToList();
   }
```

```
   public IEnumerable<Post> GetPostsByDate(int year, int month)
   {
      var posts = from post in _data.Posts
                  where post.Date.Month == month &&
post.Date.Year == year
                  orderby post.Date descending
                  select post;
      return posts.ToList();
   }

   public Post GetPost(string code)
   {
      return _data.Posts.FirstOrDefault(post => post.Code ==
code);
   }

   public void Dispose()
   {
      _data.Dispose();
   }
}
```

La siguiente captura de pantalla muestra el resultado de las tareas realizadas hasta el momento en el Modelo. Como se puede observar, se han creado en la carpeta /Models los archivos BlogContext.cs, BlogInitializer.cs, BlogManager.cs, y Post.cs:



Con esto tendríamos completamente definidos los componentes de nuestro Modelo. Si te fijas, hasta este momento no hemos introducido ninguna particularidad del framework MVC; de hecho, estos mismos componentes podrían ser útiles en una aplicación ASP.NET WebForms, o incluso de escritorio, sin ninguna modificación.

> **Los componentes del Modelo deben ser ajenos a la tecnología sobre la que está construido el sistema al que sirven.**

A continuación pasaremos a definir las rutas, estableciendo las URLs que entenderá nuestra aplicación y definiendo en qué controladores y acciones se implementará su procesamiento.

Creación de rutas

Anteriormente hemos visto que el sistema de routing actúa como *front-controller*, un controlador frontal al que llegan las peticiones para, en función de la información contenida en la tabla de rutas, delegar el proceso de las mismas a controladores más específicos de nuestra aplicación.

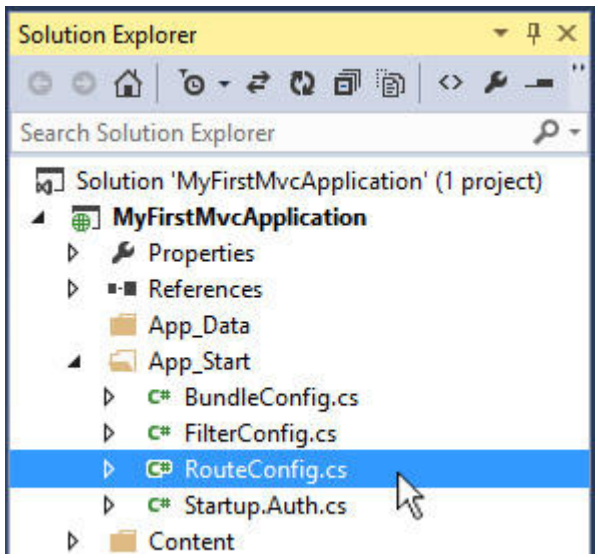
Recordemos que las rutas que habíamos planificado para acceder a las funcionalidades del sistema eran las siguientes:

Patrón de URL	Ejemplos de petición	Acción a realizar
{controller}/{action}/{id}	/blog	Muestra el listado con un resumen de las últimas entradas publicadas, ordenadas cronológicamente.
blog/archive/{year}/{month}	/blog/archive/2005/12 /blog/archive/2006/8	Muestra el resumen de las entradas publicadas en el mes y año indicado en los parámetros de ruta.
blog/{code}	/blog/welcome-to-aspnet-mvc	Muestra el artículo cuyo código coincida con el especificado en la ruta.

Veamos cada una de ellas con más detenimiento.

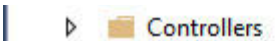
{controller}/{action}/{id}

El patrón de URLs que encontramos en primer lugar coincide con la ruta por defecto, aquella que se incluye en la plantilla de proyectos ASP.NET MVC, por lo que no tendremos que registrarla de nuevo.



Anteriormente hemos comentado que la tabla de rutas se llena durante la inicialización de la aplicación, por lo que, según la convención de ubicación de archivos, encontraremos el código que realiza el registro de rutas en la

carpeta `/App_Start`, y más concretamente en la clase `RouteConfig`. Su método estático `RegisterRoutes()` es invocado desde el evento `Application_Start` (en el archivo *`global.asax.cs`*), y es el encargado de registrar todas las rutas que utilizará nuestra aplicación.



En concreto, la ruta por defecto se define en la siguiente porción de código:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
);
```

Aunque lo veremos detalladamente más adelante, seguro que puedes intuir el significado de algunos de los parámetros utilizados para registrar la ruta:

- Mediante el parámetro `name` indicamos el nombre único que daremos a esta entrada de la tabla de rutas.
- El parámetro `url` indica el patrón de URL al que hace referencia esta ruta.
- Por último, `defaults` especifica el valor por defecto para los parámetros especificados en el patrón de URL anterior **en forma de objeto anónimo**. Aunque pueda parecer algo extraño, en realidad es sólo una forma muy compacta de escribir un diccionario clave-valor, que ya veremos que se utiliza bastante en ASP.NET MVC.

> Observa que en el parámetro {id}, en lugar de asignarle un valor por defecto se está utilizando la constante `UrlParameter.Optional` para indicar su no obligatoriedad.

Así, las peticiones a la URL `/Blog`, dado que encajan en el patrón indicado, serán procesadas por el controlador "Blog", que es el valor del parámetro `{controller}` de la URL. La acción a ejecutar, dado que no se especifica en la URL de la petición, se tomará el valor "Index", indicado por defecto para el parámetro `{action}`.

blog/archive/{year}/{month}

Como se puede observar, este tipo de direcciones no encajarían en el patrón de ruta anterior, por lo que será necesario crear una ruta específica para ello, insertando la siguiente porción en el código de inicialización:

```
routes.MapRoute(
    name: "archive",
    url: "blog/archive/{year}/{month}",
    defaults: new { controller = "Blog", action = "Archive" }
);
```

Fíjate que el patrón de URL no incluye los parámetros obligatorios `{controller}` y `{action}` porque no queremos introducir ningún factor variable en estas peticiones. Por esta razón, en el objeto anónimo definido en el tercer parámetro se establecen sus valores por defecto a "blog" y "archive", respectivamente, que corresponden con el nombre del controlador y la acción que procesará la petición. Observa también que en este mismo objeto no hemos asignado valores por defecto a los parámetros `{year}` y `{month}`, pues nos interesa que sean obligatorios.

> En este caso es indiferente la posición de esta regla dentro de la tabla de rutas, sin embargo, **hay que prestar mucha atención al hecho de que se procesan de forma secuencial** y el primer patrón que encaje con la URL del recurso solicitado será el que defina cómo va a ser procesada la petición.

Según la convención, las peticiones que se correspondan con este patrón serán procesadas por el método de acción `Archive()` que se encontrará en la clase `BlogController`.

blog/{codigo}

En este momento, si nuestro sistema recibiera una petición hacia la URL `/blog/hello-world`, ésta intentaría ser procesada por la ruta por defecto al encajar en el patrón {controller}/{action}/{id}. Sin embargo, generaría un error al interpretar que el proceso será delegado al controlador "blog", y a la acción "hello-world", lo cual es obviamente incorrecto.

Por tanto, en primer lugar, llegamos a la conclusión de que necesitamos crear una regla en la tabla de rutas para procesar las peticiones de este tipo. Además, en este caso **será necesario registrar la ruta en primer lugar**, para que se compruebe antes que las demás:

```
routes.MapRoute(
    name: "post",
    url: "blog/{code}",
```

```
    defaults: new { controller = "Blog", action = "ViewPost" }
);
```

Según la convención, las peticiones cuya URL se corresponda con este patrón, serán procesadas por el método de acción `ViewPost()` que se encontrará en la clase `BlogController`.

Resumen del registro de rutas

El código del archivo `App_Start/RouteConfig.cs`, incluyendo las definiciones que ya venían definidas por defecto, es el siguiente:

```
public class RouteConfig
{
   public static void RegisterRoutes(RouteCollection routes)
   {
      routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

      routes.MapRoute(
         name: "post",
         url: "blog/{code}",
         defaults: new { controller = "Blog", action =
"ViewPost" }
      );

      routes.MapRoute(
         name: "archive",
         url: "blog/archive/{year}/{month}",
         defaults: new { controller = "Blog", action = "Archive"
}
      );

      routes.MapRoute(
         name: "Default",
         url: "{controller}/{action}/{id}",
         defaults: new { controller = "Home", action = "Index",
id = UrlParameter.Optional }
      );
   }
}
```

A continuación, siguiendo la convención, implementaremos el controlador "Blog" en la clase `BlogController`, con los métodos de acción que necesitamos para llevar a cabo las tareas propuestas: `Index()`, `Archive()` y `ViewPost()`.

> Habrás observado que en el código por defecto de la clase `RouteConfig`, aparece una llamada al método `IgnoreRoute()`. Como se puede intuir, se trata de una ruta especial para ignorar peticiones dirigidas a URLs concretas. Lo estudiaremos más adelante.

Creación del Controlador

En este capítulo desarrollaremos los componentes de la capa Controlador. En este caso, dada la estructura de rutas definida anteriormente, únicamente será necesario crear la clase `BlogController`, a la que el sistema de routing delegará el proceso de las peticiones dirigidas hacia nuestro motor de blogs.

Según hemos ido viendo, en esta clase necesitaremos implementar tres métodos, uno para cada acción definida:

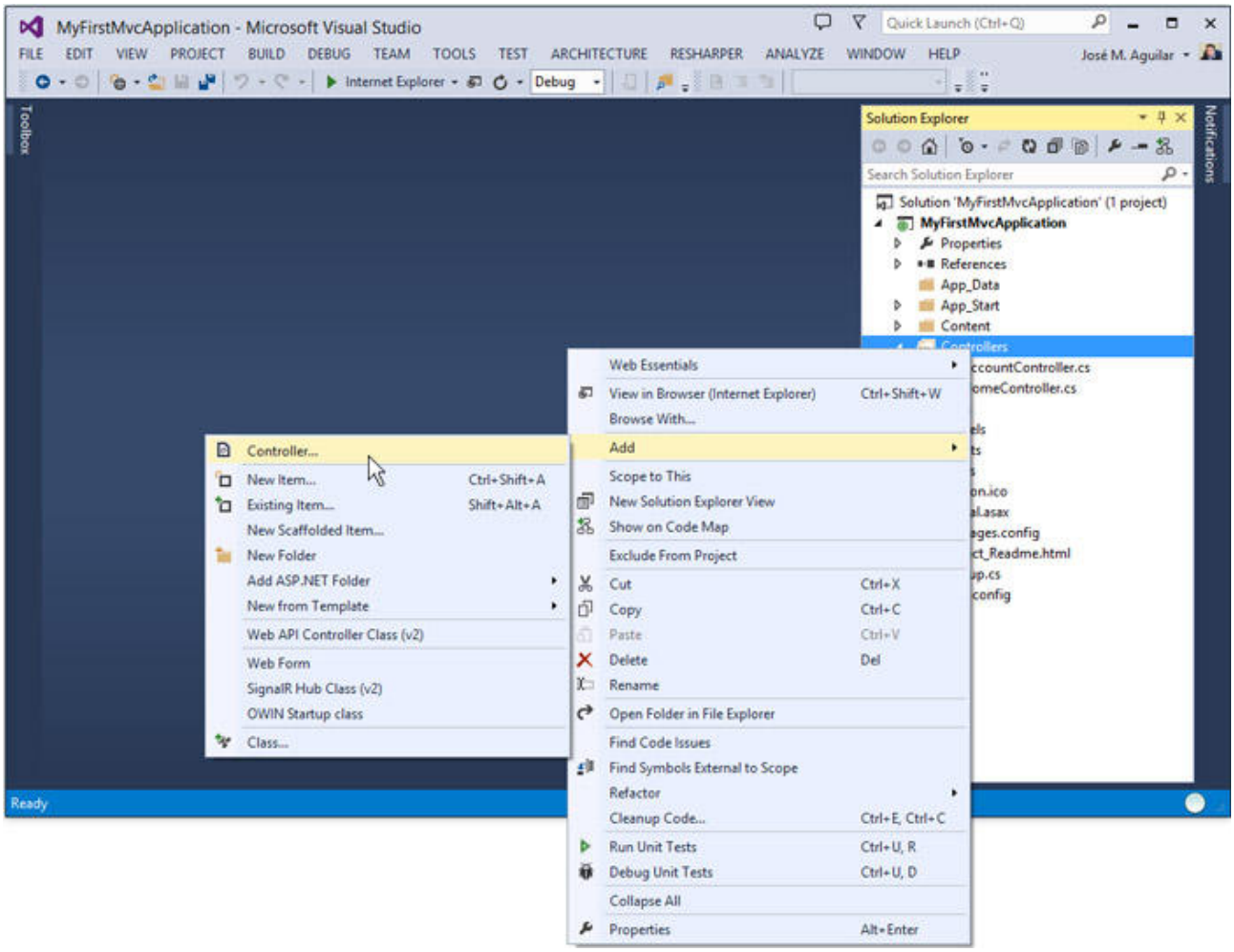
- `Index()`, que debe retornar al cliente una página mostrando una relación con los últimos artículos publicados.
- `Archive()`, para visualizar los artículos publicados en un año y mes concretos.
- `ViewPost()`, cuya misión será mostrar un artículo concreto.

Vamos a ver paso a paso el procedimiento de creación de la clase controlador de nuestro sistema, así como de los métodos de acción incluidos en ella.

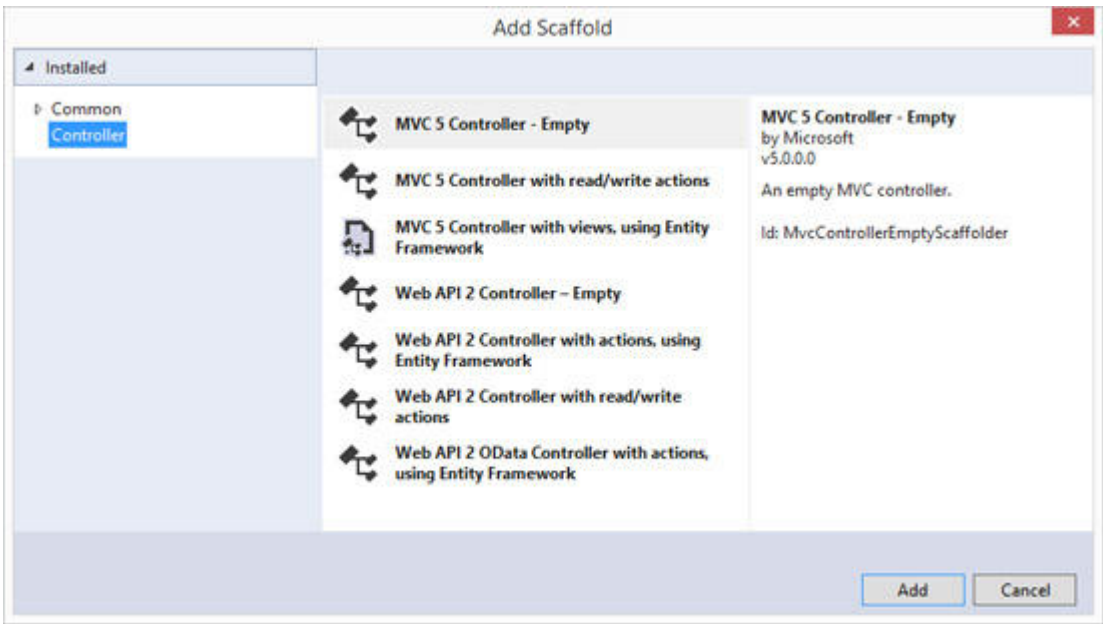
Creación de la clase controlador

Una clase controlador es sencillamente eso: una clase. Su creación, por tanto, podría realizarse siguiendo el mismo procedimiento al que estamos habituados en otro tipo de proyectos.

Sin embargo, Visual Studio incluye ayudas específicas para ASP.NET MVC que nos ayudarán en las tareas más frecuentes, como la creación de controladores. Así, podemos crear la clase pulsando el botón derecho del ratón sobre la carpeta *`/Controllers`* del proyecto y seleccionando la opción "Agregar" y seguidamente "Controlador":

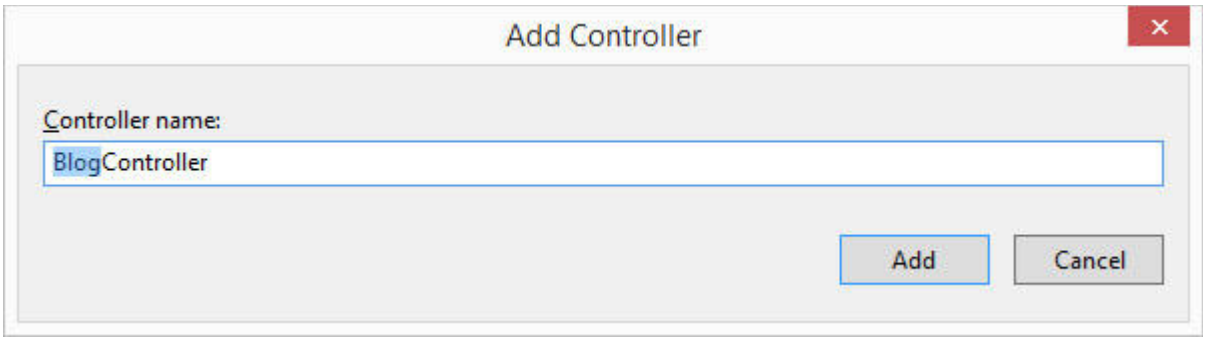


Seguidamente, Visual Studio mostrará un cuadro de diálogo consultándonos qué plantilla de controlador queremos utilizar como base. De momento, dado que queremos implementar nuestro controlador de forma manual, seleccionaremos el controlador MVC vacío:



> Usando estas opciones de scaffolding es posible generar clases controlador utilizando distintas plantillas, que podremos elegir desde este cuadro de diálogo. Por ejemplo, es posible generar controladores con la implementación completa de funcionalidades CRUD (*Create-Read-Update-Delete*, Crear-Obtener-Actualizar-Borrar) sobre una entidad del modelo de datos, o controladores Web API para ser accedidos usando OData, entre otros.

A continuación, el entorno de desarrollo solicita un nombre para la nueva clase controlador. Como podemos observar, nos sugiere la utilización de un identificador acorde con la convención de nombrado que, como recordamos, debe ser de la forma `NombreDeControlador`*`Controller`*. En nuestro caso, la clase se denominará `BlogController`:



Tras indicar el nombre y pulsar el botón "Añadir", Visual Studio habrá creado el archivo con un contenido similar al siguiente, donde podemos observar que ha sido generada una acción por defecto llamada `Index()`:

```
namespace MyFirstMvcApplication.Controllers
{
   public class BlogController : Controller
   {
      //
      // GET: /Blog/

      public ActionResult Index()
      {
         return View();
      }
   }
}
```

Acción "Index": Obtener los últimos artículos publicados

> ***Recuerda:***
> *Petición:* `GET /blog`
> *Patrón:* `{controller}/{action}/{id}`
> *Valores por defecto:*
> - `controller = "Home"`
> - `action = "Index"`
> - `id = ""`

Esta acción, que debemos implementar en el método `Index()` de la clase controlador, retornará al usuario una página con un resumen de los últimos artículos publicados en el blog. Por tanto, su primera misión será ponerse en contacto con el Modelo para obtener esta información; seguidamente, seleccionará la vista más apropiada para esta petición, le suministrará los datos que necesita para ser maquetada y la enviará como respuesta a la petición.

El código es así de sencillo, y sustituye al método `Index()` generado por defecto en el controlador por el entorno de desarrollo:

```
//
// GET: /Blog/
public ActionResult Index()
{
   using (var manager = new BlogManager())
   {
      var posts = manager.GetLatestPosts(10); // 10 latests
posts
      return View("Index", posts);
   }
}
```

¡Prácticamente tres líneas de código! La primera crea un bloque `using` para obtener la instancia del contexto de datos y liberarla automáticamente, en la segunda se obtiene la información, y en la tercera se selecciona la vista a la vez que se le envían los datos a mostrar en ella. De hecho, esta última podría haberse simplificado un poco debido a que el nombre de la vista a retornar coincide con el del método de acción:

```
// GET: /Blog/
public ActionResult Index()
{
   using (var manager = new BlogManager())
   {
```

```
      var posts = manager.GetLatestPosts(10); // 10 latests
posts
      return View(posts);                     // By default,
"Index"
   }
}
```

Acabamos de presentar una nueva convención: si el nombre de la vista que deseamos retornar al usuario coincide con el nombre de la acción en la que nos encontramos, no será necesario indicarlo de forma explícita.

Recuerda que en módulos anteriores hemos visto que era posible enviar información desde el controlador a la vista utilizando `ViewBag`, que básicamente es un diccionario clave-valor basado en tipos dinámicos de .NET, en el que podemos almacenar y recuperar cualquier tipo de datos de forma muy sencilla. En este caso estamos utilizando otro mecanismo bastante más aconsejable y potente, que describiremos más adelante en el curso.

Desde el punto de vista del método de acción, simplemente estamos retornando la vista indicada (explícita o implicitamente) a la que le enviamos como parámetro los datos que necesita para componerse, como la relación de artículos del blog. En el siguiente capítulo veremos cómo podemos acceder a estos datos desde la vista a la hora de maquetar la salida.

Un aspecto importante a tener en cuenta es que podemos nombrar los controladores, métodos y parámetros de éstos siguiendo las convenciones estándar de .NET framework en cuanto al uso de mayúsculas y minúsculas, independientemente de cómo hayan sido especificados en la tabla de rutas. Las búsquedas se realizarán mediante una comparación *case insensitive*, es decir, sin tener en cuenta este aspecto.

Acción "Archive": Obtener artículos publicados un año y mes determinado

El objetivo de esta acción es enviar al usuario una vista con el resumen de artículos publicados en el año y mes indicados en la URL empleada. Su implementación la

Recuerda:
Petición: `GET /blog/archive/2005/1`
Patrón: `blog/archive/{year}/{month}`
Valores por defecto:

encontraremos en el método `Archive()` del controlador:

```
- controller = "Blog"
- action = "Archive"
```

En capítulos anteriores, cuando estuvimos estudiando el funcionamiento de las aplicaciones ASP.NET MVC, ya comentamos que existe un componente muy potente capaz de analizar la signatura de los métodos de acción y buscar en el contexto de la petición valores apropiados para cada uno de los parámetros definidos basándose en su nombre.

Es interesante, además, saber que el *Model Binder*, que es como se llama este componente, no sólo se encarga de localizar un valor para los parámetros formales, sino también de realizar las conversiones de tipo que sean necesarias, lo cual nos ahorrará gran cantidad de trabajo cuando trabajemos en métodos más complejos.

> En realidad, esta tarea está repartida entre varios componentes de forma interna. Los *ValueProviders* son los encargados de obtener valores desde el contexto de la petición y los *ModelBinders* son los encargados de inyectarlos en los parámetros de entrada de las acciones, aunque a este conjunto de componentes se le suele denominar de forma genérica *model binders* o simplemente *binders*.

En nuestro caso, si queremos aprovechar esta capacidad, podremos definir el método de acción con los parámetros que se indican en la ruta, y el framework nos enviará sus valores de forma automática:

```
//
// GET /Blog/Archive/2005/3
public ActionResult Archive(int year, int month)
{
   using (var manager = new BlogManager())
   {
      var posts = manager.GetPostsByDate(year, month);
      return View(posts);
   }
}
```

¡De nuevo hemos resuelto la acción en tres líneas de código! En primer lugar obtenemos los artículos que cumplan los criterios desde el modelo, y retornamos la vista, a la que suministramos dicha información. Como en el caso anterior, no es necesario especificar un nombre de vista, puesto que por defecto se asume idéntico al de la acción actual.

Veremos más adelante que el *sistema de binding* es capaz incluso de pasarnos parámetros de tipos complejos, instanciando y poblando sus propiedades de forma automática, es decir, que podremos crear métodos de acción como el siguiente:

```
public ActionResult Save(Product product)
{
  ...
}
```

Acción "ViewPost": Visualizar un artículo

> ***Recuerda:***
> *Petición:* `GET /blog/hello-world`
> *Patrón:* `/blog/{code}`
> *Valores por defecto:*
> - `controller = "Blog"`
> - `action = "ViewPost"`

La última acción que implementaremos como ejemplo es igual de sencilla que las anteriores. Su misión es retornar al usuario una página con el contenido del artículo cuyo código se incluye en la ruta, y como veremos, será implementada en el método `ViewPost()` del controlador `BlogController`.

De nuevo sacaremos provecho del *binding* para obtener los parámetros que necesitamos para ejecutar la acción:

```
//
// GET /Blog/Hello-world
public ActionResult ViewPost(string code)
{
   using (var manager = new BlogManager())
   {
      var post = manager.GetPost(code);
      if (post == null)
         return HttpNotFound();
      else
         return View(post);
    }
}
```

Como se puede observar, se ha incluido algo de lógica de control para impedir la generación de una vista de un artículo inexistente. Por ello, hemos tenido que utilizar... ¡seis líneas! A pesar de duplicar en tamaño las acciones anteriores, se mantiene intacta la

legibilidad del código y su intención queda bastante clara, lo cual debería ser un objetivo cada vez que vayamos a escribir un controlador o sus acciones.

> **Se considera buena práctica el hecho de mantener los controladores ligeros**, es decir, utilizar en ellos el menor código posible.

En el código se puede observar también la forma en que hemos decidido gestionar las solicitudes de visualización relativas a un artículo existente. Por simplificar, hemos elegido retornar un error HTTP 404 (no encontrado), pero igualmente podríamos haberle retornado una vista descriptiva del problema, por ejemplo mediante una instrucción `return View("PostNotfound")`.

A continuación, para finalizar la implementación de nuestro mini-motor de blogs, crearemos las vistas para cada una de las acciones anteriormente definidas.

Creación de Vistas

En este capítulo desarrollaremos la Vista de nuestro sistema, es decir, los componentes encargados de generar el interfaz de usuario. Según hemos ido viendo a lo largo de este módulo, serán necesarias tres vistas para nuestro sistema:

- `ViewPost`, que muestra el contenido completo de un artículo.
- `Index`, destinada a mostrar una lista con los últimos artículos publicados.
- `Archive`, que permitirá el acceso al archivo, mostrando los posts creados en un año y mes indicado por el usuario.

Vamos a ver cómo se crean estos componentes; en el primer caso, que es la vista más sencilla, lo haremos muy detenidamente, aprovechando cada ocasión para comentar detalles interesantes sobre el proceso, y los siguientes lo haremos ya de forma más rápida, aunque igualmente detallada.

Creación de la Vista "ViewPost" (visualización de un artículo)

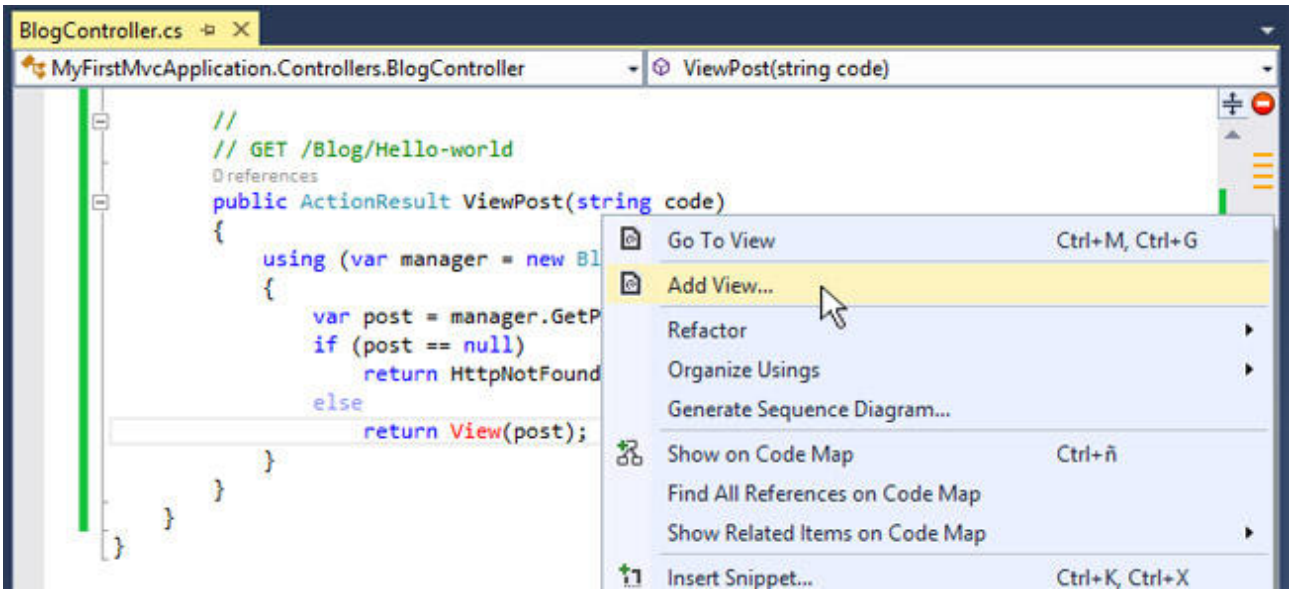
> ***Recuerda:***
> *Vista:* *ViewPost*
> *Petición:* `GET /blog/hello-world`
> *Controlador:* *Blog (* `BlogController` *)*
> *Acción:* `ViewPost(string code)`
> *Datos enviados a la vista:*
> *- Artículo (*`Post`*)*

Como recordarás, en la implementación de la acción "ViewPost" del controlador lo único que hacíamos era obtener un post desde el Modelo y, si no había ningún problema, retornar una vista, a la que suministrábamos el objeto de tipo `Post` a visualizar. Ahora nos centraremos en la creación de la vista que recibirá esta información y generará la página HTML a retornar al usuario.

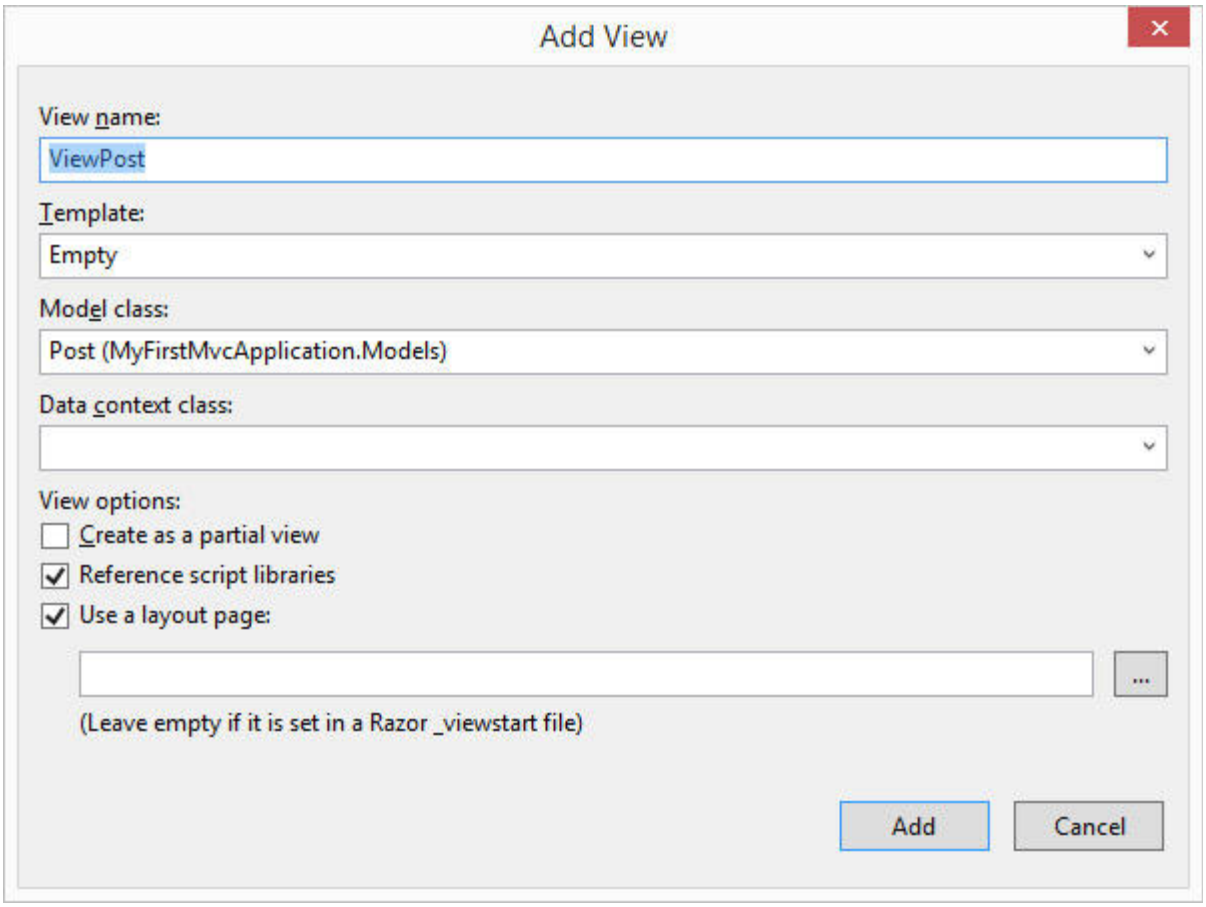
Usando el motor Razor, el más recomendable y utilizado, una vista no es más que un fichero con extensión .cshtml ubicado en el lugar donde el framework MVC espera encontrarlo según la convención de ubicación de archivos. Este archivo contiene la plantilla que será procesada en servidor y, utilizando los datos suministrados por el controlador, generará el contenido a enviar al cliente como respuesta a su petición.

En caso de utilizar el motor de vistas WebForms, el concepto es idéntico, aunque la extensión del archivo será .aspx.

La creación de una vista podría realizarse utilizando las opciones del entorno que permiten agregar elementos al proyecto, pero Visual Studio nos ofrece herramientas para facilitarnos aún más esta tarea y ayudarnos a ser más productivos, aprovechando las convenciones. Cuando se trata de una vista asociada a una acción concreta, la forma más rápida de crearla es pulsando el botón derecho del ratón sobre cualquier punto del cuerpo del método de acción y seleccionar la opción "Agregar vista" del menú contextual. Por ejemplo, en nuestro escenario, bastaría con desplegar el menú sobre la acción "ViewPost" del controlador y hacer clic sobre esta opción:



A continuación nos aparecerá un cuadro de diálogo solicitando algunos datos sobre la vista a crear:



De arriba a abajo, la información solicitada es:

- **El nombre de la vista**. Como podemos comprobar, se ha seguido la convención, sugiriéndonos para ella la denominación del método de acción sobre el que estamos trabajando ("ViewPost"). El identificador que utilicemos aquí será usado como nombre del archivo a crear. Así, usando Razor, en este caso se creará el archivo ViewPost.cshtml.

- **El desplegable "Template"** (plantilla para andamiaje) indica qué tipo de contenido deseamos generar automáticamente para la vista. En función de nuestra elección, Visual Studio generará el código de la vista que podremos más tarde adaptar a nuestras necesidades. Las posibilidades que tenemos, que como se puede observar están muy orientadas a la creación rápida de funcionalidades de tipo CRUD, son:
  - *Empty (without model)*, que indica que el IDE no debe generar código alguno en el interior de la vista a crear y seremos nosotros los responsables de hacerlo.
  - *Empty*, es idéntica a la anterior, aunque en este caso indicaremos que en la vista recibirá desde el controlador un objeto del tipo especificado en el campo "Model class" del cuadro de diálogo. Esta es la opción recomendada, pues nos permite crear vistas usando tipado fuerte y las ventajas de éste en cuanto al uso de intellisense o comprobación de errores en tiempo de diseño.
  - *Create*, que implica que la vista es un formulario para la creación de una entidad del tipo especificado como "model class". Visual Studio generará un formulario con controles de edición para las propiedades del tipo indicado.
  - *Edit*, muy similar al anterior, salvo que se refiere a la edición de una instancia existente del tipo especificado.

    - *Details*, que hace que Visual Studio genere código de visualización de la entidad, mostrando el contenido de sus propiedades.
    - *Delete*, generando una vista de confirmación de la eliminación de la entidad.
    - *List*, indicando que se pretende mostrar una lista de entidades, lo cual implica que la información suministrada por el controlador es una enumeración de objetos del tipo indicado en el desplegable "Model class" del cuadro de diálogo. El entorno de desarrollo generará automáticamente código para visualizar en forma de tabla esta información.

- **El desplegable "Model class"** (clase del modelo) nos permite indicar el tipo de datos concreto (la clase) que contiene los datos que el controlador enviará a la vista. Inicialmente se muestran en el desplegable todas las clases definidas en nuestra aplicación susceptibles de ser utilizadas para este fin. En nuestro caso, debemos indicar que la vista recibe desde el controlador un objeto del tipo `Post` (la entidad de datos definida en el modelo).

> ***Importante: cuando añadas una clase al Modelo, a veces no aparecerá en el desplegable hasta que compiles el proyecto.***

- **El checkbox "Create as partial view"** (crear como vista parcial) permite crear vistas que representan el contenido de una porción de la página, algo parecido a los controles de usuario de ASP.NET Webforms, lo que permite reutilizar código de vistas y evitar duplicidades. Esto lo veremos detalladamente en módulos posteriores, de momento no nos interesa marcarlo.
- **El checkbox "Reference script libraries"** (referencias bibliotecas de scripts) indica que se deben añadir a la vista referencias (tags `<script>`) hacia las bibliotecas utilizadas en la misma, como jQuery y algunos plugins necesarios para realizar validaciones de formularios en cliente.

- Por último, podemos seleccionar el **layout o página maestra** que utilizará la vista. Si simplemente marcamos la casilla, indicaremos que la nueva vista utilizará el Layout establecido por defecto.

En nuestro caso, el valor de los campos del cuadro de diálogo será:

- View name: "ViewPost".
- Template: "Empty".
- Model class: "Post".
- En el resto de campos, dejamos los valores por defecto.

Pulsando el botón "Añadir" de la ventana de diálogo el IDE creará una vista en la ubicación apropiada, según la convención. En este caso, dado que se trata de una vista utilizada desde el controlador "Blog", se generará el archivo `ViewPost.cshtml` en el interior de la carpeta `/Views/Blog`.

Contenido de ViewPost.cshtml

Si acudimos a dicho directorio y abrimos la vista `ViewPost.cshtml`, veremos en ella un código como el siguiente:

```
@model MyFirstMvcApplication.Models.Post

@{
    ViewBag.Title = "ViewPost";
}

<h2>ViewPost</h2>
```

La primera de las líneas que encontramos es una directiva Razor, `@model`, que especifica el tipo de datos que estamos recibiendo desde el controlador. De hecho, es el tipo que seleccionamos previamente en el desplegable "Model class" del cuadro de diálogo.

En la práctica esto sólo significa que a nivel de la vista Razor tendremos disponible una propiedad, llamada `Model`, a través de la cual podremos acceder de forma directa, y usando tipado fuerte, a los datos suministrados por el controlador. De esta forma, en nuestro escenario podremos acceder de forma directa a propiedades como `Model.PostId`, `Model.Title` o `Model.Text`, que es justo lo que necesitamos para generar una página donde se muestre el contenido del artículo.

El bloque encerrado entre llaves que aparece precedido del símbolo arroba (@) es un bloque de código de servidor escrito con sintaxis Razor. Ese símbolo es el utilizado para alternar entre código de cliente y de servidor, lo cual, como veremos más adelante, nos facilitará la escritura de vistas de forma mucho más compacta y elegante que como lo veníamos haciendo con la sintaxis ASPX, que usa las tradicionales secuencias "<%" y "%>".

Este bloque de servidor contiene por defecto una asignación en la que se establece el título de la página en `ViewBag.Title`. Como veremos más adelante, `ViewBag` es un contenedor donde podemos almacenar información arbitraria con objeto de compartirla entre varios componentes, y en este caso se está utilizando para enviar información a la página maestra, o Layout, del sitio web. De hecho, si abres el layout (disponible en *`/Views/Shared/_Layout.cshtml`*) verás que dicho valor es utilizado para generar parte del contenido del tag `<title>` de la página.

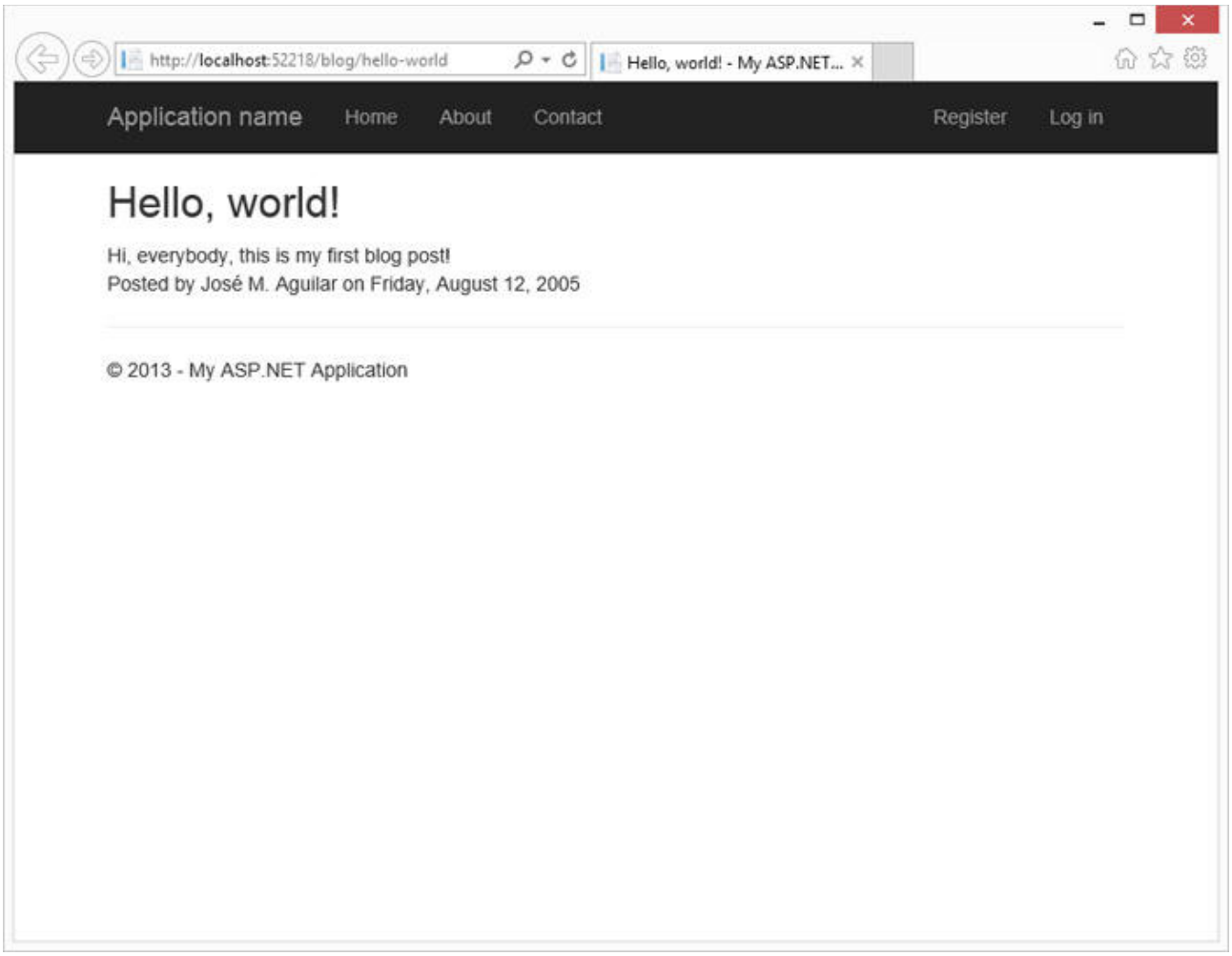
Y ya por último, en el código generado encontraríamos el marcado (HTML, Javascript, etc.) utilizado para componer la página a retornar. Uniendo todos estos aspectos, podríamos implementar la vista completa como sigue:

```
@model MyFirstMvcApplication.Models.Post
@{
   ViewBag.Title = Model.Title;
}

<h2>@Model.Title</h2>
<div class="post-body">
   @Model.Text
</div>
<p>
   Posted by @Model.Author
   on @Model.Date.ToLongDateString()
</p>
```

Observa que también utilizamos el carácter arroba (@) para enviar al cliente el contenido de propiedades o el resultado de la evaluación de expresiones. Se trata de una fórmula mucho más cómoda que la utilizada con el motor de vistas Webforms (ASPX), que usaba una sintaxis mucho más verbosa heredada de ASP clásico, `<%= expresión %>`.

> En las vistas encontramos código de servidor, escrito con sintaxis Razor o ASPX según el motor elegido, y código cliente como HTML, CSS o Javascript.

Una vez introducidos estos cambios en la vista, ya hemos finalizado la implementación de esta funcionalidad en todas las capas: Modelo, Controlador (rutas, controladores y acciones), y Vista. Así, podemos ejecutar nuestra aplicación y acceder, por ejemplo, a la URL /blog/hello-world. Si existe un post con este código en la base de datos, veremos en pantalla algo similar a lo siguiente:



Creación de la vista "Index" (listado de últimos posts)

Pasamos ahora a implementar la vista "Index" que, según habíamos establecido en los requisitos del sistema, debía mostrar un listado con los últimos posts publicados en nuestro blog.

Recuerda:

Vista: *Index*

Petición: `GET /blog`

Controlador: *Blog (* `BlogController` *)*

Acción: `Index()`

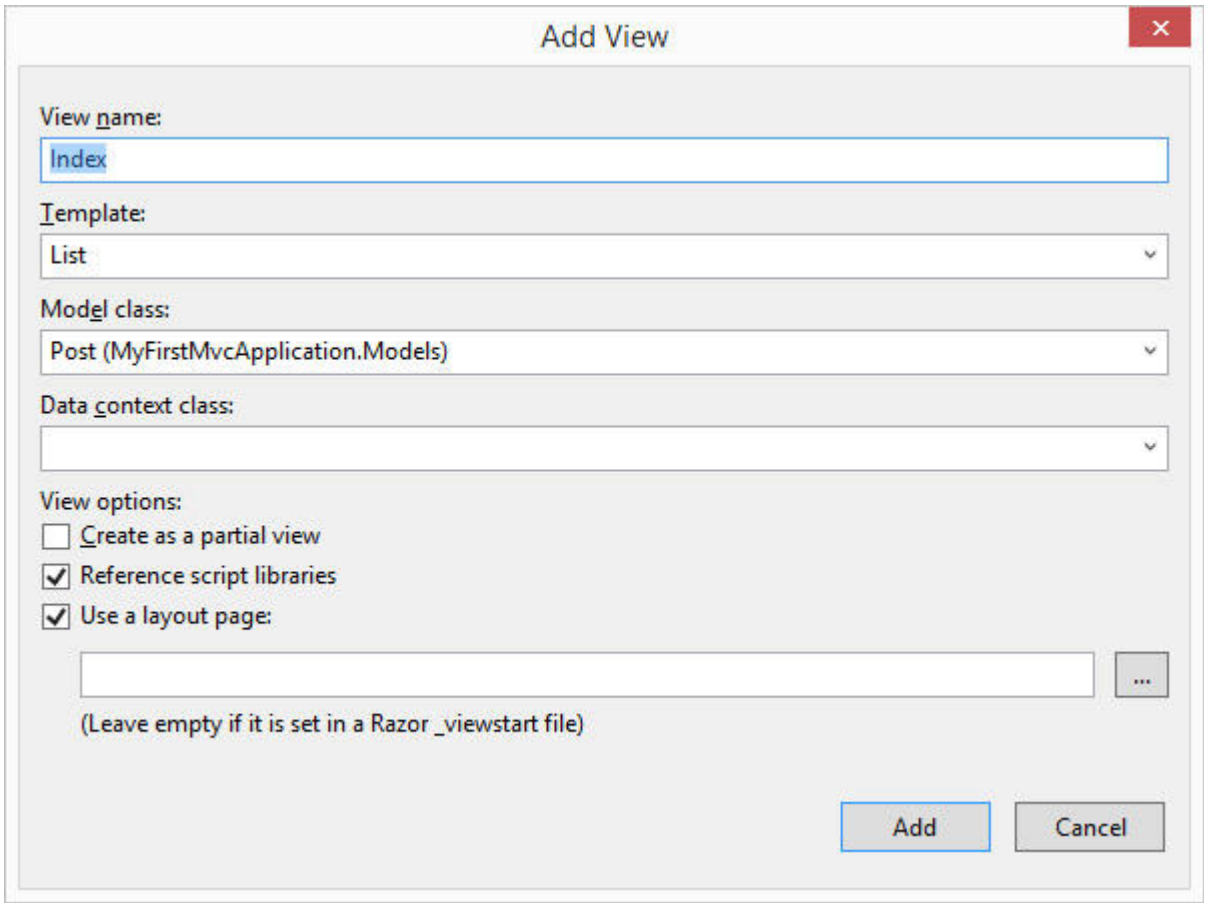
Datos enviados a la vista:

- Lista de posts (`IEnumerable<Post>`*)*

El Controlador, en su método `Index()`, invocaba al Modelo para obtener la información requerida, una colección de objetos `Post`, y tras ello retornaba la vista (denominada "Index" según la convención) adjuntándole dichos datos.

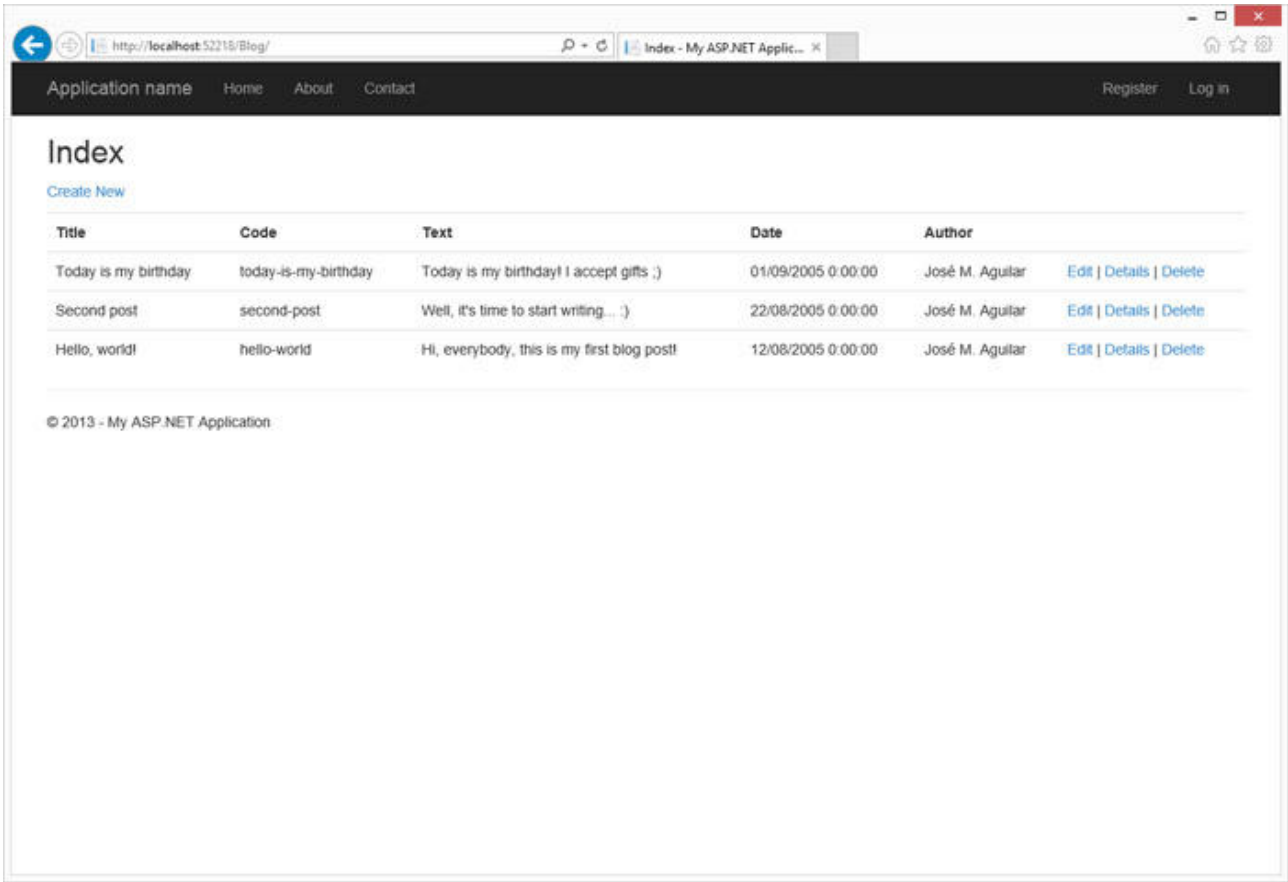
Para crear esta vista seguiremos un procedimiento similar al visto anteriormente, acudimos al cuerpo del método `Index()` del controlador `BlogController`, desplegamos el menú contextual y seleccionamos la opción "Añadir vista". En esta

ocasión, especificaremos los siguientes datos en el cuadro de diálogo de creación de vistas:



Como en la anterior ocasión, vamos a generar una vista tipada (es decir, desde el controlador le pasaremos datos de un tipo concreto), y queremos que la clase del Modelo sobre la que vamos a trabajar sea `Post`, pero esta vez vamos a utilizar la plantilla "List", puesto que queremos mostrar una colección de objetos. Pulsando el botón "Añadir" del cuadro de diálogo, tendremos la vista creada.

De hecho, si ejecutas la aplicación y accedes a la URL /blog, podrás ver un listado de los posts almacenados en la base de datos:



Visual Studio ha examinado la clase del Modelo que hemos indicado al generar la vista, y ha creado una tabla HTML mostrando cada propiedad en una columna, más una columna adicional con enlaces hacia funcionalidades (en estos momentos inexistentes) de edición, visualización y borrado de filas de datos.

> El sistema de scaffolding (generación de andamiaje) de ASP.NET MVC está muy orientado a generar código para CRUD (altas, bajas, modificaciones, consultas).

Aunque el código generado por el IDE nos puede ser de utilidad en algunas ocasiones, en este caso símplemente vamos a eliminarlo para crear algo más adaptado a nuestras necesidades. Obviamente no vamos a prestar atención al diseño, por lo que el contenido de nuestra vista podría ser similar al siguiente:

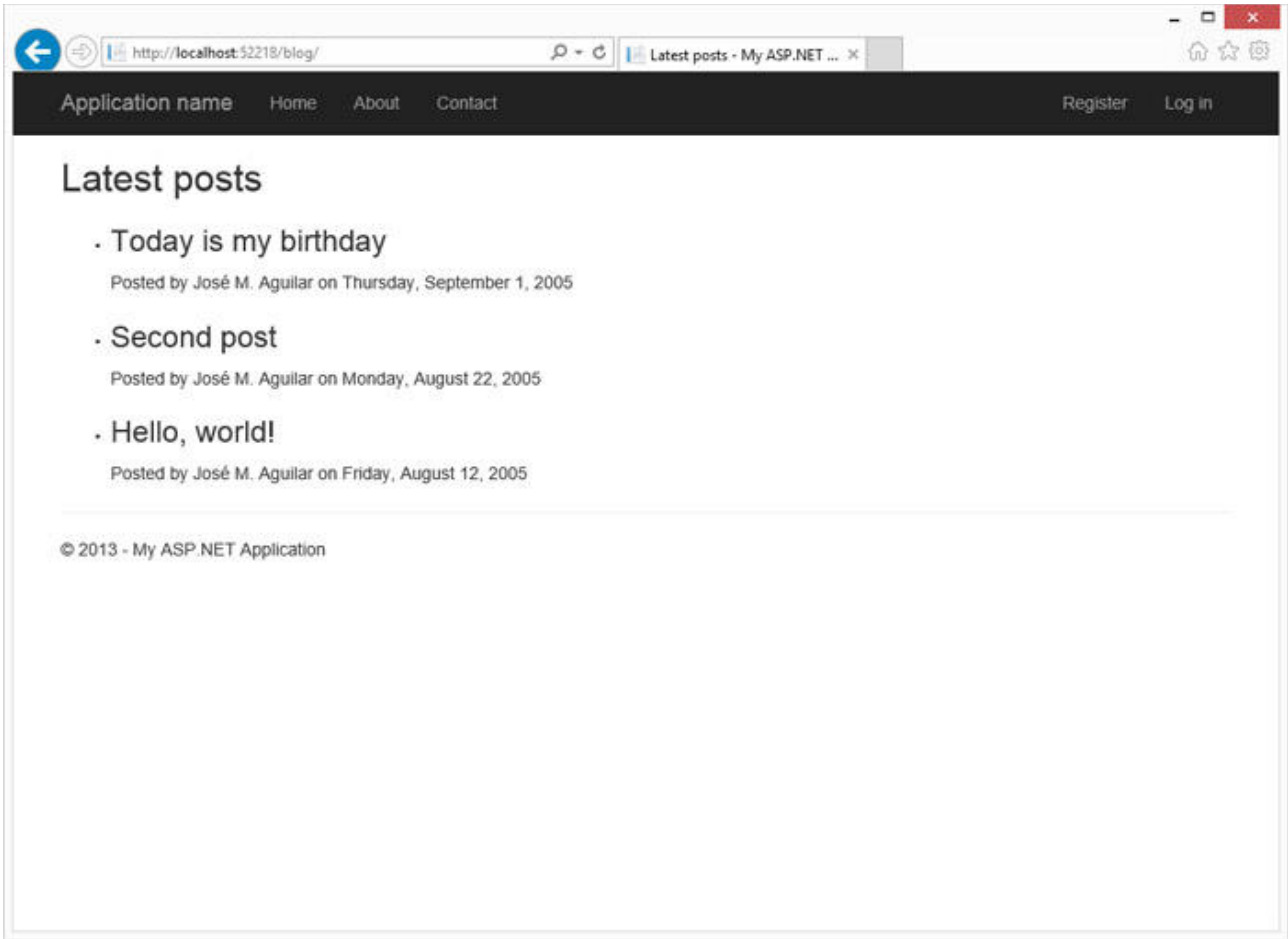
```
@model IEnumerable<MyFirstMvcApplication.Models.Post>
@{
   ViewBag.Title = "Latest posts";
}
<h2>Latest posts</h2>
```

```
<ul>
@foreach (var post in Model)
{
   <li>
      <h3>@post.Title</h3>
      <div>
         Posted by @post.Author on @post.Date.ToLongDateString()
      </div>
   </li>
}
</ul>
```

En esta ocasión la vista es un poco más compleja que la que creamos anteriormente, puesto que incluye una lista `<ul>`, y un bucle mediante el cual recorremos el `IEnumerable<Post>` que recibimos desde el Controlador para mostrar cada uno de los artículos que, a su vez, éste obtuvo desde el Modelo.

> Observa la forma tan limpia de mezclar código de cliente y servidor en Razor. Su parser, bastante más inteligente que el incorporado en Webforms/ASPX, es capaz de determinar de forma automática dónde empiezan y acaban los bloques de código, cuándo se trata de instrucciones de servidor y cuándo se está escribiendo marcado.

Por último, dado que para esta funcionalidad tenemos implementadas todas las capas (Model, Controlador y Vista), ya sería posible ver el resultado en ejecución lanzando la aplicación y accediendo a la URL /blog:

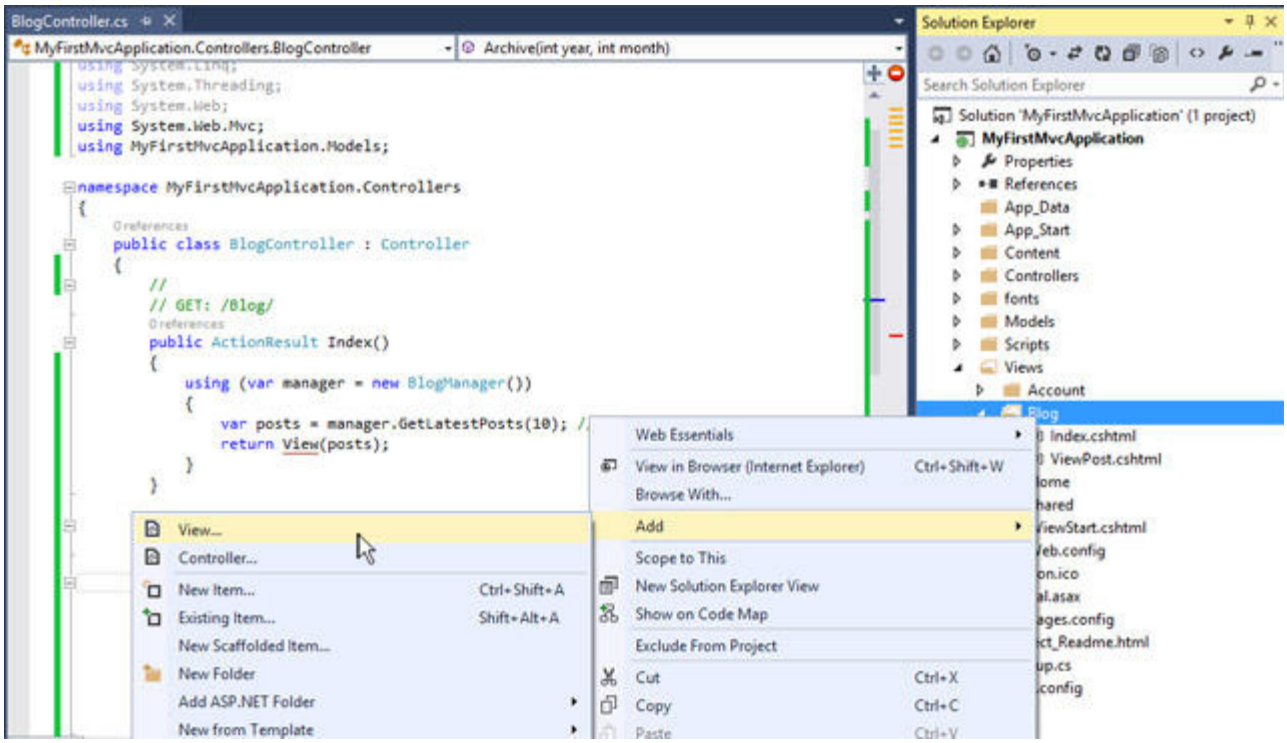


Vista Archive (Acceso al archivo por año y mes)

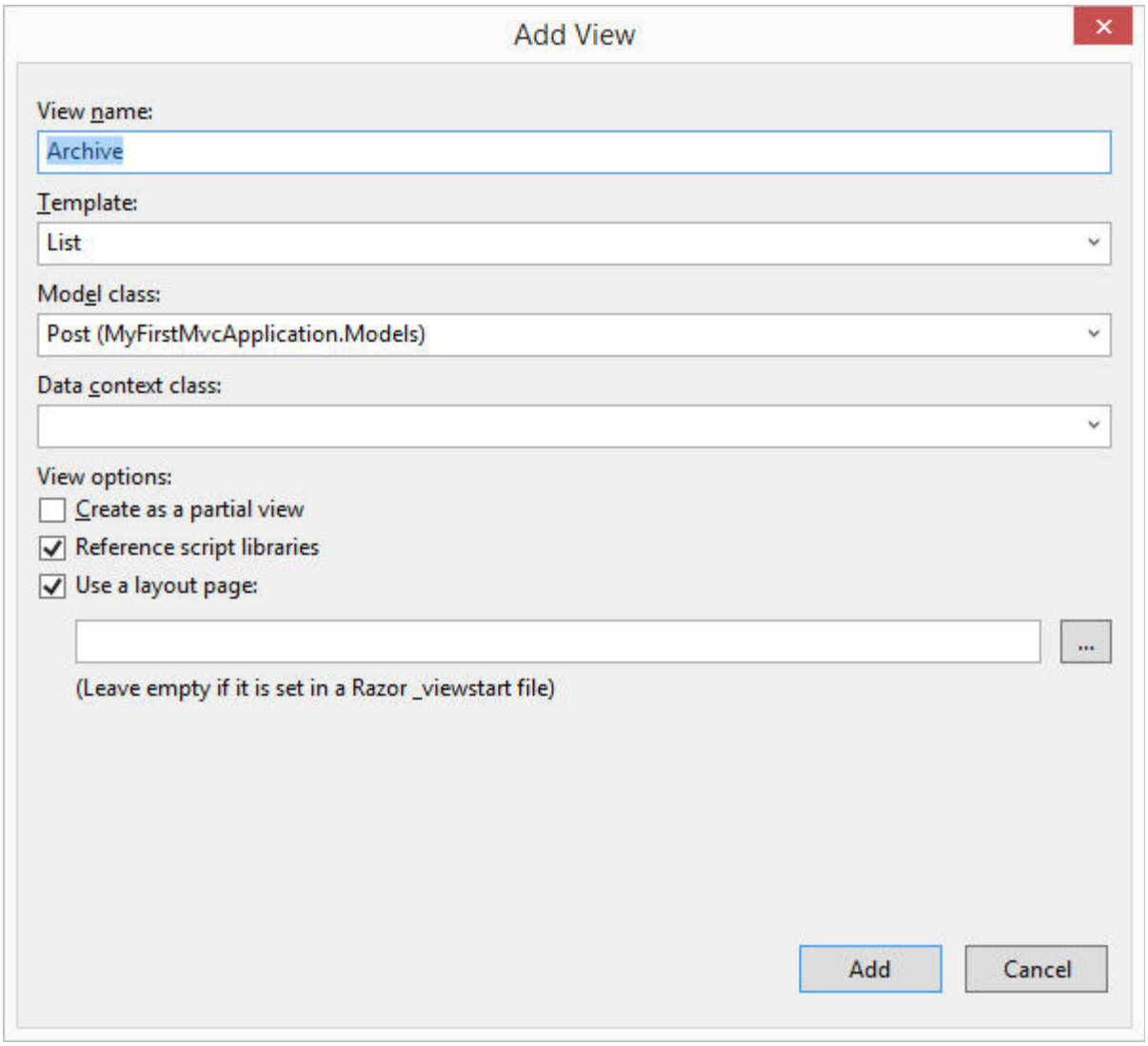
> ***Recuerda:***
> *Vista:* *Archive*
> *Petición:* `GET /blog/archive/2005/1`
> *Controlador: Blog (* `BlogController` *)*
> *Acción:* `Archive(int year, int month)`
> *Datos enviados a la vista:*
> *- Lista de posts (*`IEnumerable<Post>`*)*

Esta vista es prácticamente igual a la anterior, por lo que vamos a recorrer el proceso más rápidamente, aunque aprovecharemos para estudiar otra forma de crear estos componentes.

Anteriormente creamos la vista abriendo el menú contextual sobre el cuerpo del método de acción. En esta ocasión, la crearemos directamente sobre la carpeta donde sabemos que, por convención, debe encontrarse la misma. Para ello, podemos hacer click con el botón derecho del ratón sobre el directorio */views/blog* del proyecto y seleccionar la opción Añadir > Vista:



A continuación, debemos proporcionar cierta información para crear la vista, como su nombre (que en este caso el entorno no puede deducirlo del contexto, como antes), y el resto de datos serán idénticos a los introducidos anteriormente puesto que el tipo de información que recibirá la vista es la misma:



Una vez pulsado el botón "Añadir", la vista será creada, y dado que ya tenemos todas las capas implementadas, podrías acceder a ella con tu navegador utilizando una URL del tipo "/blog/archive/2005/9". Pero no es esto lo que queremos: como en el caso anterior, el contenido generado por defecto vamos a sustituirlo por uno más apropiado, similar al de la vista Index, aunque aprovecharemos para introducir el concepto de "lógica de presentación".

Por ejemplo, parece lógico pensar que nuestra vista debería incluir un encabezado indicando qué filtro temporal estamos aplicando durante la consulta al archivo para indicar al usuario que los posts mostrados pertenecen al mes X del año Z. Sin embargo, si observas los datos suministrados por el controlador (de tipo `IEnumerable<Post>`), esa información no aparece en ninguna parte...

Una posibilidad sería enviar estos datos, el mes y año de la consulta, desde el controlador utilizando `ViewBag`, de la misma forma que hemos visto que se hacía en el controlador Home creado por defecto. Otra forma de conseguirlo sería utilizar una clase

de trabajo que expusiera como propiedades dichos datos, además de la lista de artículos; más adelante veremos que esta forma de implementarlo, que es la más recomendable por cierto, se denomina *view-model pattern* (patrón modelo-vista), y se considera una buena práctica en el desarrollo con el framework MVC.

Sin embargo, vamos a solucionarlo de forma más sencilla. Dado que todos los artículos corresponden a dicho mes y año porque el controlador ya nos los está haciendo llegar filtrados, podríamos tomar el valor de cualquiera de los elementos de la colección, así que lo obtendremos del primero de ellos.

Por otra parte, para complicar un poco más el ejemplo, vamos a incluir algo de código para hacer que, cuando mostremos el listado de artículos, los elementos pares aparezcan resaltados con un color de fondo (sí, se puede hacer con CSS, pero entonces no valdría como ejemplo ;-)). Introduciremos un contador en el bucle que recorre los artículos e iremos generando el estilo del elemento de lista en función de la paridad del índice.

Observa que en ambos casos, estamos hablando de introducir cierta lógica en la vista, que es lo que se suele denominar lógica de presentación, es decir, código destinado a solucionar problemas relativos exclusivamente a la generación del interfaz, y que nada tienen que ver con el dominio o reglas de negocio de la aplicación. Veamos cómo quedaría la vista:

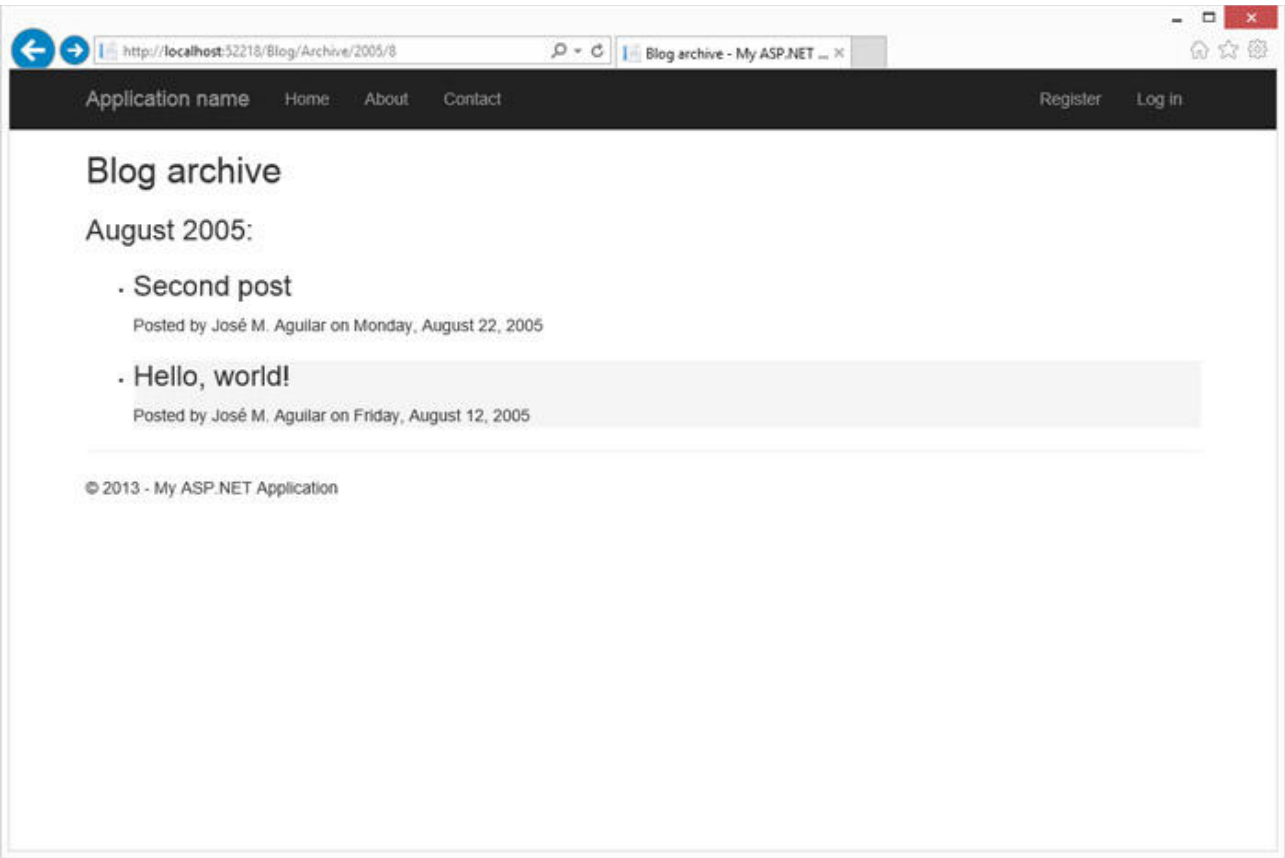
```
@model IEnumerable<MyFirstMvcApplication.Models.Post>
@{
   ViewBag.Title = "Blog archive";
}
<h2>@ViewBag.Title</h2>
@{
   var firstPost = Model.FirstOrDefault();
   var index = 0;
   if (firstPost != null)
   {
      <h3>@firstPost.Date.ToString("MMMM yyyy"):</h3>
      <ul>
      @foreach (var post in Model)
      {
         string style = (index++%2 == 0) ? null : "background-
color: #f5f5f5";
         <li style="@style">
            <h3>@post.Title</h3>
```

```
            <p>Posted by @post.Author on
@post.Date.ToLongDateString()</p>
         </li>
      }
      </ul>
   }
}
```

Como se puede observar, al principio obtenemos el primer elemento y lo utilizamos para generar el encabezado con información sobre el filtro, y a continuación recorremos la colección mostrando cada uno de sus elementos con el color de fondo apropiado en las filas pares.

> **Importante: Las vistas pueden contener lógica, pero exclusivamente de presentación. De no ser así, supondría una violación del patrón MVC.**

El resultado en ejecución sería el siguiente, una vez iniciada la aplicación e introduciendo en el navegador la URL `/blog/archive/2005/8` para visualizar los artículos publicados en agosto de 2005:



Práctica 2: Añadiendo funcionalidades

Objetivo

El objetivo de esta práctica es que profundices en la estructura de las aplicaciones y proyectos ASP.NET MVC, mediante la observación e investigación del proyecto que hemos ido desarrollando a lo largo de este módulo.

1. Enlazar el blog con el sitio web

1. Hasta ahora hemos accedido a las distintas funcionalidades del motor de blogs introduciendo directamente la URL en el navegador. Esta primera parte de la práctica consiste en crear una nueva opción en el menú principal de la aplicación (donde se encuentran los enlaces a las páginas "Home", "About" y "Contact") y hacer que enlace con la visualización del listado de últimos posts publicados.

2. En primer lugar, abre el Layout del sitio web (`/Views/Shared/_Layout.cshtml`) y observa que existe una lista sin orden (`<ul>`) en la que se encuentran definidas las opciones del menú. Fíjate en la forma en que se están creando los enlaces llamando a `Html.ActionLink()`. Se trata de un método helper que nos ayuda a crear etiquetas `<a>` dirigidas hacia controladores y acciones concretas; ya lo estudiaremos más adelante, de momento simplemente intenta adivinar qué significan cada uno de los parámetros utilizados.

3. Incluye un nuevo elemento en la lista, e introduce en él un tag `<a>` con el atributo `href` apuntando hacia la URL en la que se puede encontrar la funcionalidad de visualización del listado de los últimos posts publicados (/blog). Ejecuta la aplicación para comprobar que todo funciona correctamente.

4. Prueba a continuación, aunque sea de forma intuitiva en este momento, a utilizar el helper `Html.ActionLink()` para generar el enlace. Reflexiona sobre la ventaja de utilizar este método sobre la codificación manual del enlace que hiciste previamente.

   (Pista: ¿qué ocurriría si modificamos la tabla de rutas y hacemos que cambie la URL de acceso a las funcionalidades del blog?)

2. Enlazar el listado de posts con la visualización del post

1. En esta sección vamos a enlazar el listado de posts (al que ya debes poder acceder desde el menú principal de la aplicación), con la visualización de cada post, es decir, con la

acción `ViewPost()` que hemos ido implementando a lo largo del módulo.

2. Accede a la vista */Views/Blog/Index.cshtml*, y modifícala para que en cada elemento se incluya un enlace hacia la página donde se puede visualizar el post completo (la acción "ViewPost"), de forma que ya tengamos la navegación completa de nuestro pequeño motor de blogs. La fórmula más sencilla de hacerlo sería incluir algo como lo siguiente:

```
<a href="/blog/@post.Code">Read post...</a>
```

3. Observa que, de nuevo, estamos introduciendo URLs en nuestro código, lo cual introduce en el código de la vista una dependencia muy rígida respecto al contenido de la tabla de rutas actual. Es decir, si cambiamos la ruta de acceso a la acción "ViewPost", nuestros usuarios no podrían visualizar los posts completos, puesto que la URL especificada en la vista no se modificaría de forma automática.

   Comprueba que el sistema funciona de forma correcta, y después realiza este pequeño cambio en la tabla de rutas:

```
routes.MapRoute(
   name: "ViewPost",
   url: "view/{code}",
   defaults: new { controller = "Blog", action = "ViewPost" }
);
```

   Ejecuta de nuevo la aplicación, y verás cómo el link entre el listado de posts y la visualización individual ha dejado de funcionar.

4. Sustituye la etiqueta `<a>` que has introducido anteriormente por el siguiente código:

```
@Html.ActionLink("Read post...", "ViewPost", new { code = post.Code} )
```

   En él, estamos generando un enlace de forma sensible al contenido de la tabla de rutas de nuestra aplicación, por lo que cualquier cambio en ésta será tenido en cuenta. La sobrecarga de `Html.ActionLink()` que estamos utilizando asume que el controlador será el mismo que está siendo ejecutado en este momento (Blog), y observa que estamos suministrando el parámetro que necesita la acción ViewPost utilizando un objeto anónimo.

   Si vuelves a probar la aplicación, verás que los cambios en la tabla de rutas ahora se ven reflejados de forma inmediata en los enlaces.

3. Añadir visualización de comentarios al blog

En esta tercera parte de la práctica vamos a plantearnos un objetivo algo más ambicioso: añadir a la vista "ViewPost" (donde se visualizan los posts completos) los comentarios asociados al artículo que esté siendo visualizado.

Los pasos que debes seguir son los descritos a continuación:

1. Añade al modelo de datos de la aplicación una nueva entidad, llamada `Comment`, definida de la siguiente forma:

```
public class Comment
{
   public int Id { get; set; }
   public string Text { get; set; }
   public string Author { get; set; }
   public DateTime Date { get; set; }
}
```

2. Actualiza la entidad `Post` para añadirle una propiedad que permita acceder a la colección de objetos `Comment` asociados al artículo. El resultado debe ser algo así:

```
public class Post
{
   public Post()
   {
      Comments = new HashSet<Comment>();
   }
   public int PostId { get; set; }
   public string Title { get; set; }
   public string Code { get; set; }
   public string Text { get; set; }
   public DateTime Date { get; set; }
   public string Author { get; set; }
   public ICollection<Comment> Comments { get; set; }
}
```

3. Añade algunos comentarios a los posts de prueba que introdujimos en el inicializador de Entity Framework. Puedes seguir, por ejemplo, el siguiente patrón para añadir artículos con sus correspondientes comentarios:

```
context.Posts.Add(
   new Post()
   {
      Author = "José M. Aguilar",
      Title = "Hello, world!",
      Code = "hello-world",
```

```
      Date = new DateTime(2005, 8, 12),
      Text = "Hi, everybody, this is my first blog post!",
      Comments = new[] {
         new Comment() { Author = "John Doe", Date = new
DateTime(2005, 8,13), Text="Hey, this is great!" },
         new Comment() { Author = "Peter Petersen", Date =
new DateTime(2005, 8,14), Text="Good news! Keep writing,
please :)" }
      }
   }
);
```

4. A continuación, debemos modificar ligeramente la clase gestora para asegurar que cuando se obtenga un objeto de tipo `Post`, se incluya en él la colección de comentarios asociados al mismo. Los cambios a realizar en BlogManager.cs son:

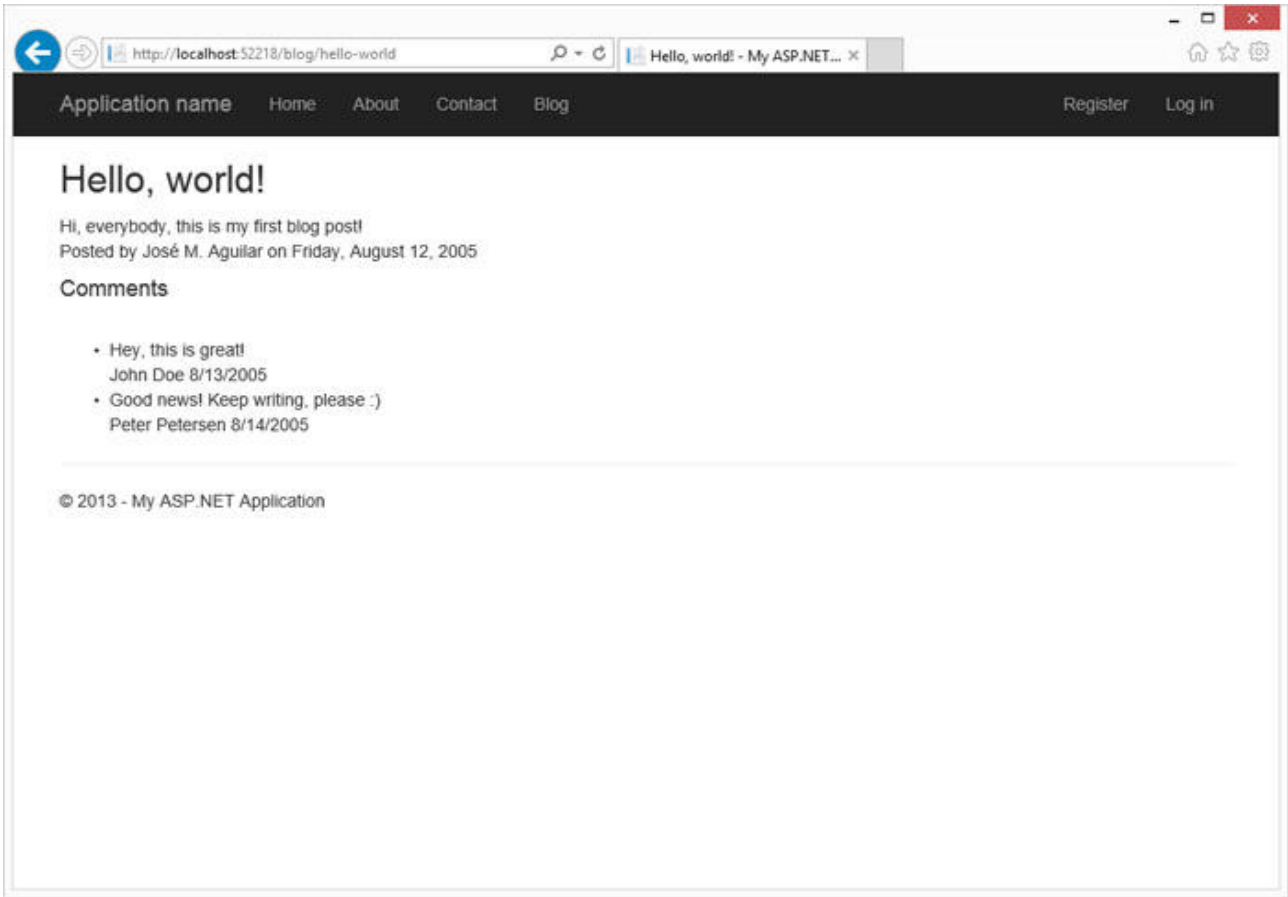
```
using System.Data.Entity;
...
public class BlogManager: IDisposable
{
   ...
   public Post GetPost(string code)
   {
      return _data.Posts.Include
(p=>p.Comments).FirstOrDefault(post => post.Code == code);
   }
}
```

5. Por último, actualiza la vista "ViewPost" para que muestre los comentarios asociados al Post actual, por ejemplo insertando este código:

```
@if(Model.Comments.Any())
{
        <h4>Comments</h4>
   <ul>

        @foreach(var comment in Model.Comments)
   {
      <li>

        <div>@comment.Text</div>
        <div>@comment.Author
        @comment.Date.ToShortDateString()</div>
      </li>
   }
```

```
    </ul>
}
```

El resultado obtenido debería ser aproximadamente como el mostrado en la siguiente captura de pantalla:



Encontrarás el proyecto completo para Visual Studio 2015 y 2013 en el área de recursos. En este proyecto, además, se ha implementado una funcionalidad adicional, el acceso desde el menú principal al archivo del blog, que puedes estudiar para comprender aún mejor la forma de programar con ASP.NET MVC.