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:

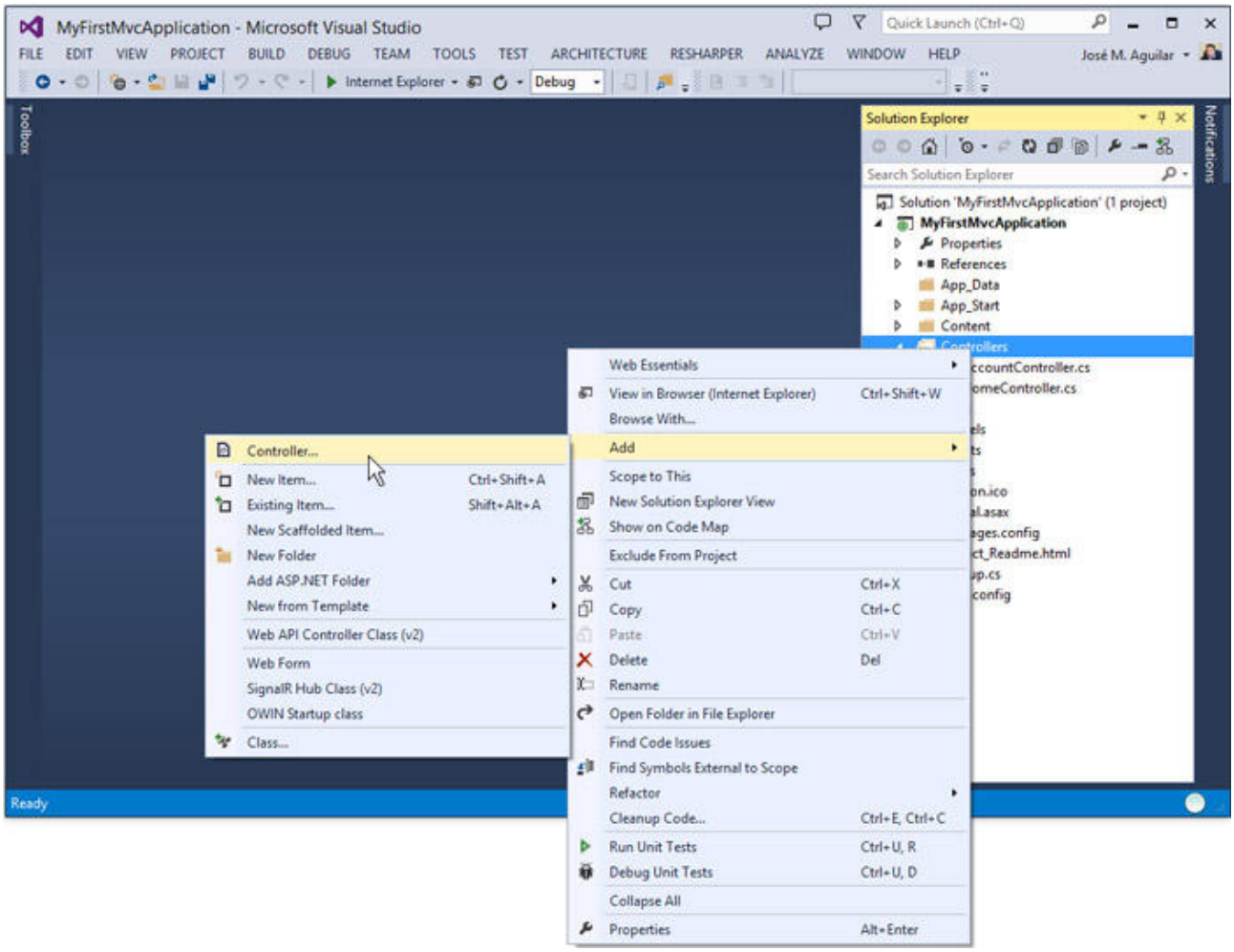
- `Index()`, que debe retornar al cliente una página mostrando una relación con los últimos artículos publicados.
- `Archive()`, para visualizar los artículos publicados en un año y mes concretos.
- `ViewPost()`, cuya misión será mostrar un artículo concreto.

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.

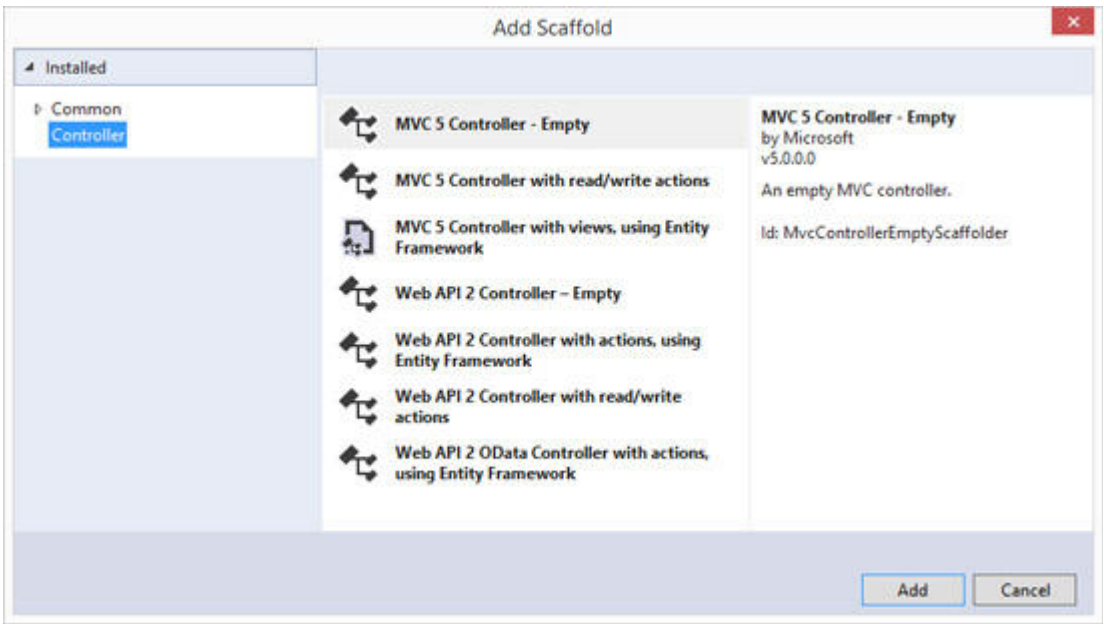
Creación de la clase controlador

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":

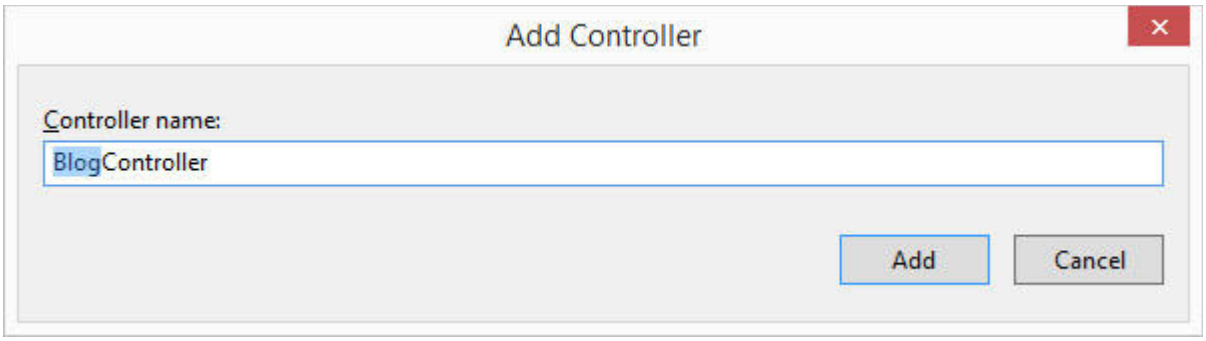


Seguidamente, Visual Studio mostrará un cuadro de diálogo consultándonos qué plantilla de controlador queremos utilizar como base. De momento, dado que queremos implementar nuestro controlador de forma manual, seleccionaremos el controlador MVC vacío:



> 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.
