Tema 6

Desarrollo de aplicaciones Web MVC (I)

Objetivos

- Reconocer los diversos modelos de desarrollo de aplicaciones Web.
- Comprender el patrón arquitectónico Modelo-Vista-Controlador (MVC), así como los elementos que lo forman y las interacciones entre ellos.
- Identificar las ventajas que aporta un bajo nivel de acoplamiento entre interfaz y lógica.
- Identificar y comprender el funcionamiento de los componentes que actúan como modelo, vista y controlador en las aplicaciones basadas en ASP.NET MVC.

Introducción

Uno de los factores más importantes a considerar durante el diseño de una aplicación Web, y que influirá de manera decisiva en todas las fases posteriores de su desarrollo, es el modelo de arquitectura del software a aplicar. En los temas anteriores, se ha estudiado el desarrollo de aplicaciones Web basadas en un modelo de arquitectura de software de tres capas, mediante la tecnología de desarrollo ASP.NET Web Forms. En este tema se inicia el estudio del desarrollo de aplicaciones Web basadas en el modelo de arquitectura del software Modelo-Vista-Controlador (MVC), empleando la tecnología ASP.NET MVC. Aunque ASP.NET MVC se apoya en los mismos servicios de base que ASP.NET, su funcionamiento es totalmente distinto. La principal ventaja de la aplicación del patrón arquitectónico MVC es que proporciona un mayor nivel de desacoplamiento entre la interfaz y el resto de los elementos que forman una aplicación Web.

Índice

6.1 MODELOS DE DESARROLLO DE APLICACIONES WEB CON ASP.NET

La tecnología ASP.NET está concebida para crear aplicaciones Web interactivas que emplean tecnologías Web estándar, como HTML, CSS, etc., y secuencias de código lógico que se procesa en el servidor Web. El marco de desarrollo de ASP.NET *(ASP.NET Development framework)* admite tres modelos de desarrollo de aplicaciones Web bien diferenciados. La existencia de diversos modelos de desarrollo no implica ningún tipo de substitución entre ellos. Al contrario, se trata, más bien, de la coexistencia de diversos enfoques de implementación que emplean patrones arquitectónicos diferentes. Este planteamiento amplía las capacidades de desarrollo de aplicaciones Web, según la infraestructura de software que se pretenda desarrollar.

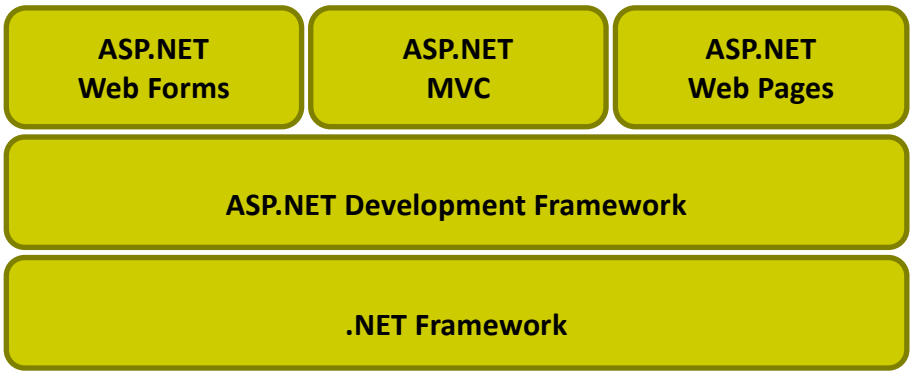

Ilustración 7.1. Modelos de desarrollo de ASP.NET

El hecho de disponer de tres modelos de desarrollo diferentes permite desarrollar tres tipos de aplicaciones Web diferentes:

- **Aplicación Web basada en ASP.NET Web Forms**. Este es el tipo de aplicación Web original de ASP.NET que sustituyó a la tecnología ASP. Se basa en el diseño de la interfaz Web a través de controles de servidor utilizando las herramientas RAD *(Rapid Application Development)* incorporadas en Visual Studio. La especificación del comportamiento de estos controles, y del Web Form, se realiza a través de código lógico asociado a eventos.

- **Aplicación Web basada en ASP.NET MVC**. Este tipo de aplicación Web utiliza la tecnología de desarrollo ASP.NET MVC que facilita el desarrollo de aplicaciones Web basadas en el patrón arquitectónico MVC (Modelo-Vista-Controlador). El desarrollo de este tipo de aplicación Web se basa en la existencia de un módulo, conocido como controlador, que genera la respuesta que corresponda a las acciones solicitadas por el usuario desde la interfaz. De este modo, la interfaz no es el centro de la aplicación Web y no está tan estrechamente relacionada con el resto de los elementos de la aplicación Web. El trabajo de desarrollo de este tipo de aplicaciones Web no utiliza herramientas de tipo RAD, sino que consiste, principalmente, en escribir código.

- **Aplicación Web basada en ASP.NET Web Pages**. Es el tipo de aplicación Web más simple. La aplicación Web está formada por páginas independientes en las que el contenido y el código lógico se combinan, utilizando para ello la sintaxis del lenguaje de vistas Razor.

6.2 EL PATRÓN MODELO-VISTA-CONTROLADOR (MVC)

Al desarrollar software es habitual utilizar patrones de software. Los patrones de software, o sencillamente patrones, son la base para la búsqueda de soluciones a problemas comunes en el ámbito del desarrollo de software. Se trata de esquemas lógicos que tienen un objetivo concreto aplicable a diversas situaciones de desarrollo de software, con independencia de: el tipo de proyecto de software, los lenguajes empleados y las herramientas de desarrollo que se utilicen. Los patrones de arquitectura de software, también conocidos como patrones arquitectónicos, son un tipo de patrones que no afectan a un componente de software concreto, sino que afectan a la estructura general de un proyecto de software. A este grupo de patrones pertenece el conocido como patrón arquitectónico MVC (Modelo-Vista-Controlador).

El patrón MVC fue introducido por **Trygve Reenskaug** en 1979. Este científico de ciencias de la computación y profesor de la Universidad de Oslo es autor del primer documento[1] en que se formula una separación de elementos de un sistema de software en: un modelo, un software controlador y una vista manipulable por el usuario para el diseño de una interfaz gráfica. Posteriormente, en 1983 se realizó la primera implementación MVC como patrón de arquitectura de software para el lenguaje de programación Smalltalk-80. Y posteriormente, el patrón MVC se expresó formalmente como concepción general. A partir de ese momento comenzó la extensión y popularidad de su uso como patrón arquitectónico aplicable independientemente del lenguaje de programación utilizado.

En la actualidad, algunos de los *Frameworks* MVC más utilizados pueden ser los siguientes:

***Framework* MVC**	**Lenguaje**	**Licencia**
Ruby on Rails	Ruby	MIT
Spring MVC	Java	Apache
Struts	Java	Apache
JSR	Java	Oracle
AngularJS	Javascript	MIT
Symfony	PHP	MIT
CakePHP	PHP	MIT
Cocoa	ObjectiveC	Apple
Django	Python	BSD
ASP.NET MVC	C#, VB, etc. (.NET Framework)	Microsoft

Nota de referencia al documento:

[1] http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

Arquitectura de software Modelo-Vista-Controlador

El patrón de arquitectura de software Modelo-Vista-Controlador (MVC) se basa en separar la lógica de una aplicación de la interfaz de usuario. Por un lado, se definen componentes para la representación de la información, y por otro los componentes para la interacción del usuario. Para ello, el patrón MVC propone que la arquitectura de una aplicación esté compuesta por tres partes bien diferenciadas que son: el Modelo, la Vista y el Controlador.

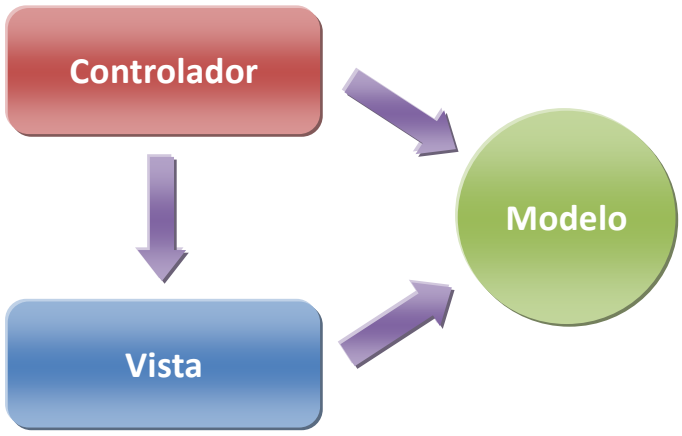

Ilustración 7.2. Patrón arquitectónico Modelo-Vista-Controlador

En el diagrama anterior, Ilustración 7.2, puede apreciarse los tres elementos fundamentales del patrón arquitectónico MVC, así como las conexiones directas entre ellos y la dirección en que se efectúa la comunicación. Esos elementos son los siguientes:

- **Modelo.** Representa el estado de la aplicación y proporciona los medios para consultar y modificar la información que define dicho estado. Así, el modelo es la representación de la información que maneja la aplicación en un momento dado y gestiona los accesos a dicha información: consultas, inserciones, eliminaciones y actualizaciones. El modelo puede estar constituido, por ejemplo, por una serie de clases que representan las entidades de una base de datos, junto con sus propiedades y los métodos para actuar sobre ellas.

- **Controlador.** Procesa la interacción del usuario. Así, se encarga de responder a las acciones solicitadas por el usuario, traduciéndolas en alteraciones sobre la vista o sobre el modelo. Se podría decir que hace de intermediario entre la vista y el modelo. Es habitual que exista un controlador por cada funcionalidad de la aplicación, determinando globalmente el comportamiento de la aplicación. Los controladores son los componentes que manejan la interacción con el usuario, actúan sobre el modelo y seleccionan la vista a desplegar. Pueden estar constituidos por una clase que incluye una serie de métodos.

- **Vista.** Genera interfaz de usuario a partir de la información obtenida del modelo. La vista cambiará a demanda del controlador, cuando una acción de usuario así lo requiera. Habitualmente, se utiliza un lenguaje o motor de vistas para generar las representaciones visuales del estado de la aplicación.

La conexión existente entre el controlador y el modelo se emplea, principalmente, para que el controlador actúe sobre el estado de la aplicación. El mantenimiento de la información que maneja la aplicación es responsabilidad del modelo, de manera que el modelo es el único responsable de establecer las conexiones sobre una base de datos y de ejecutar las consultas o modificaciones de datos cuando el controlador lo demande. Por otra parte, la vista usará la conexión con el modelo para obtener información desde el modelo, con la finalidad de componer la interfaz. Y finalmente, por otra parte, el controlador se comunica con la vista para establecer la interfaz de usuario.

Habitualmente, en una aplicación MVC, ni el controlador ni la vista tienen acceso directo al modelo, no conocen las clases concretas que lo implementan, sino que se comunican con él a través de una interfaz pública expuesta por el modelo. Igualmente, el controlador se comunica con la vista a través de una interfaz pública. Esto contribuye a que exista desacoplamiento entre los componentes de la aplicación.

Aunque originalmente el patrón arquitectónico MVC fue desarrollado para aplicaciones de escritorio, su uso ha sido ampliamente aceptado como arquitectura para diseñar e implementar aplicaciones Web interactivas. Se han desarrollado multitud de *Frameworks* Web que implementan este patrón y que han sido adaptados a los principales lenguajes de programación. Es necesario tener en cuenta que el patrón MVC es aplicable exclusivamente a sistemas software en los que se precisa interacción por parte del usuario, no teniendo sentido fuera de dicho contexto.

Características y ventajas del patrón de arquitectura MVC

El patrón arquitectónico MVC favorece el diseño de sistemas de software que permiten obtener un bajo nivel de acoplamiento y, al mismo tiempo, un alto nivel de cohesión entre los componentes de software que los forman. El **bajo nivel de acoplamiento** se refiere a que cada componente puede resolver su funcionalidad completamente, sin necesidad de recurrir a otros componentes. Así en este tipo de aplicaciones se reduce la dependencia entre los diferentes componentes del software. La **alta cohesión** entre los componentes se refiere a que cada componente ofrece una funcionalidad clara y específica sobre una estructura de datos concreta. En efecto, en este tipo de aplicaciones solo el modelo representa y gestiona el estado de la información que se maneja, solo la vista genera representaciones visuales de dicho estado sobre la interfaz y, solo el controlador gestiona la interacción del usuario y selecciona la vista a desplegar. Habitualmente, se considera que conseguir estas características simultáneamente suele ser uno de los objetivos principales de los procesos de ingeniería del software. Las características del patrón de arquitectura del software MVC consiguen obtener importantes ventajas, algunas de las cuales son las siguientes:

- Ayuda a manejar la complejidad del desarrollo de una aplicación Web, ya que permite centrarse en cada uno de los aspectos del desarrollo.

- Posibilita la división de un proyecto de desarrollo de software en partes, de forma que cada una de ellas puedan ir desarrollándose en paralelo por varios equipos de desarrolladores diferentes. Incluso cabe la posibilidad, si la complejidad y dimensión del software a construir lo requiere, de abordar el desarrollo de la vista, el controlador y el modelo como proyectos diferentes. En estos casos, lo único que precisan conocer los diferentes equipos de desarrollo son las interfaces públicas que hacen posible la comunicación entre los componentes.

- Evita la duplicación de código en distintas partes de la aplicación, proque cada componente tiene asignada una responsabilidad clara y concreta. Además, se facilita la reutilización de código en otros proyectos.

- Simplifica la realización de las pruebas del software, ya que es posible realizar pruebas completas sobre cada una de las partes. Así, es posible realizar pruebas sobre el modelo de manera independiente al controlador y la vista.

- Facilita el mantenimiento y los cambios futuros en la aplicación, al posibilitar la modificación y la sustitución de cualquier componente sin que esto afecte al resto de componentes de la aplicación. Así, por ejemplo, para actualizar la interfaz de usuario empleando nueva tecnología Web, solamente habría que trabajar sobre la vista, sin que ello afecte a las demás partes de la aplicación.

En definitiva, la utilización del patrón arquitectura del software MVC puede proporcionar ventajas que se traducen en un incremento de la productividad durante el proceso del desarrollo del software. En general, se pueden acortar los tiempos del desarrollo de las aplicaciones web, si se emplean tecnologías y herramientas de desarrollo que faciliten el trabajo de los desarrolladores.

La tecnología de desarrollo ASP.NET MVC. Dinámica de la interacción en una Aplicación Web

La tecnología de desarrollo ASP.NET MVC es un *framework* para el desarrollo de aplicaciones Web interactivas basadas en el patrón de arquitectura de software Modelo-Vista-Controlador. La construcción de aplicaciones Web basadas en ASP.NET MVC representa una alternativa al desarrollo clásico de ASP.NET basado en Web Forms. El planteamiento de Microsoft es que ambos modelos de desarrollo convivan, sin que ninguno de ellos sustituya al otro. La estructura de una aplicación Web de ASP.NET MVC se apoya en los mismos servicios que forman el núcleo de ASP.NET, sin embargo, el nivel de acoplamiento entre los componentes es bajo.

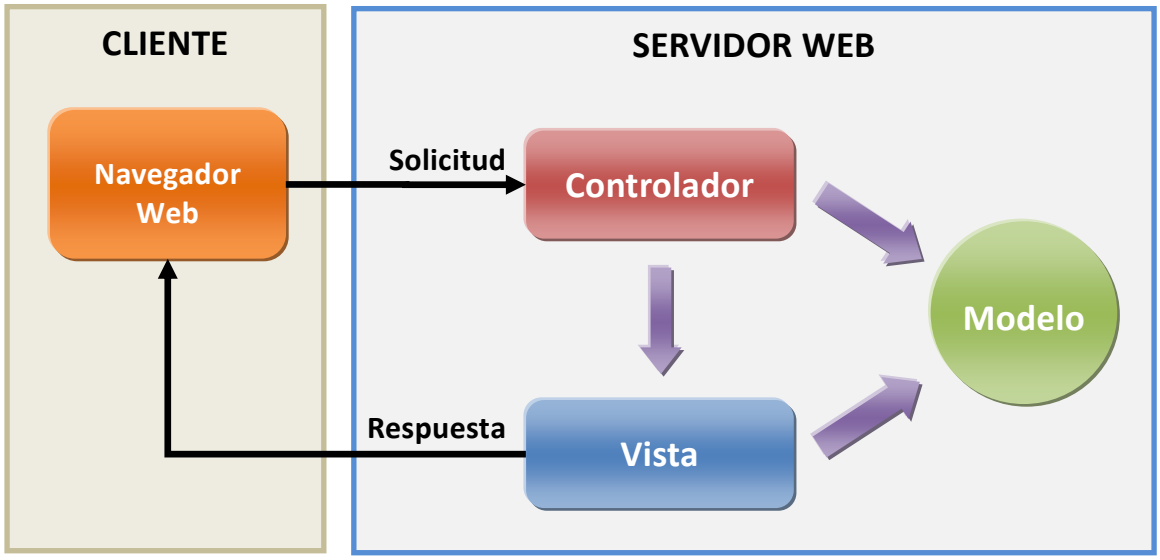

Ilustración 7.3. Dinámica de la interacción en una aplicación Web basada en ASP.NET MVC

La dinámica del proceso de interacción, que se muestra en la Ilustración 1.3, es el siguiente:

1. Una vez que la solicitud del cliente ha sido analizada se desencadena la ejecución de una acción en el controlador. El controlador puede recurrir al modelo para consultar o actualizar (añadir, eliminar o modificar) la información requerida.

2. A continuación, es responsabilidad también del controlador activar la vista que corresponda, en función de los parámetros asociados a la solicitud.

3. Finalmente, la vista activada por el controlador genera el contenido HTML que se envía como respuesta al cliente. Las vistas también pueden acceder al modelo para obtener información del estado para de componer la interfaz.

La solicitud y la respuesta, que se produce en la interacción con el usuario a través del protocolo HTTP, se representan mediante los objetos *HttpRequestBase* y *HttpResponseBase*. Estos objetos facilitan el acceso a la información precedente de la solicitud y a la información que se enviará como respuesta, respectivamente.

Diferencias entre las aplicaciones Web basadas en Web Forms y en ASP.NET MVC. Acoplamiento entre interfaz y lógica

Aunque utilizan los mismos servicios de base de ASP.NET, la concepción, la arquitectura y el funcionamiento las aplicaciones Web basadas ASP.NET MVC es completamente diferente al que tienen las aplicaciones Web basadas en Web Forms.

La diferencia principal entre ambos tipos de aplicaciones Web reside en el modelo de arquitectura del software que se emplea en cada caso, lo que influye de manera decisiva en todas las fases del desarrollo. Al desarrollar una aplicación Web basada en ASP.NET Web Forms, cada página de ASP.NET (.aspx) va forzosamente unida a un archivo de código subyacente (.aspx.cs). De manera que la definición de la interfaz de usuario se encuentra directamente asociada con el código lógico que gestiona los eventos. El resultado es la existencia de un alto nivel de acoplamiento entre la interfaz y la lógica de la aplicación. Por el contrario, las aplicaciones Web basadas en ASP.NET MVC permiten obtener un bajo nivel de acoplamiento entre los componentes que las forman, lo que facilita el diseño, la implementación, la realización de pruebas y el mantenimiento del software.

Otra diferencia que cabe destacar está relacionada con el procedimiento de desarrollo de ambos tipos de aplicaciones. El trabajo de desarrollo en las aplicaciones Web basadas en Web Forms utiliza herramientas de tipo RAD y se escribe el código lógico en los archivos de código subyacente. El trabajo de desarrollo en las aplicaciones Web basadas en ASP.NET MVC consiste, principalmente, en escribir código asociado al modelo, a los controladores y a las vistas.

Finalmente, otra diferencia significativa es que en las aplicaciones Web basadas en ASP.NET MVC no existe el mecanismo de mantenimiento del estado de la página, ya que no existe una relación directa entre la interfaz y la lógica. Pueden utilizarse variables de sesión y otros mecanismos específicos para compartir datos entre páginas.

6.3 APLICACIONES WEB BASADAS EN ASP.NET MVC

La tecnología de desarrollo ASP.NET MVC es una capa de software que facilita el desarrollo de aplicaciones Web utilizando el patrón arquitectónico Modelo-Vista-Controlador. En este apartado se aborda el estudio de la estructura y los principales componentes de una aplicación Web basada en ASP.NET MVC.

El entorno de desarrollo Visual Studio utiliza plantillas de proyecto predeterminadas para crear las aplicaciones Web basadas en ASP.NET MVC. Al crear un nuevo proyecto de ASP.NET MVC, un asistente genera un prototipo de aplicación Web que incluye: diversos controladores y vistas predeterminados, el menú de la aplicación, la página de inicio, el sistema de registro para la autentificación de usuarios, etc. Por razones de productividad, cualquier aplicación Web basada en ASP.NET MVC se basará en una de estas plantillas de proyecto, por lo que es necesario conocer su estructura y el funcionamiento de sus principales componentes.

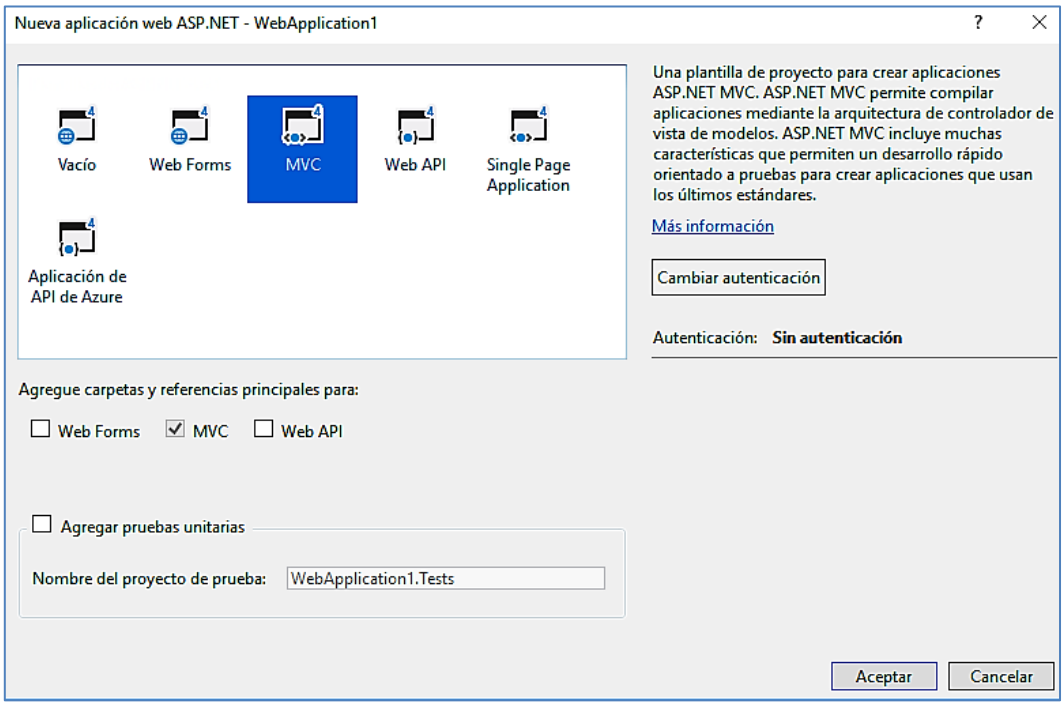

Ilustración 7.4. Plantillas para aplicaciones Web basada en ASP.NET MVC 5 con Visual Studio

Las plantillas de proyecto para crear aplicaciones Web interactivas basadas en ASP.NET MVC están disponibles solo en el caso de crear un nuevo Proyecto de aplicación Web.

Estructura de una aplicación Web basada en ASP.NET MVC

Al crear un nuevo Proyecto de ASP.NET MVC el asistente generará una aplicación Web predeterminada según la plantilla de diseño seleccionada. Una vez finalizado el trabajo del asistente, en el Explorador de soluciones de Visual Studio pueden observarse la estructura básica de carpetas y archivos que forman el Proyecto de aplicación Web de ASP.NET MVC.

La Ilustración 7.5 muestra la estructura del proyecto de ASP.NET MVC recién creado usando la plantilla de proyecto de Aplicación de ASP.NET (.NET Framework). Esta plantilla de proyecto incorpora, de forma predeterminada, los componentes básicos de un proyecto Web de ASP.NET MVC. Además, dependiendo del tipo de autentificación seleccionada, también pueden incluirse un modelo, unos controladores y varias vistas que se encargan de implementar la seguridad de acceso a la aplicación Web, mediante la autentificación de cuentas de usuario.

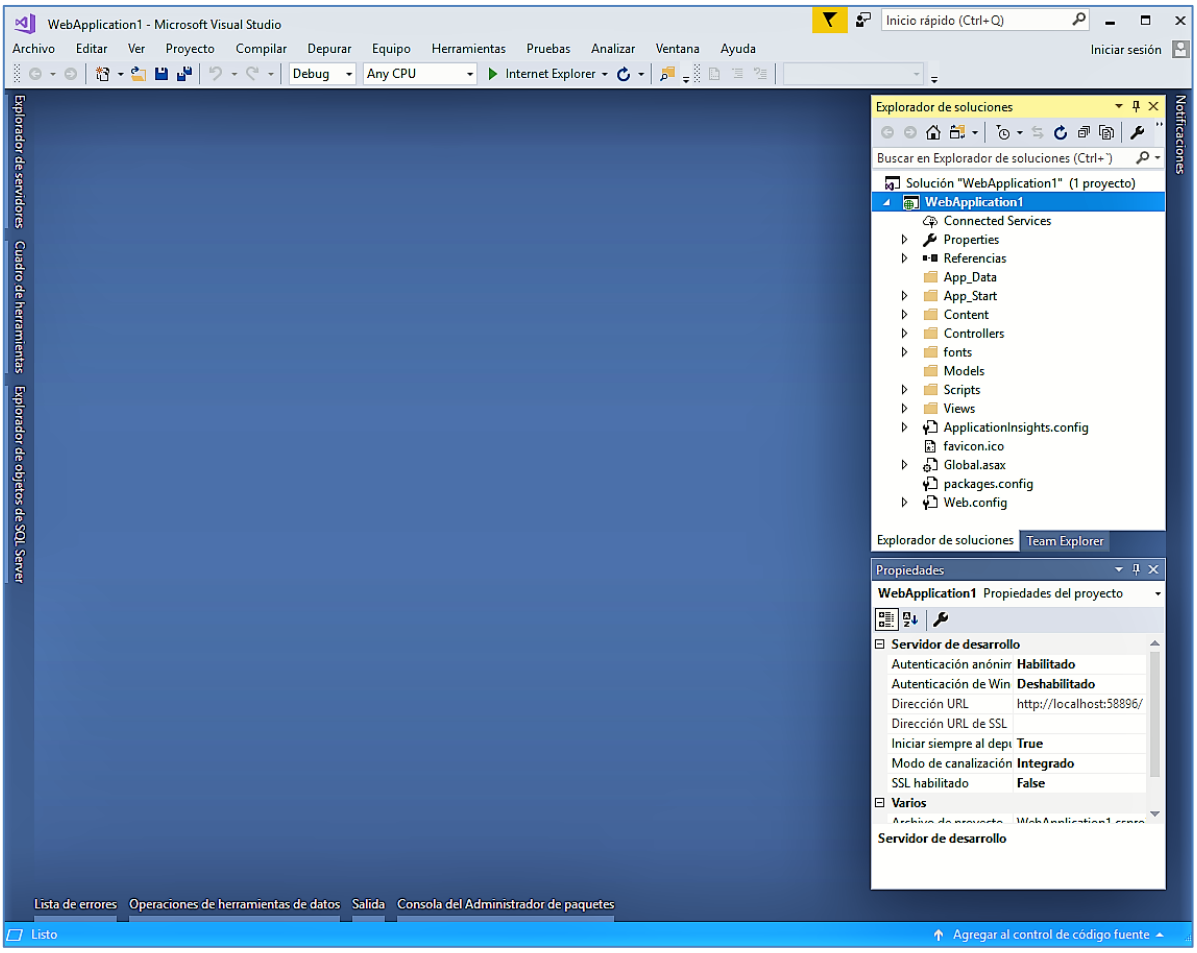

Ilustración 7.5. Estructura del Proyecto de ASP.NET usando la plantilla Aplicación de Internet

Los nombres de las carpetas predeterminadas que forman un proyecto de ASP.NET MVC, así como los nombres de los archivos que se almacenan en estas carpetas, no son arbitrarios. Al contrario, el funcionamiento de ASP.NET MVC se basa en esa nomenclatura o convención de nombres para localizar a cada componente de la aplicación Web. La funcionalidad de cada una de las carpetas de una aplicación Web basada en ASP.NET MVC es la siguiente:

- **App_Data**. Contiene los archivos de bases de datos que maneja la aplicación Web.

- **App_Start**. Contiene archivos de código que se ejecutan al inicializar la aplicación Web y que permiten establecer diversos aspectos sobre su configuración.

- **Content**. Incluye el contenido estático y visual de la aplicación Web, especialmente: archivos de hojas de estilo CSS, imágenes y demás recursos compartidos.

- **Controllers**. Contiene los controladores destinados a procesar las acciones. Cada controlador está representado por una clase que incorpora varios métodos. Cada uno de estos métodos especifica las acciones a ejecutar cuando son invocados, por este motivo reciben el nombre de **métodos de acción**. Lo más habitual es que los controladores devuelvan la ejecución de una vista, sobre la que se inyecta información proveniente del modelo, aunque también pueden devolver otras acciones. La denominación estándar de los controladores finaliza con el sufijo Controller, por ejemplo: *HomeController.cs*.

- **fonts**. Contiene los archivos de las fuentes personalizadas para la aplicación Web.

- **Models**. Incluye las clases que representan el modelo de la aplicación. Estas clases definen y representan la información con la que trabaja la aplicación Web.

- **Scripts**. Es la carpeta donde se ubican los archivos de Javascript (*.js). El código javascript se ejecuta en el entorno del cliente. Habitualmente, esta carpeta suele incluir diversas librerías javascript, como pueden ser: jQuery, Bootstrap, Respond, etc.

- **Views**. Contiene las vistas. Las vistas son archivos (.cshtml) donde se especifica código HTML estático entremezclado con áreas de código que son ejecutadas en el servidor y que están escritas usando un lenguaje de vistas, que habitualmente suele ser Razor. De manera que **una vista genera dinámicamente una página de respuesta en HTML**. Las vistas se agrupan en subcarpetas denominadas zonas, las cuales se corresponden con los nombres de los controladores, pero sin la terminación Controller. A su vez, en cada zona de vistas se define un archivo de vista por cada método de acción del controlador que devuelve la ejecución de esa vista. El nombre del archivo de la vista y del método de acción correspondiente debe coincidir, de modo que la ejecución de ese método de acción devolverá la ejecución de la vista correspondiente. Hay que tener presente que no siempre los métodos de acción devuelven la ejecución de una vista, también pueden realizar otras acciones.

    - **Views/Shared**. Dentro de la carpeta Views aparece la subcarpeta **Shared** que no corresponde a ningún controlador. Esta subcarpeta contiene las plantillas de diseño comunes a las vistas, así como las vistas parciales que van a ser reutilizadas por otras vistas. Además, esta subcarpeta puede alojar vistas compartidas por distintos controladores, lo que evita tener que duplicar vistas en distintas zonas de la carpeta /Views cuando controladores distintos activan la misma vista.

Además de las carpetas anteriores, la estructura predeterminada de un proyecto de ASP.NET MVC incluye los archivos **Web.config** y **Global.asax** que especifican diversos aspectos sobre la configuración de la aplicación Web basada en ASP.NET MVC. Además, esta estructura predeterminada de un proyecto de ASP.NET MVC, también incluye los elementos **Properties**, **packages.config** y **Referencias** que establecen las referencias a los paquetes de software necesarios para que el proyecto funcione correctamente.

Configuración del enrutamiento y de la página de inicio de la aplicación Web

La **ejecución de un método de acción de un controlador se desencadena cuando el servidor Web recibe una solicitud URL**. La tecnología ASP.NET MVC usa un formato URL predeterminado para especificar el método de acción cuyo código se va a invocar. Este formato define la lógica del enrutamiento de las direcciones URL entrantes o solicitudes. El formato predeterminado del enrutamiento MVC o mapeo MVC es el siguiente:

/[NombreControlador]/[NombreMétodoDeAcción]/[Parámetros]

Por ejemplo, al solicitar la URL http://localhost:xxxx/HelloWorld/Bienvenida se establece: que el controlador a utilizar es **HelloWorld** que se encuentra implementado en la clase **HelloWorldController.cs** de la carpeta **/Controllers**; que el método de acción del controlador HelloWorld a ejecutar es **Bienvenida()**; y que si el método Bienvenida() devolviera como resultado la ejecución de una vista, que es el caso más habitual, se ejecutaría la vista **Bienvenida.cshtml,** que estará alojada en la subcarpeta o zona **/Views/HelloWord**, generándose la página de respuesta HTML resultante.

La configuración del formato de enrutamiento URL se establece al inicio de la aplicación, cuando se ejecuta el código definido en el archivo RouteConfig.cs de la carpeta /App_Start. En el código de este archivo puede apreciarse cómo se utiliza el método MapRoute() del objeto routes para definir la convención de nombres de la política de enrutamiento a través del parámetro *url*,

```
namespace MvcApplication1
{
    public class RouteConfig
    {
        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 }
            );
        }
    }
}
```

Además, el archivo RouteConfig.cs también define la ruta URL por defecto que se desencadenará cuando un cliente accede a la aplicación Web. Para ello, se especifica la página de inicio o ruta raíz de la aplicación Web mediante el parámetro *defaults*. El código anterior especifica que la ruta URL predeterminada conduce a la acción *Index* del controlador *Home*. De manera que, por ejemplo, al solicitar la URL http://localhost:xxxx/ se producirá el mismo efecto que al solicitar la URL http://localhost:xxxx/Home/Index. Además, debe tenerse en cuenta que el enrutamiento predeterminado de la acción Index afecta a todos los controladores. Así, al solicitar la dirección URL http://localhost:xxxx/HelloWorld/Index se producirá el mismo efecto que al solicitar la dirección URL http://localhost:xxxx/HelloWorld, en ambos casos se procesará la acción Index.

En las aplicaciones Web basadas en ASP.NET MVC la mayor parte del trabajo de desarrollo consiste en escribir código asociado al modelo, el controlador y la vista. En los siguientes apartados se estudia con detalle cada uno de sus elementos principales: el modelo, el controlador y la vista.

6.4 EL MODELO

El modelo es una representación en objetos de la información que maneja la aplicación Web. En las aplicaciones Web basadas en ASP.NET MVC se utiliza ***Entity Framework*** para vincular el contenido de la base de datos con las entidades que representan la información en la aplicación Web, a través de un modelo conceptual de datos de alto nivel. Los componentes que definen el modelo son archivos de clase que se almacenan en la carpeta /Models, de manera que cada clase representa una parte del modelo.

```
using System;
using System.Data.Entity;

namespace MvcPeliculas.Models
{
    public class Pelicula
    {
        public int ID { get; set; }
        public string Titulo { get; set; }
        public string Director { get; set; }
        public DateTime FechaLanzamiento { get; set; }
        public string Genero { get; set; }
        public decimal Precio { get; set; }
    }

    public class PeliculaDBContext : DbContext
    {
        public DbSet<Pelicula> Peliculas { get; set; }
    }

}
```

El código de ejemplo anterior, definido en el archivo de clase Pelicula.cs de la carpeta /Models, especifica solo una clase de datos, denominada **Pelicula**. Las **clases de datos**, o clases de modelo de datos, son las clases del modelo que actúan como entidades de datos. Así, la clase Película representa la información sobre las películas que almacenará una base de datos. Cada clase de datos expone sus propiedades, aunque también puede exponer métodos y atributos. De modo que cada instancia de un objeto de una clase de datos corresponde con una fila de una tabla de la base de datos. Y, además, cada propiedad de una clase de datos se asigna a una columna de la fila correspondiente.

La **clase del contexto de datos** es la clase del modelo que se encarga de buscar, almacenar y actualizar las instancias de las clases de datos en la base de datos que maneja la aplicación. En el código anterior, la clase **PeliculaDbContext** representa la clase del contexto de datos. También puede apreciarse que se trata de una clase derivada de la clase base **DbContext**, proporcionada por Entity Framework. Para poder referenciar las clases DbContext y DbSet debe agregarse el espacio de nombres **System.Data.Entity**.

Debe tenerse siempre presente que para poder asociar el contexto de Entity Framework con una base de datos externa, **debe coincidir el nombre de la clase del contexto de datos**, en este ejemplo PeliculaDBContext, **con el nombre de la cadena de conexión que referencia a la base de datos** y que está especificado en el archivo de configuración *Web.Config*.

```
...
<connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=(localdb)\MSSQLLocalDB;
       Initial Catalog=aspnet-MvcPeliculas-20150729184111; Integrated Security=SSPI;
       AttachDBFilename=|DataDirectory|\aspnet-MvcPeliculas-20150729184111.mdf"
       providerName="System.Data.SqlClient" />
    <add name="PeliculaDBContext" connectionString="Data Source=(localdb)\MSSQLLocalDB;
       AttachDbFilename=|DataDirectory|\Peliculas.mdf; Integrated Security=True"
       providerName="System.Data.SqlClient"/>
</connectionStrings>
...
```

Validación de datos en aplicaciones Web basadas en ASP.NET MVC

El proceso de desarrollo de la validación de la información introducida por los usuarios en las aplicaciones Web basadas en ASP.NET MVC es muy distinto al empleado en las aplicaciones Web basadas en Web Forms. Al desarrollar una aplicación Web basada en ASP.NET MVC no existe un diseñador visual de tipo RAD, ni controles de validación especializados que puedan agregarse mediante el procedimiento de arrastrar y soltar.

En una aplicación Web basada en ASP.NET MVC, las condiciones de validación de los datos que los usuarios introducirán en un formulario están vinculadas siempre a un modelo. Es, por tanto, en el modelo donde se definen las reglas de validación, empleando para ello un esquema declarativo basado en atributos que permite establecer las anotaciones de validación en el código. Estas anotaciones definidas en el espacio de nombres System.ComponentModel.DataAnnotations. En la siguiente tabla se muestran los atributos de validación más usuales.

Atributo	Descripción de la regla de validación
Compare	Proporciona un atributo que compara dos propiedades
CreditCard	Valida un formato de un número de tarjeta de crédito
DataType	Especifica el nombre de un tipo adicional asociado con un campo de datos
Display	Permite especificar la cadena de texto de los campos asociados a la propiedad
Editable	Indica si un campo de datos es editable
EmailAddress	Valida un formato de una dirección de correo electrónico
MaxLength	Especifica la longitud máxima del campo de datos
MinLength	Especifica la longitud mínima del campo de datos
Phone	Valida un formato de un número de teléfono
Range	Especifica las limitaciones numéricas de rango para el valor de un campo

Atributo	Descripción de la regla de validación
RegularExpression	El valor del campo debe coincidir con la expresión regular establecida
Required	Especifica que se requiere un valor en el campo de datos
StringLength	Establece la longitud máxima del campo de datos y establece esta restricción en la definición del campo de la tabla en la base de datos
UrlAttribute	Valida un formato de una dirección URL

Para especificar la validación en código del modelo, se escriben las anotaciones de validación que afectan a cada una de las propiedades de la clase de datos correspondiente. Las anotaciones que se escriben inmediatamente antes de la declaración de una propiedad afectan a esa propiedad. Tras introducir las anotaciones en el modelo, las condiciones de validación declaradas para una propiedad se aplican inmediatamente a todos los campos de los formularios asociados al valor de la propiedad correspondiente. El alcance de las anotaciones de validación es el proyecto de ASP.NET MVC.

```
namespace MvcPeliculas.Models
{
    public class Pelicula
    {
        public int ID { get; set; }

        [Display(Name = "Título")]
        [Required(ErrorMessage = "El campo Título es requerido")]
        [StringLength(60, MinimumLength = 3)]
        public string Titulo { get; set; }

        public string Director { get; set; }

        [Display(Name = "Fecha de Lanzamiento")]
        [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}", ApplyFormatInEditMode = true)]
        public DateTime FechaLanzamiento { get; set; }

        [Display(Name = "Género")]
        [Required(ErrorMessage = "El campo Género es requerido")]
        public string Genero { get; set; }

        [Range(1, 5000, ErrorMessage = "Precio debe estar entre 1 y 5.000 €")]
        [DisplayFormat(DataFormatString = "{0:n2}", ApplyFormatInEditMode = true)]
        public decimal Precio { get; set; }
    }

    public class PeliculaDBContext : DbContext
    {
        public DbSet<Pelicula> Peliculas { get; set; }
    }
}
```

En el código anterior se aprecia la declaración de anotaciones de validación. El mecanismo interno de validación en las aplicaciones Web basadas en ASP.NET MVC se ejecuta tanto en el cliente como en el servidor. La parte de cliente utiliza el método no intrusivo basado en *jQuery*, mientras que la parte de servidor utiliza el mecanismo de validación específico del *Framework* ASP.NET MVC.

6.5 EL CONTROLADOR

Se encarga de responder a las acciones del usuario. Un controlador es una clase que deriva de la clase base Controller. Las clases que actúan como controlador se alojan en la carpeta /Controllers y han de utilizar necesariamente el sufijo Controller en su nombre. Una clase controlador se activa mediante una solicitud URL, siguiendo un formato de enrutamiento MVC concreto, tal como se ha explicado anteriormente.

La característica esencial de un controlador consiste en proporcionar una o más acciones. Las acciones son los métodos públicos definidos en una clase que actúa como controlador. Estos métodos también suelen denominarse métodos de acción. En el código siguiente, se define la clase HelloWorldController que expone los métodos de acción Index() y Bienvenida().

```
using System.Web.Mvc;

namespace MvcPeliculas.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HellowWorld/
        public ActionResult Index()
        {
            ViewBag.Mensaje = "Index";

            return View();
        }

        //
        // GET: /HelloWorld/Welcome/
        public ActionResult Bienvenida()
        {
            ViewBag.Mensaje = "Bienvenida";

            return View();
        }
    }
}
```

El resultado más frecuente de un método de acción consiste en devolver la ejecución de una vista. Para ello, el método de acción devuelve una llamada al método View() que define la vista a activar. En el código anterior, puede apreciarse cómo cada método de acción activará la vista que le corresponda, puesto que el nombre de la vista a activar debe coincidir con el nombre del método de acción del controlador correspondiente. Los archivos de las vistas a activar por cada acción se encontrarán alojadas en la carpeta /Views.

Es habitual necesitar transferir datos desde el controlador hacia la vista. Para ello, puede recurrirse a: la propiedad dinámica ViewBag que ofrece propiedades en el controlador que pueden ser recuperadas en la vista activada por este, el objeto ViewData que funciona como una colección de pares clave-valor que almacenan los datos a transferir y la propiedad Model que pasa a la vista un objeto cuya clase está definida en el modelo de la aplicación.

En el código anterior puede apreciarse el uso de la propiedad Mensaje de ViewBag para poder transferir textos de mensaje que se definen en el controlador y que se podrán mostrar en la vista correspondiente.

El resultado de una acción: la clase ActionResult

Los métodos de acción se caracterizan por implementarse como funciones que devuelven un valor de tipo ActionResult. De esta manera, la clase ActionResult encapsula el resultado de un método de acción. El resultado más habitual de acción consiste en devolver la ejecución de la vista correspondiente. En estos casos, se emplea una llamada al método View() de la clase ActionResult para establecer la vista activa. Es decir, para devolver la ejecución de la vista que activa esa acción, de acuerdo con la convención de nombres establecida.

Sin embargo, los métodos de acción pueden devolver otros resultados de acción diferentes, dependiendo de la respuesta que se desee dar. En estos casos, pueden utilizarse otros métodos de la clase ActionResult para generar resultados de acción diferentes. La siguiente tabla muestra los resultados de acción de ASP.NET MVC y los métodos de acción que los devuelven.

Resultado de la acción	Método de acción	Descripción
ViewResult	**View()**	Representa una vista como una página web
PartialViewResult	**PartialView()**	Representa una vista parcial, que define una sección de una vista que se puede representar dentro de otra vista
RedirectResult	**Redirect()**	Redirecciona a otro método de acción utilizando su dirección URL
RedirectToRouteResult	**RedirectToAction()** **RedirectToRoute()**	Redirecciona a otro método de acción
ContentResult	**Content()**	Devuelve un tipo de contenido definido por el usuario
JsonResult	**Json()**	Devuelve un objeto JSON serializado
JavaScriptResult	**JavaScript()**	Devuelve un script que se puede ejecutar en el cliente
FileResult	**File()**	Devuelve la salida binaria para escribir en la respuesta
EmptyResult	**(Ninguno)**	Representa un valor que se utiliza si el método de acción debe devolver un resultado *null*

La clase ActionResult es la base de todos los resultados de acciones de ASP.NET MVC. Los métodos de acción pueden devolver un objeto de cualquier tipo de datos: cadena, entero, booleano, etc. En estos casos, se realiza una conversión de tipos automática a un tipo ActionResult antes de representarse en la secuencia de respuesta.

Acciones GET y POST

En los procesamientos que incluyen formularios para que el usuario pueda introducir datos, suelen definirse dos métodos de acción con el mismo nombre en el mismo controlador. En estos casos, el segundo método de acción estará precedido por el atributo **[HttpPost]**. Este atributo especifica que esta sobrecarga del método se invoca únicamente para las peticiones POST. El primer método se invoca para las peticiones GET y se le aplica el atributo **[HttpGet]**, aunque suele quedar implícito al ser este el valor predeterminado. Las acciones **GET** se relacionan con la obtención de información desde el servidor hacia el cliente, por ejemplo, cuando se solicita información sobre un determinado cliente que está almacenada en una base de datos o cuando se solicita una URL. Las acciones **POST**, sin embargo, se relacionan con el envío de información desde el cliente para que sea procesada en el servidor, por ejemplo, cuando se solicita la actualización de los datos de un cliente después de su edición. Se ha de tener en cuenta que ambas acciones requieren de una solicitud *(Request)* y de una respuesta *(Response)*.

En el siguiente código puede observarse que se ha aplicado el atributo [HttpPost] en el método de acción Edit() que aparece escrito en segundo lugar, lo que significa se invocará únicamente en las peticiones POST. Mientras que el primer método Edit() se invocará en las peticiones GET.

```
...
        // GET: /Peliculas/Edit/5
        public ActionResult Edit(int id = 0)
        {
            Pelicula pelicula = db.Peliculas.Find(id);
            if (pelicula == null)
            {
                return HttpNotFound();
            }
            return View(pelicula);
        }

        // POST: /Peliculas/Edit/5
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Edit(Pelicula pelicula)
        {
            if (ModelState.IsValid)
            {
                db.Entry(pelicula).State = EntityState.Modified;
                db.SaveChanges();
                return RedirectToAction("Index");
            }
            return View(pelicula);
        }
...
```

Filtros de acción *(Action Filter)*

Los atributos [HttpPost] y [HttpGet], estudiados en el punto anterior, son filtros de acción. Un filtro de acción es un atributo que puede asociarse a una acción de un controlador, o al controlador en su totalidad, para modificar la forma en la que se ejecutan las acciones. La tabla siguiente incluye los atributos de ASP.NET MVC más utilizados para definir filtros de acción:

Atributos	Descripción
Authorize	Restringe una acción para los roles o usuarios especificados
HandleError	Permite especificar una vista que se mostrará en caso de excepción no controlada
HttpGet	Filtra la invocación del método de acción solo para peticiones GET
HttpPost	Filtra la invocación del método de acción solo para peticiones POST
OutputCache	Almacena en caché la salida de un controlador
ValidateAntiForgeryToken	Facilita un mecanismo de seguridad de accesos indebidos
ValidateInput	Desactiva las validaciones de solicitud

Además de los filtros de acción predeterminados de ASP.NET MVC, pueden crearse filtros de acción personalizados para cubrir necesidades específicas.

6.6 LA VISTA

La ejecución de una vista genera dinámicamente en el servidor una página de respuesta HTML que se envía al cliente. Los archivos de vista permiten encapsular el procesamiento que genera las respuestas HTML hacia un cliente. Los nombres de los archivos de vista tienen una extensión de nombre .cshtml. Para crear los archivos de vista se suele emplear el **lenguaje de vistas Razor** que facilita la creación de documentos HTML utilizando el lenguaje de programación C#. El proceso de creación de nuevas vistas resulta sencillo, puesto que Visual Studio proporciona un asistente para la creación de vistas a partir del modelo y otras indicaciones del desarrollador.

En las aplicaciones Web basadas en ASP.NET MVC el trabajo de desarrollo de las vistas está completamente orientado al código. No existe un diseñador visual ni controles que puedan agregarse mediante el mecanismo de arrastrar y soltar. En general, en las aplicaciones Web basadas en ASP.NET MVC el trabajo de desarrollo consiste, básicamente, en escribir código.

Convención de nombres para las vistas

Como se ha estudiado anteriormente, un método de acción de un controlador puede devolver la llamada a un método View() del objeto ActionResult. En este caso, que es el más habitual, el método View() se emplea para activar la vista correspondiente a ese método de acción. La ejecución de la vista activada genera una página de respuesta HTML que se envía al cliente. Para establecer la correspondencia entre la acción del controlador y la vista a ejecutar, se emplea un mecanismo de convención de nombres sencillo. En la carpeta /Views de un Proyecto de ASP.NET MVC existirá una subcarpeta o zona de vistas, por cada uno de los controladores existentes, y con su mismo nombre. A su vez, cada una de estas subcarpetas almacenará los archivos de las vistas asociadas a las acciones del controlador correspondiente, y con el mismo nombre que el método de acción que le corresponda.

La siguiente ilustración muestra un ejemplo de un Proyecto de ASP.NET MVC en el que se aprecia la correspondencia existente entre el nombre del controlador Home y la subcarpeta o zona de vistas /Views/Home, así como de la correspondencia entre los nombres de los métodos de acción del controlador Home y los nombres de las vistas alojadas en esa subcarpeta de vistas.

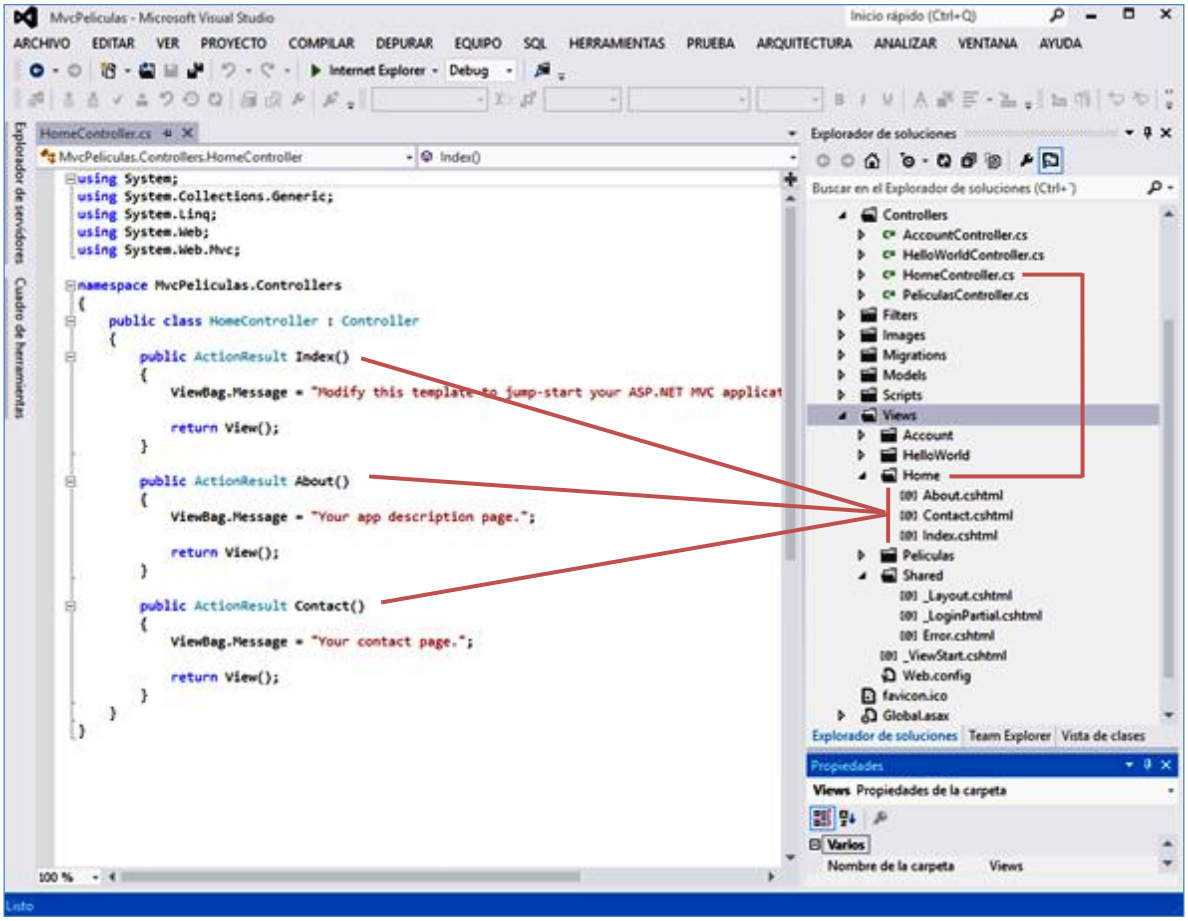

Ilustración 7.6. Explorador de soluciones mostrando la estructura de un Proyecto de ASP.NET MVC

Ejecución de código común a todas las vistas

En la carpeta Views, además de las zonas de vistas, se encuentra alojado un archivo específico denominado **_ViewStart.cshtml**, cuya finalidad es establecer la plantilla de diseño común para todas las vistas del Proyecto de ASP.NET MVC. Su contenido predeterminado es:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

La marca @ especifica el inicio de una sección de código del lenguaje de vistas Razor que queda delimitada entre llaves. La línea de código contenida asigna a la propiedad Layout la ruta y nombre del archivo de vista que define la plantilla de diseño común. Al definirse una plantilla de diseño común, cada archivo de vista contendrá únicamente el código específico de esa vista.

De manera predeterminada, el archivo de vista que define la plantilla de diseño común se denomina **_Layout.cshtml** y se aloja en la subcarpeta /Views/Shared. Esta subcarpeta almacena archivos de código compartido por las vistas, como pueden ser: plantillas de diseño, vistas parciales que son reutilizadas por otras vistas y vistas compartidas por varios controladores. Se pueden crear varios archivos de plantilla de diseño en un mismo Proyecto de ASP.NET MVC. En estos casos, se asigna el nombre de una de las plantillas de diseño existentes al valor de la propiedad Layout en cada archivo de vista de manera específica, añadiendo el código anterior al inicio de cada archivo de vista.

El siguiente código es un ejemplo de un archivo _Layout.cshtml que especifica los aspectos comunes de presentación que van a afectar a todas las vistas del proyecto de ASP.NET MVC.

```
<!DOCTYPE html>
<html lang="es">
    <head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
        <meta charset="utf-8" />
        <title>@ViewBag.Title - Mi aplicación ASP.NET MVC</title>
        <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
        <meta name="viewport" content="width=device-width" />
        @Styles.Render("~/Content/css")
        @Scripts.Render("~/bundles/modernizr")
    </head>
    <body>
        <header>
            <div class="content-wrapper">
                <div class="float-left">
                    <p class="site-title">MvcPelículas</p>
                </div>
                <div class="float-right">
                    <section id="login">
                        @Html.Partial("_LoginPartial")
                    </section>
                    <nav>
                        <ul id="menu">
                            @if (Request.IsAuthenticated)
                            {
                     <li>@Html.ActionLink("Inicio", "Index", "HelloWorld")</li>
                     <li>@Html.ActionLink("Bienvenida", "Bienvenida", "HelloWorld")</li>
                     <li>@Html.ActionLink("Peliculas", "Index", "Peliculas")</li>
                            }
                            else
                            {
                                <li>@Html.ActionLink("Inicio", "Index", "Home")</li>
                                <li>@Html.ActionLink("Acerca de", "About", "Home")</li>
                                <li>@Html.ActionLink("Contacto", "Contact", "Home")</li>
                            }
                        </ul>
                    </nav>
                </div>
            </div>
        </header>
        <div id="body">
            @RenderSection("featured", required: false)
            <section class="content-wrapper main-content clear-fix">
                @RenderBody()
```

```
            </section>
        </div>
        <footer>
            <div class="content-wrapper">
                <div class="float-left">
                    <p>&copy; @DateTime.Now.Year - Mi aplicación ASP.NET MVC</p>
                </div>
            </div>
        </footer>

        @Scripts.Render("~/bundles/jquery")
        @RenderSection("scripts", required: false)
    </body>
</html>
```

Generar los enlaces para la solicitud URL de las acciones: el método ActionLink()

Como se ha estudiado anteriormente, una acción de un controlador se activa mediante una solicitud URL. Por tanto, para que desde una página de la aplicación Web se pueda accederse a otras acciones será necesario especificar los enlaces correspondientes. Además, también podrán especificarse en la plantilla de diseño común, especialmente cuando haya que especificar el menú de la aplicación Web. En el código anterior, que muestra el contenido de un archivo _Layout.cshtml de ejemplo, pueden apreciarse las líneas de código que especifican los enlaces que conforman las opciones del menú. En este ejemplo, existe un elemento <ul> con el identificador "menu" que define los enlaces que forman dos menús diferentes, el primero será accesible solo para usuarios autentificados y, el segundo para la parte pública.

Se utiliza el método **ActionLink()** del objeto Html para generar sobre la página de respuesta los enlaces hacia las acciones correspondientes de los controladores. Los enlaces que son generados mediante el método ActionLink() tienen en cuenta el formato de enrutamiento URL establecido en el módulo RouteConfig.cs de la carpeta /App_Start. Así, por ejemplo, la línea de código:

```
<li>@Html.ActionLink("Acerca de", "About", "Home")</li>
```

generará un enlace hacia la solicitud URL http://localhost:xxxx/Home/About cuyo texto será "Acerca de" y al hacer clic sobre el enlace, se ejecutará el método de acción About() de controlador Home y, como resultado, se activará, posiblemente, la vista About.cshtml.

El lenguaje o motor de vistas Razor

El motor de vistas Razor permite integrar el lenguaje C# en las vistas. La especificación del código de una vista es, por tanto, una mezcla de sintaxis de C# y código de marcado HTML. Los nombres de los archivos de vista tiene la extensión: **.cshtml**. El motor de Razor se encarga de la ejecución de los bloques de código C#, mientras que las etiquetas HTML se añadirán directamente al flujo de salida de la página de respuesta hacia el cliente, para que sean interpretadas por el navegador. El carácter **@** sirve de marcador al motor de vistas para distinguir las secuencias de código de las etiquetas de HTML estático. Si se necesitara escribir un carácter @ sobre la página, por ejemplo, para añadir a la presentación de la página una dirección de correo electrónico, puede doblarse (@@) para que el motor Razor inhiba su comportamiento predefinido.

El objeto de Razor más utilizado es el **objeto Html**. Los métodos del objeto Html se caracterizan por devolver el texto HTML de la operaqción que realizan con su ejecución. Por esta razón suelen denominarse **Helpers** o **HTML Helpers**. La siguiente tabla incluye algunos de los helpers más utilizados, principalmente los que generan campos de formulario.

Nombre de método	Descripción
@Html.Action	Invoca una acción
@Html.ActionLink	Obtiene el enlace de una acción considerando el formato de enrutamiento o mapeo URL establecido
@Html.AntiForgeryToken	Genera un código de seguridad para evitar ataques
@Html.BeginForm	Inicio de formulario. Añade una etiqueta <form>
@Html.CheckboxFor	Genera una casilla de verificación para marcar un campo
@Html.EditorFor	Genera un campo para introducir de texto
@Html.EndForm	Fin de formulario. Añade una etiqueta </form>
@Html.LabelFor	Genera un campo de etiqueta
@Html.ListboxFor	Genera un campo de cuadro de lista
@Html.PasswordFor	Genera un campo que permite introducir una contraseña
@Html.RadiobuttonFor	Genera un campo botón de tipo radio
@Html.TextareaFor	Genera una zona de texto multilínea
@Html.TextBoxFor	Genera un campo de cuadro de texto
@Html.ValidationMessageFor	Devuelve un mensaje de error de validación
@Html.ValidationSummary	Devuelve una lista desordenada de los mensajes de validación. Muestra únicamente los errores de validación a nivel de modelo

Puede obtenerse información más completa sobre Razor en las siguientes direcciones Web:

http://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-(c)
http://www.tutorialsteacher.com/mvc/html-helpers

Glosario de términos

Acoplamiento de componentes de software. El acoplamiento indica el nivel de dependencia entre las unidades de software que conforman una aplicación informática. Es decir, el grado en que una unidad de software puede funcionar sin tener recurrir a otras. Se dice que dos componentes de software están desacoplados, cuando son absolutamente independientes entre sí, es decir, cuando uno puede realizar su funcionalidad completamente sin tener que recurrir al otro. El bajo acoplamiento entre las unidades de software es el estado ideal que siempre se intenta obtener para lograr un buen desarrollo o un buen diseño. Cuanto menos dependiente sean las partes que constituyen un sistema de software, mejor será el resultado final.

Cohesión de componentes de software. La cohesión tiene que ver con que cada unidad de software, que forma parte de una aplicación informática, se refiera a un único proceso o entidad. A mayor cohesión de un componente de software, éste será más sencillo de diseñar, programar, probar y mantener. En el diseño estructurado, se logra alta cohesión cuando cada unidad de software (componente, módulo, función, etc.) realiza una única tarea trabajando sobre una estructura de datos concreta, sin tener la necesidad de conocer detalles sobre otras unidades de software.

Desarrollo rápido de aplicaciones o *Rapid Application Development* (RAD). Es un proceso de desarrollo de software que comprende el desarrollo interactivo, la construcción de prototipos y el uso de utilidades CASE (Sistema de ingeniería asistida por computadora). Actualmente, este término suele utilizar para referirse al desarrollo rápido de interfaces gráficas de usuario y utiliza entornos de desarrollo integrado.

Entity Framework. Es un conjunto de tecnologías de ADO.NET de tipo ORM *(Object Relational Mapper)* que permiten el desarrollo de aplicaciones de software orientadas a datos. Estas tecnologías permiten a los desarrolladores trabajar con datos en forma de objetos y propiedades específicos del dominio, sin tener que pensar en las tablas de las bases de datos subyacentes y en las columnas de las tablas en las que se almacenan estos datos.

Framework. La palabra inglesa *framework* significa marco de trabajo. Un *Framework* define un conjunto estandarizado de conceptos, prácticas y criterios que se pueden emplear para orientar y resolver un tipo de problemática particular y que sirve como referencia para resolver nuevos problemas de índole similar. En el desarrollo de software, un *framework* es una estructura conceptual y tecnológica que sirve de base organizativa para el desarrollo de software. Puede incluir soporte, bibliotecas, lenguajes, etc. para ayudar a desarrollar y unir los diferentes componentes de un proyecto de desarrollo de software.

Helper. En general, es cualquier método o función que devuelve un dato de tipo *string*. En las tecnologías MVC, se refiere a los métodos de Razor que se incluyen en las vistas para devolver un texto HTML, según su funcionalidad, y componer dinámicamente la página de respuesta.

Interfaz pública. Una interfaz pública contiene las definiciones de las funcionalidades que una clase puede implementar. Las interfaces no contienen implementación de métodos, porque en ellas solo se especifica qué se debe hacer, pero no su implementación. La lógica del comportamiento de los métodos se define en las clases que implementan una interfaz pública. Las clases que implementan

una interfaz pública pueden interactuar entre sí en una misma máquina o a través de una red de ordenadores. Una clase puede implementar varias interfaces. Las interfaces pueden contener como miembros: eventos, indizadores, métodos y propiedades. Los miembros de una interfaz son públicos y no pueden incluir modificadores de acceso. No se puede crear una instancia de una interfaz directamente, ya que sus miembros se implementan por medio de una clase que implementa la interfaz. Las interfaces públicas deben estar diseñadas para soportar cambios futuros y mejoras.
Patrón arquitectónico. Los patrones arquitectónicos, o patrones de arquitectura del software, ofrecen soluciones a problemas de arquitectura de software en ingeniería de software. Proporcionan una descripción de los elementos, así como del tipo de relación existente entre ellos, y definen un conjunto de restricciones sobre cómo pueden ser usados. Un patrón arquitectónico expresa un esquema de organización estructural elemental para un sistema de software, que consta de subsistemas, sus responsabilidades e interrelaciones. Aunque un patrón arquitectónico propone una estructura u organización de un sistema de software, no es una arquitectura como tal. Un patrón arquitectónico es más un concepto que define los elementos de una arquitectura de software.

Patrón de software. En general, los patrones de software son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y en otros ámbitos relacionados con el diseño de la interacción o de la interfaz. Un patrón es una solución fiable a un problema de software. Para que una solución sea considerada un patrón debe ser fiable y reutilizable. Se debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Además, debes ser aplicable a diferentes problemas de diseño en distintas circunstancias de procesamiento. Suelen diferenciarse entre los conceptos patrón arquitectónico y patrón de diseño, atendiendo a su nivel de abstracción. Así, a menudo se suele considerar como **patrones arquitectónicos** a aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software. Mientras que se considera como **patrones de diseño** a aquellos que expresan esquemas para definir estructuras de diseño, o sus relaciones, con las que construir sistemas de software. De este modo, se atribuye un mayor nivel de abstracción a los patrones arquitectónicos.

Razor. Es un lenguaje de marcado en el lado del servidor. Es decir, es un lenguaje de marcado de procesamiento en el servidor. Se trata, por tanto, de una sintaxis de marcado que permite incorporar código de procesamiento en el servidor *(server-based code)* en páginas Web empleando Visual Basic y C#. El código de procesamiento en el servidor permite crear contenido web dinámico sobre la marcha, mientras que una página web se escribe en el navegador. Cuando se solicita una página web, el servidor ejecuta el código de procesamiento en el servidor incluido dentro de la página antes de que se devuelva la página al navegador. Mediante la ejecución en el servidor, el código puede realizar tareas complejas, tales como el acceso a bases de datos. Razor está especialmente diseñado para la creación de aplicaciones web y es fácil de usar y de aprender.

Smalltalk. Es un lenguaje de programación reflexivo de programación, orientado a objetos y con tipado dinámico. Por sus características, puede ser considerado también como un entorno de objetos, donde incluso el propio sistema es un objeto. Metafóricamente, se puede considerar que un programa Smalltalk es un mundo virtual donde conviven objetos que se comunican entre sí, mediante el envío de mensajes. De modo que un programa Smalltalk consiste únicamente en un conjunto de objetos que presentan características comunes: tienen memoria propia, pueden comunicarse con otros objetos mediante el envío de mensajes, poseen la capacidad de heredar características de objetos ancestros y tienen capacidad de procesamiento.