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)

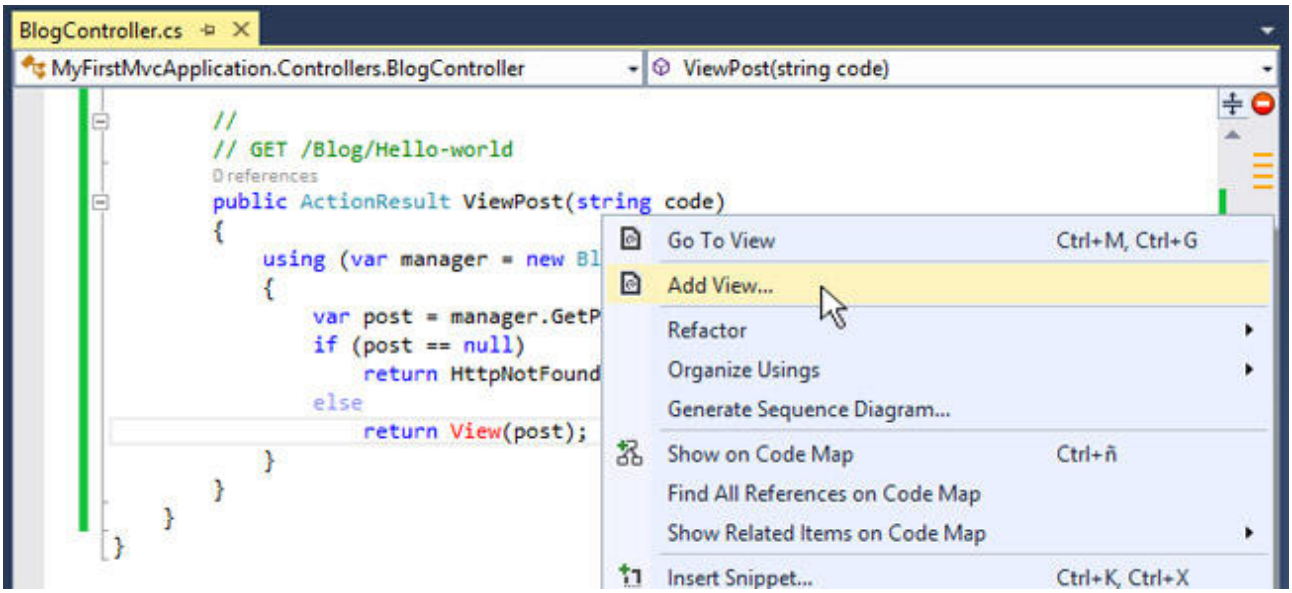
> ***Recuerda:***
> *Vista:* *ViewPost*
> *Petición:* `GET /blog/hello-world`
> *Controlador:* *Blog (* `BlogController` *)*
> *Acción:* `ViewPost(string code)`
> *Datos enviados a la vista:*
> *- Artículo (*`Post`*)*

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.

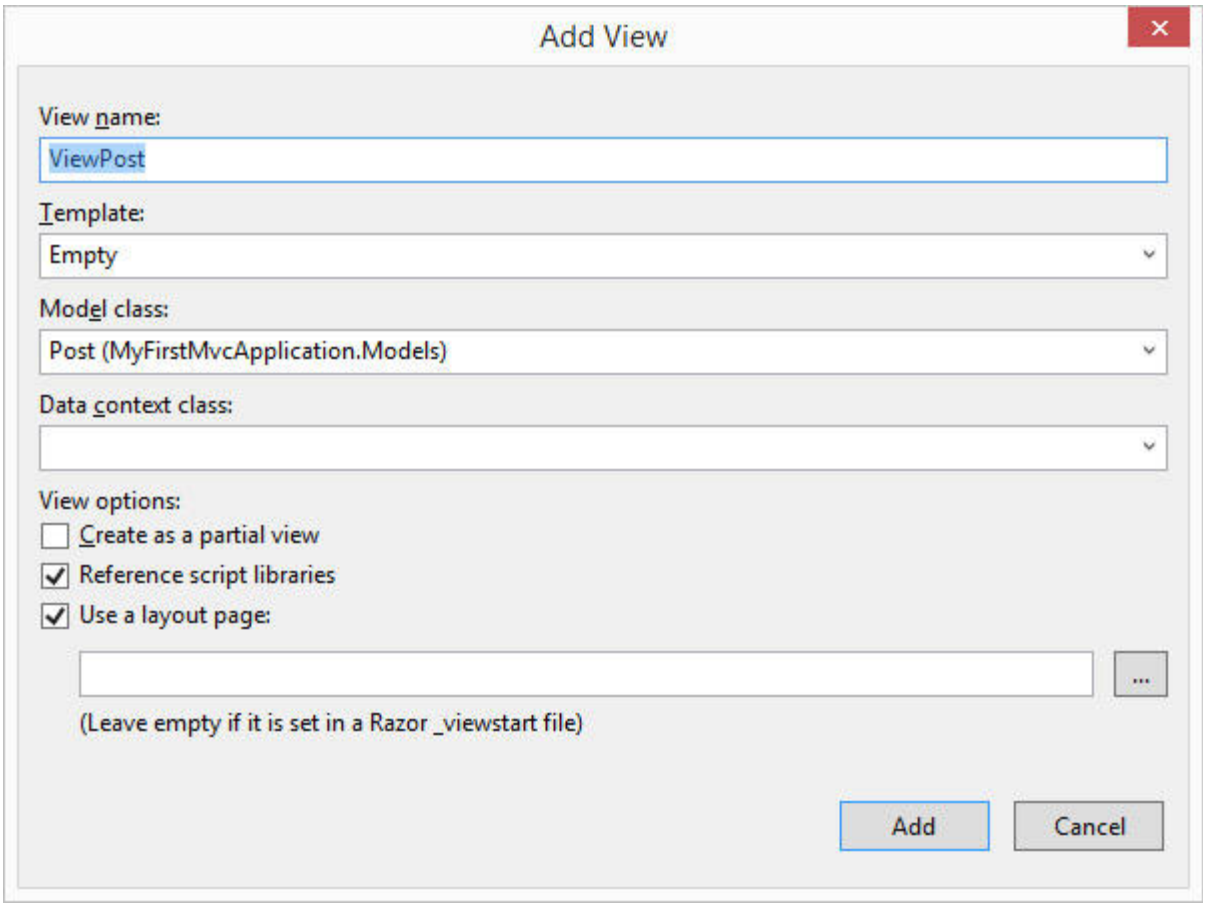
Usando el motor Razor, el más recomendable y utilizado, una vista no es más que un fichero con extensión .cshtml ubicado en el lugar donde el framework MVC espera encontrarlo según la convención de ubicación de archivos. Este archivo contiene la plantilla que será procesada en servidor y, utilizando 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:

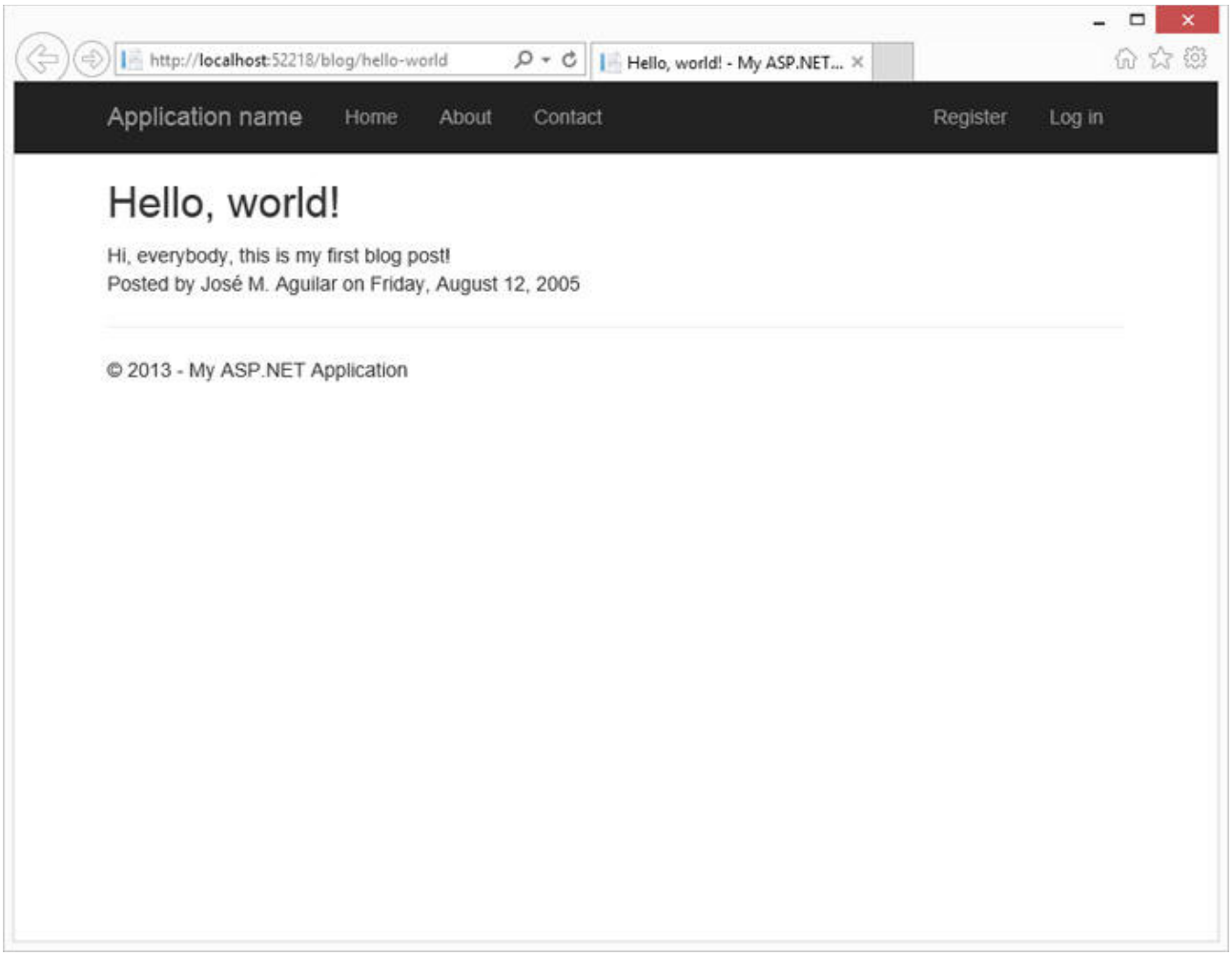
```
@model MyFirstMvcApplication.Models.Post
@{
   ViewBag.Title = Model.Title;
}

<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()`

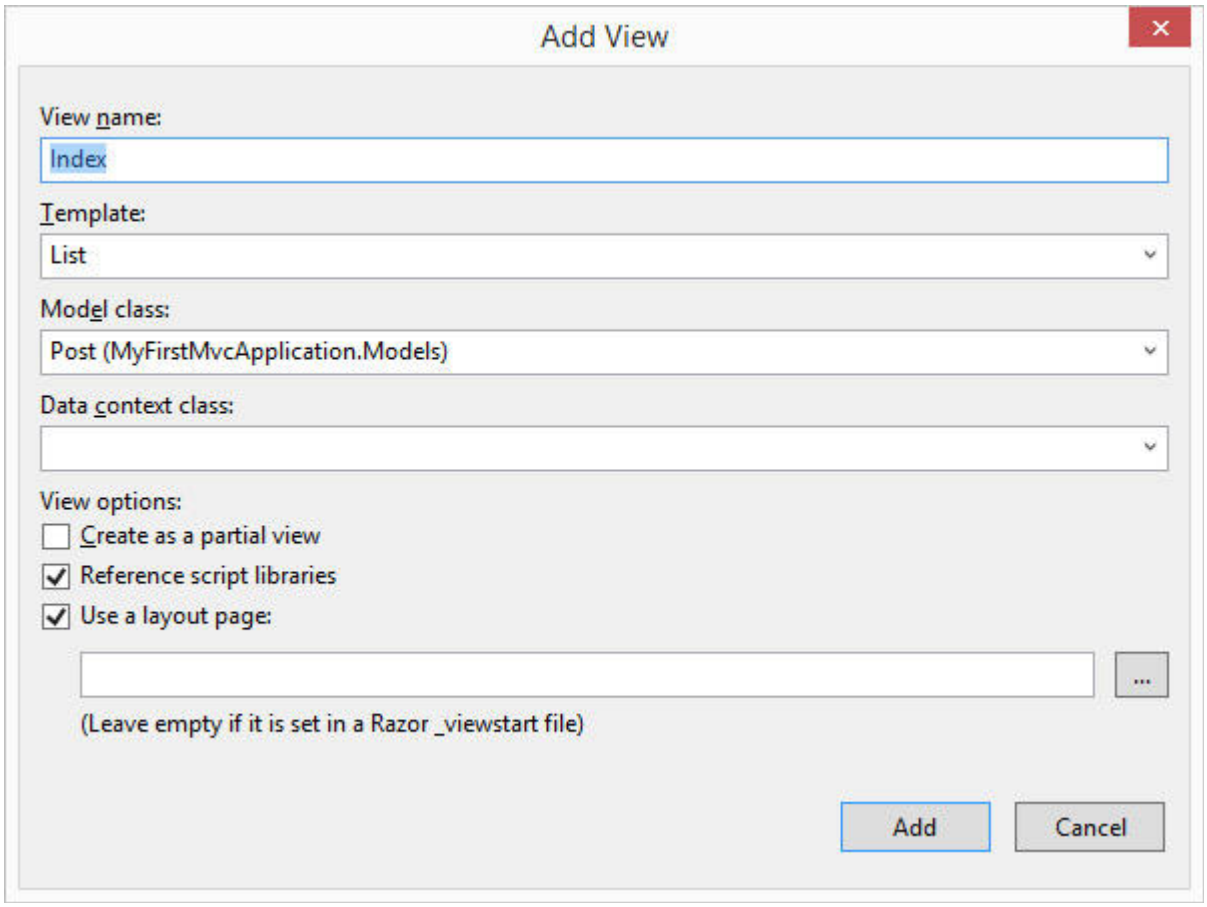
Datos enviados a la vista:

- 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.

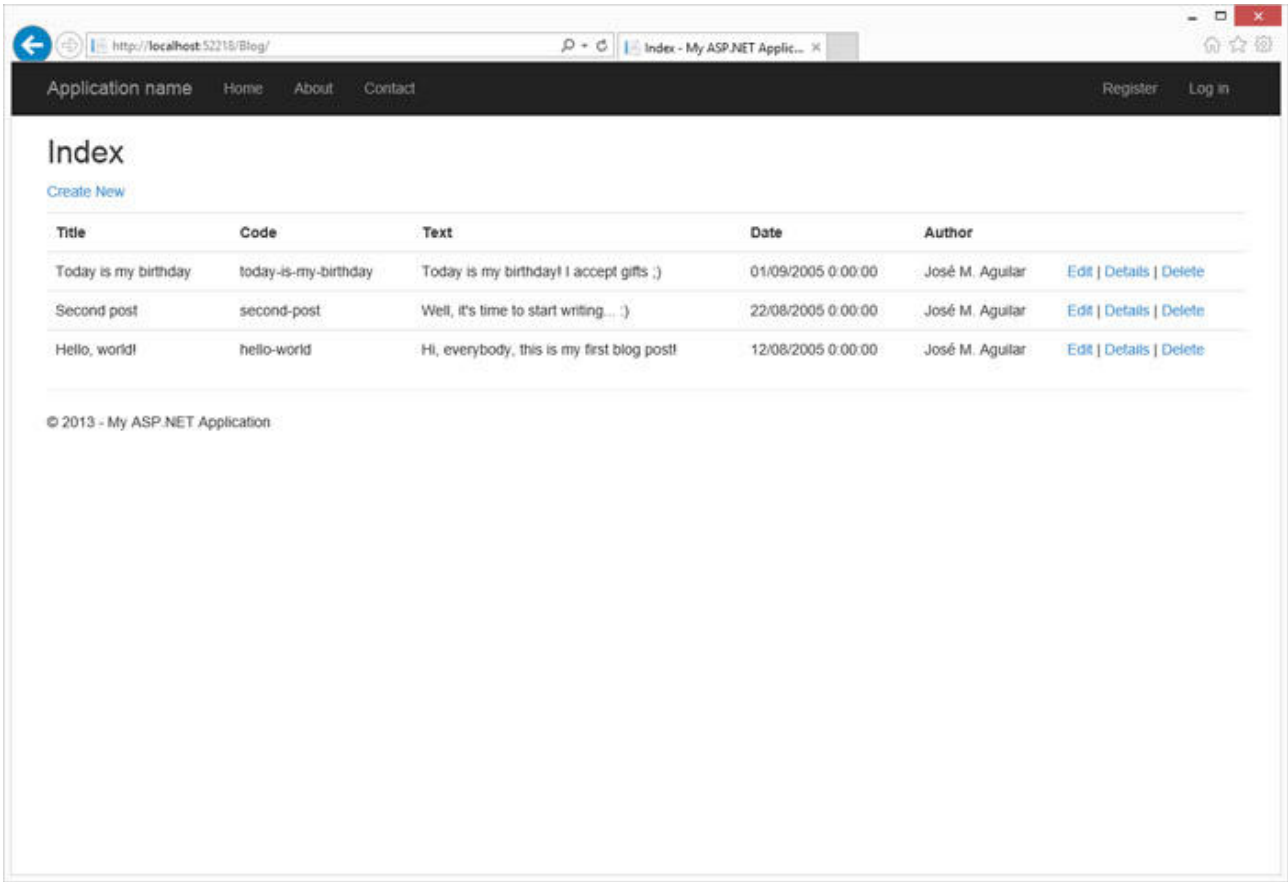
Para crear esta vista seguiremos un procedimiento similar al visto anteriormente, acudimos al cuerpo del método `Index()` del controlador `BlogController`, desplegamos el menú contextual y seleccionamos la opción "Añadir vista". En esta

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:

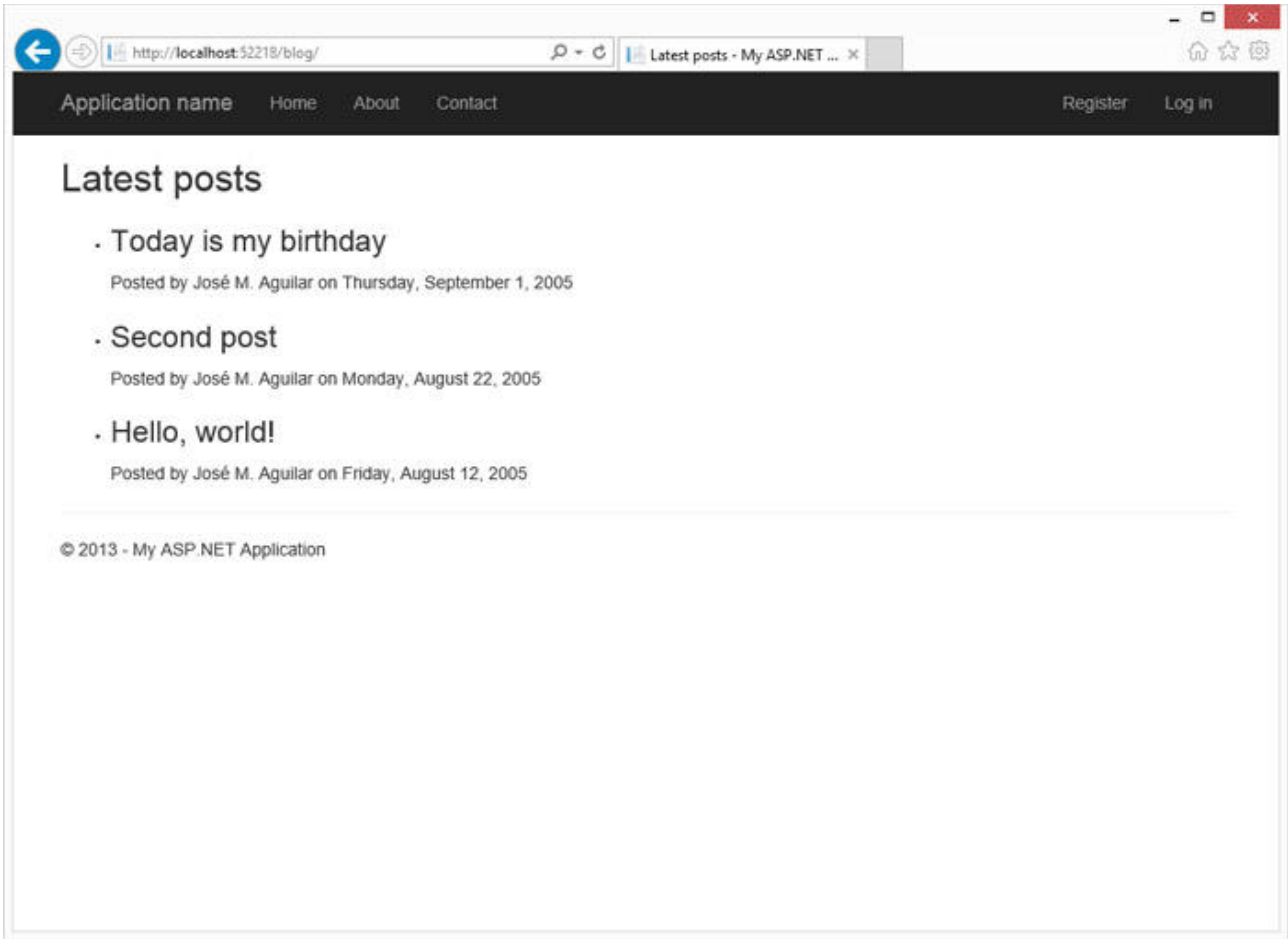
```
@model IEnumerable<MyFirstMvcApplication.Models.Post>
@{
   ViewBag.Title = "Latest posts";
}
<h2>Latest posts</h2>
```

```
<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:

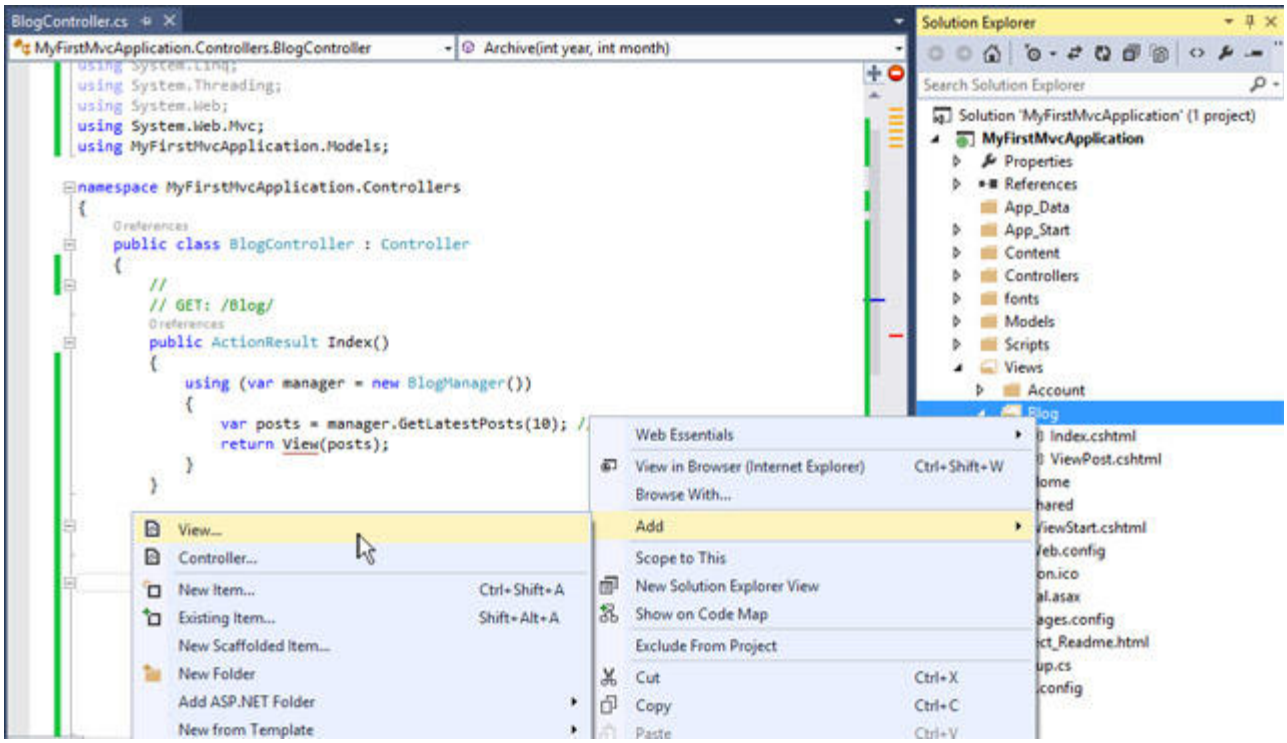


Vista Archive (Acceso al archivo por año y mes)

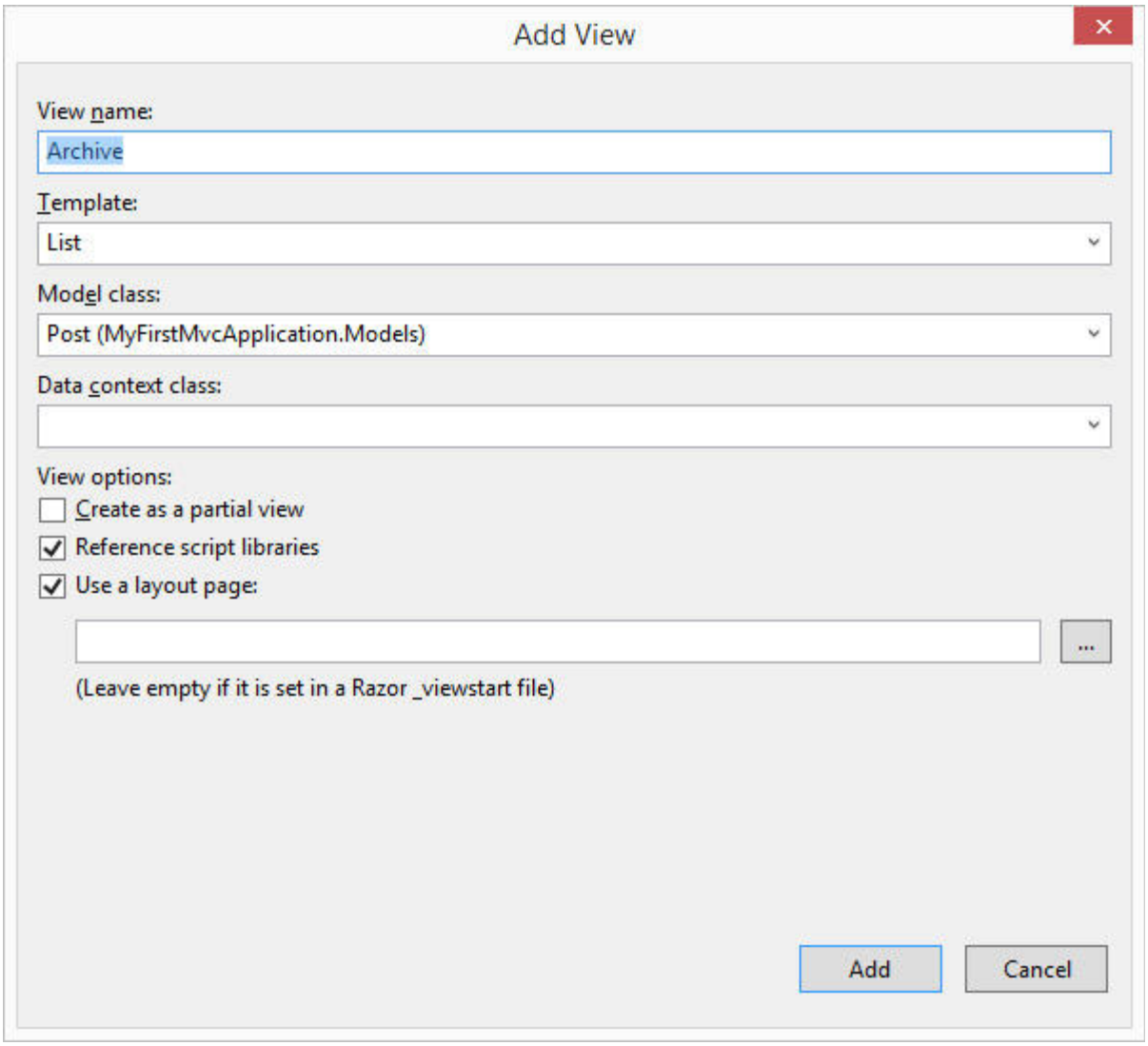
> ***Recuerda:***
> *Vista:* *Archive*
> *Petición:* `GET /blog/archive/2005/1`
> *Controlador: Blog (* `BlogController` *)*
> *Acción:* `Archive(int year, int month)`
> *Datos enviados a la vista:*
> *- Lista de posts (*`IEnumerable<Post>`*)*

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: