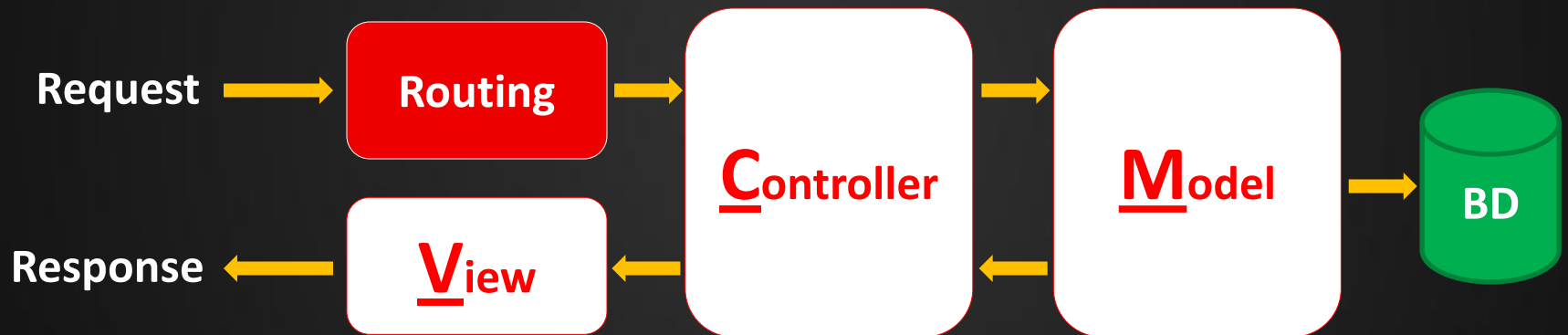


ASP.NET MVC

Angel Núñez Salazar

Email: snahider@gmail.com
Blog: <http://snahider.blogspot.com>
Twitter: [@snahider](https://twitter.com/snahider)

Model – View - Controller



Qué es ASP.NET MVC

« **ASP.NET MVC** es una framework de desarrollo web **open source**, que combina la efectividad y ventajas de una **arquitectura MVC**, prácticas del **desarrollo ágil** y las mejores partes de la plataforma **ASP.NET existente** »

ASP.NET Framework

ASP.NET MVC

ASP.NET WebForms

ASP.NET Core

(Caching, Sessions, Security
Cookies, QueryString, Master Pages)

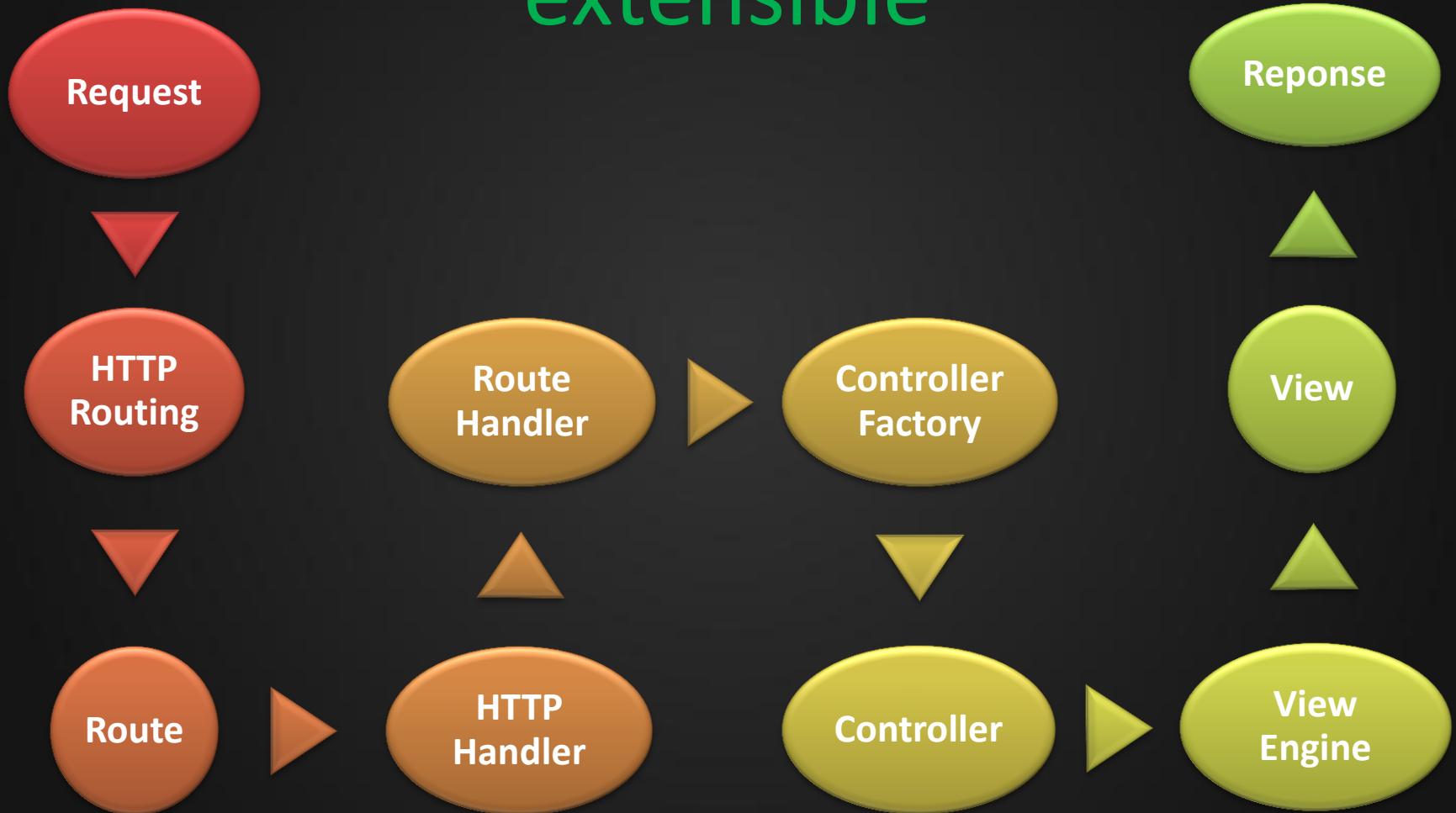
.NET Framework

Demo

Explorando un nuevo proyecto
ASP.NET MVC

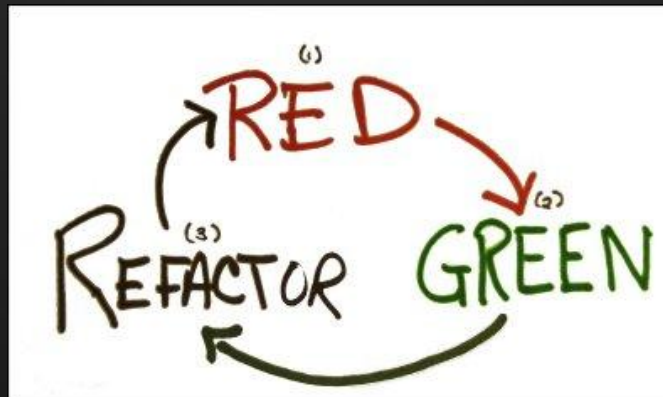
¿Porqué utilizar ASP.NET MVC?

1.- Arquitectura flexible y extensible



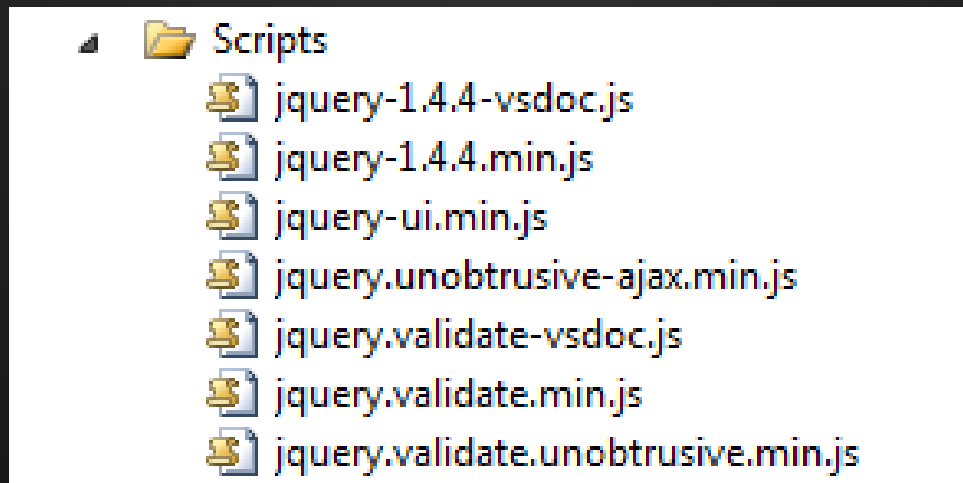
2.- Testeabilidad

- Cualquier Unit Testing Framework es soportada.
 - NUnit, MSTest, MBUnit, XUnit.Net
- Partes específicas de la framework son fácilmente mockeables.
- Facilita la aplicación de prácticas ágiles.



3.- Control sobre el HTML

- ASP.NET MVC reconoce la importancia de producir **HTML limpio, semántico y compatible con los estándares**.
- Fácil integración con herramientas a nivel de cliente.



4.- Routing

- Las Urls no corresponden a archivos ya que los request son manejados por los controllers.
- Completo control de las URLs permitiendo cualquier patron de mapeo URL – controller.
- El sistema de ruteo fue diseñado para ASP.NET MVC pero está en «system.web.routing» permitiendo su uso por WebForms.

5.- Convention over Configuration

Goodbye XML Hell

*“From now on anyone who considers themselves to be a serious professional must refuse to write another line of **XML**. When asked, **say NO**”*

Robert C. Martin (Uncle Bob)

6.- Community and Environment

- Gran cantidad de proyectos creados alrededor de MVC



- Conferencias y encuentros continuos.



Community For MVC.Net

- Actualmente existen 241 resultados para "ASP.NET MVC" dentro de



7.- Open Source

- ASP.NET MVC ha sido liberado bajo MS-PL que es una licencia open source aprobada por la OSI.
- Podemos descargar el código original e inclusive modificarlo y compilar una nueva versión de el.
- Depurar componentes del sistema y navegar por el código para entenderlo o ver las posibilidades de desarrollo.

ASP.NET "Tradicional" - WebForms

ASP.NET significó un gran cambio cuando apareció por primera vez ya que pretendía cerrar la brecha entre el desarrollo en windows y el desarrollo web.

- Jerarquía de controles que renderizaban automáticamente HTML.
- UI que mantenía su estado y orientada a eventos lo que permitía al desarrollador despreocuparse de las llamadas y respuestas HTTP.

¿Cuál es el problema con el uso de WebForms ?

- ViewState
- Page LifeCycle
- Limitado control del HTML
- Casi imposible de realizar test unitarios..

WebForms o MVC

- MVC no es un reemplazo frente a WebForms.
- Las ventajas de MVC y la presión de la comunidad están convirtiendo a WebForms en obsoleto.
- Se pueden usar ambas tecnologías al mismo tiempo.
- Si recién estás utilizando ASP.NET, utiliza de frente MVC y ya no aprendas WebForms.

Conclusiones

- ASP.NET MVC fue diseñado para aplicar los mejores principios, patrones y prácticas dentro del desarrollo de software.
- ASP.NET MVC es la mejor alternativa para el desarrollo web utilizando la plataforma .NET.

PRERREQUISITOS

ASP.NET MVC

Angel Núñez Salazar

Email: snahider@gmail.com

Blog: <http://snahider.blogspot.com>

Twitter: [@snahider](https://twitter.com/snahider)

UNIT TESTING

ASP.NET MVC

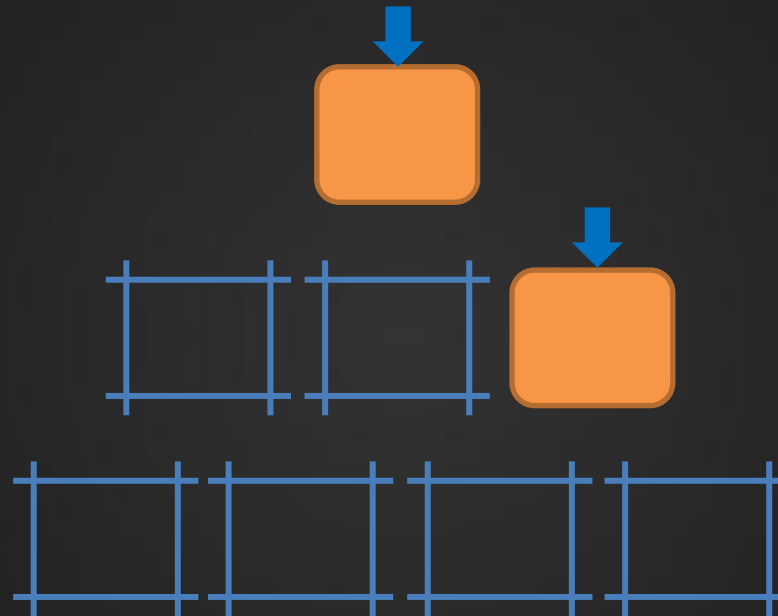
Angel Núñez Salazar

Email: snahider@gmail.com
Blog: <http://snahider.blogspot.com>
Twitter: [@snahider](https://twitter.com/snahider)

3 Tipos Importantes de Test



Test Unitarios



Se encargan de verificar asunciones sobre
piezas lógicas de código y en aislamiento

Test Unitarios

- **Código Lógico:** Pequeñas unidades de código con lógica (ifs, loops, cálculos, etc)

```
public int CalculateDiscount(decimal amount, UserCategory userCategory)
{
    int discount=0;

    if (amount > 1000)
        discount = 50;

    if (userCategory == UserCategory.Platinum)
        discount += 10;

    return discount;
}
```

Test Unitarios

- **Aislamiento:** Se realizan de manera separada al resto de la aplicación, de sus dependencias y no acceden a recursos del sistema.
 - Un test unitario no se comunica con la base de datos.
 - Un Test Unitario no depende de archivos de configuración.
 - Un Test Unitario no ejercita la clase y todas sus dependencias en simultáneo.

Propiedades de un buen Test Unitario

Fast: Unos cuantos milisegundos en ejecutarse.

Isolated: Enfocarse en una única unidad de código.

Repeatable: Ejecutarse de manera repetitiva sin intervención.

Self-validating: Sin necesidad de reexaminar los resultados.

Timely: Escritos en el momento adecuado, antes del código.

Herramientas

- Podríamos realizar los test de forma manual pero puede ocasionar errores y tomar mucho tiempo.
- Estas herramientas se denominan XUnit Testing Frameworks:
 - NET: NUnit, MSTest, XUnit.net, Mbunit
 - Java: JUnit, TestNG, JTiger.....

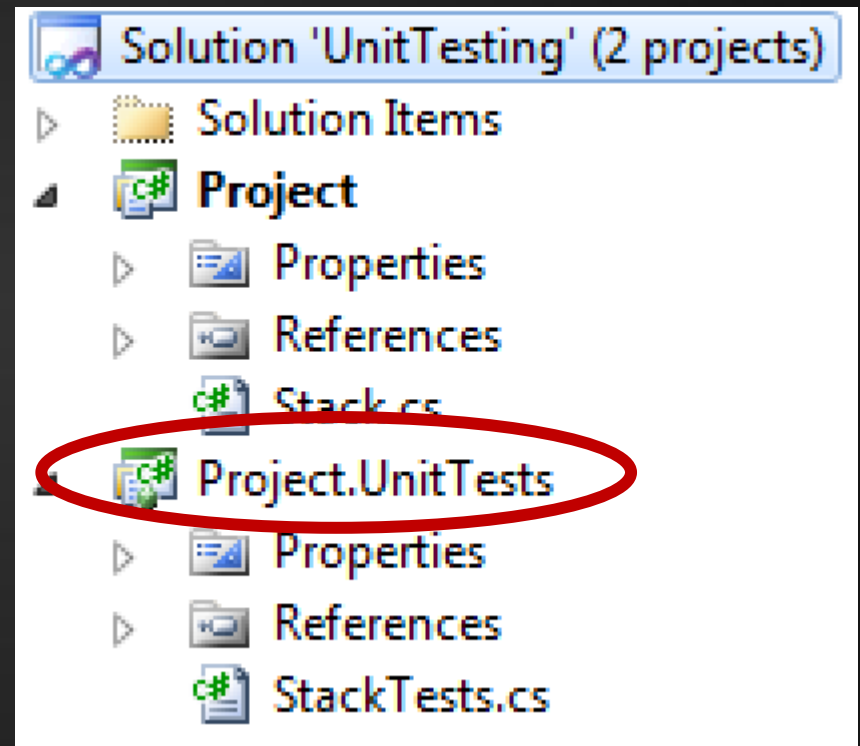
Nuestro Primer Test Unitario

Generar los test unitarios para la clase Stack dentro de la carpeta ejercicios/UnitTesting.

Convención de Nombres

Proyectos

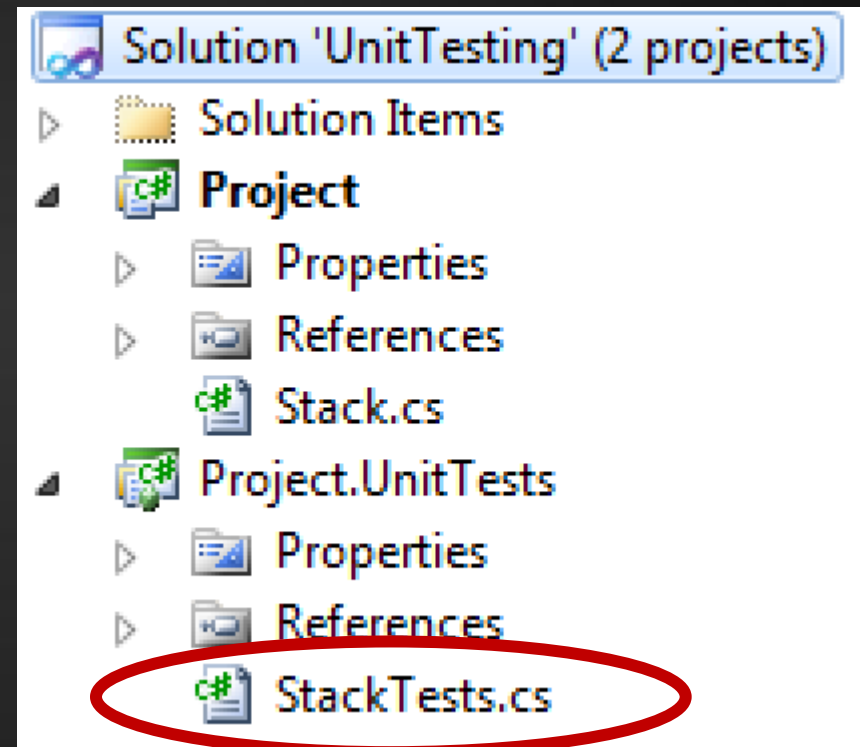
- Crear un nuevo proyecto para los test.
- Llamar al proyecto `[ProjectUnderTest].UnitTests`



Convención de Nombres

Clases

- Por cada clase de producción crear una nueva clase para los tests.
- Llamar a las clases **[ClassUnderTest]Tests**



Convención de Nombres

Métodos

- Por cada test a la clase de producción crear un nuevo método
- Llamar a estos métodos

[Método_Escenario_ResultadoEsperado]

```
public void IsEmpty_NuevoStack_RetornaVerdadero()
```

Marcar con Atributos los Test

- **[TestClass]** Denota una clases que contendrá los test automatizados

```
[TestClass]  
public class StackTest
```

- **[TestMethod]** Denota que es un test automatizado.

```
[TestMethod]  
public void IsEmpty_NuevoStack_RetornaVe
```

Estructura de un Test

ARRANGE Creamos todas las precondiciones y entradas necesarias.

```
Stack stack=new Stack();
```

ACT Realizamos la acción del objeto que estamos probando.

```
bool estaVacio = stack.IsEmpty;
```

ASSERT Verificamos los resultados esperados.

```
Assert.IsTrue(estaVacio);
```

Los Assertos

Nuestros resultados los podremos verificar a través de métodos ya definidos dentro del framework y que a su vez nos proveerán de mensajes útiles si estos fallan.

- Assert.IsTrue
- Assert.IsNull
- Assert.AreEqual
- Muchos más

```
Assert.IsTrue(estaVacio);
```


Ejecutar nuestras pruebas

- **Visual Studio** A través del menú o una combinación de teclas.
- **Consola NUnit** Aplicación externa capaz de ejecutar pruebas que se encuentren dentro de un ensamblado.

TIP: Ejecutar la pruebas desde el VS: CTRL +R +A

Nuestro Segundo Test Unitario

Realizar la prueba unitaria que verifique :
«El Stack no se encuentra vacío cuando se ingresa un elemento»

```
[TestMethod]
public void IsEmpty_IngresoUnElemento_RetornaFalso()
{
    Stack stack=new Stack();
    stack.Push(1);

    bool estaVacio = stack.IsEmpty;

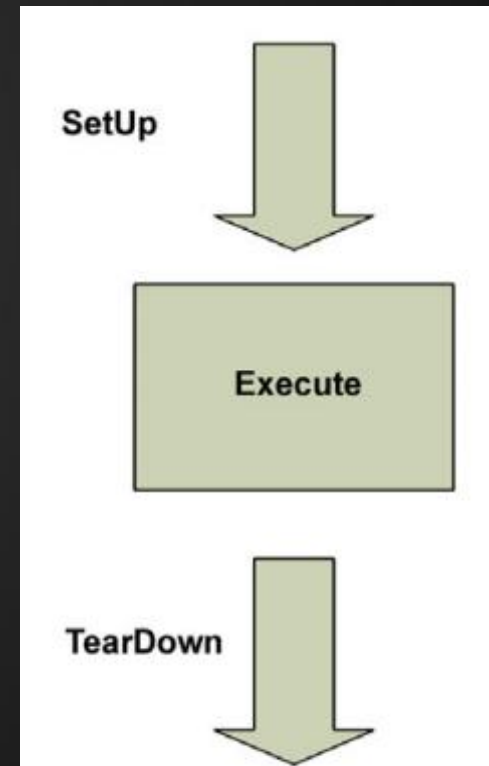
    Assert.IsFalse(estaVacio);
}
```

Test Fixtures

Nos permiten establecer un estado consistente al inicio y al final de los tests. (Constructores /Destructores de cada test)

[TestInitialize] Se ejecuta siempre antes de cada test.

[TearCleanup] Se ejecuta siempre al finalizar un test.



Ejercicio – Completar las siguientes pruebas unitarias

1. Al ingresar y sacar un elemento, el stack está vacío.
2. Al ingresar dos y sacar uno, el stack no está vacío.
3. Al ingresar y sacar un elemento, el elemento debe ser igual al ingresado.
4. Al ingresar dos y obtener un elemento, el elemento debe ser igual al segundo ingresado
5. Al ingresar dos y obtener dos, el segundo elemento obtenido es igual al primero que se ha ingresado.

ENTITY FRAMEWORK ASP.NET MVC

Angel Núñez Salazar

Email: snahider@gmail.com
Blog: <http://snahider.blogspot.com>
Twitter: [@snahider](https://twitter.com/snahider)

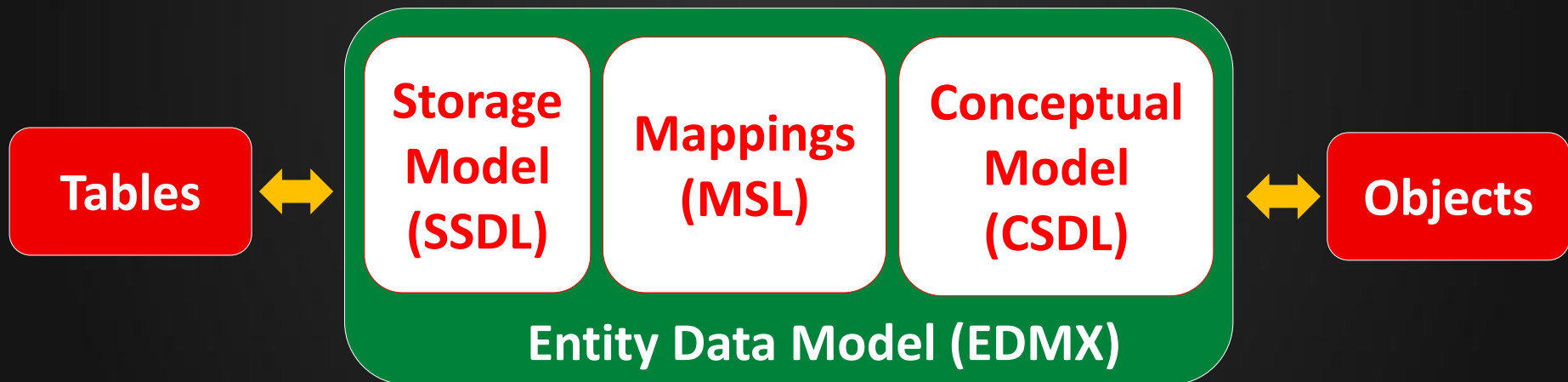
Entity Framework

Entity Framework es ORM (Objet/Relational mapping) conformado por un conjunto de tecnologías ADO.NET para el desarrollo de aplicaciones que interactúan con datos.

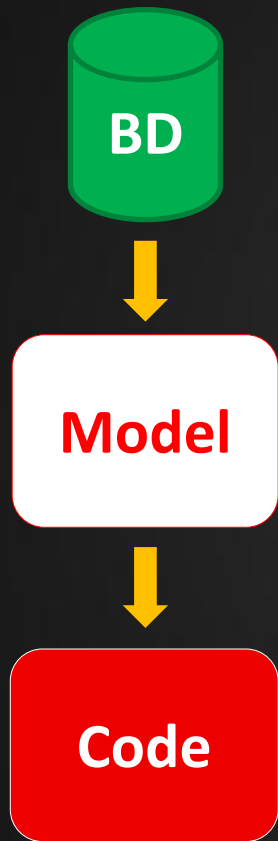
- Eliminar la fricción entre los modelos de datos y el trabajo con objetos.
- Reducir el tiempo y cantidad de infraestructura a desarrollar en aplicaciones orientadas a datos.

Entity Data Model

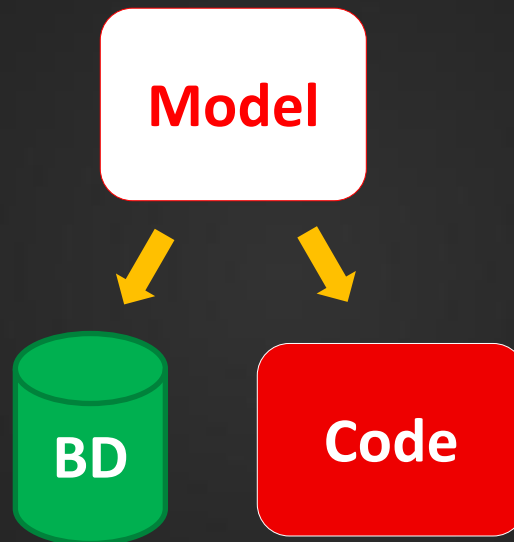
Define la estructura de nuestros objetos y sus relaciones y como estos serán mapeados dentro de un esquema de tablas.



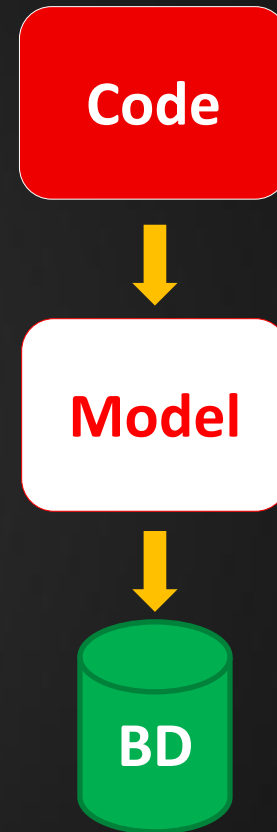
3 Enfoques de Trabajo



DB First



Model First



Code First

Ejemplos

Realizaremos ejemplos desde el VS donde veremos los siguientes temas:

- Enfoque DB First.
- Enfoque Model First
- Consumir el modelo desde un proyecto diferente
- Linq to Entities para consultar datos del modelo.
- Manipulando datos del Modelo

PROYECTO BASE TIENDA VIRTUAL ASP.NET MVC

Angel Núñez Salazar

Email: snahider@gmail.com
Blog: <http://snahider.blogspot.com>
Twitter: [@snahider](https://twitter.com/snahider)

URL Y ROUTING

ASP.NET MVC

Angel Núñez Salazar

URLs y Routing

Antes

`http://site.com/login.aspx` == `e:\webroot\login.aspx`

- Las URLs corresponden directamente a archivos en el servidor.
- URLs poco amigables y poco favorables para el SEO.
- Uso Handlers y Filtros pero traen más problemas que soluciones.

URLs y Routing

ASP.NET MVC

- Las Urls no corresponden a archivos ya que los request son manejados por los controllers.
- Completo control de las URLs permitiendo cualquier patron de mapeo URL – controller.
- El sistema de ruteo fue diseñado para ASP.NET MVC pero está en «system.web.routing» permitiendo su uso por WebForms.

URLs y Routing

ASP.NET MVC

<http://site.com/fotos>

```
{ controller = Galeria,  
  action = Mostrar }
```

<http://site.com/admin/login>

```
{ controller = Autorización,  
  action = Login }
```

<http://site.com/articulos/misvacaciones>

```
{ controller = Articulos,  
  action = Ver,  
  parameter = misvacaciones }
```

Configurando las Rutas

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.MapRoute(
        "Default",                      // Route name
        "{controller}/{action}/{id}",   // URL with parameters
        new {controller = "Home",       // Parameter defaults
            action = "Index",
            id = UrlParameter.Optional}
    );
}
```

Urls

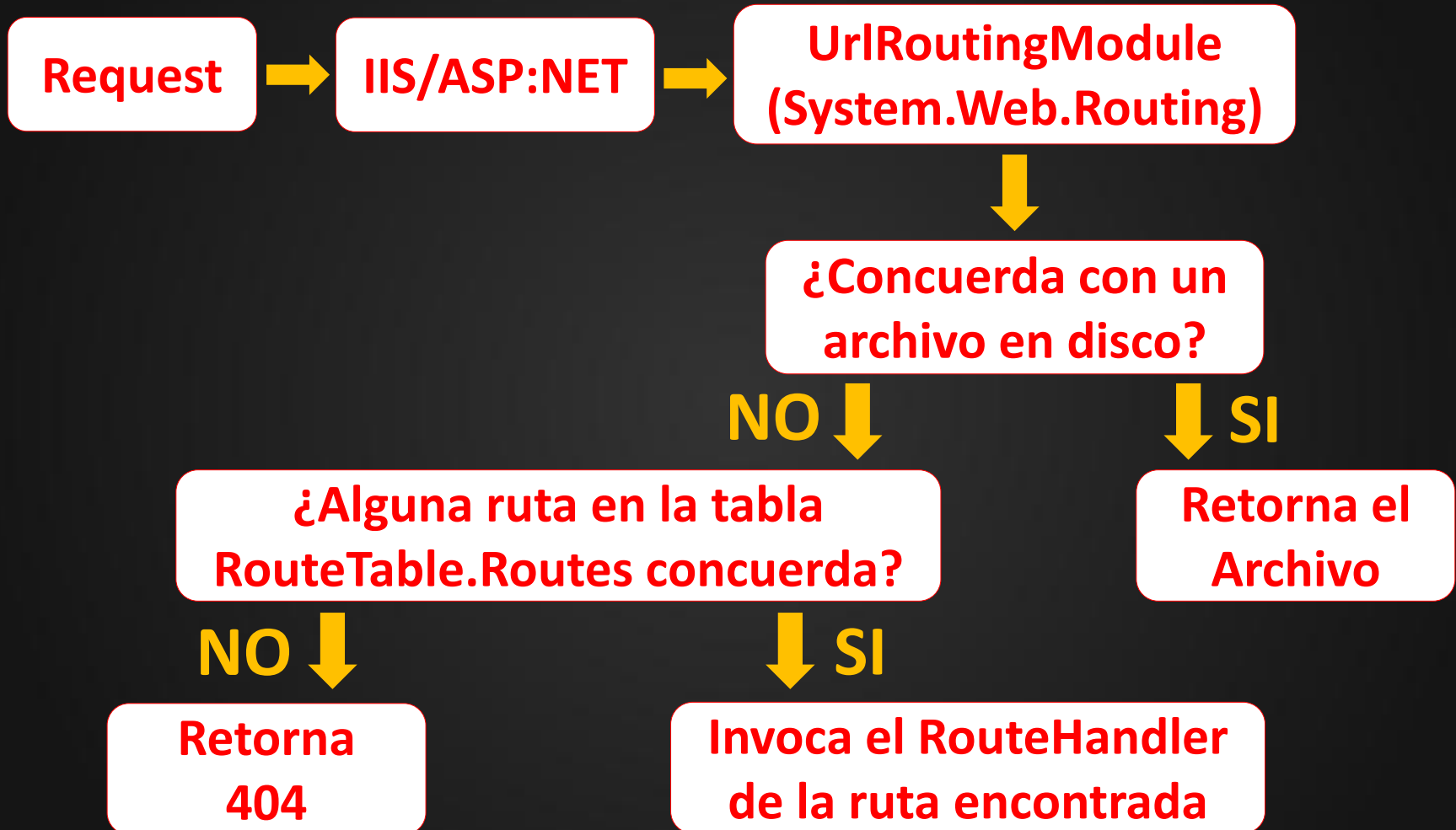
MAPEO

/	{ controller = Home, action = Index}
/Articulos	{ controller = Articulos, action = Index}
/Articulos/Mostrar	{ controller = Articulos, action = Mostrar}
/Articulos/Editar/1	{ controller = Articulos, action = Editar, id = 1}

Propiedades de una Ruta

```
Route route = new Route(  
    "{controller}/{action}/{id}", URL  
    new MvcRouteHandler() RouteHandler  
{  
    Defaults = new RouteValueDictionary(new  
        {  
            controller = "Home",  
            action = "Index",  
            id = UrlParameter.Optional  
        }  
    ),  
  
    Constraints = new RouteValueDictionary(new  
        {  
            id = @"\d+"  
        }  
    )  
};  
routes.Add("Default", route);
```


ASP.NET MVC Routing



Búsqueda de Rutas

Para encontrar una ruta que concuerde con el request se deben satisfacer las siguientes condiciones:

- La URL del request debe seguir el patrón de la URL de la ruta.
- Todos los parámetros entre llaves deben estar presentes en la URL o en los Defaults.
- Todas las Constraints debe cumplirse.

El orden de las rutas es importante

Las búsquedas se realizan de inicio a fin en la tabla y toma la primera que encuentra, por lo tanto **colocar las rutas más específicas antes que las generales.**

```
routes.MapRoute(  
    "Ofertas",  
    "ofertas/{fecha}",  
    new { controller = "Catalogo", action = "Ofertas" }  
);  
  
routes.MapRoute(  
    "Default",  
    "{controller}/{action}/{id}",  
    new { controller = "Home", action = "Index",  
          id = UrlParameter.Optional }  
);
```

Ejercicio: Modificando el orden de las rutas y

- Agregar la siguiente ruta al final de todas las demás rutas.


```
routes.MapRoute("Producto-Detalles"  
    , "Productos/{id}",  
    new { controller = "Productos", action = "Detalle" });
```

- Analizar las rutas utilizando el route debugger.
- Mover la ruta previamente creada antes de la ruta por defecto y analizar con el route debugger.

URL Pattern

Para encontrar cual es la ruta que corresponde a una URL solo se considera el path de la URL.

`http://sitio.com/catalogo/libros?id=1234&tipo=historia`



The diagram illustrates the components of the URL `http://sitio.com/catalogo/libros?id=1234&tipo=historia`. Red brackets are used to group the components: the first bracket under `http://sitio.com` is labeled **Host**; the second bracket under `/catalogo/libros` is labeled **Path**; and the third bracket under `?id=1234&tipo=historia` is labeled **QueryString**.

Parámetros

```
routes.MapRoute(null, "productos/{categoria}",  
    new { controller = "Productos", Action = "Mostrar" });
```

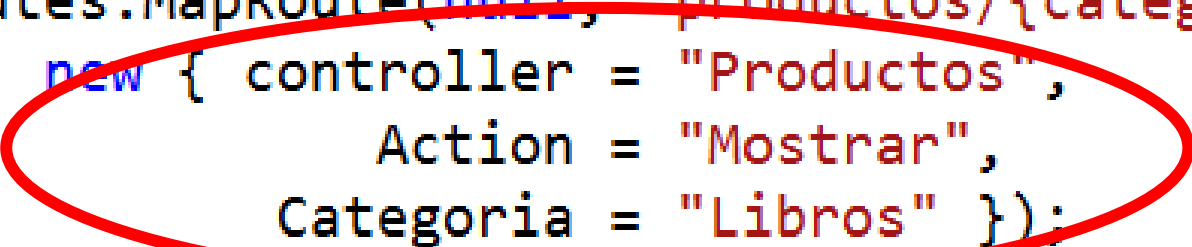
- Concuerda con las siguientes URLs:
 /products/libros /products/12354
- El parámetro es agregado al objeto **RouteData** (RouteData.Values["categoria"]) y por lo tanto puede ser pasado a la acción.

```
public ActionResult Mostrar(string categoria)  
{  
    return View();  
}
```

Parámetros por Defecto

- Podemos indicar que un parámetro sea opcional si le asignamos un valor por defecto.

```
routes.MapRoute(null, "productos/{categoria}",  
    new { controller = "Productos",  
          Action = "Mostrar",  
          Categoria = "Libros" });
```



- Concuerda con las siguientes URLs:
 /productos/libros /productos/12354
 /productos

Parámetros sin Valor

- Podemos indicar que determinados parámetros por defecto tengan valor 0 o null.

```
routes.MapRoute(null, "productos/{categoria}",  
    new { controller = "Productos",  
          action = "Listar",  
          categoria = UrlParameter.Optional }  
);
```

- El valor que tomará será 0 o null dependiendo de su tipo de dato en la acción del controller.

Restricciones

Podemos agregar condicionales adicionales que se deben satisfacer antes de encontrar la ruta correcta.

- Peticiones que únicamente sean POST o GET.
- Que algunos parámetros cumplan determinados patrones. Ejm: Id solo numérico.
- Peticiones de determinados navegadores o dispositivos.C

```
routes.MapRoute("catalogo", "productos/detalle/id",  
    new { controller = "Productos", action = "Detalle" },  
    new { id = @"\d" }));
```

Parámetro Catchall

Parámetros especial que permite que la ruta satisfaga una URL con un número arbitrario de parámetros.

```
routes.MapRoute(  
    null,  
    "catalogo/{*rutaCatalogo}",  
    new {controller="Catalogo", action="Mostrar"});
```

Ejm:

/catalogo/libros/arte/renacentismo

RouteData.Values["rutaCatalogo"]=libros/arte/renacimiento

Evitando las rutas

Existen casos especiales en los cuales queremos evitar el sistema de ruteo cuando se cumpla determinado patrón.

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");  
  
routes.IgnoreRoute("{*favicon}",  
    new { favicon = @"(.*/*)?favicon.ico(/.*)?" });
```

- Colocar estas sentencias antes que todas las demás.
- No perder el tiempo excluyendo determinadas rutas al menos que haya una buena razón para ello.

Ejercicio: Creando Nuevas Rutas

- Crear las rutas necesarias para que se cumplan los siguientes casos.
- Solo se debe utilizar a la acción Index del controlador Productos.

Urls	Descripción
/	Primera página con todos los productos de todas las categorías
/pagina2	Segunda página de todos los productos de todas las categorías
/libros	Primera página con únicamente los productos de la categoría libros
/libros/pagina2	Segunda página con únicamente los productos de la categoría libros

Generando URLs

- Podemos hardcodear y concatenar la url.

```
<a href="/productos/detalle/@ViewBag.Id">  
    Ver Detalle  
</a>
```

- Si cambiamos la Url para el ProductoController y la acción Detalle, el link ya no funcionará.
 - Concatenar Urls complejas es propenso a errores.
- El sistema de Ruteo nos provee un mejor mecanismo aprovechando el conocimiento de las rutas que hemos ingresado.

Html.ActionLink

```
@Html.ActionLink("Mostrar Todos","Listar","Productos")
```

Para la configuración de rutas por defecto mostrará:

```
<a href="/productos/listar" />
```

- Mostrará un link con cualquier URL que hayamos configurado en nuestra tabla de rutas
- Cualquier cambio en la configuración de rutas se reflejará automáticamente en las URLs.
- Si no especificamos un controller asumirá el mismo de la acción actual.

Otras opciones Html.ActionLink

Utilizando parámetros

```
@Html.ActionLink("Primera Página","Listar","Productos",  
    new {pagina=1},null)
```

Para la configuración de rutas por defecto mostrará:

```
<a href="/catalogo/mostrar?pagina=1" />
```

Para la ruta /catalogo/mostrar/pagina{pagina} mostrará:

```
<a href="/catalogo/mostrar/pagina1" />
```

Parámetros por defecto

```
@Html.ActionLink("Mostrar Todos","Listar","Productos")
```

Si "Mostrar" es un parámetro por defecto:

```
<a href="/productos" />
```

Url.Action

Funciona exactamente igual que `Html.ActionLink` con la diferencia que genera únicamente el url y no el link.

```
<a href="@Url.Action("Listar","Productos")">  
    Mostrar Todos  
</a>
```


Generando Urls desde el Controller

Un uso muy común es generar la url para realizar una redirección desde la acción de un controller.

```
public ActionResult Nuevo(Producto producto)
{
    return RedirectToAction("Listar", "Productos");
}
```

- También podemos utilizar `Url.Action` para obtener únicamente la Url.

Utilizando el nombre de las Rutas

```
routes.MapRoute("catalogo", "productos/{categoria}",  
    new { controller = "Productos", action = "Listar" });
```

Nos permite referenciar directamente a la ruta y evitar que esta se busque de manera ordenada en la tabla.

```
@Html.RouteLink("Mostrar Libros", "catalogo", new { categoria = "Libros" })
```

```
return RedirectToRoute("catalogo", new { categoria = "libros" });
```

- Los nombres de las rutas son opcionales.
- Referenciar directamente el nombre rompe la separación de conceptos entre el controller y las rutas.

Ejercicio: Creando un Menú de Navegación

- Abrir el archivo _Layout.cshtml y dirigirnos a la sección que se muestra a continuación:

```
<div class="left_content">
  <div class="title_box">Categorías</div>
  <ul class="left_menu">
    <li class="odd">@*Link de libros aquí*@</li>
    <li class="even">@*Link de deportes aquí*@</li>
  </ul>
</div>
```

- Agregar 2 links para las categorías: "Libros" y "Deportes", estos links deben llevarnos a una página que muestre únicamente los productos de dichas categorías.

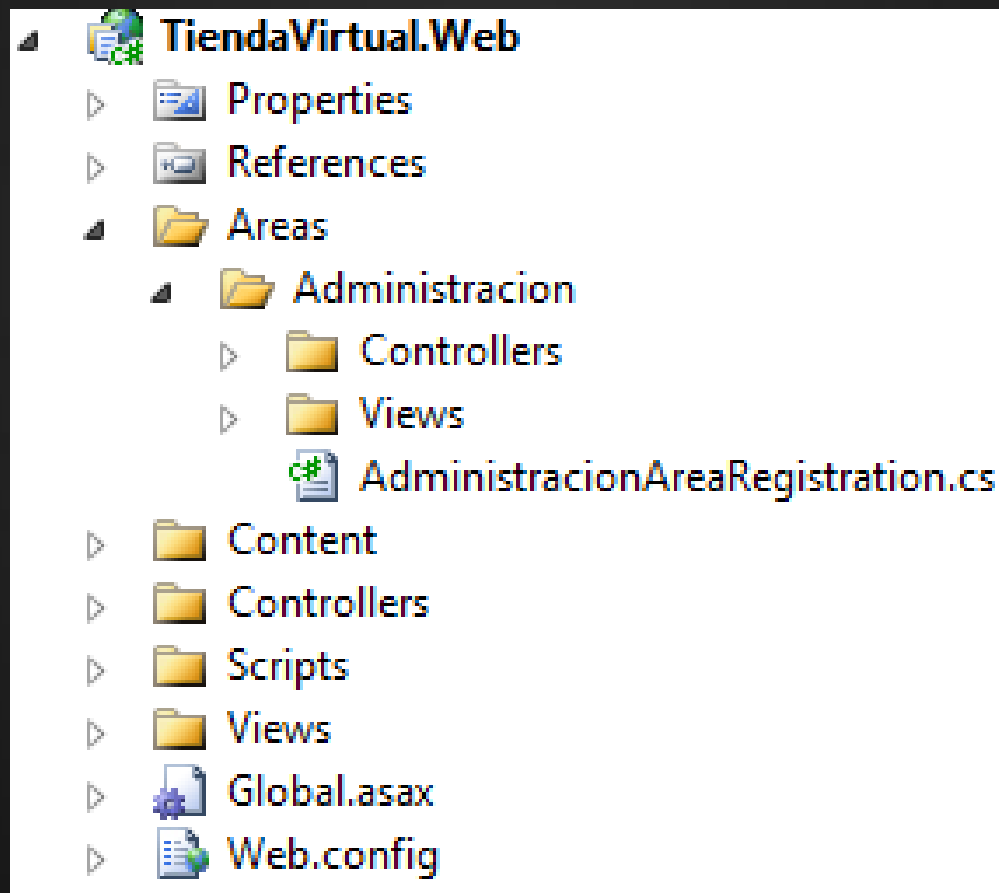
Áreas

Nos permite **dividir nuestra aplicación en segmentos funcionales** denominados áreas (forum, administración) cada uno con sus propios controllers, vistas, estilos, etc.

- **Organización:** Cada área tiene su propia estructura de directorios relacionada a una funcionalidad específica.
- **Aislamiento:** Múltiples equipos trabajando en áreas diferentes sin que esto cause conflictos.
- **Reutilización:** Las áreas pueden ser independientes de los proyectos donde se encuentran y podemos copiarlas y usarlas en uno nuevo.

Creando un Área

Para crear un área damos clic derecho sobre nuestro proyecto web y seleccionamos agregar área.



Áreas y el Sistema de Rutas

```
protected void Application_Start()  
{  
    AreaRegistration.RegisterAllAreas();  
    RegisterRoutes(RouteTable.Routes);  
}
```

A través de ese método se escanea todas las referencias en búsqueda de clases AreaRegistration

```
public class AdministracionAreaRegistration : AreaRegistration {  
  
    public override string AreaName { get { return "Administracion"; } }  
    public override void RegisterArea(AreaRegistrationContext context){  
        context.MapRoute(  
            "Administracion_default",  
            "Administracion/{controller}/{action}/{id}",  
            new { action = "Index", id = UrlParameter.Optional }  
        );  
    }  
}
```

Generando Urls en la misma Área

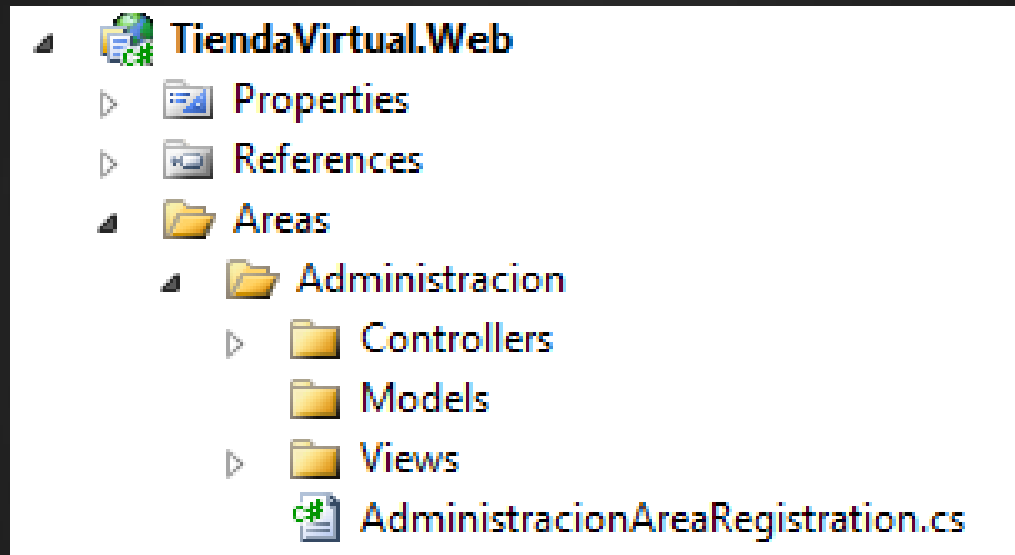
Al trabajar dentro de un misma área no tenemos que preocuparnos sobre el concepto de área.

Por ejemplo si utilizamos: `Url.Action("Productos","Mostrar")`

- `/Administracion/Productos/Mostrar` si nos encontramos en el área administración y existe el controller adecuado dentro del mismo namespace del área.
- `/Productos/Mostrar` si no pudo asociar el request dentro del mismo área y lo buscó en el resto de la aplicación.

Ejercicio: Crear un Área para la administración.

- Crear una nueva área Administración



- Crear el HomeController dentro del área.
- Modificar el AdministracionAreaRegistration para que el controller por defecto sea Home.
- Crear el link "Admin" en el menú de navegación superior que muestre el Home del área Administración

Buenas Prácticas con URLs

Desde el avance de la web se ha empezado a tomar el diseño de urls de manera seria y se deben tomar en cuenta ciertas prácticas:

- Tener URLs fácilmente entendibles por personas.
- Crear URLs amigables según SEO.
- Evitar muchos querystrings.
- Hacer un buen uso de las redirecciones.
- Utilizar correctamente los POST o GET.

CONTROLLERS Y ACTIONS ASP.NET MVC

Angel Núñez Salazar

Email: snahider@gmail.com
Blog: <http://snahider.blogspot.com>
Twitter: [@snahider](https://twitter.com/snahider)

Controllers

Son los responsables de manejar el flujo de la aplicación, esto incluye:

- Recibir las peticiones y datos de los usuarios.
- Realizar consultas o ejecutar comandos en el dominio.
- Determinar cual es la vista adecuada que se mostrará al usuario.

Criterios que debe cumplir un controller

Se deben cumplir los siguientes criterios para que una clase sea considerada como controller en ASP.NET MVC,

- Ser una clase pública y concreta.
- Terminar con el postfijo "Controller".
- Implementar la interface IController.

```
public class HomeController : IController
{
    public void Execute(RequestContext requestContext)
    {
        requestContext.HttpContext.Response.Write("Hola Mundo");
    }
}
```

La clase Controller

La clase Controller que es una implementación por defecto de `IController` y que además nos brinda otras facilidades: Action Methods, Action Results y Filters.

```
[HandleError] Filter
public class HomeController : Controller
{
    public ActionResult Saludo()
    {
        ViewData["Mensaje"] = "Hola Mundo";
        return View(); Action Result
    }
}
```

Action Method

Recibiendo información entrante

Existen varias alternativas, una de ellas es accediendo directamente a los conjuntos de objetos del contexto.

```
public ActionResult EditarProducto()
{
    string usuario = User.Identity.Name;
    DateTime fechahora = HttpContext.Timestamp;
    Auditar(usuario, fechahora);

    string id = Request.QueryString["id"];
    string nombre = Request.Form["Nombre"];
    string precio = Request.Form["precio"];
    ActualizarProducto(id, nombre, precio);

    return View();
}
```

Usando Action Parameters

Es la forma más limpia de recibir la información sin que tengamos que extraerla manualmente y dependiendo de los contextos

```
public ActionResult EditarProducto  
    (string id,string nombre,string precio)  
{  
    ActualizarProducto(id, nombre, precio);  
    return View();  
}
```

Para obtener la información de los parámetros, se escanea internamente varios contextos y les asigna un valor.

Model Binders and Value Providers

Model Binders: Son los encargados de proveer los valores para los actions parameters, permitiendo que las acciones se mantengan limpias sin consultar directamente la información del request.

Value Providers: Son las fuentes de información que son consultadas por los model binders. Por defecto existen 4 value providers:

- Request.Form
- Request.QueryString
- Request.Files
- RouteData.Values

Model Binders and Value Providers

<http://site.com/buscar?nombre=mvc&precio=30>

Action Parameters

String nombre mvc

Int precio 30

Producto producto

String nombre mvc

Int precio 30

Model Binder

category nombre

ValueProviderFactories.Factories

FormValueProvider

RouteDataValueProvider

QueryStringValueProvider

HttpFileCollectionValueProvider

Invocando al Model Binder

Se puede invocar al model binding de manera explícita para actualizar las propiedades de cualquier objeto creado previamente.

```
public ActionResult CrearProducto()
{
    var producto = new Producto();
    UpdateModel(producto);

    GrabarProducto(producto);
    return View();
}
```

Devolviendo una respuesta

Los objetos derivados de la clase **ActionResult** nos permiten generar una respuesta al usuario.

Object Type	Descripción
ViewResult	Devuelve Html mostrando una vista.
RedirectToRouteResult	Lanza una redirección (HTTP 302) hacia una acción o ruta, generando la url de acuerdo a sistema de rutas.
FileResult	Transmite datos binarios al navegador.
JsonResult	Transforma un objeto .Net en JSON y lo devuelve como respuesta.
HttpStatusCodeResult	Nos permite establecer un código de respuesta HTTP. Ejm: 404 Not Found.

Retornando una Vista HTML

Para esto hacemos uso del tipo ActionResult.

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

La llamada View genera un ActionResult e intenta encontrar la vista en las siguientes ubicaciones

- /Views/[ControllerName]/[ViewName].aspx
- /Views/[ControllerName]/[ViewName].ascx
- /Views/Shared/[ViewName].aspx
- /Views/Shared/[ViewName].ascx

Enviando datos a la vista: ViewBag

Podemos utilizar el tipo dinámico ViewBag para enviar información desde el controller y recibirla en la vista.

```
public ActionResult DetalleProducto()  
{  
    var producto = new Producto {Nombre = "MVC Cookbook",  
                                   Precio = 15.5m };  
    ViewBag.Nombre = producto.Nombre;  
    ViewBag.Precio = producto.Precio;  
    return View();  
}
```

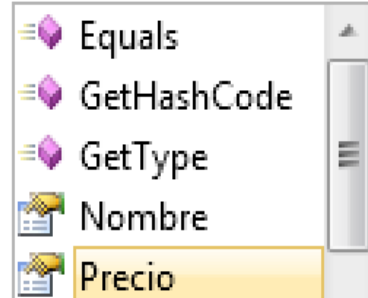
```
<h2>Nombre: @ViewBag.Nombre</h2>  
<strong>Precio: @ViewBag.Precio</strong>
```

Enviando un objeto "Strongly Typed"

Podemos pasar un objeto como parámetro al View y podremos acceder a este objeto a través de la propiedad Model desde la vista.

```
public ActionResult DetalleProducto()  
{  
    var producto = new Producto {Nombre = "MVC Cookbook",  
                                   Precio = 15.5m};  
    return View(producto);  
}
```

```
<h2>Nombre: @Model.Nombre</h2>  
<strong>Precio: @Model.</strong>
```



Ejercicio: Enviando datos utilizando el ViewBag

- En el área de administración, crear dos acciones (GET, POST) para agregar un nuevo producto.
- Enviar desde el controller una lista de categorías utilizando el ViewBag.
- Crear la vista para la acción anteriormente creada, esta debe ser fuertemente tipada al objeto Producto.
- Crear en la vista un DropDownList (<Select />) para asignar una categoría al producto.

Ejercicio: Enviando datos utilizando un ViewModel

- Crear un ViewModel para la edición de productos.
- Crear 2 nuevas acciones (GET, POST) para la edición de productos
- Crear la vista para la acción anteriormente creada, esta debe ser fuertemente tipada al viewmodel.

Redireccionando

Después de procesar información enviada desde un formulario, no se debe mostrar una vista, sino lanzar una redirección a una acción (Post-Redirect-Get Pattern)

```
public ActionResult Crear(Producto producto)
{
    GuardarProducto(producto);
    return View("Listar");
}
```

```
public RedirectToRouteResult Crear(Producto producto)
{
    GuardarProducto(producto);
    return RedirectToAction("Listar");
}
```

Mantener data a través redirecciones

Una redirección ocasiona un nuevo request y por lo tanto ya no se tiene a los valores anteriores a este. Si necesitamos mantener información entre request podemos utilizar el TempData.

```
public RedirectToRouteResult Crear(Producto producto)
{
    GuardarProducto(producto);
    TempData["Mensaje"] = "El nuevo producto ha sido guardado";
    return RedirectToAction("Listar");
}
```

```
@if (@TempData["Mensaje"]!=null){
    Mensaje: @TempData["Mensaje"]
}
```

Ejercicio: Mostrar un listado de productos desde una vista

- Crear un controller y vista que muestra un listado de todos los productos.
- La vista debe mostrar el nombre de la categoría del producto.
- Analizar el carga "lazy" y "eager" para las categorías de los productos.
- Mostrar un mensaje en el listado cuando se haya creado un nuevo producto correctamente.

Enviado archivos y datos binarios

ASP.NET MVC nos permite enviar archivos al navegador para que se mostrar su contenido o para que sean descargados.

Existen tres clases que extienden de **FileResult** nos permiten trabajar con datos binarios:

- **FilePathResult**: Archivos desde el File System.
- **FileContentResult**: Contenido desde un arreglo de bytes en memoria.
- **FileStreamResult**: Contenido que se encuentra en un objeto IO.Stream.

Utilizando el método File

El método File() nos permite retornar alguna de las clases que extienden de FileResult.

```
public ActionResult DescargarReporte()  
{  
    return File("c:\\archivo.pdf", "application/pdf", "Reporte.pdf");  
}
```

Parámetro	Descripción
file[Name,Content,Stream] (Obligatorio)	El nombre de archivo , arreglo binario o Stream de donde se obtendrá el contenido.
contentType (Obligatorio)	El tipo MIME que se enviará en la cabecera de la respuesta.
fileDownloadName (opcional)	Cuando este parámetro es especificado, se mostrará el dialogo de save-or-open.

Ejercicio: Agregar una Imagen al Producto

- Modificar la acción de edición del producto para que reciba una imagen y la guarde en un directorio.
- Crear una acción que recupere devuelva la imagen que le corresponde a un producto.
- En la edición del producto y el listado de la página principal, mostrar la imagen del producto solo si esta existe.


Filters

Nos permite inyectar comportamiento adicional al flujo de procesamiento de un request dentro de ASP.NET MVC.

```
[Authorize]  
public ActionResult AdministrarPedidos()  
{  
    return View();  
}
```

Tipos Básicos de Filtros

Nos permite inyectar comportamiento adicional al flujo de procesamiento de un request dentro de ASP.NET MVC.

```
[Authorize]  
public ActionResult AdministrarPedidos()  
{  
    return View();  
}
```


Tipos Básicos de Filtros

Filter Type	Interface	Default Imp.
Authorization Filter	IAuthorizationFilter	AuthorizedAttribute
Action Filter	IActionFilter	ActionFilterAttribute
Result Filter	IResultFilter	ActionFilterAttribute
Exception Filter	IexceptionFilter	HandleErrorAttribute

Filter Type	Cuando se lanza
Authorization Filter	Antes que el Action Method y antes que cualquier otro filtro.
Action Filter	Antes y después que el Action Method
Result Filter	Antes y después que el Action Result es ejecutado.
Exception Filter	Solo si algún filtro, Action Method o Action result lanza una exception

Creando un Nuevo Filtro

Para crear un nuevo filtro podemos utilizar cualquiera de las siguientes alternativas:

- Crear una clase que derive de `FilterAttribute` (clase base de todos los filtros) e implementar una o más de las 4 interfaces.
- Crear una clase que derive de alguno de los filtros ya existentes y sobre escribir alguno de sus métodos.

try{

Authorization Filters



Action Filters: Before Action

Action Method

Result Filters: After Action



Result Filters: Before Result

Ejecución Action Result

Result Filters: After Result

}

catch (**Exception**){

Exception Filters

}

Ejercicio: Creando un Filtro de Logging

- Agrega al proyecto web el archivo de configuración de log4net.
- Configurar log4net en el global.asax
- Crear un filtro que registre un mensaje en el log:
 - Antes de llamar a la acción.
 - Después que se ha llamado la acción.
 - Si se ha producido un error en la acción.
- Agregar el filtro a nivel de método y acción.

Algunos Built-in Filters

- **Authorize:** Nos permite indicar que un acción solo podrá ser accedida por un usuario autenticado.
- **HandleError:** Nos permite mostrar una vista especial cuando se produzca una excepción no manejada dentro la acción.
- **OutputCache:** Nos permite agregar a la cache el resultado devuelto por la acción.
- **RequireHttps:** Fuerza que una conexión no segura se muestre a través de HTTPS.

Ejercicio: Utilizando el filtro Authorize

- Crear un controller y sus acciones necesarias dentro de área administración, para mostrar y recibir los datos de un formulario de login.
- Crear una vista que contenga el formulario de login.
- Utilizar el filtro Authorize para no permitir que ningún usuario no logeado ingrese al área de administración.
- Configurar el web.config para activar la autenticación por formulario.

Global Filters

Podemos indicar que filtros serán aplicados a todas las acciones de la aplicación.

```
GlobalFilters.Filters  
    .Add(new HandleErrorAttribute());
```

Sin realizar ningún cambio en el controller, agregamos cualquier filtro a la colección GlobalFilters.Filters dentro del Global.asax.

VIEWS

ASP.NET MVC

Angel Núñez Salazar

Email: snahider@gmail.com
Blog: <http://snahider.blogspot.com>
Twitter: [@snahider](https://twitter.com/snahider)

Views

Son las responsables de tomar las salidas del controller y renderizarlas en un formato HTML.

```
@model Views.Models.Post

<html>
<head>
    <title>@Model.Titulo</title>
</head>
<body>
    @Model.Contenido
</body>
</html>
```

View Engine

Se encarga de realizar las siguientes tareas:

- Localizar la vista y proveerle la información necesaria (IViewEngine)
- Renderizar la vista (IView)
- Compilar el archivo de la vista en código ejecutable

Por defecto existen 2 view engines: **Razor y Webforms**, pero podemos extenderlas o reemplazarlas por otra diferente.

Razor ViewEngine

- Razor es un motor de plantillas de propósito general el cuál no está acoplado a ASP.NET.
- RazorViewEngine ha sido creado con el objetivo de simplificar la creación de vistas en ASP.NET MVC.

```
<% foreach (var post in Model) { %>
    @foreach (var post in Model) {
        <tr>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=post.Id }) |
                @Html.ActionLink("Delete", "Delete", new { id=post.Id })
            </td>
            <td>
                @post.Titulo
            </td>
            <td>
                @post.Contenido
            </td>
        </tr>
    }
}
```

Agregar contenido dinámico

ASP. NET MVC nos ofrece muchas opciones para agregar contenido dinámico dentro de nuestras vistas.

Técnica	Cuando usarlo
Inline Code	Mostrar salidas en la respuesta o colocar pequeñas porciones de lógica en la vista (if,foreach)
HTML Helpers	Generar tags HTML en base a la información del ViewData o Model
Partial Views	Compartir segmentos HTML a través de múltiples vistas.
Child Actions	Secciones o widgets reusables que pueden contener lógica de negocio.

Inline Code

Es la manera más fácil de agregar contenido dinámico a la vista. Se **precede el símbolo @ antes del código** que necesitamos insertar.

```
<h1>@Model.Titulo</h1>
<p>@Model.Contenido</p>
<ul>
    @foreach (var comentario in Model.Comentarios)
    {
        <li>@comentario.Texto</li>
    }
</ul>
<div>
    <p>Autor: Angel Núñez Salazar</p>
    <p>Email: snahider@gmail.com</p>
    <p>Twitter: @@snahider</p>
</div>
```

Html Helpers

Métodos que generan fragmentos HTML que son comúnmente usados.

```
<input type="text" id="titulo" name="titulo"  
value="@ViewBag.Titulo" />
```

```
@Html.TextBox("titulo")
```

- Podemos acceder a los helpers a través de la propiedad `HTML(System.Web.Mvc.HtmlHelper)` de la vista.
- Buscan si la información está presente dentro de `ViewBag`, `ViewData` o `Model`.

Built-In Helpers

- Existen alrededor de 50 helpers que podemos utilizar para generar diferentes tags HTML. Ejm:

CheckBox, Hidden, RadioButton, Password, TextBox, TextArea...

```
@Html.TextBox("usuario")  
@Html.Password("contraseña")  
@Html.CheckBox("recordarContraseña")
```

- Si la vista es fuertemente tipada, podemos utilizar helpers tipados que funcionan exactamente igual a los basados en cadenas.

```
@Html.TextBoxFor(x => x.Usuario)  
@Html.PasswordFor(x => x.Contraseña)  
@Html.CheckBoxFor(x => x.RecordarContraseña)
```

Agregando atributos HTML

- Podemos agregar una lista arbitraria de atributos html pasando un objeto anónimo por el parámetro *htmlAttributes*.

```
@Html.TextBox("titulo", "Mi primer post",  
    new { atributo = "valor" })  
  
<input type="text" name="titulo"  
    value="Mi primer post" atributo="valor" />
```

- Para incluir una palabra reservada dentro del lenguaje C# podemos utilizar el símbolo @.

```
new { @class = "cssClass" }
```


Form Tag Helpers

Tienen como ventaja la generación de la url correcta en base a una acción y controller.

```
@using (Html.BeginForm("editar", "productos",  
    new { id=Model.Id}))  
{  
    @*Elementos del formulario*@  
}
```

Si no ingresamos ningún parámetro, se utilizará la URL actual de la página.

Creando un Helper Method

- Extender la clase HtmlHelper utilizando extensiones de métodos.
- Retornar una clase del tipo MvcHtmlString

```
public static MvcHtmlString Video(this HtmlHelper htmlHelper,
                                string src){
    var videoTag = @"<video src='{0}' controls='controls'>
                    Tag no soportado
                    </video>";
    return MvcHtmlString.Create(String.Format(videoTag, src));
}
```

- Uno de los siguientes pasos para utilizar el helper:
 - Agregar `@using {namespace}` al inicio de la vista.
 - Agregar `<add namespace="{namespace}" />` en el nodo `system.web.webpages.razor/pages/namespace` dentro del archivo `/views/web.config`.

Ejercicio: Helper para la paginación de productos.

- Crear una clase PagedList que maneje los datos de la paginación.
- Crear una extensión de método para pasar los resultados de la bd a la clase PagedList.
- Crear un HtmlHelper que muestre links para navegar a través del paginado utilizando la información de la clase PagedList.
- Utilizar el nuevo Helper en el listado de productos de la parte pública de la tienda.

Partial Views

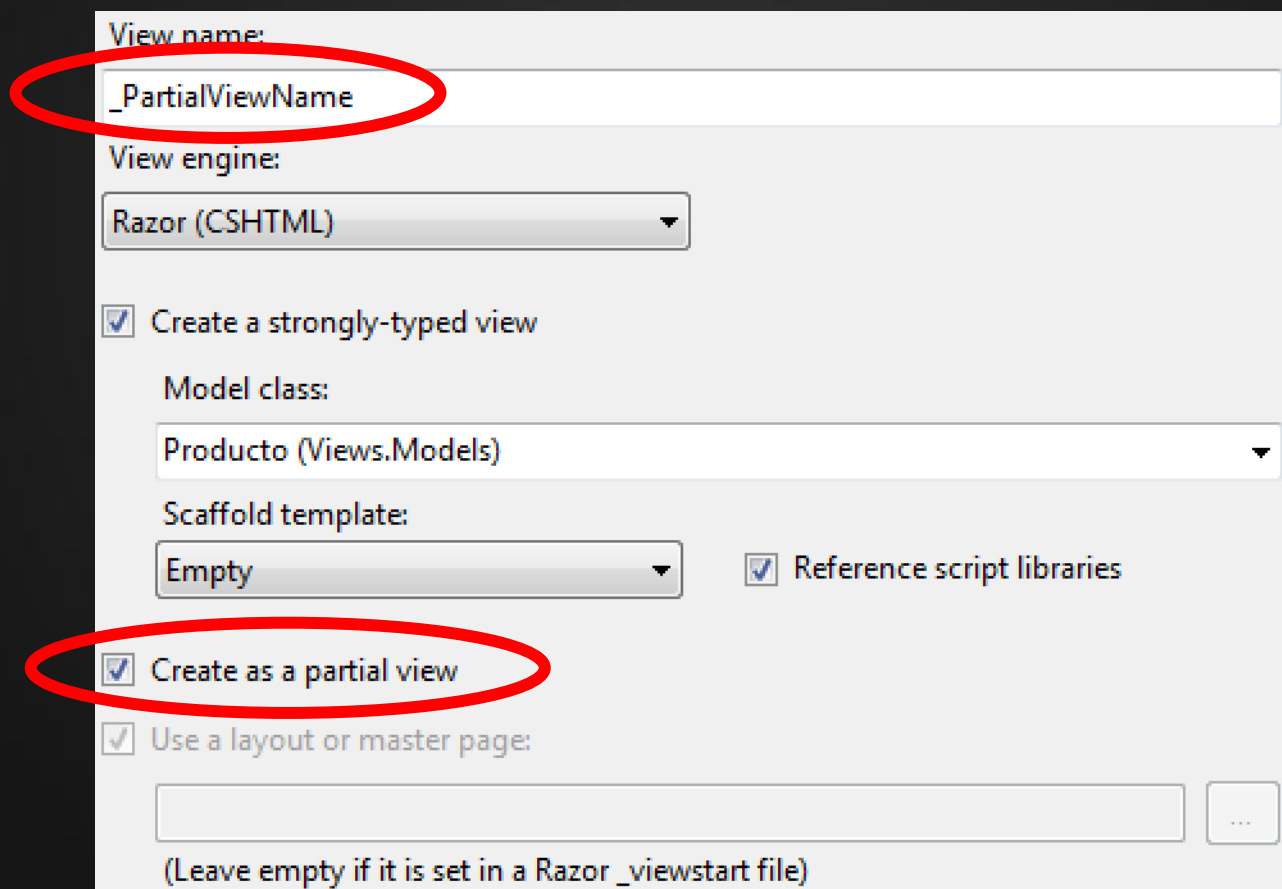
- Sirven para **compartir segmentos HTML** a través de múltiples vistas.
- Muy similares a los html helpers con la diferencia que **se usa la sintaxis del view engine y no código c#.**
- Muy similares a las vistas normales y dentro también **se puede utilizar el Model y el ViewBag.**

```
@model Views.Models.Producto

<div>
    @Model.Nombre
    @Model.Precio
</div>
```

Crear un Partial View

- Seleccionando la casilla **create as partial view** al momento de crear una vista.
- Se utiliza la sintaxis "_{nombre}" para el nombre.



The screenshot shows the 'Add View' dialog box in an IDE. Two red circles highlight specific settings: the 'View name' field and the 'Create as a partial view' checkbox. The 'View name' field contains the text '_PartialViewName'. The 'View engine' is set to 'Razor (CSHTML)'. The 'Create a strongly-typed view' checkbox is checked, and the 'Model class' is set to 'Producto (Views.Models)'. The 'Scaffold template' is set to 'Empty'. The 'Reference script libraries' checkbox is checked. The 'Create as a partial view' checkbox is checked. The 'Use a layout or master page' checkbox is checked. The 'Layout' field is empty. The text '(Leave empty if it is set in a Razor _viewstart file)' is at the bottom.

View name:
_PartialViewName

View engine:
Razor (CSHTML)

☒ Create a strongly-typed view

Model class:
Producto (Views.Models)

Scaffold template:
Empty

☒ Reference script libraries

☒ Create as a partial view

☒ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

Mostrar un Partial View

Para mostrar un partial view hacemos uso del helper `Html.Partial()`.

```
@Html.Partial("_producto")
```

Por defecto se buscará el partial view en las siguientes ubicaciones:

- `/Views/[ControllerName]/[PartialView].cshtml`
- `/Views/Shared/[ViewName].cshtml`

Si el partial view es fuertemente tipado, podemos pasarle su Model como segundo parámetro.

```
@Html.Partial("_producto", producto)
```

Ejercicio: Producto Partial View

- Crear un partial view para mostrar la información de un producto.
- Del listado de productos en la vista principal, cortar el segmento html que corresponde a la información de un producto y pegarlo en el partial view creado.
- Desde el mismo listado de productos, hacer un llamado al partial view por cada producto de la iteración.

Child Actions

- Tipicamente las acciones devuelven vistas que devuelven una respuesta HTML, pero las vistas a su vez pueden llamar a acciones (child actions) para que el contenido devuelto sea inyectado en la respuesta principal.
- Cuando realizamos una llamada a una child action, se inicia otro nuevo request que pasará por toda el pipeline MVC.
- Sirven para mostrar secciones de información que no tienen relación con el contenido principal.

Utilizando un Child Action

- Para llamar a una child action desde la vista se utiliza el helper `Html.Action()`.

```
@Html.Action("Menu", "Navegacion")
```

- Podemos restringir que una acción solo sea child.

```
[ChildActionOnly]  
public ActionResult Menu()
```

- La vista devuelta por la acción no debe tener un **master page** o utilizar el action result `PartialView` desde la acción.

```
public PartialViewResult Menu(){  
    return PartialView();  
}
```

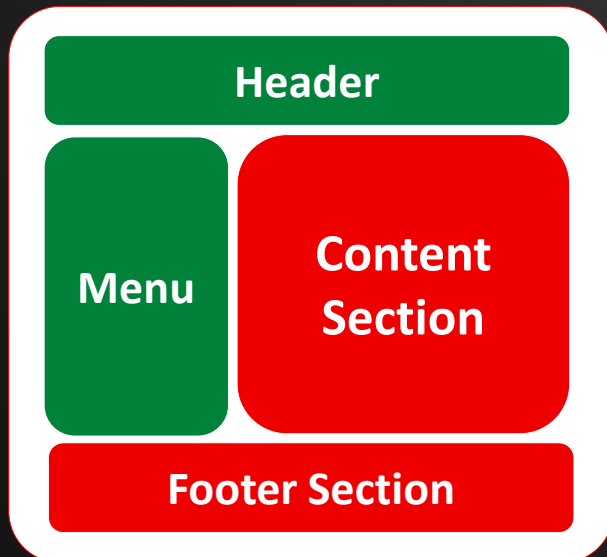
Ejercicio: Menu de Categorías utilizando un Child Action

- Crear un controller y una acción que nos devuelva una vista con todas las categorías que existen.
- Crear una vista que muestre un menú de navegación utilizando las categorías que vienen del controller.
- Realizar una llamada al child action que devuelve la vista que muestra el menú.

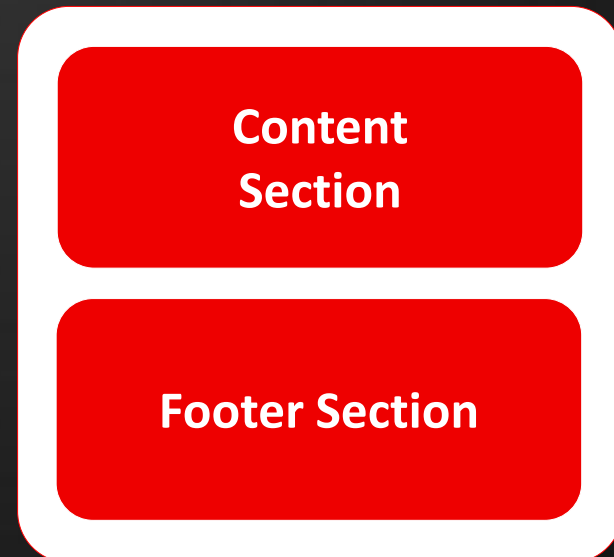
Layout Views

- Podemos colocar los elementos comunes que existen en la interfaz web dentro de unas plantillas denominadas layout views.
- A través de otras vistas (content views) podemos llenar únicamente los contenidos faltantes de la plantilla.

Layout Views

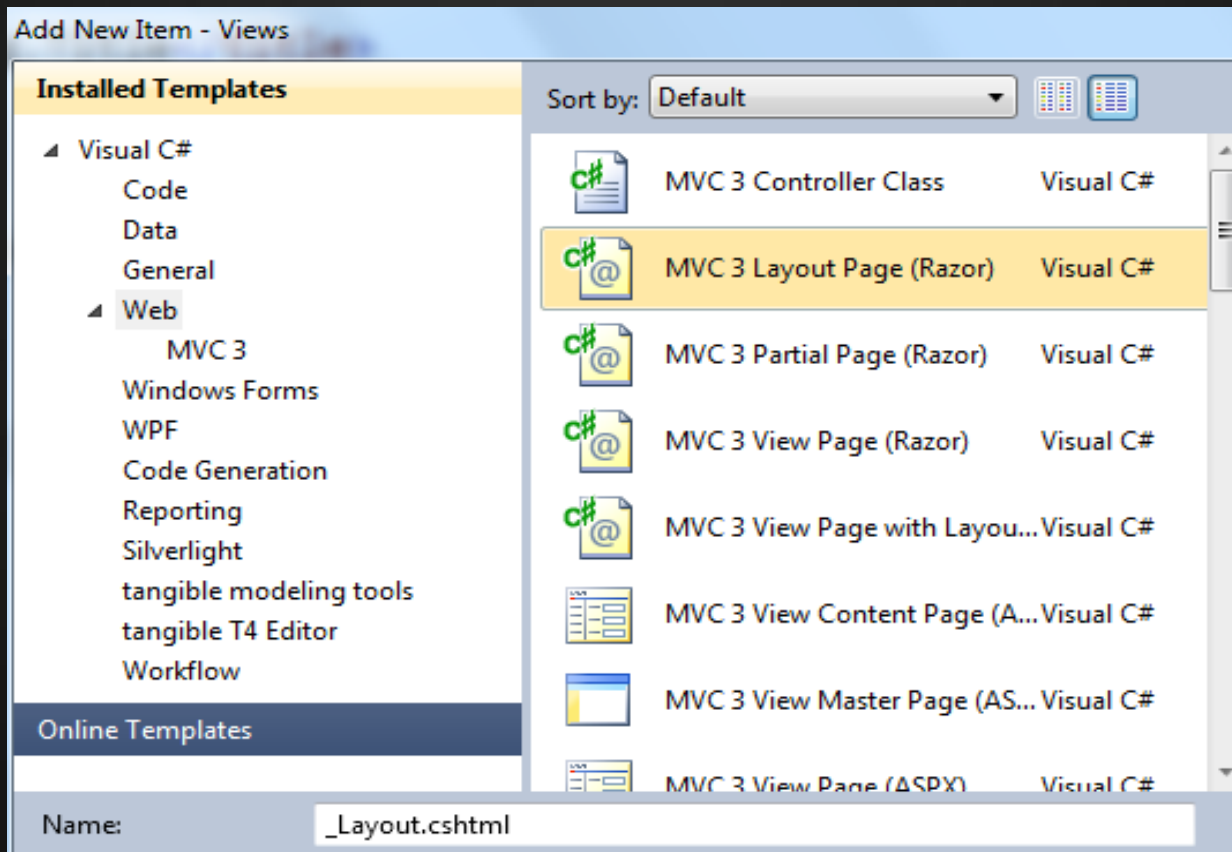


Content Views



Creando un Layout View

- Podemos crear un layout haciendo clic derecho -> add -> new item -> y seleccionar Layout Page.
- La sintaxis del nombre es "_{layout_name}".
- Por convención se utiliza la carpeta /views/shared.



Editar un Layout View

Podemos utilizar las sentencias ViewBag, RenderBody y RenderSection para especificar la información que deberá ser inyectada a través de un content view.

```
<html>
<head>
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
    <div>
        @RenderSection("footer")
    </div>
</body>
</html>
```

Content Views

- Para especificar que una vista será un content view se, debe escribir la ruta del layout al que pertenece, en caso contrario dar el valor de null.
- Debe contener las propiedades y las secciones obligatorias que se encuentran declaradas en el layout.

```
@{
    ViewBag.Title = "ContentView";
    Layout = @"~/Views/Shared/LayoutViews/
                _LayoutView.cshtml";
}

<h2>ContentView</h2>

@section footer {
    <strong>Footer</strong>
}
```

ViewStart File

- Es un archivo especial que contiene código que se ejecutará antes que se muestre una vista.
- Podemos tener más archivos ViewStart dentro de otras carpetas, el contenido que se encuentre en estos archivos sobrescribirá al ViewStart padre.
- Por defecto contiene la ruta se encuentra el layout.

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

Ejercicio: Creando un Layout View para el Área de Administración

- Crear un layout view para el área de Administración.
- Mover las secciones comunes de las vistas del área a el layout.
- En la vista LogOn, especificar la ruta del layout creado.
- Crear un archivo _ViewStart y especificar la ruta del layout para todas las vistas dentro del área.

MODELS

ASP.NET MVC

Angel Núñez Salazar

Email: snahider@gmail.com
Blog: <http://snahider.blogspot.com>
Twitter: [@snahider](https://twitter.com/snahider)

Models

- El término models para referirnos a los objetos que se transmiten entre los controllers y las vistas.
 - **Domain models** que encapsulan la lógica de negocio y son persistidos en la bd.
 - **View models** que nunca son persistidos y guardan estrecha relación con la información de la vista.
- No se exige una implementación especial para el dominio ni tampoco una tecnología de acceso a datos.
- ASP.NET MVC sin conocer nuestros modelos nos provee varias alternativas para manejar el ingreso y presentación de datos.

Template Helpers

Html Helpers

Sirven para indicar específicamente el HTML que se mostrará.

```
@Html.TextBoxFor(x => x.Nombre)
```

Template Helpers

Determinan un template en base a la información con la cuál ya cuentan los objetos

```
@Html.EditorFor(x => x.Nombre)
```

Si la propiedad es un string se mostrará una caja de texto.

Son útiles para establecer convenciones que se utilicen para determinar como mostrar un modelo. Por ejemplo podemos establecer que todos los tipos datetime se muestren como un calendario.

Built-In Helpers

Helper	Descripción
Html.Display Html.DisplayFor	Muestra html de solo lectura para una propiedad ingresada como parámetro, selecciona el template a mostrar en base al tipo de dato o metadata asociada.
Html.DisplayForModel	Muestra html de solo lectura para todas las propiedades del Model.
Html.Editor Html.EditorFor	Muestra html con el cuál se pueda editar la propiedad ingresada como parámetro, selecciona el template a mostrar en base al tipo de dato o metadata asociada.
Html.EditorForModel	Muestra html de edición para todas las propiedades del Model.

Ejemplo de vistas equivalentes

Vistas

```
<div class="editor-label">  
    @Html.LabelFor(x => x.Nombre)  
</div>  
<div class="editor-field">  
    @Html.EditorFor(x => x.Nombre)  
</div>  
    <div class="editor-label">  
        @Html.LabelFor(x => x.ExisteStock)  
</div>  
<div class="editor-field">  
    @Html.EditorFor(x => x.ExisteStock)  
</div>  
<input type="submit" value="Enviar" />
```

```
@Html.EditorForModel()  
<input type="submit" value="Enviar" />
```

Resultado

Nombre

ExisteStock

☐

Enviar

Templates

- Son partial views que contienen el html con el cuál se mostrará determinado objeto.
- El template aducado se revuelven en base a un algoritmo en el cuál interviene el tipo de dato y metadata adicional.
- Existen 2 categorías **DisplayTemplates** y **EditorTemplates**.
- Cuando ya se ha resuelto cuál es el template adecuado, este se busca en las siguientes rutas:
~/Views/ControllerName/{categoriatemplate}/{nombretemplate}.cshtml
~/Views/Shared/{categoriatemplates}/{nombretemplate}.cshtml

Default Templates

- Ya existen varios templates que podemos utilizar.
Boolean,Decimal,String,Object,EmailAddress...
- Los default templates son realmente código, pero podríamos ilustrarlos de la siguiente manera.

EditorTemplates/String.cshtml

```
@Html.TextBox("", ViewData.TemplateInfo.FormattedModelValue,  
    new { @class = "text-box single-line" })
```

- Podemos sobre escribir su contenido, creando un partial view con el nombre del template a sobre escribir y colocándolo en la carpeta adecuada.

Ejm : /Views/Shared/EditorTemplates/String.cshtml

Ejercicio: Explorando los Default Templates

- Modificar la vista de Login en el área de administración para que utilice template helpers.
- Colocar explícitamente el nombre del template para mostrar la etiqueta de password adecuada
- Descargar MvcFutures y explorar los default templates object, string y password.
- Sobre escribir el template de String para que utilice otra clase.

Metadata

Podemos declarar metadata en los objetos para influenciar como estos se muestran o editan. Esta metadata se declara a través de atributos o data annotations.

```
public class Producto
{
    [DisplayName("Nombre del Producto")]
    public string Nombre { get; set; }

    [DisplayFormat(DataFormatString = "{0:c}")]
    public decimal Precio { get; set; }
}
```

Data Annotations

Attributo	Descripción
[DisplayName]	El nombre a mostrar de la propiedad.
[DisplayFormat]	El fomato a mostrar para el valor de la propiedad.
[UIHint]	El nombre del template a utilizar.
[DataType]	Información sobre el tipo de dato de la propiedad. Ejm: Que un string es realmente un EmailAddress
[ScaffoldColumn]	Flags para indicar si la prooiedad se usará al momento en los templates.

Custom Templates

- Crear un partial view en la carpeta de templates adecuada.
Ejm: Views/Shared/EditorTemplates/DateTime.cshtml
- Para acceder al valor del elemento:
 - Si el template es fuertemente tipado podemos utilizar la propiedad Model, pero al utilizar esta propiedad saltamos el formato establecido en la metadata.
 - Utilizando ViewData.TemplateInfo.FormattedModelValue que nos dará el formato y valor correcto.

DisplayTemplates/EmailAddress.cshtml

```
<a href="mailto:@Model">  
    @ViewData.TemplateInfo.FormattedModelValue  
</a>
```

Búsqueda de Templates

1. Template explícito en el helper.

`Html.EditorFor(x=>x.Password,"password")`

2. Template explícito en la metadata.

`[UIHint("password")]`

3. El data type name especificado en la metadata.

`[DataType(DataType.EmailAddress)]`

4. El nombre del tipo de dato.

`DateTime,Decimal,Boolean.....`

5. Si el objeto no es complejo (puede ser convertido a string): **String**

6. Si el objeto es complejo y no interfaz, escala a través de todos los tipos base.

7. Si el objeto implementa IEnumerable: **Collection**

8. Finalmente utiliza: **Object**

Ejercicio: Crear un DateTime Template

- Modificar el LoginViewModel para que utilice metadata para encontrar el template de password.
- Crear un nuevo template DateTime que muestre un calendario de selección de fecha.
- Agregar una nueva propiedad DateTime al objeto EditarViewModel.
- Modificar la vista de edición de producto para editar la nueva propiedad del viewmodel, utilizar un template helper para mostrar el tag adecuado.

Validación

- Dentro de las validaciones podemos encontrar los siguientes casos:
 - Exigir la presencia o formato de la información ingresada por el usuario a través de la interfaz gráfica.
 - Determinar si un objeto se encuentra en un estado inválido.
 - Aplicar reglas de negocio para prevenir que ciertas operaciones se lleven a cabo.
- ASP.NET MVC cuenta con un sistema de validación que se ejecuta tanto a nivel de cliente como servidor.
- Podemos mostrar los errores producidos en la vista a través de html helpers.

Validación Manual

- Podemos registrar errores directamente en el ModelState.
- La propiedad IsValid del ModelState nos permite verificar si existe algún error registrado.

```
public ActionResult Crear(Producto producto)
{
    if (string.IsNullOrEmpty(producto.Nombre))
        ModelState.AddModelError("Nombre", "Por favor, ingrese el nombre");

    if (producto.Precio == 0)
        ModelState.AddModelError("Precio", "El precio debe ser mayor a 0");

    if (!ModelState.IsValid)
    {
        return View();//si hay error retorna a la misma vista
    }

    return RedirectToAction("Index");
}
```

Data Annotations

- ASP.NET MVC está configurado por defecto para registrar reglas utilizando atributos en el model.

```
public class Producto
{
    [Required(ErrorMessage = "Por favor, ingrese el nombre")]
    public string Nombre { get; set; }

    [Range(1, 1000000, ErrorMessage = "El precio debe ser mayor a 0")]
    public decimal Precio { get; set; }
}
```

- Ya existen las siguientes data annotations de validación:
Required, Range, StringLength, RegularExpression, Compare....

Custom Attributes

Extendiendo ValidationAttribute y sobre escribiendo IsValid.

```
public class FileExtensionsAttribute : ValidationAttribute
{
    private readonly string extensions;
    public FileExtensionsAttribute(string extensions){
        this.extensions = extensions;
        ErrorMessage = "{0} no es una extensión válida";
    }

    public override bool IsValid(object value){
        return extensions.Split('|').Contains(value);
    }
}
```

Extendiendo cualquier atributo ya existente.

```
public class EmailAttribute : RegularExpressionAttribute
{
    private const string EmailPattern = @"^([0-9a-zA-Z]([-\.\\w]*[0-9a-zA-Z])*"
        @"([0-9a-zA-Z]([-\\w]*[0-9a-zA-Z]\\.)+[a-zA-Z]{2,9}))$";
    public EmailAttribute(): base(EmailPattern){}
}
```

ValidationContext

Nos permite acceder al objeto que se está validando con el fin de acceder a información de otra propiedad diferente a la del atributo.

```
public class GreaterThanNumberAttribute : ValidationAttribute
{
    private readonly string otherPropertyName;
    public GreaterThanNumberAttribute(string otherPropertyName): base("{0} must be greater than {1}"){
        this.otherPropertyName = otherPropertyName;
    }

    public override string FormatErrorMessage(string name){
        return String.Format(ErrorMessageString, name, otherPropertyName);
    }

    protected override ValidationResult IsValid(object value, ValidationContext validationContext){
        var otherPropertyInfo = validationContext.ObjectType.GetProperty(otherPropertyName);
        var otherNumber = (int)otherPropertyInfo.GetValue(validationContext.ObjectInstance, null);
        var thisNumber = (int)value;
        if (thisNumber <= otherNumber)
            return new ValidationResult(this.FormatErrorMessage(validationContext.DisplayName));
        return null;
    }
}
```

Self Validation

La interfaz `IValidatableObject` nos permite agregar reglas adicionales dentro del objeto. Estas solo se ejecutan si no se ha encontrado ningún error en las reglas de los atributos.

```
public class RegistroNuevoUsuario : IValidatableObject
{
    [Required]
    public string Password { get; set; }
    [Required]
    public string ConfirmacionPassword { get; set; }

    public IEnumerable<ValidationResult>
        Validate(ValidationContext validationContext)
    {
        if (!Password.Equals(ConfirmacionPassword))
            yield return new ValidationResult(
                "Los passwords deben ser iguales",
                new[] { "ConfirmacionPassword" });
    }
}
```

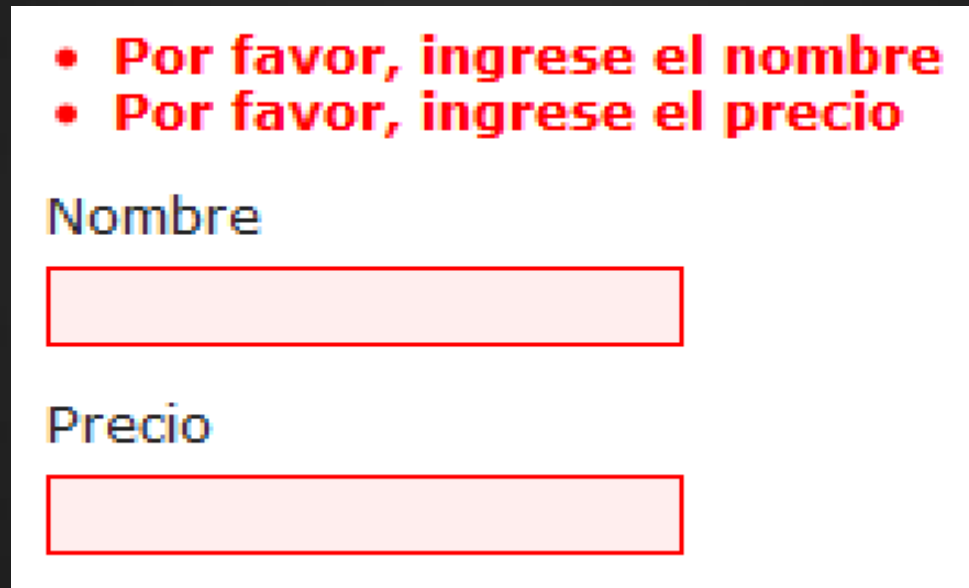
Fases de Validación

- Existen 2 fases dentro del proceso de validación:
 - El `DefaultModelBinder` asegura que valores los ingresados tengan el formato correcto antes de asignarlos al objeto.
 - Luego que el model binding se ha completado, se verifica si se cumplen reglas adicionales agregadas por código.
- Verificamos si existe algún error registrado, a través de cualquier técnica, utilizando `IsValid` del `ModelState`.

```
public ActionResult Editar(Producto producto)
{
    if (!ModelState.IsValid)
        return View();
    //Actualizar el producto en la BD
    return RedirectToAction("Index");
}
```

Mostrando los Errores en la Vista

- El helper `Html.ValidationSummary()` nos permite mostrar un listado de todos los errores (propiedad y modelo)
- Podemos excluir los errores a nivel de propiedad.



A screenshot of a web form with two input fields. Above the fields, there are two red error messages: "• Por favor, ingrese el nombre" and "• Por favor, ingrese el precio". The first field is labeled "Nombre" and the second is labeled "Precio". Both fields have a red border and a light red background, indicating they are required and currently empty.

- Los helpers verifican los errores registrados en el ModelState y se auto asignan las clases CSS: `input-validation-error` y `validation-summary-errors`.

Mostrando Errores Individuales

De manera alternativa podemos utilizar el helper **Html.ValidationMessage** para mostrar los errores relacionados a una sola propiedad.

```
<div>
    Nombre: @Html.EditorFor(x => x.Nombre)
    @Html.ValidationMessageFor(x => x.Nombre)
</div>
<div>
    Precio: @Html.EditorFor(x => x.Precio)
    @Html.ValidationMessageFor(x => x.Precio)
</div>
```

Nombre: Por favor, ingrese el nombre
Precio: Por favor, ingrese el precio

Ejercicio:

Agregar Validación al Login

- Agregar atributos que validen que todas las propiedades del LoginViewModel son obligatorias.
- En el acción de Login devolver un mensaje si la autenticación es inválida.
- En el acción de Login regresar a la misma vista si se ha producido algún error.
- Modificar la vista de Login para mostrar los mensajes de error.

Ejercicio:

Agregar Validación al Crear Producto

- Agregar atributos que validen que todas las propiedades del `EditarProductoViewModel` son obligatorias.
- Crear un nuevo atributo que valide un valor mínimo para el precio del producto.
- Modificar la acción `Crear` del `HomeController` para que retorne la misma vista si se produce un error.
- Modificar la vista de `Login` para mostrar los mensajes de error.

Client Side Validation

- ASP.NET MVC tiene la capacidad de generar scripts de validación en el cliente utilizando la metadata del modelo.
- Para habilitar este comportamiento a nivel de vista se utiliza los helpers **Html.EnableClientValidation** y **Html.EnableUnobtrusiveJavaScript** o directamente para todas las vistas editando el web.config.

```
<appSettings>  
  <add key="ClientValidationEnabled" value="true"/>  
  <add key="UnobtrusiveJavaScriptEnabled" value="true"/>  
</appSettings>
```

- Se debe agregar los siguientes scripts a las vistas

```
<script src="jquery-1.4.4.min.js" type="text/javascript" />  
<script src="jquery.validate.min.js" type="text/javascript" />  
<script src="jquery.validate.unobtrusive.min.js" type="text/javascript" />
```

Como funciona la Validación de Cliente

- El Html.BeginForm crea un contexto donde se almacena metadata necesaria en el cliente
- Los helpers dentro del formulario acceden a la metadata del contexto para renderizar **atributos data-***.



```
[Required]  
public string Nombre { get; set; }
```

```
@Html.TextBoxFor(x => x.Nombre)  
@Html.ValidationMessageFor(x => x.Nombre)
```

```
<input type="text" value="" name="Nombre" id="Nombre"  
data-val-required="The Nombre field is required."  
data-val="true"/>  
<span data-valmsg-replace="true" data-valmsg-for="Nombre"  
class="field-validation-valid"></span>
```

Ejercicio:

Validación Cliente Area Administración

- Activar la validación a nivel de cliente en el Web.config.
- Agregar los scripts necesarios de validación en el master page.
- Probar la validación de cliente en alguno de los formularios del área de administración.
- Desabilitar y habilitar "Unobtrusive Validation" para ver la diferencia en los resultados.

Custom Client Validation

- Por defecto solo los atributos previamente creados pueden generar información de validación en el cliente.
- Para extender la generación de información a nuevos atributos se deben seguir los siguientes pasos:
 - Implementar la interface `IClientValidatable` en nuestro custom attribute.
 - Crear un nuevo método de validación en el cliente utilizando la API de JQuery Validation.
 - Implementar un adaptador utilizando JS que pase la información de los atributos `data-*` a la metadata de JQuery Validation.

Ejercicio:

Validación Cliente en Nuevos Atributos

- Implementar la interface `IClienteValidatable` en `MinAttribute` y sobre escribir el método `GetClientValidationRules`.
- Crear un nuevo archivo javascript e implementar el método de validación y el adaptador para el `MinAttribute`.
- Agregar el script a la vista de creación de Producto y probar la funcionalidad.

Remote Validation

- ASP.NET MVC nos permite realizar llamadas asíncronas al servidor para realizar validaciones que no se puedan realizar a nivel de cliente.
- Podemos utilizar el atributo remote e indicar un controller y acción que se encargarán de realizar la validación de manera asíncrona.

```
[Remote("VerificarNombreUsuario", "Usuario")]
```

```
public class UsuarioController : Controller
{
    public ActionResult VerificarNombreUsuario(string usuario)
    {
        var esValido = !usuario.Equals("Admin");
        return Json(esValido, JsonRequestBehavior.AllowGet);
    }
}
```

Ejercicio:

Verificar si el nombre ya existe

- Agregar el attribute remote en el nombre del producto.
- Crear la acción que se encargue de verificar que no existe un producto con el mismo nombre al momento de crear uno nuevo.

AJAX CLIENT SCRIPT ASP.NET MVC

Angel Núñez Salazar

Email: snahider@gmail.com
Blog: <http://snahider.blogspot.com>
Twitter: [@snahider](https://twitter.com/snahider)

Controllers and Ajax

Pueden devolver datos asíncronos de múltiples maneras:

- **ContentResult:** Retorna una cadena de texto.

```
return Content("<strong>Success!</strong>");
```

- **JsonResult:** Los datos serán transformados a formato json.

```
var productos = new[] {  
    new Producto{Nombre = "nombre1", Precio = "10"},  
    new Producto{Nombre = "nombre2", Precio = "20"}};  
return Json(productos);
```

```
[{"Nombre": "nombre1", "Precio": "10"}, {"Nombre": "nombre2", "Precio": "20"}]
```

- **ViewResult/PartialViewResult:** Retornará toda la vista (datos + html)

```
return PartialView("_Productos", productos);
```

Detectando Ajax Request

- ASP.NET MVC nos permite detectar si un request se está realizando por ajax.
- Permite tener en una misma acción lógicas distintas si es un request normal o ajax.

```
public ActionResult Delete()
{
    if (Request.IsAjaxRequest())
    {
        return Content("OK");
    }
    return RedirectToAction("Index");
}
```

Ajax Helpers

- Muy fáciles de usar y orientados a casos simples.
- Simplemente Inyectan el contenido devuelto dentro de la página y no permiten procesar los datos.
- Requieren los scripts: JQuery y jquery.unobtrusive-ajax

Ajax Helper	Descripción
Ajax.ActionLink Ajax.RouteLink	Muestran un link al igual que sus html helpers. Realizan llamadas asíncronas solicitando contenido para mostrar.
Ajax.BeginForm Ajax.BeginRouteForm	Muestran un formulario html similar a su correspondiente html helper. Envían asíncronamente los datos del formulario y muestran el contenido devuelto.

JQuery

- Librería javascript que facilita y simplifica la programación a nivel de cliente.
 - Selectores tipo CSS3.
 - API fluente para manipular el DOM.
 - Funciones AJAX muy simples.



- Es la librería principal de trabajo a nivel de cliente dentro de ASP.NET MVC.
- Ya viene integrado por defecto al crear un nuevo proyecto.

Ejercicio:

Eliminar un producto utilizando Ajax

- Realizar una acción que elimine un producto en el área de Administración.
- Agregar un link en el listado de productos para que llame a la acción previamente creada.
- Modificar el listado de productos para que realice la llamada a la acción de eliminar utilizando ajax.
- Modificar la acción de eliminar para que verifique si la llamada es por ajax y devuelva el resultado adecuado.