



SERVICIO DE INFORMÁTICA | UNIVERSIDAD DE ALICANTE

ASP.NET MVC 3 y 4

CONTACTO CON MVC

Andrés Vallés Botella | Analista | Desarrollos propios
Servicio de Informática | Universidad de Alicante
Campus de Sant Vicent del Raspeig | 03690 | España
<http://si.ua.es/es/>

1º DÍA – CONTACTO CON MVC

Introducción a la historia de ASP.NET MVC	3
Modelo Vista Controlador	3
Evolución ASP (VBscript) a ASP.NET WebForms en el SI	6
Crear un proyecto básico	8
Controlador	14
Enrutamiento	15
Parámetros	16
Configurar enrutamiento en MVC 3	17
Configurar enrutamiento en MVC 4	18
Vista	20
Razor	23
Modelo	26
Usar un modelo dentro de un controlador / vista	27
Ejemplo práctico	29
Gestión de libros y revistas del SI	29
Integrar en base de datos con OracleDB	34



INTRODUCCIÓN A LA HISTORIA DE ASP.NET MVC

Es un entorno de trabajo cuya primera versión salió en marzo de 2009. Creado por Microsoft con objeto de ayudarnos a desarrollar aplicaciones que sigan la filosofía MVC, muy divulgada en otros lenguajes o entornos, sobre ASP.NET. Hasta el momento, para el desarrollo de aplicación web ASP.NET sólo nos permitía trabajar con Webforms. El objetivo de este curso es dar a conocer otra filosofía de trabajo, que por una parte nos dará la sensación de que volvemos al pasado (perdemos toda la potencia visual y de eventos de los Webforms), pero la claridad para trabajar, para el mantenimiento y sobre todo el control sobre el código (quebradero para más de uno en los últimos meses) puede que en muchos casos sea la mejor solución.

Además del conjunto de librerías (ensamblados) que proporcionan las nuevas funcionalidades a nivel de API, incluye plantillas y herramientas que se integran en Visual Studio 2008 y 2010 (tanto en la versión Express de Visual Web Developer como en sus hermanas mayores) para facilitarnos un poco las cosas.

En marzo de 2010 apareció la 2ª versión, un año después la 3ª versión, que es la estable actualmente y en la que centraremos casi todo el curso. Es inminente la salida de la 4ª versión que se encuentra en la fase Release Candidate, a la que dedicaremos dos sesiones para ver las novedades que aporta.

Visual Studio 2010 ya incorpora ASP.NET MVC 2 de serie, mientras que en la versión 2008 es necesario descargar e instalar el software (<http://www.asp.net/mvc>). ASP.NET MVC 3 y 4 RC sólo funciona con Visual Studio 2010 y 2012 y la podemos descargar de la dirección anterior o usar la herramienta [Web Platform Installer](#).

Por último indicar que ASP.NET MVC Framework es software libre, lo que ha permitido que se integre en otros entornos (por ejemplo Mono y MonoDevelop) y su código fuente está disponible en Codeplex.

MODELO VISTA CONTROLADOR

Modelo Vista Controlador (MVC) es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

Se trata de un modelo muy maduro y que ha demostrado su validez a lo largo de los años en todo tipo de aplicaciones, y sobre multitud de lenguajes y plataformas de desarrollo.

- El **Modelo** que contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia.
- La **Vista**, o interfaz de usuario, que compone la información que se envía al cliente y los mecanismos interacción con éste.
- El **Controlador**, que actúa como intermediario entre el Modelo y la Vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.



El modelo es el responsable de:

- Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.
- Define las reglas de negocio (la funcionalidad del sistema). Un ejemplo de regla puede ser: "Si la mercancía pedida no está en el almacén, consultar el tiempo de entrega estándar del proveedor".
- Lleva un registro de las vistas y controladores del sistema.
- Si estamos ante un modelo activo, notificará a las vistas los cambios que en los datos pueda producir un agente externo (por ejemplo, un fichero por lotes que actualiza los datos, un temporizador que desencadena una inserción, etc.).

EL CONTROLADOR ES RESPONSABLE DE:

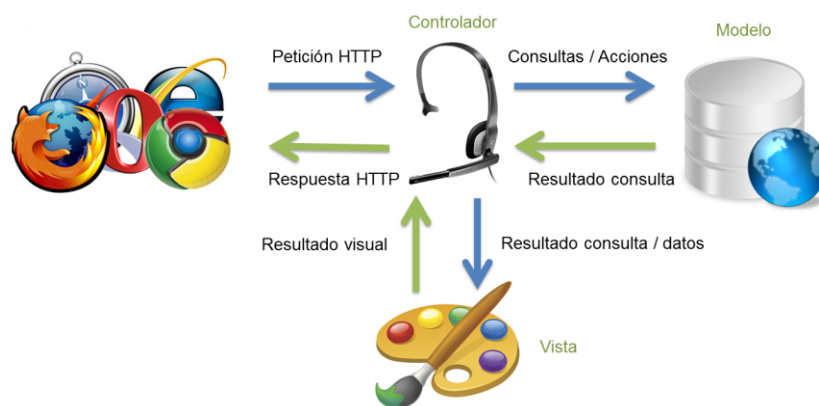
- Recibe los eventos de entrada (un clic, un cambio en un campo de texto, etc.).
- Contiene reglas de gestión de eventos, del tipo "SI Evento Z, entonces Acción W". Estas acciones pueden suponer peticiones al modelo o a las vistas. Una de estas peticiones a las vistas puede ser una llamada al método "Actualizar()". Una petición al modelo puede ser "Obtener_tiempo_de_entrega (nueva_orden_de_venta)".

LAS VISTAS SON RESPONSABLES DE:

- Recibir datos del modelo y los muestra al usuario.
- Tienen un registro de su controlador asociado (normalmente porque además lo instancia).
- Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).



El flujo que sigue el control generalmente es el siguiente:



1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)
2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, se podría utilizar el patrón Observador para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio. Un objeto vista puede registrarse con el modelo y esperar a los cambios, pero aun así el modelo en sí mismo sigue sin saber nada de la vista. El controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice. Nota: En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista.
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.



EVOLUCIÓN ASP (VBSCRIPT) A ASP.NET WEBFORMS EN EL SI

Aunque han pasado más de 15 años desde que se comenzó a desarrollar con ASP en el Servicio de Informática, si que podemos destacar algunos hitos que nos permiten mostrar como poco a poco estábamos evolucionando a un modelo MVC.

ASP (VBSCRIPT)

FASE INICIAL

Se desarrollaba todo en uno, es decir el mismo ASP incluía toda las funcionalidades:

- Incluía código HTML y dentro se embebía el código ASP.
- Había pocos includes que hacían referencias a funciones muy generales o muy utilizadas
- Se hacía muy complicado de gestionar por el tamaño que llegaban a ocupar los ficheros, en muchos casos de miles de líneas
- Se reutilizaba poco código

1º EVOLUCIÓN

Se comienzan a crear clases reutilizables para centralizar la mayoría de las funciones generales:

- Conexión y operaciones con bases de datos, control de errores, control de la seguridad, validación de campos, trabajo con ficheros PDF o Excel, etc.
- Por optimización de tiempos se generaliza el uso de los includes con elementos de diseño comunes(cabecera, cuerpo y pie) y el código HTML restante se suele generar desde programación

Aunque de forma muy preliminar se comienza a trabajar con un modelo modelo-controlador con aquellos elementos más generales o utilizados.

- Modelo: clases generales
- Controlador: el propio ASP.

2º EVOLUCIÓN

Se incluye el trabajo con plantillas y con todas las clases creadas se dispone de un Framework completo de trabajo:

- Se dispone de dos tipos de plantillas las que se almacenan en memoria de cada aplicación para mejorar los tiempos de carga y las que se cargan al vuelo (nos permitía obtener plantillas de otros entornos).
- Con el framework y con las plantillas se reduce en un 90% el tamaño de los ficheros ASP. También se reducen el número de includes con elementos de diseño.
- Se compilan las clases para evitar duplicidad de includes.



- Muchos programadores comienzan a desarrollar todo con objetos, de manera que se reaprovecha mucho código y se descentraliza el papel del propio ASP a las clases que incluya.
- Al usuario acostumbrado a trabajar como en la fase inicial se le hace muy difícil centrarse en la lógica del programa y de lo que se visualiza por pantalla, porque lo primero se gestiona en las clases y lo segundo en las plantillas.

En este punto realmente se trabajaba con un sistema MVC rudimentario pero completo.

- Modelo: clases
- Vista: plantillas
- Controlador: el propio ASP.

ASP.NET (WEBFORMS)

FASE INICIAL

Se parte de un modelo parecido al del ASP (VBScript) en su 1ª evolución (el cambio a .NET nos pilló en la transición de la 1ª a la 2ª fase):

- Se crea una framework básico con los objetos más utilizados en ASP, destaco algunos:
 - Trabajo con base de datos (Oracle y SQL Server)
 - Gestión de errores, con avisos personalizados a los programadores
 - Gestión de Seguridad, sql injection, validación de campos, conversiones, etc.
 - Envíos de correos, descargas, etc.
- Se desarrolla buena parte de la lógica dentro del ASPX aunque se comienza a desarrollar con clases/objetos

FASE ACTUAL

Personalmente creo que se trabaja con modelo MVC bastante completo y que se acerca mucho a los que va a ver en este curso:

- Los programadores se conciencian completamente del uso de clases.
- Los ASPX se encargan exclusivamente de gestionar los eventos, y muy poco código para crear y usar objetos.
- El Framework ha madurado, se utilizan en muchas aplicaciones y por la totalidad de los programadores. Se realiza una buena documentación y curso de formación.
- Se crea un nuevo concepto de plantilla que unifica el entorno en todas las aplicaciones, las del Campus Virtual, las que se desarrollan como aplicación externa o las que se integran en las páginas web de la universidad.
 - El programador selecciona el entorno para el que va orientada su aplicación, y en tiempo de ejecución se ajusta a éste. Es totalmente transparente para éste.



Cada elemento del modelo quedaría definido como.

- Modelo: clases
- Vista: plantillas y masterpages
- Controlador: el propio ASPX

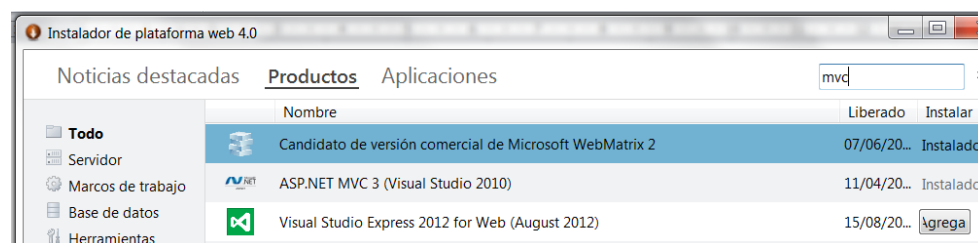
CREAR UN PROYECTO BÁSICO

ELEMENTOS NECESARIOS

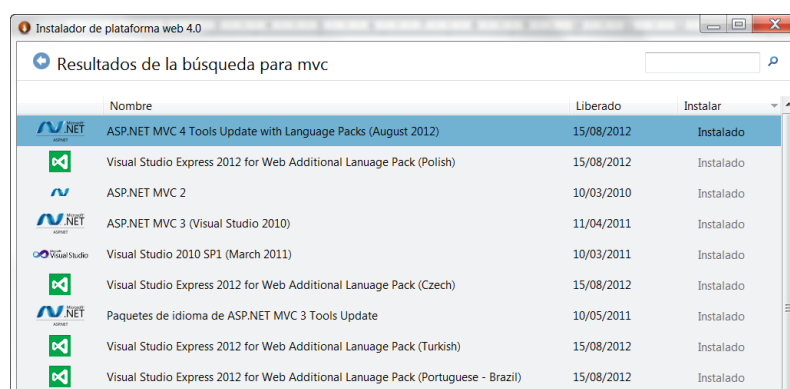
Para poder desarrollar en nuestro equipo con MVC 3 o MVC 4 debemos tener instalado el Visual Studio 2010. Como he comentado antes lo más sencillo para instalarlo es usar la herramienta Web Platform Installer. Ocupa muy poco espacio y es muy sencilla de utilizarla. La podemos descargar de

<http://www.microsoft.com/web/downloads/platform.aspx>

Seleccionamos la opción Productos y buscamos por el término mvc.



En los resultados podemos pulsar el botón de Agregar que hay a la derecha de cada producto. Debemos marcar ASP.NET MVC 3 y una actualización.



En cuanto a ASP.NET MVC 4, en caso de que no os aparezca en este listado, lo podéis descargar desde la siguiente dirección:

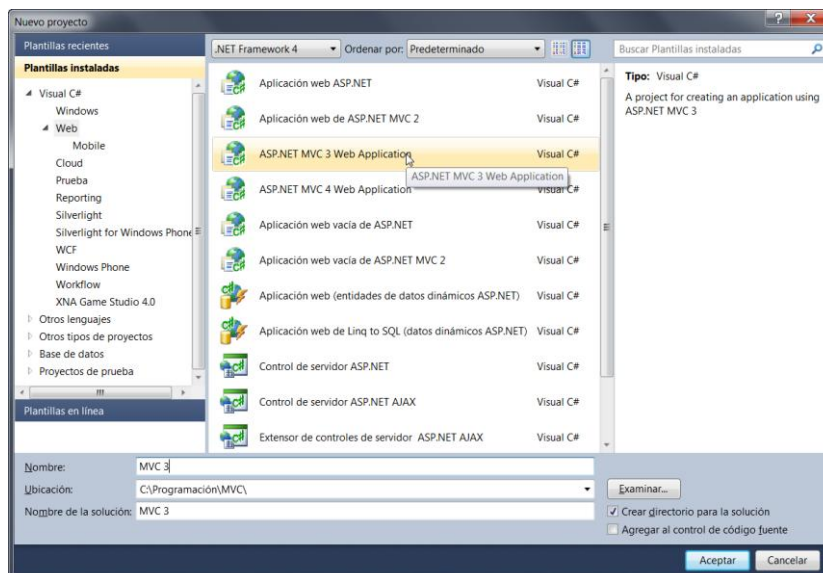
<http://www.asp.net/mvc/mvc4>



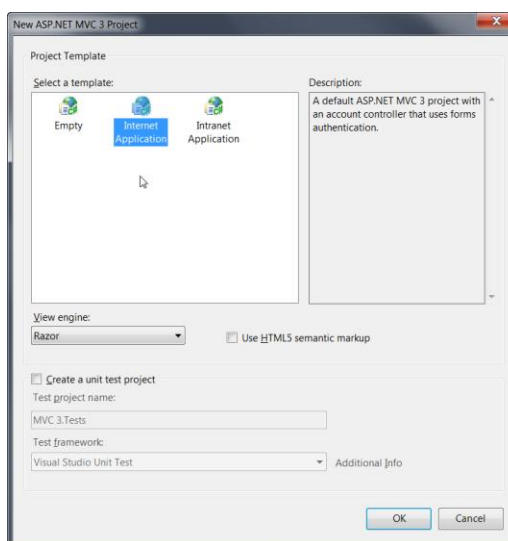
A continuación vamos a explicar brevemente como se crea una aplicación MVC 3 y MVC 4 y todos los elementos que lo componen.

MVC 3

Abrimos un nuevo proyecto (Archivo->Nuevo->Proyecto) y seleccionamos ASP.NET MVC 3 Web Application



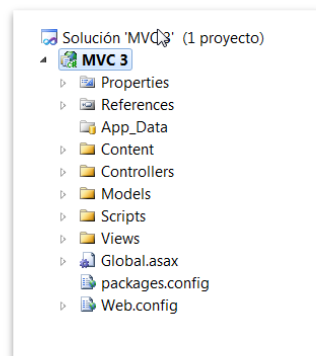
e indicamos el nombre del proyecto y dónde queremos almacenarlo. Pulsamos Aceptar.



Seleccionamos Internet Application y dejamos el resto de valores a los que vienen por defecto. Pulsamos Ok.

Veremos un proyecto con la siguiente estructura





- **Content**, carpeta donde se almacena el contenido estático de la aplicación. Generalmente se alojan las hojas de estilo, las imágenes, etc.
- **Controllers**, carpeta donde almacenamos los controladores. El marco de MVC requiere que los nombres de todos los controladores terminen con "Controller", como HomeController, CatalogadorController o UsuariosController.
- **Models**, carpeta donde almacenamos las clases que representan los modelos que usaremos en nuestra aplicación.
- **Scripts**, carpeta dónde alojamos todos los ficheros javascript que necesitemos en nuestra aplicación. Por defecto ya encontramos un conjunto de ficheros que usaremos para las llamadas AJAX y la biblioteca de jQuery (que es la que usa por defecto Visual Studio).
- **Views**, que es la ubicación recomendada para las vistas.

Las vistas usan archivos ViewPage (.aspx), ViewUserControl (.ascx) y ViewMasterPage (.master), además de otros archivos relacionados con la representación de vistas.


La carpeta Views contendrá muchas carpetas, al menos tantas como controladores. Al generar una vista se crea una carpeta con el nombre del controlador (quitando la palabra "Controller", y a su vez dentro de ésta, creará tantos ficheros como acciones disponga el controlador.

Dentro de la carpeta Views nos encontramos de una carpeta denominada Shared que usaremos para alojar todo aquello que sea común para todos los controladores. Por ejemplo podemos colocar las páginas maestras, podemos alojar ViewUserControl comunes. Todo esto lo veremos con detalle en la sección de plantillas.

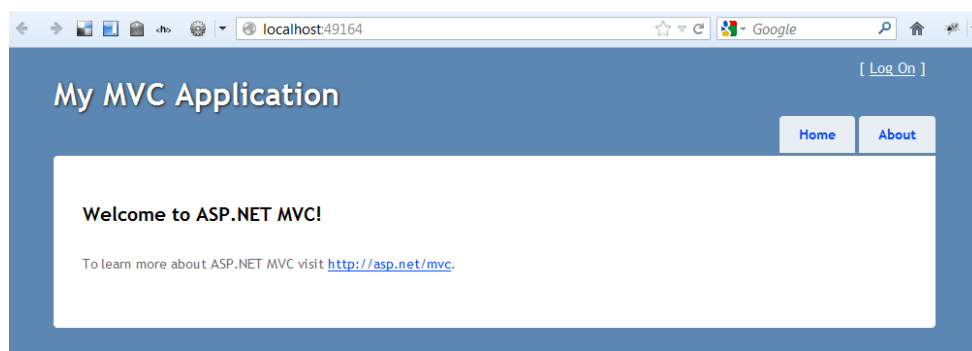
- **Ficheros especiales**

Como en toda aplicación ASP.NET, se dispone de dos carpetas para configuración y definición de la aplicación; Global.asax y Web.config.

El fichero Global.asax la usaremos para definir el enrutamiento de las direcciones URL.

Para probar si funciona, pulsamos la tecla F5 o pulsamos la imagen de reproducir  y se abrirá el navegador por defecto con la siguiente página





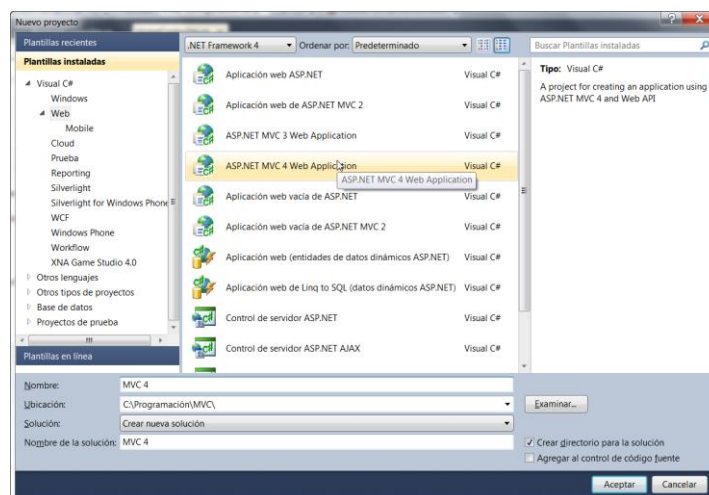
Se abre en nuestro ordenador (localhost) y usa un puerto no estándar cada vez que ejecutemos un proyecto diferente (49164).

Esta plantilla es muy interesante porque incluye dos opciones, Home y About que se corresponden con los métodos del fichero HomeController.cs y porque integra una identificación y alta de usuarios que se gestionan en AccountController.cs que se encuentra, como el controlador anterior, en la carpeta Controllers.

Por defecto cuando hemos pulsado F5 ha abierto el controlador HomeController y el método Home.

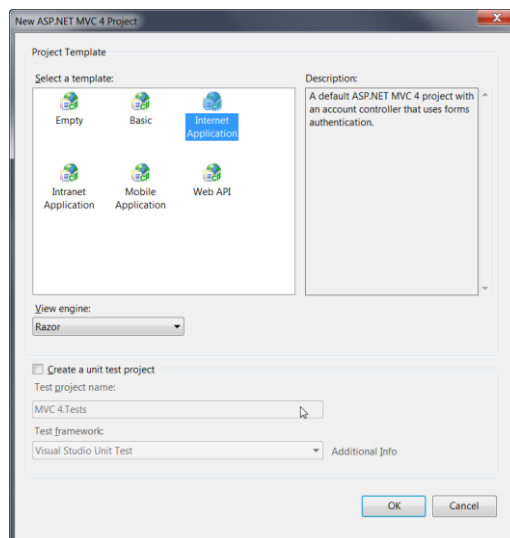
MVC 4

Abrimos un nuevo proyecto (Archivo->Nuevo->Proyecto) y seleccionamos ASP.NET MVC 4 Web Application

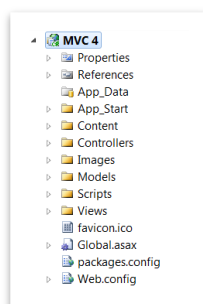


Aparecen algunos tipos de proyectos nuevos con respecto a MVC 3 y algunas opciones ya no son personalizables. Seleccionamos Internet Application y dejamos el resto de valores a los que vienen por defecto. Pulsamos Ok.




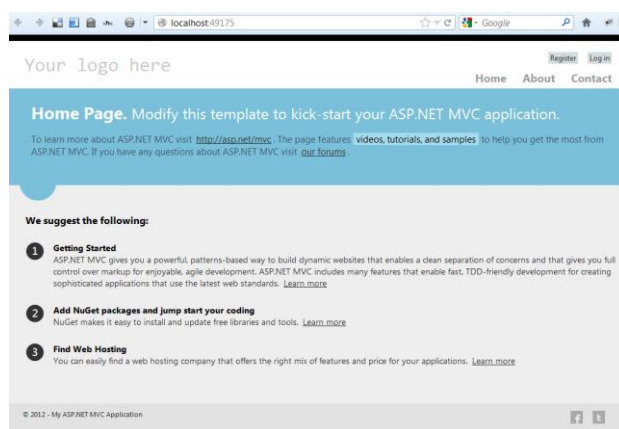


Veremos un proyecto con la siguiente estructura



Comparte la mayoría de los elementos de MVC 3, pero incorpora a primera vista una nueva carpeta App_Start y un icono favicon.ico para personalizar la imagen que aparece junto a la URL de nuestra Web. Además ha desaparecido el fichero Global.asax. Sus funciones ahora las asumen el fichero RouteConfig.cs que se encuentran en la carpeta App_Start. Los otros ficheros los explicaremos posteriormente.

Pulsamos la tecla F5 o pulsamos la imagen de reproducir  y se abrirá el navegador por defecto con la siguiente página



La plantilla es diferente, algo más moderna, pero incluye muchos de los elementos de la aplicación MVC 3. Por destacar cambios aparece una nueva opción Contact, integración con las redes sociales, enlace directo al registro sin pasar por el Log In.

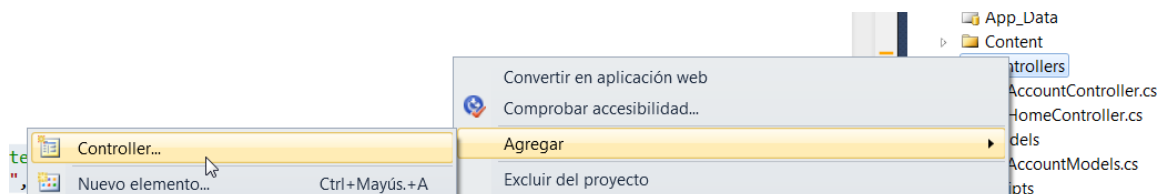
Por defecto cuando hemos pulsado F5 ha abierto el controlador HomeController y el método Home.



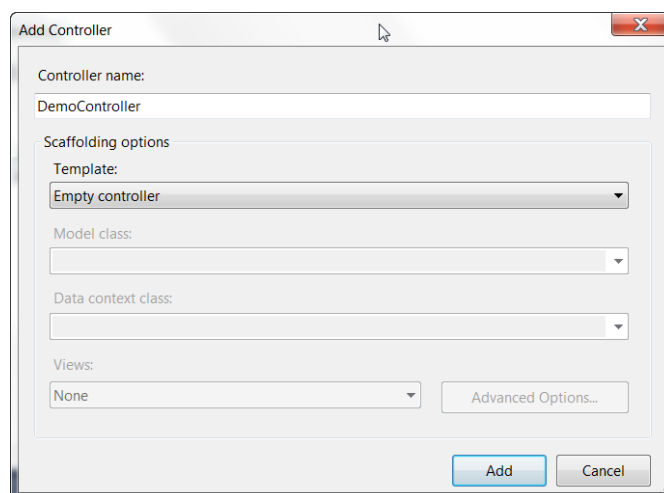
CONTROLADOR

El controlador es el primero de los elementos que vamos a ver porque es totalmente independiente de los otros y además es el que nos va a permitir realizar las primeras pruebas con nuestra aplicación.

La forma mas sencilla de crear un nuevo controlador es pulsar el botón derecho sobre la carpeta Controllers, y seleccionar Agregar > Controller.



Lo obligatorio es ponerle un nombre, en nuestro caso DemoController y no seleccionamos ninguna plantilla/template



El ejemplo que nos genera es el siguiente

```
public class DemoController : Controller
{
    //
    // GET: /Demo/


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

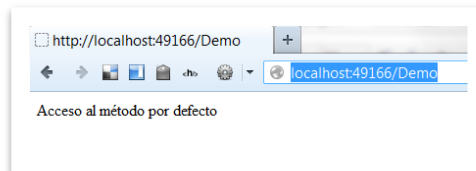
Nos crea una nueva clase y dentro de ésta un único método Index. Este sería el método de acceso a nuestro controlador. Como comentario nos indica como podremos acceder a nuestro controlador. No indica nada de la acción Index, porque es el método por defecto y no es necesario indicarlo.



El método devuelve un tipo ActionResult que lo debe generar la llamada a View(). Para ver de una forma sencilla como funciona la llamada a controladores, vamos a reemplazarlo por string y vamos a devolver realmente una cadena.

```
public string Index()
{
    return "Acceso al método por defecto";
}
```

Pulsamos la tecla F5 o pulsamos la imagen de reproducir  para visualizar el controlador por defecto y le añadimos /Demo (en mi caso http://localhost:49166/Demo)



ENRUTAMIENTO

El enrutamiento ASP.NET permite usar direcciones URL que no es necesario asignar a archivos específicos de un sitio web. Dado que la dirección URL no tiene que asignarse a un archivo, se pueden usar direcciones URL que describan la acción del usuario y, por tanto, sean más fáciles de comprender.

En una aplicación ASP.NET que no utiliza el enrutamiento, una solicitud entrante de una dirección URL normalmente se asigna a un archivo físico que controla la solicitud, como un archivo .aspx. Por ejemplo, una solicitud de http://server/application/Products.aspx?id=4 se asigna a un archivo denominado Products.aspx que contiene código y marcado para representar una respuesta al explorador. La página web utiliza el valor de cadena de consulta id=4 para determinar el tipo de contenido que se va a mostrar.

En el enrutamiento de ASP.NET, se pueden definir modelos de dirección URL que se asignen a archivos de controlador de solicitudes pero que no necesariamente incluyan los nombres de esos archivos en la dirección URL. Además, se pueden incluir marcadores de posición en un modelo de dirección URL de modo que se puedan pasar datos variables al controlador de solicitudes sin necesidad de una cadena de consulta.

Los modelos de dirección URL para las rutas en las aplicaciones de MVC suelen incluir los marcadores de posición {controller} y {action}.

Por ejemplo, una dirección URL que incluye la ruta de acceso /Products está asignada a un controlador denominado ProductsController. El valor del parámetro action es el nombre del método de acción que se invoca. Una dirección URL que incluye la ruta de acceso /Products/show daría lugar a una llamada al método Show de la clase ProductsController.

El Modelo de dirección URL predeterminado es el siguiente

{controller}/{action}/{id}



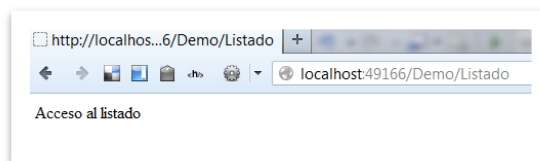
y una dirección que se corresponde con este enrutamiento sería:

`http://server/Catalogo/Listado/cursos`

- Controller: Catalogo
- Action: Listado
- Id: cursos

En caso de que el método no sea Index entonces debemos especificarlo.

```
public string Listado()
{
    return "Acceso al listado";
}
```



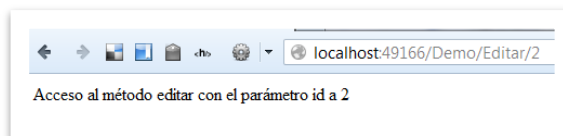
PARÁMETROS

Lo normal es que al menos en los métodos de edición, borrado o detalle enviemos un parámetro, que puede ser el ID del modelo con el que trabaje el controlador.

En ese caso lo más sencillo es usar el parámetro por defecto que nos ofrece el entorno que no es otro que id.

```
public string Editar(int id)
{
    return "Acceso al método editar con el parámetro id a " + id.ToString(CultureInfo.InvariantCulture);
}
```

Generamos la solución (no os olvidéis cada vez) y veremos el resultado



Lo primero que nos preguntamos es, ¿qué ha pasado con los parámetros por QueryString?. No hay que preocuparse sigue funcionando igual, y se puede reemplazar y acceder como hasta ahora.

Si ponemos la dirección `http://localhost:52314/UA/Editar/?id=2`, veremos que el resultado es el mismo.

Incluso si cambiamos la llamada al método y quitamos el parámetro y lo leemos desde dentro funciona igual.

```
public string Editar()
{
```




```
return "Acceso al método editar con el parámetro id a " + Request.QueryString["id"];
}
```

El resultado es el mismo

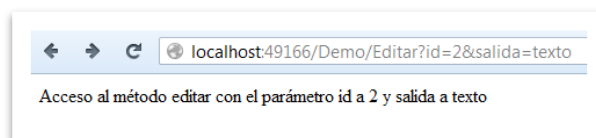
PASAR MÁS DE UN PARÁMETRO

En ocasiones necesitaremos pasar más de un parámetro al método por ejemplo el id y el tipo de salida.

El proceso es muy parecido a como se hacía antes pero se añade tantos parámetros al método como se necesiten.

```
public string Editar(int id, string salida)
{
    return "Acceso al método editar con el parámetro id a " + id.ToString(CultureInfo.InvariantCulture) +
        " y salida a " + salida;
}
```

Y lo llamaríamos con dos parámetros



Obtendríamos el mismo resultados con:

<http://localhost:49166/Demo/Editar/2?salida=texto>

<http://localhost:49166/Demo/Editar?id=2&salida=texto>

CONFIGURAR ENRUTAMIENTO EN MVC 3

Es posible crear nuestra propia implementación de controladores con clases que deriven de la clase ControllerBase. Este proceso es algo laborioso con lo que nos vamos a centrar el enrutamiento que se define el fichero global.asax.

A continuación se define el enrutamiento por defecto que genera la plantilla de proyecto de Visual Studio para las aplicaciones de MVC.

```
public class MvcApplication : System.Web.HttpApplication
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }

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

        routes.MapRoute(
            "Default", // Route name
            "{controller}/{action}/{id}", // URL with parameters
            new RouteValueDictionary()
        );
    }
}
```



```

        new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter defaults
    );
}

protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
}

```

En la sección MapRoute podemos ver que se define con el formato anteriormente comentado. En la línea siguiente define los valores por defecto.

Si quisiéramos añadir un tercer parámetro a la ruta que indicara el idioma de la aplicación, sería tan sencillo como añadir en la sección MapRoute el parámetro {language}. En la tercera sesión veremos como hacer uso del parámetro y poder crear aplicaciones multiidiomáticas.

```

public class MvcApplication : System.Web.HttpApplication
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }

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

        routes.MapRoute(
            "Default", // Route name
            "{language}/{controller}/{action}/{id}", // URL with parameters
            new { language = "es", controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter defaults
        );
    }

    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();

        RegisterGlobalFilters(GlobalFilters.Filters);
        RegisterRoutes(RouteTable.Routes);
    }
}

```

CONFIGURAR ENRUTAMIENTO EN MVC 4

El proceso es muy parecido pero en ficheros diferentes. En MVC 4 no se usa global.asax para estas funciones y se gestiona en el fichero RouteConfig.cs, dentro de la carpeta App_Start. En este fichero se crea la clase RouteConfig, en la cual registramos las rutas por defecto de la siguiente manera.

```

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

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

```



```
};  
}
```



VISTA

El siguiente paso después de crear el controlador es personalizarlo para visualizarlo. Esta fase se desarrolla con las vistas. Hoy veremos la parte más básica y en días posteriores veremos como personalizarlas.

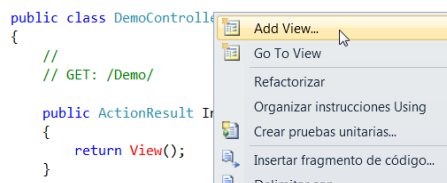
Si seguimos con el controlador demo que hemos usado anteriormente, lo primero que debemos hacer es volver a cambiar el tipo de devolución de datos de nuestros métodos de string a ActionResult. Luego cambiar el return “cadena de texto” por return View().

```
public string Index()
{
    return "Acceso al método por defecto";
}
```

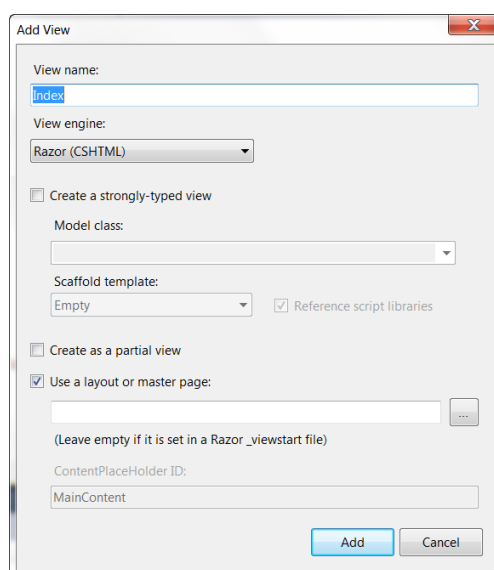
Pasa a

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

El que la palabra View aparezca en rojo nos indica que no hemos declarado la vista para ese método. Para hacerlo nos ponemos dentro del código del método (lo más cómodo es hacerlo sobre la declaración de éste) y pulsamos botón derecho y seleccionar Add View...



Aparece una ventana con muchas opciones, pero la mayoría desactivadas. Las opciones por defecto seleccionan el motor de trabajo Razor y que utilice layout o master page. La dejamos tal cual y pulsamos Add.



Nos genera un código muy básico

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
```

Que si lo visualizamos en el navegador obtendremos



Del código generado destacar tres cosas

1. La sintaxis `@{ .. }` es de Razor y nos permite ejecutar código C# o el lenguaje con el que se trabaje en el proyecto. Hoy y mañana lo veremos con detalle.
2. Aparece un objeto `ViewBag` que no hemos declarado, pero que tiene propiedades, en este caso `Title`. Este objeto se usa para pasar datos entre nuestro controlador y la vista.
3. Por defecto usa como master page el que esté definido por defecto (Views > Shared > `_Layout.cshtml`)

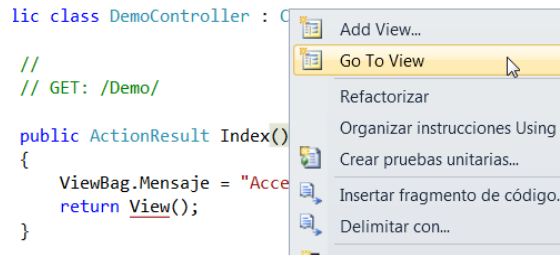
Por ejemplo si queremos mostrar el mensaje "Acceso al método por defecto" dentro de la vista y que sea el propio controlador quien se lo mande, deberíamos incluir una línea más en el controlador, asignando una nueva propiedad a `ViewBag` el texto que queramos. En este caso se los asigno a `Mensaje`, pero podría usarse cualquier otro nombre.

```
public ActionResult Index()
{
    ViewBag.Mensaje = "Acceso al método por defecto";
    return View();
}
```

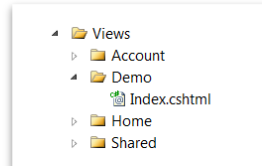
Ahora para visualizarlo volvemos a la vista. Hay varias maneras.

4. Pulsando e botón derecho dentro del código del método y en esta ocasión seleccionar `Go To View`





5. Accediendo a la capeta Views, desplegar Demo y visualizaremos Index.cshtml. Doble clic sobre el fichero y lo tendremos abierto para trabajar.



Ahora incorporamos código para visualizar el mensaje. Con Razor se hace con @NombreVariable o @NombreObjeto.Propiedad

```

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
@ViewBag.Mensaje

```

El resultado sería el siguiente



La primera impresión que tenemos al trabajar con Razor es que estamos volviendo al ASP clásico (VBScript) o que tenemos que aprender un nuevo lenguaje.

Con un ejemplo creo que quedará más claro. Con WebForms cuando queríamos escribir código teníamos dos formas en el ASPX o en el CS.

ASPX

```

<% if(User.Type == "admin") { %>
<span>Hola, <%= User.Username %></span>
<% } %>
<% else { %>
<span>Debes identificarte para poder acceder a esta sección</span>
<% } %>

```



CS

```

if(User.Type == "admin") {
    Response.Write("<span>Hola, " + User.Username + "</span>");
}
else {
    Response.Write("<span>Debes identificarte para poder acceder a esta sección</span>");
}

```

En Razor queda menos engorroso el código, ya que el HTML y la sintaxis propia, se integran perfectamente.

```

@if(User.Type == "admin") {
    <span>Hola, @User.Username</span>
} else {
    <span> Debes identificarte para poder acceder a esta sección </span>
}

```

RAZOR

En una sintaxis basada en C# (aunque se puede programar en Visual Basic) que permite usarse como motor de programación en las vistas o plantillas de nuestros controladores. Es una de las novedades de ASP.NET MVC 3.

No es el único motor para trabajar con ASP.NET MVC. Entre los motores disponibles destaco los más conocidos: Spark, NHaml, Brail, StringTemplate o NVelocity, algunos de ellos son conversiones de otros lenguajes de programación.

Con ASP.NET MVC 4 se ha incorporado nuevas funcionalidades a Razor que simplifican algunas tareas cotidianas. Lo veremos los últimos días.

También destacar que dispone IntelliSense dentro de Visual Studio con lo que agiliza enormemente programar dentro de las vistas.

CÓDIGO GENERAL (ASIGNACIÓN, CONDICIONES, ETC.)

```
@{ var miVariable = valor; }
```

```

@{
    var miVariable1 = valor1;
    var miVariable2 = valor2;
    ...
}

```

Cada línea debe llevar su ; al final de ésta.

Mi recomendación es utilizar var en vez del tipo de dato, siempre que inicialicemos la variable, independiente del tipo.

```

@{
    var mensaje = "";
}

```



```

if(User.Type == "admin") {
    mensaje = "Administrador del sistema";
}
else {
    mensaje = "Gestor";
}
}

```

MUESTRA DE VARIABLES

<p>El valor de la variable es @miVariable</p>

Todos las variables que mostremos con @ son parseadas con HTML Encode. Lo que hace es remplazar símbolos <, > o & por sus correspondientes códigos. De esta manera evitemos que nos introduzcan código mal intencionado que pueda alterar el comportamiento de nuestro programa.

Si lo que quiere mostrar está dentro de etiquetas HTML (por ejemplo span) se puede escribir

```

@if(true){
    <span>La hora es: @DateTime.Now</span>
}
else
{
    <span>Aquí no se debería acceder nunca</span>
}

```

En caso de que no hubiera etiquetas este código daría error

```

<span>
@if(true){
    La hora es: @DateTime.Now
}
else
{
    Aquí no se debería acceder nunca
}
</span>

```

Porque no sabe si es código HTML o código C#. Para definírselo anteponeamos @: (si sólo es una línea) para decirle que es HTML

```

<span>
@if(true){
    @:La hora es: @DateTime.Now
}
else
{
    @:Aquí no se debería acceder nunca
}
</span>

```

O etiquetando el bloque (o una sola línea) de código con <text>

```

<span>
@if(true){

```




```

    <text>La hora es: @DateTime.Now</text>
}
else
{
    <text>Aquí no se debería acceder nunca</text>
}
</span>

```

COMENTARIOS

Se usa la etiquetas @* y *@ con el texto dentro.

```

@* Una línea de comentario *@
@*
Más de una línea de comentario
Puede tener tantas como se necesiten
*@

```

Dentro de bloques se código también se permite // y /* */

```

@{
    @* Título de la página *@
    ViewBag.Title = "Index";
}

@{
    // Título de la página
    ViewBag.Title = "Index";
}

@{
    /* Título de la página
       que se visualiza en la parte superior del navegador */
    ViewBag.Title = "Index";
}

```

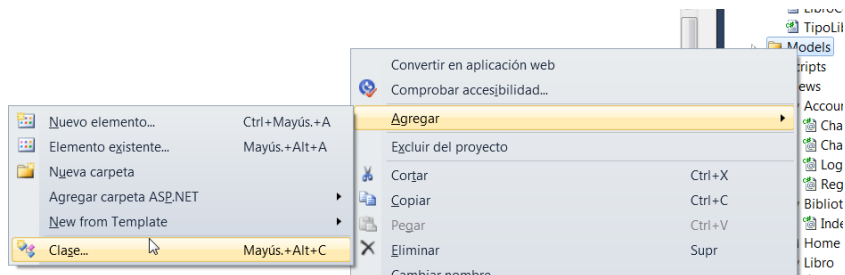


MODELO

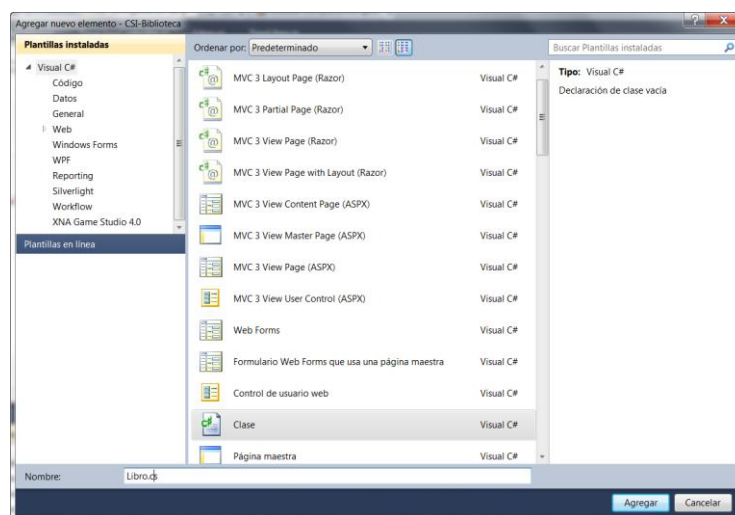
El proceso de traspaso de información entre el controlador y la vista se puede hacer interminable cuando queremos pasar mucha información.

Es el momento de crear un modelo y comenzar a trabar con los 3 pilares de MVC.

Para crear un modelo nos vamos a la carpeta Models, y con el botón derecho seleccionamos Agregar > Clase ...



En la siguiente ventana estará seleccionada la plantilla Clase. Le ponemos como nombre Libro.cs



El código que nos genera

```
namespace MVC_3.Models
{
    public class Libro
    {
    }
}
```

Es importante destacar el namespace porque para acceder a la clase Libro desde fuera (por ejemplo controlador) deberemos incluir el namespace con using.

Le añadimos propiedades. No es necesario crear unos privados y luego los públicos que hacen referencia a los privados. Se declaran públicos y con {get; set; } indicamos si se puede leer su valor (get) o modificar su valor (set). Lo normal es poner ambos.

```
public class Libro
{
    public string Isbn { get; set; }
```



```

    public string Titulo { get; set; }
    public string TipoLibro { get; set; }
}

```

USAR UN MODELO DENTRO DE UN CONTROLADOR / VISTA

Para llamarlo desde nuestro controlador Demo, deberemos hacer una llamada al namespace del modelo.

```
using MVC_3.Models;
```

A continuación en el método del controlador creamos un libro y se lo pasamos a la vista

```

public ActionResult Index()
{
    var libro = new Libro {Isbn = "1122", Titulo = "El principito", TipoLibro = "Novela"};
    return View(libro);
}

```

Por último dentro de la vista tenemos que indicar que tipo de datos vamos a recibir. En la primera línea añadimos

```
@model MVC_3.Models.Libro
```

Y donde lo queramos mostrar `Model.Propiedad`

El título del libro es `@Model.Titulo`

El resultado es



Otro caso que nos encontraremos habitualmente será el generar un listado de datos, en nuestro caso de libros.

El proceso es muy similar, por una parte el controlador deberá enviar el listado de libros

```

public ActionResult Index()
{
    var libros = new List<Libro>
    {
        new Libro {Isbn = "1122", Titulo = "El principito", TipoLibro = "Novela"},
        new Libro {Isbn = "1122", Titulo = "Steve Jobs", TipoLibro = "Biografía"}
    };

    return View(libros);
}

```



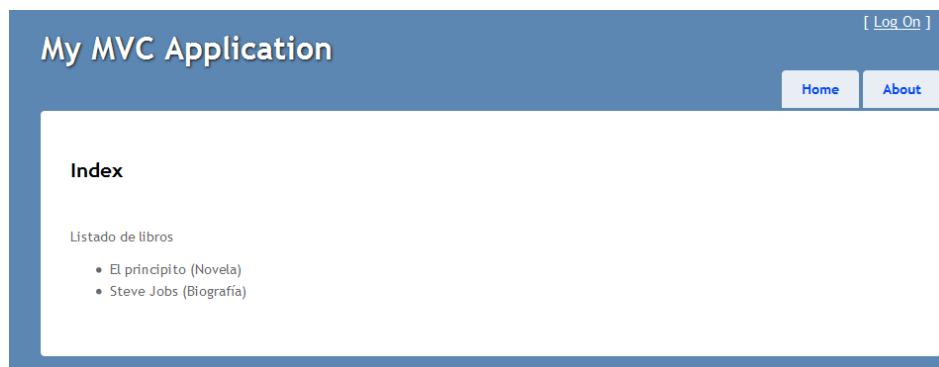
Y en la vista debemos indicar el tipo de datos del modelo, en este caso un listado de libros. Para ello en vez de usar `List<MVC_3.Models.Libro>` se debe remplazar por `<MVC_3.Models.Libro>`
`@model IEnumerable<MVC_3.Models.Libro>`

Para recorrer los datos con Razor, aunque lo veremos con detalle el 2º día, os dejo el código básico.

Listado de libros

```
<ul>
  @foreach (var libro in Model)
  {
    <li>@libro.Titulo (@libro.TipoLibro)</li>
  }
</ul>
```

El resultado sería:



Cuando tengamos que pasar un conjunto de datos a la vista podemos usar dos técnicas para crear un nuevo modelo que recoja todos los que se necesiten enviar (lo recomendado) o usar el objeto ViewBag. Voy a aclarar el segundo caso con un ejemplo porque el primer caso lo veremos varias veces en días posteriores.

```
public ActionResult Index()
{
    var libros = new List<Libro>
    {
        new Libro { Isbn = "1122", Titulo = "El principito", TipoLibro = "Novela"},
        new Libro { Isbn = "1122", Titulo = "Steve Jobs", TipoLibro = "Biografía"}
    };

    ViewBag.Libros = libros;

    return View();
}
```

La diferencia en la vista es que es recomendable especificar el tipo de datos de la propiedad Libros de ViewBag para que el asistente nos ayude cuando escribimos.

Listado de libros

```
<ul>
  @foreach (var libro in (IEnumerable<MVC_3.Models.Libro>)ViewBag.Libros)
  {
    <li>@libro.Titulo (@libro.TipoLibro)</li>
  }
</ul>
```



EJEMPLO PRÁCTICO

GESTIÓN DE LIBROS Y REVISTAS DEL SI

Para poner en práctica como funcionan los controladores vamos a desarrollar una sencilla web que gestionen los libros o revistas del Servicio de Informática.

No vamos a usar base de datos o para entender todos los conceptos. Cuando lo tengamos acabado, con unos pequeños cambios, lo tendremos en base de datos.

El proyecto lo llamaremos CSI-Biblioteca y usamos el modelo de libro que hemos utilizado anteriormente (se arrastra desde el explorador)

Crearemos un modelo llamado Biblioteca que incluya un listado de libros y que al crearse, añada unos libros por defecto (por tener unos datos por efecto)

```
public class Biblioteca
{
    public List<Libro> Libros { get; set; }

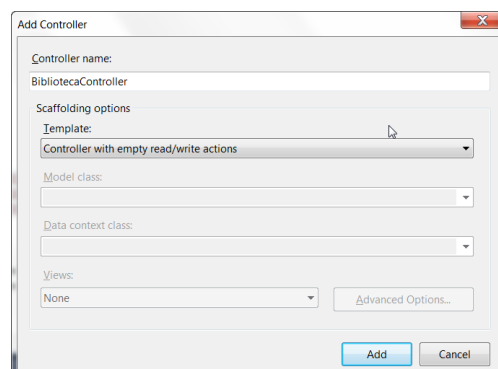
    public Biblioteca()
    {
        Libros = new List<Libro>
        {
            new Libro { Isbn = "11122", Titulo = "Los Piratas del Caribe", TipoLibro = "Novela"},
            new Libro { Isbn = "22211", Titulo = "Los Pilares de la tierra", TipoLibro = "Novela"},
            new Libro { Isbn = "33311", Titulo = "Steve Jobs", TipoLibro = "Biografía"}
        };
    }
}
```

Ahora sería el momento de crear el controlador con todos los métodos generales para poder dar de alta, editar o borrar libros de la biblioteca.

Aunque se puede hacer a mano como lo hemos hecho hasta ahora, ASP.NET MVC incluye muchos asistentes para facilitar tareas rutinarias.

Pulsamos botón derecho en la carpeta Controllers, y seleccionar Agregar > Controller.

En la ventana que habíamos visto antes, ponemos el nombre del nuevo controlador BibliotecaController y en la plantilla/template seleccionamos "Controller with empty read/write actions"



Veremos que nos aparecen muchos métodos vacíos, con la forma de llamarlos vía web como comentario. Los métodos se corresponden con todas las operaciones generales. Podemos añadir las que necesitemos o quitar las que no se vayan a usar.

```
//
// GET: /Biblioteca/

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

//
// GET: /Biblioteca/Details/5

public ActionResult Details(int id)
{
    return View();
}
```

Como nosotros queremos hacer uso del modelo Biblioteca que hemos definido anteriormente, antes de cualquier método del controllador debemos crear un objeto del tipo de esa clase.

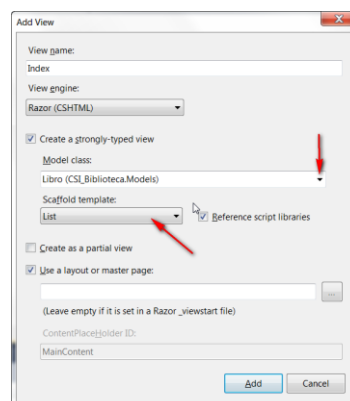
```
public class BibliotecaController : Controller
{
    Biblioteca miBiblioteca = new Biblioteca();
}
```

Una posible implementación del método Index podría ser

```
public ActionResult Index()
{
    return View(miBiblioteca.Libros.ToList());
}
```

Ahora para crear la vista, pulsamos dentro del método el botón derecho y seleccionamos la primera opción Add View...

Ahora vamos a probar la potencia de las vistas marcando la casilla “Create a strongly-typed view” y seleccionado en Model class la clase Libro (no confundir con Biblioteca porque lo que queremos listar son libros y es lo que pasamos a la vista) y en Scaffold template seleccionamos List.



En muchas ocasiones no os aparecerán los modelos en el mismo momento que se crean, deberemos pulsar F6 para que se compile el proyecto y todos los modelos sean visibles desde el resto de elementos del proyecto.

Se genera un código que permite la visualización, el alta, la edición y el borrado de libros.

```
@model IEnumerable<CSI_Biblioteca.Models.Libro>
```

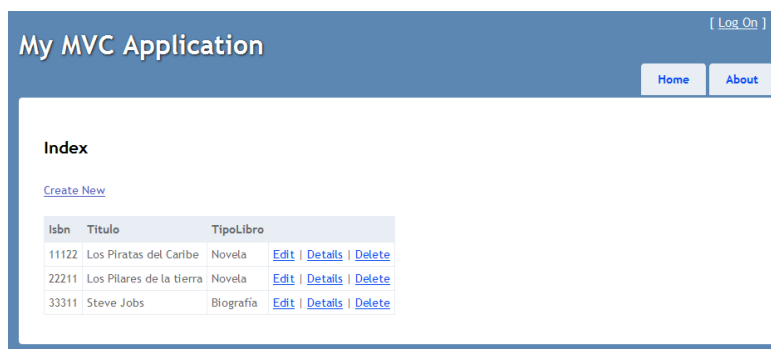
```
@{
    ViewBag.Title = "Index";
}
```

```
<h2>Index</h2>
```

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
```

```
<table>
<tr>
<th>
    Isbn
</th>
<th>
    Titulo
```

El resultado es



El resto de métodos se implementan de una forma similar.

Vamos a incorporar dos métodos adicionales al modelo Biblioteca que usaremos desde el controlador.

```
public int NumeroLibros()
{
    return Libros.Count();
}

public Libro ObtenerPorIsbn(string isbn)
{
    foreach (var libroBuscar in Libros)
    {
        if (libroBuscar.Isbn == isbn)
        {
            return libroBuscar;
        }
    }

    return null;
}
```

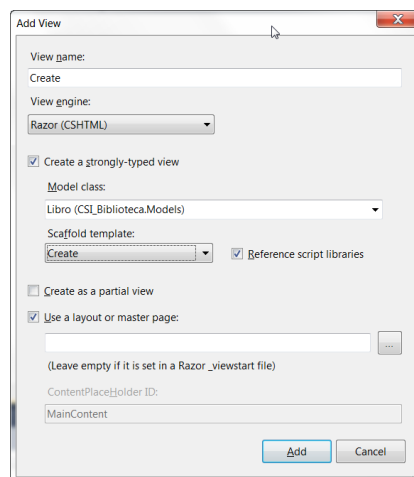


En el controlador vamos a cambiar la declaración del objeto porque sino los cambios que hagamos alta o baja no se refejarán (en cada llamada se redeclara el objeto). Añadimos static para que los datos se mantengan entre llamadas.

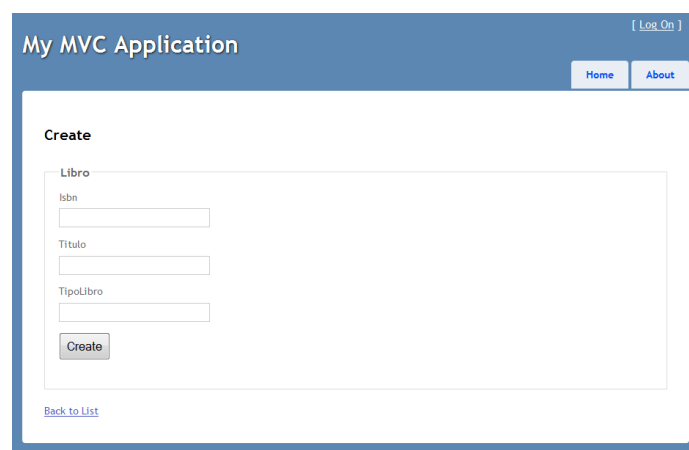
```
static Biblioteca miBiblioteca = new Biblioteca();
```

Detallamos la operación de creación. Nos vamos al método Create. Lo primero que nos damos cuenta es que hay dos. El primero es el que nos solicita los datos del libro y el segundo el que lo crea realmente (funciona con el método POST).

En el primer método Create pulsamos el botón derecho Add View... y seleccionamos Libro como Model class y Create como Scaffold template



Si lo visualizamos (pulsando en Create New del listado de libros), visualizaremos todos los cambios de un libro. No pulsar Create hasta que definamos el segundo método.



Al segundo método le añadimos el código para dar de alta un libro a la biblioteca

```
[HttpPost]
public ActionResult Create(FormCollection collection)
{
```




```

try
{
    miBiblioteca.Libros.Add(new Libro
    {
        Isbn = (miBiblioteca.Libros.Count() + 1).ToString(),
        Titulo = collection["Titulo"],
        TipoLibro = collection["Categoria"]
    });

    return RedirectToAction("Index");
}
catch
{
    return View();
}
}

```

En caso de que sea todo correcto le redirigimos al listado de libros (donde ya se verá el nuevo) y en caso de que falle volvemos a mostrar la vista de creación que hemos definido anteriormente.

Rellenamos el formulario

Pulsamos Create y vemos en el listado el nuevo libro.

Isbn	Titulo	TipoLibro	
11122	Los Piratas del Caribe	Novela	Edit Details Delete
22211	Los Pilares de la tierra	Novela	Edit Details Delete
33311	Steve Jobs	Biografia	Edit Details Delete
11223344	Mi primer libro		Edit Details Delete

Se deja como ejercicio implementar el resto de métodos.



INTEGRAR EN BASE DE DATOS CON ORACLEDB

<http://www.entrelectores.com/>

El siguiente paso va a ser trabajar con una base de datos Oracle. Hoy vamos a realizar todas las operaciones con ClaseOracleBD y mañana ya nos meteremos de fondo con las nuevas funcionalidades que ofrece MVC en este aspecto.

Disponemos de dos tablas que mañana crearemos de cero. Tenemos permiso INSERT, UPDATE y DELETE sobre ellas con lo que vamos a crear una gestión básica.

CSI_LIBRO x						
Columnas Datos Restricciones Permisos Estadísticas Disparadores Flashback Dependencias Detalles Particiones						
Acciones...						
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER	No	(null)	1 (null)	
2	ISBN	VARCHAR2 (50 BYTE)	No	(null)	2 (null)	
3	TITULO	VARCHAR2 (300 BYTE)	No	(null)	3 (null)	
4	TIPOLIBRO	NUMBER	Yes	(null)	4 (null)	

CSI_TIPOLIBRO x						
Columnas Datos Restricciones Permisos Estadísticas Disparadores Flashback Dependencias Detalles Particiones Índices SQL						
Acciones...						
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER	No	(null)	1 (null)	
2	DESCRIPCION	VARCHAR2 (250 BYTE)	No	(null)	2 (null)	

Desde el controlador llamaremos a los métodos de la clase Biblioteca para realizar las operaciones básicas de alta, baja y modificación. El código del controlador debe ser de lógica, validación y de llamadas a métodos de los modelos, nunca debemos escribir consultas o recorrer recordsets.

```
[HttpPost]
public ActionResult Create(FormCollection collection)
{
    try
    {
        bool correcto = biblioteca.AltaLibro(collection["Isbn"], collection["Titulo"],
        Int32.Parse(collection["TipoLibro"]));

        if (correcto)
            return RedirectToAction("Index");
        return View();
    }
    catch
    {
        return View();
    }
}
```

La clase que vamos a utilizar en el curso difiere un poco de la que estamos acostumbrados a utilizar. Se basa en Oracle.DataAccess.Client y no en System.Data.OracleClient.

```
using Oracle.DataAccess.Client;
```

El nombre de la clase pasa de ClaseOracleBD a ClaseOracleBd y los tipos de datos ahora se definen con OracleDbType. El resto es idéntico. Veamos un ejemplo para identificar los cambios.

```
public bool AltaLibro(string isbn, string titulo, int tipolibro)
{
```



```

        bool correcto = true;

        using (var bd = new ClaseOracleBd())
        {
            try
            {
                bd.CadenaConexion =
                ConfigurationManager.ConnectionStrings["PoolLibros"].ConnectionString;
                bd.TipoComando = CommandType.StoredProcedure;
                bd.TextoComando = "PKG_CSI.ALTA_LIBRO";

                bd.CrearParametro("pisbn", OracleDbType.Varchar2,
                ParameterDirection.Input, 50, isbn);
                bd.CrearParametro("ptitulo", OracleDbType.Varchar2,
                ParameterDirection.Input, 300, titulo);
                bd.CrearParametro("ptipolibro", OracleDbType.Int32,
                ParameterDirection.Input, 0, tipolibro);

                bd.Ejecuta();
            }
            catch
            {
                correcto = false;
            }
            finally
            {
                bd.Close();
            }
        }

        return correcto;
    }
}

```

Usaremos los mismos modelos del ejercicio anterior al que le añadiremos los métodos para interactuar con las tablas de la base de datos. No usaremos paquetes e interactuaremos directamente con las tablas.

Creamos las vistas para el método Index, Create (GET) y Create (POST) y el resultado es que cuando damos de alta un libro.

Create

Libro

Isbn
9788435021043

Título
Cita con Rama

TipoLibro
66

Ya se visualiza en el listado

Isbn	Título	TipoLibro	
11	AA2	66	Edit Details Delete
9788435021043	Cita con Rama	66	Edit Details Delete

Y se ha guardado en la base de datos



ID	ISBN	TITULO	TIPOLIBRO
67	11	AA2	66
82	9788435021043	Cita con Rama	66

Acabar el resto de operaciones.

