Desarrollador de Aplicaciones Web
Programación Web III

Departamento de Ingeniería e Investigaciones Tecnológicas

Introducción a ASP.NET MVC

Ing. Mariano Juiz
Ing. Matias Paz Wasiuchnik
Ing. Pablo Nicolás Sanchez

ASP.NET: Evolución

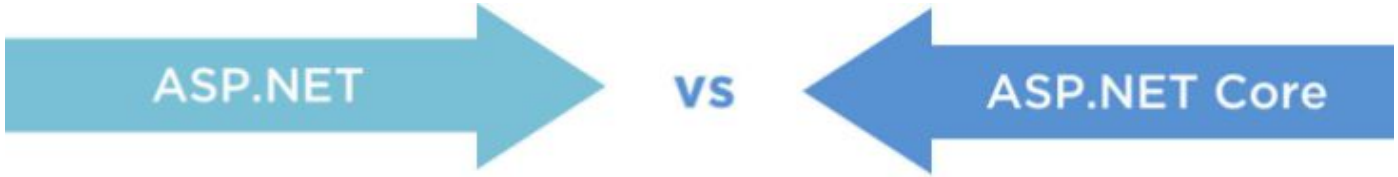
1996
Active Server Pages

2003
ASP.NET Web Forms

2009
ASP.NET MVC, Web Pages and Web API

2016
ASP.NET Core

ASP.NET vs ASP.NET Core



Criteria	ASP.NET	ASP.NET Core
Platform support	Windows only	Windows, Linux, Mac OS
Performance	High-performing framework	Demonstrated a better performance than ASP.NET
Architecture	Based on .NET Framework only	Based on .NET Framework and Core Framework
Components	Web Form, MVC, Web API	MVC, Web pages, Web API - no WebForms support
Dependencies	More dependencies, less monitoring options	A stricter control over the number of dependencies
Supported files	WCF, WF, WPF, VB, Web config and others	No support for Global.asax files and Web config, but appsettings.json is integrated
Isolation, modules, containers	Low level of isolation, not highly suitable for microservice and container development	Improved modular support, integration with docker, powerful isolation algorithms
Visual Studio Support	Supports all versions	Only the latest versions are supported starting from 2015

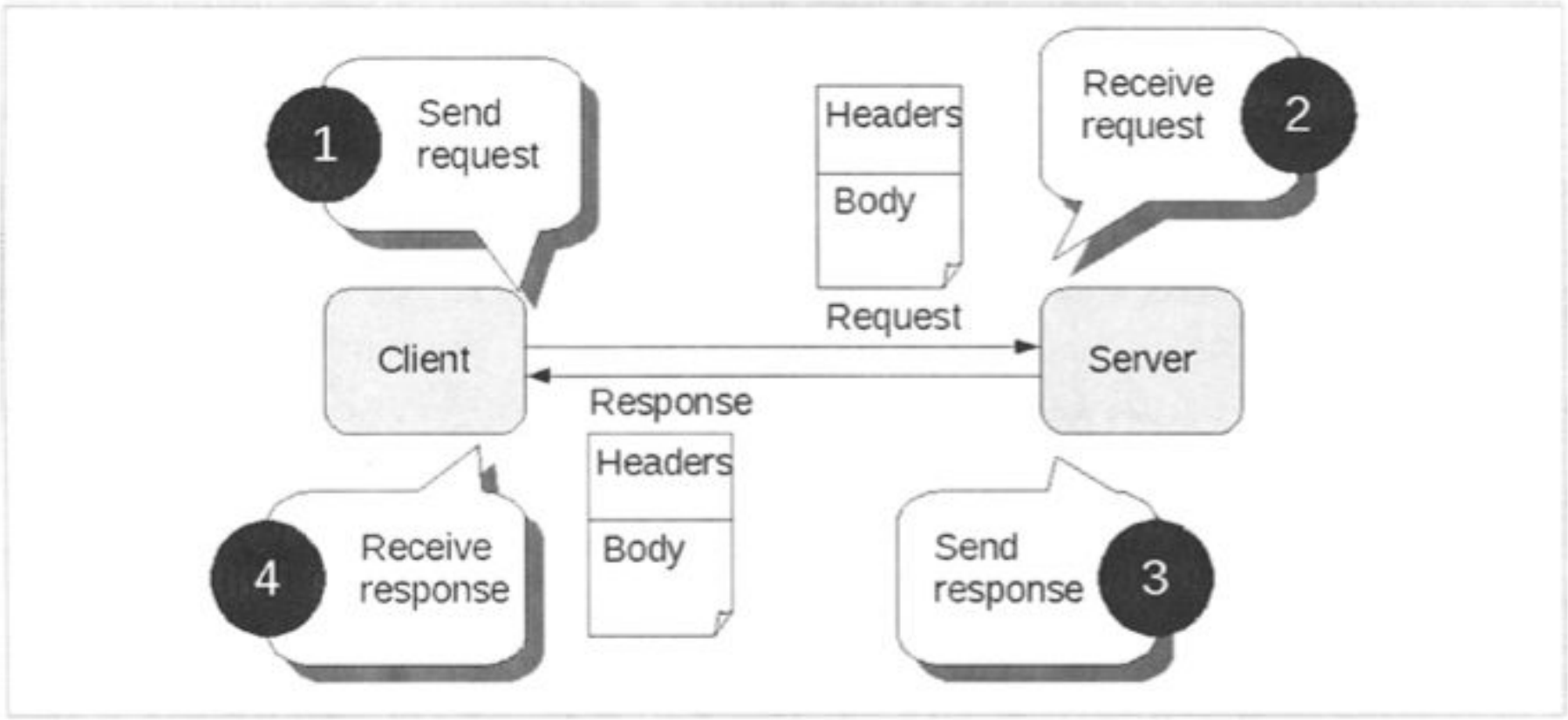
ASP.NET vs ASP.NET Core

Selección del marco

En la tabla siguiente se compara ASP.NET Core en ASP.NET 4.x.

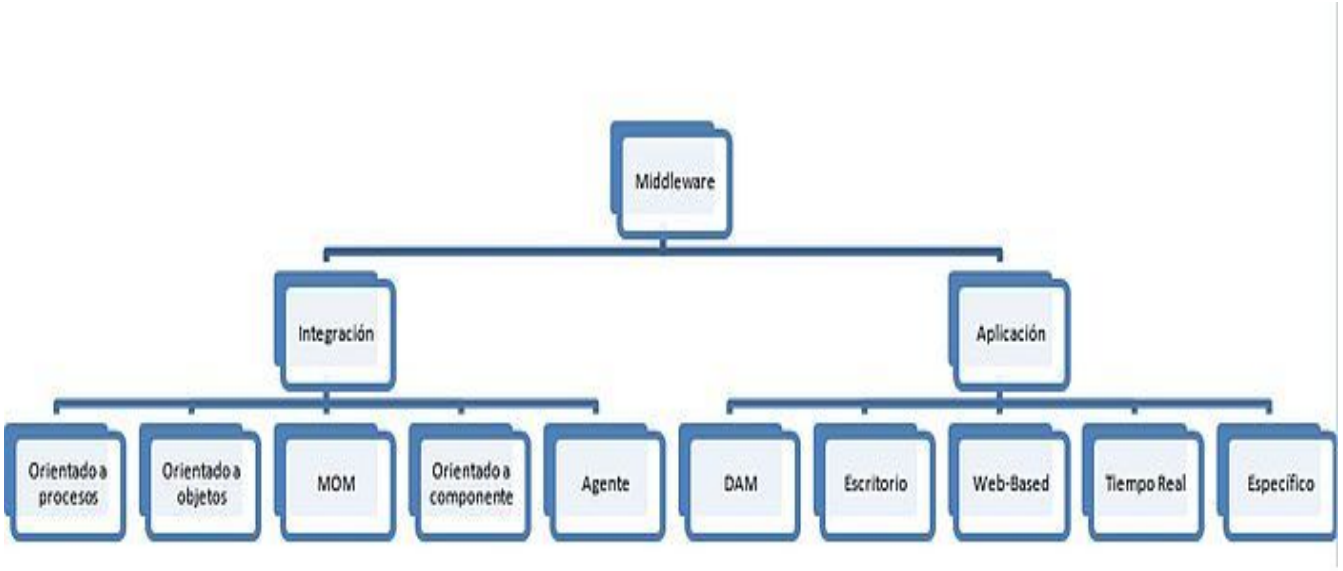
ASP.NET Core	ASP.NET 4.x
Compilación para Windows, macOS o Linux	Compilación para Windows
Razor Pages es el método recomendado para crear una interfaz de usuario web a partir de ASP.NET Core 2.x. Consulte también MVC, Web API y SignalR.	Use Web Forms, SignalR, MVC, Web API, WebHooks o Web Pages
Varias versiones por equipo	Una versión por equipo
Desarrollo con Visual Studio, Visual Studio para Mac o Visual Studio Code con C# o F#	Desarrollo con Visual Studio con C#, VB o F#
Mayor rendimiento que ASP.NET 4.x	Buen rendimiento
Uso del entorno de ejecución de .NET Core	Usar el tiempo de ejecución de .NET Framework

Repasando una vez más http:



Middleware

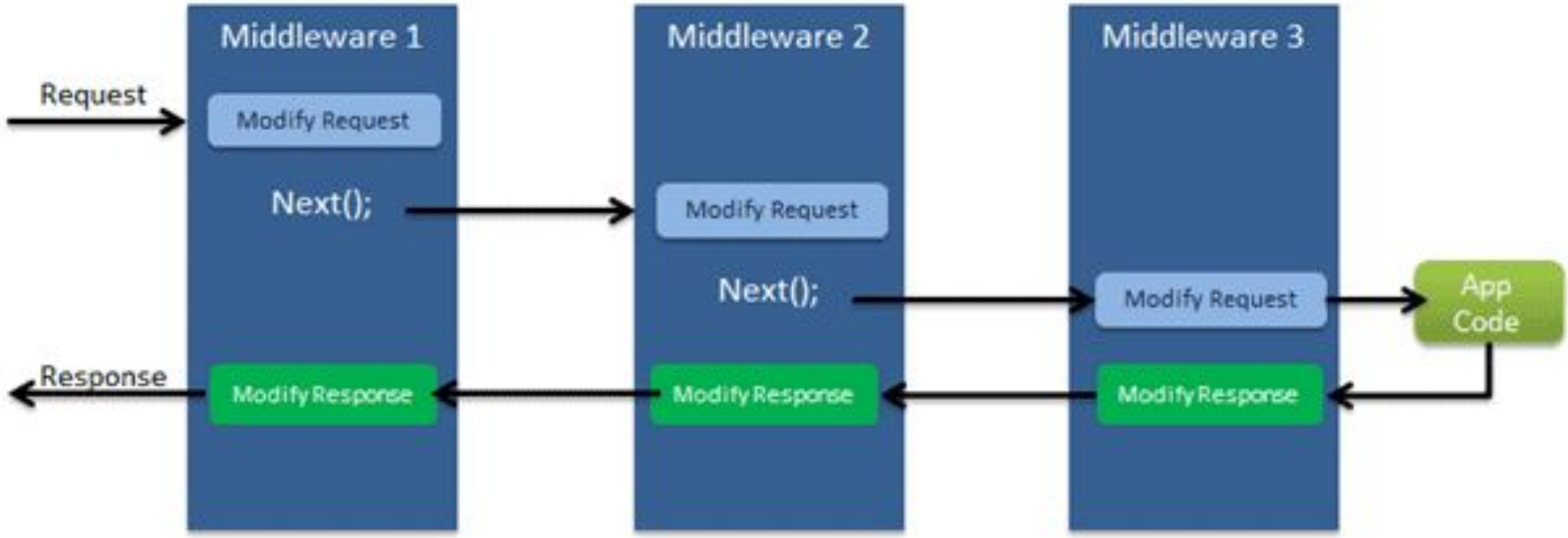
Middleware o **lógica de intercambio de información entre aplicaciones** o *Agente Intermedio*, es un *software* que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, o paquetes de programas, redes, *hardware* o sistemas operativos. Este simplifica el trabajo de los programadores en la compleja tarea de generar las conexiones y sincronizaciones que son necesarias en los sistemas distribuidos. De esta forma, se provee una solución que mejora la calidad de servicio, así como la seguridad, el envío de mensajes, la actualización del directorio de servicio, etc.



Middleware

Un middleware en ASP.NET Core es una porción de código que se incrusta en el ciclo de vida de un contexto HTTP, donde se puede ejecutar código antes y después de que la petición HTTP entre en un controlador, de esta manera, poder controlar las solicitudes y las respuestas.

Un middleware en ASP.NET Core no se ejecuta en un programa aparte. En ASP.NET Core estamos llenos de middlewares.



Middleware

Los Middleware son configurados en la Clase **Startup** en el método "**Configure**"

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
        });
    }
```

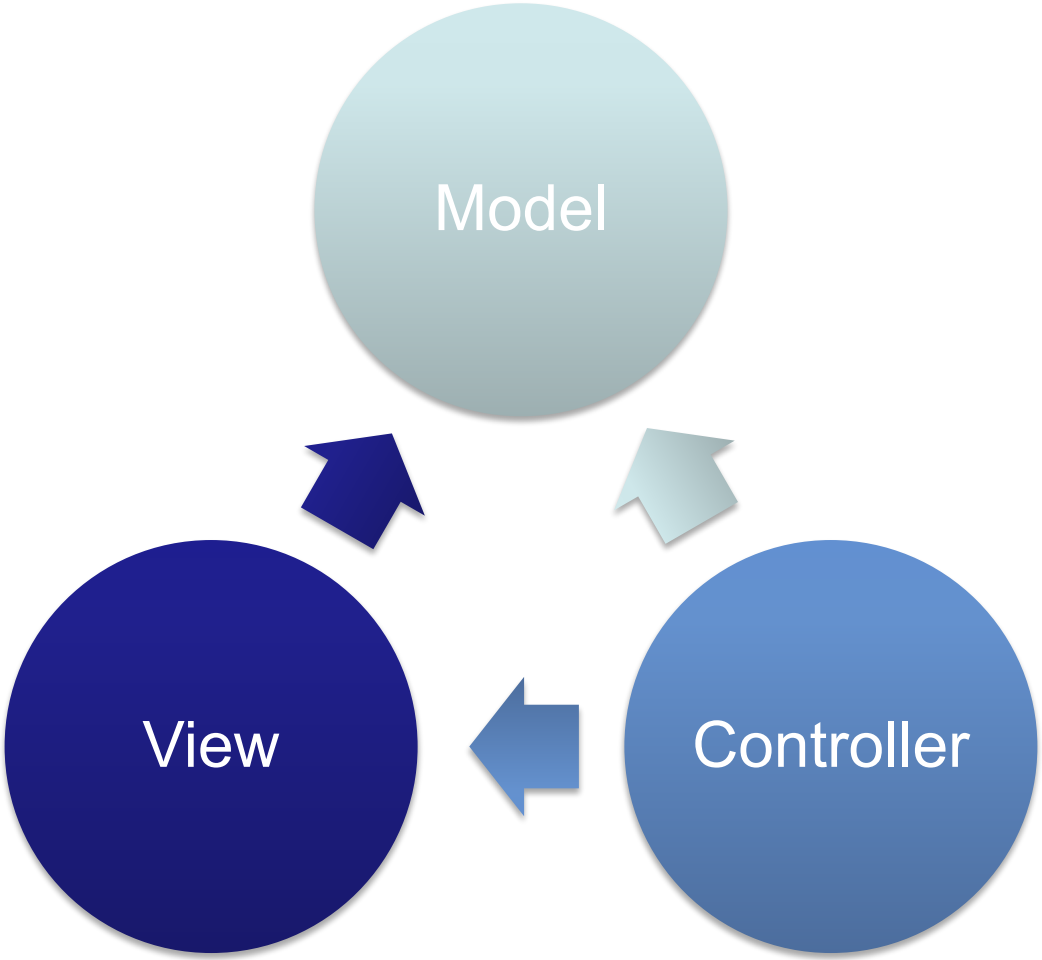
Patrón MVC

El patrón de diseño MVC (Modelo-Vista-Controlador) ha existido desde hace algunas décadas, y se ha usado en muchas tecnologías diferentes. Desde Smalltalk, C ++, Java, .NET sea ha utilizado este patrón de diseño para construir una interfaz de usuario.

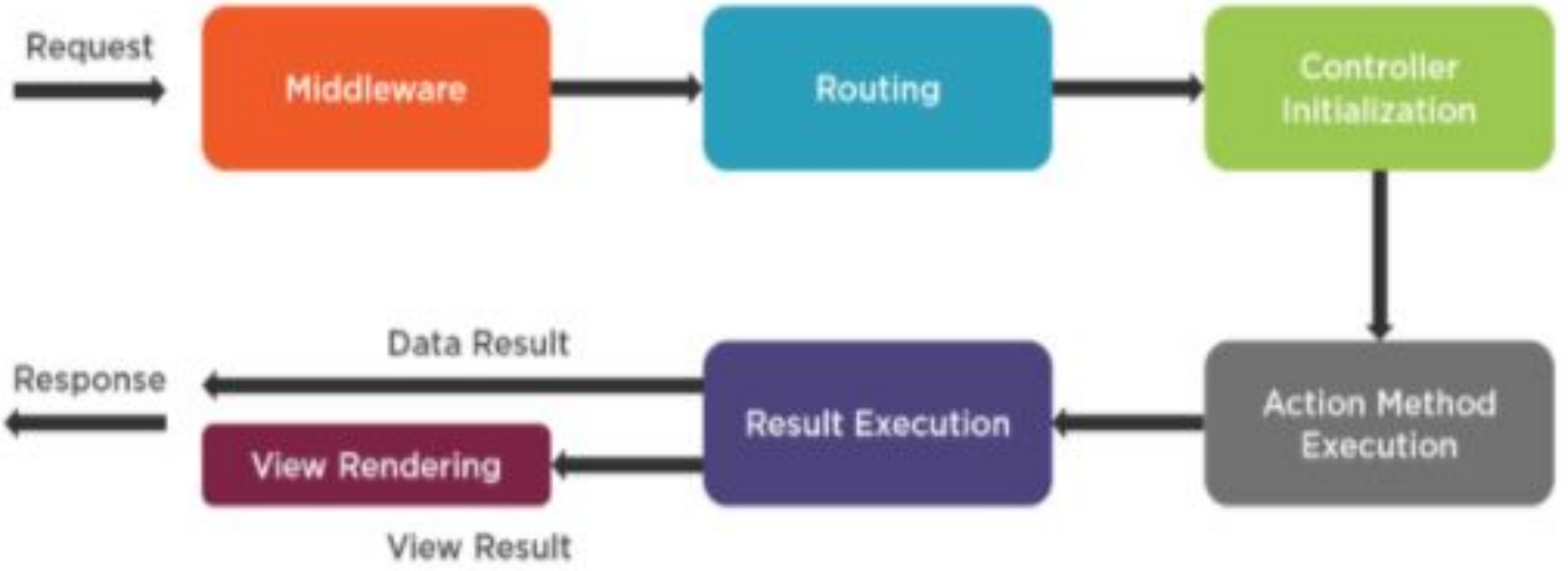
Es un medio poderoso y elegante de la separación de las responsabilidades dentro de una aplicación (por ejemplo, la separación de la lógica de acceso a datos de la lógica de visualización) y se aplica en sí muy bien a las aplicaciones web.

El patrón de arquitectura MVC separa la interfaz de usuario (UI) de una aplicación en tres partes principales: Modelo - Vista – Controlador.

Modelo Vista Controlador



Ciclo de vida:



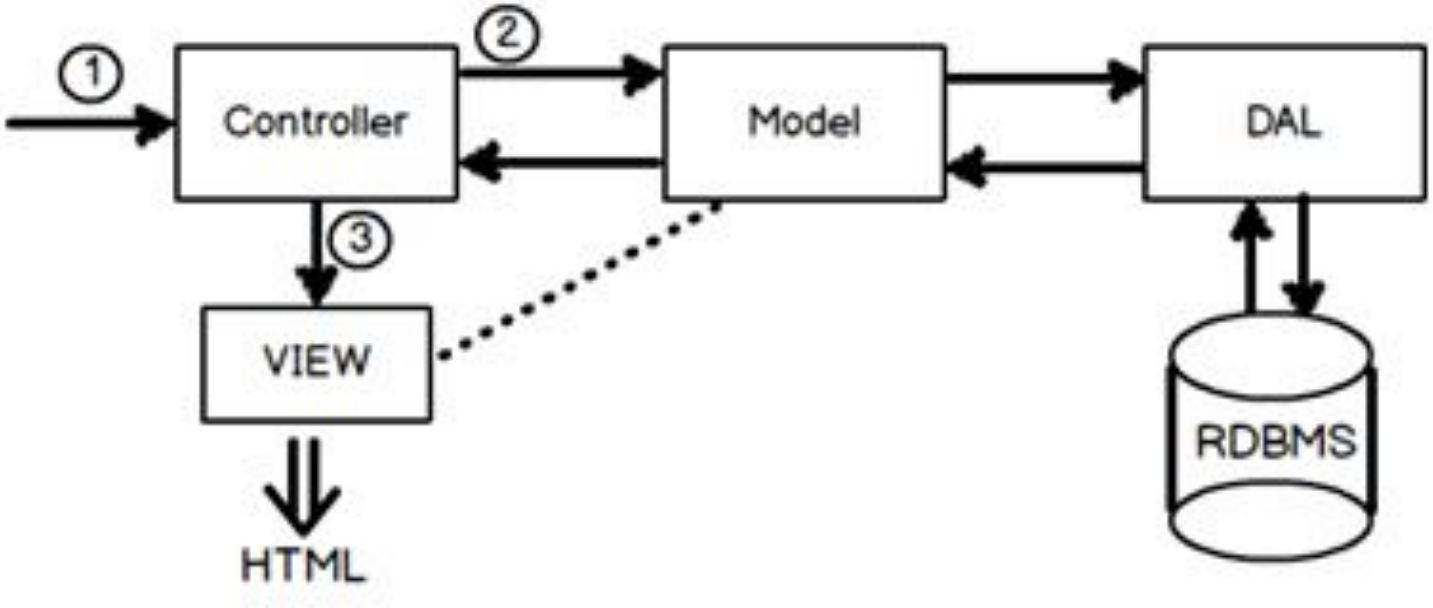
Ciclo de vida:

El servicio o marco MVC se encarga de resolver la ruta escogiendo un controlador concreto que puede manejar las peticiones. Una vez elegido el controlador, el siguiente paso importante es **la acción de ejecución.** Un componente llamado **invocador de acción encuentra y selecciona un método de acción apropiado sobre el controlador**

Después que la acción es invocada, la siguiente etapa activa es **el resultado de ejecución.** Si el resultado es una vista, la vista es invocada y procesada

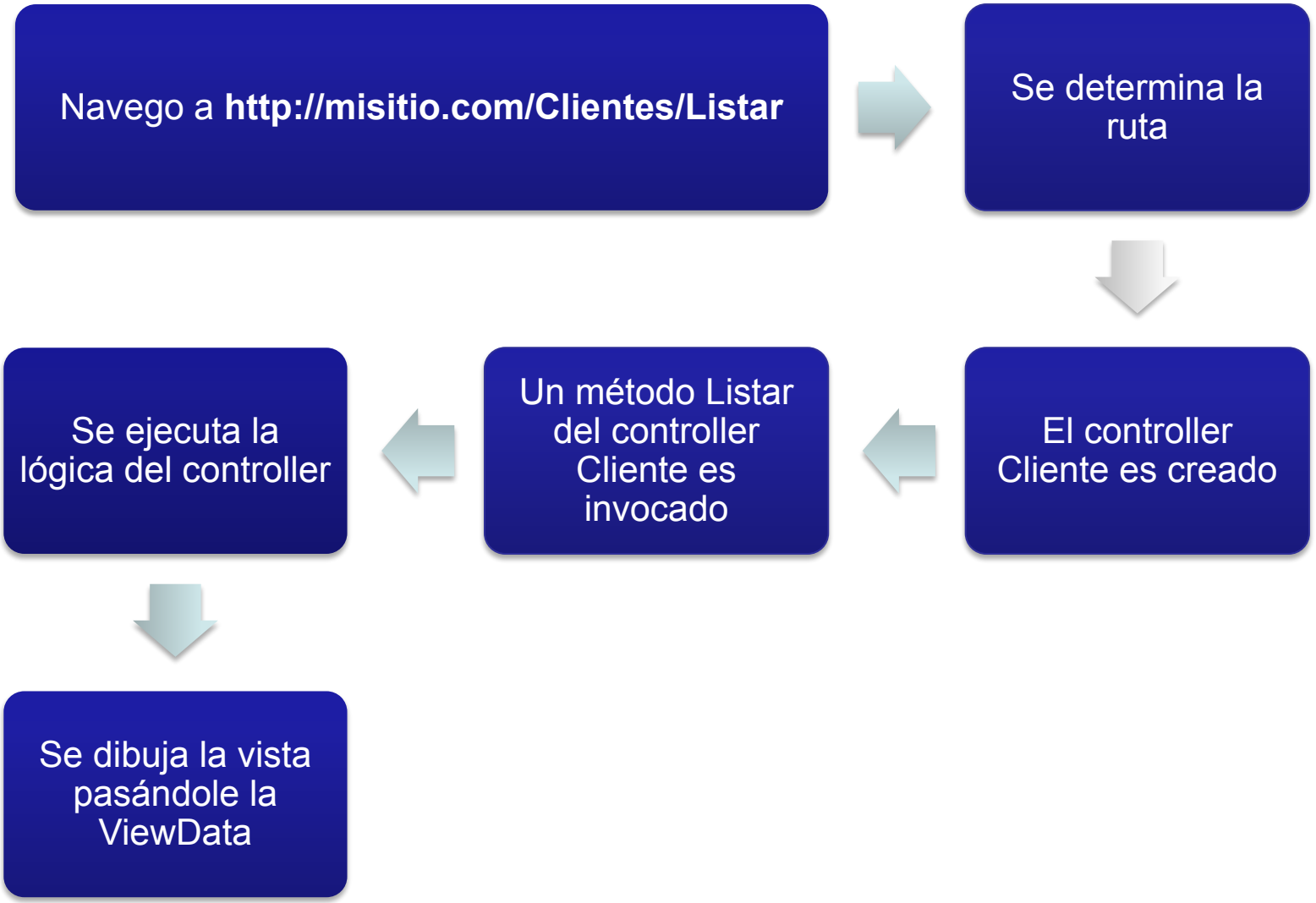
Si el resultado no es una vista, el resultado de la acción se ejecutará por su cuenta (ejemplo un string). Este resultado de ejecución es lo que genera una respuesta real a la solicitud HTTP inicial.

Funcionamiento MVC integrado a la arquitectura



El controller atiende todas las solicitudes (Request), resuelve lógica a través del modelo y devuelve como respuesta una vista html u otros formatos de salida. El modelo puede interactuar con una capa de menor jerarquía, por ejemplo con una capa de servicios o una capa de acceso a datos DAL, como se muestra en la figura.

Como Funciona ??



Introducción ASP.NET MVC

- Un framework para Web Development
- Más control sobre el HTML
  - Más Web-Frendly
- Más testeable
- **Pensado para no ocultar la naturaleza de la arquitectura Web**
- **No es una nueva versión de ASP.NET Web Forms**
- Esta construido sobre en ASP.NET

Ventajas:

- SoC (Separation of Concerns)
  - TDD por default
  - Mantenibilidad
- Url y HTML mas limpio
  - SEO y REST friendly
    - /Alumno/BuscarMateria/pw3
  - CSS Friendly
    - <html> <label>
- Modelo de programación mas performante
  - No hay ViewState
  - No hay modelo de eventos

Rutas

www.misitio.com/products/reports/1/06/2008

```
using System;

public class ProductsController : Controller
{
    public ActionResult Reports(int id, int month, int year)
    {
        //...
        View();
    }
}
```

URLs Amigables

Legibles:

www.sitio.com/products.aspx?module=reports&productId=1&month=6year=2008 => ☹

www.sitio.com/products/report/1/6/2008 => ☺

Predecibles

http://es.**wikipedia**.org/**wiki**/MVC

Enrutamiento

El enrutamiento es el proceso mediante el cual la aplicación hace coincidir una ruta URL entrante y ejecuta los métodos de acción correspondientes. ASP.NET Core MVC usa un **middleware de enrutamiento** para hacer coincidir las URL de las solicitudes entrantes y asignarlas a métodos de acción específicos.

Podemos definir las rutas en una clase de inicio (de forma predeterminada **Startup.cs) o como atributos.**

Hay dos tipos de enrutamiento para métodos de acción:

1) **Enrutamiento convencional (ver próximos slides).**
2) **Enrutamiento por atributos:** Se define dentro del Controlador, sobre una acción en particular, mediante el atributo **[Route()] :**

Ejemplo enrutamiento por atributos en un controller "Cliente":

```
[Route("Cliente/Saludo/{nombre?}")]
public string Saludo(string nombre)
{
   return $"Bienvenido a la aplicación Cliente: {nombre}";
}
```

Enrutamiento Convencional

Cuando creamos una nueva aplicación **ASP.NET Core MVC** usando la plantilla predeterminada, la aplicación configura un enrutamiento predeterminado.

En el archivo y clase **Program.cs** es donde se define la **clase de Inicio** (por defecto **Startup**) para su posterior aplicación y configuración:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Enrutamiento: Startup.cs

Después de crear un nuevo proyecto con la plantilla ASP.NET Core MVC predeterminada, podemos revisar la clase **Startup.cs** y ver que la aplicación ha configurado un enrutamiento predeterminado en el método **Configure()**:

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
   endpoints.MapControllerRoute(
       name: "default",
       pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

`MapControllerRoute():`
Define rutas válidas por medio del patrón definido en el parámetro **url.** Esas rutas válidas existen en una tabla de enrutamiento o routing. Luego esas rutas se mapean a las URLs solicitadas a una acción de un controlador específico. Una acción es sólo un método en el controlador. También puede recoger parámetros de la URL y pasarlos como parámetros en el método.
El código anterior corresponde al patrón de urls válidas por defecto. De esta forma cuando llegue una solicitud tipo: **(algo) / (algo) / (algo)**, MVC interpretará el primer "algo" como el nombre del controlador, el 2do "algo" como el nombre de la acción y el 3er algo como el parámetro.

Enrutamiento: Startup.cs

La llamada default anterior, permitirá URLs de la forma:

http://www.mitisio.com/Home/Inicio
http://www.mitisio.com/Home
http://www.mitisio.com/Clientes/Todos
http://www.mitisio.com/Clientes/Buscar/1
http://www.mitisio.com

En cambio la siguiente url **NO** sería válida:
www.misitio.com/productos/reportes/15/06/2006

Ya que el patrón de url aceptado es /controlador/acción/id (con valores por defecto), y el middleware de enrutamiento configurado no encuentra en su tabla una url con ese patrón o características.

.

Para hacer valida la solicitud anterior se debería agregar explícitamente una nueva entrada en la tabla de enrutamiento para el patrón:

```
endpoints.MapControllerRoute(
   "Fecha",
   "productos/{action}/{dia?}/{mes?}/{anio?}");
```

Controlador Uso Básico

El controller por medio de la acción, no necesariamente tiene que devolver una vista html.

En el siguiente caso se está devolviendo un string como respuesta a la solicitud (request): **/TestNoSoloHTML/SimpleString**

```
public class TestNoSoloHTMLController : Controller
{
    public string SimpleString()
    {
        return "Hola Clase, esto no es html";
    }
}
```

Controlador Uso Básico

- Escenarios, Objetivos y Diseño:
  - Las URLs indican "acciones" del Controlador, **NO páginas**
  - Las acciones se declaran en el Controlador.
  - El controlador ejecuta lógica y elige la vista.

```
public IActionResult MostrarCliente(int id)
{
    Cliente c = clienteRepository.GetClienteById(id);
     if (c != null)
     {
        return View(c);
     } else
     {
        return View("noencontrado", id);
     }
}
```

Vistas

- Generan HTML u otro tipo de contenido.
  - Helpers pre-definidos.
- Pueden ser .ASPX, .ASCX, .MASTER, etc.
- Pueden reemplazarse con otras tecnologías:
  - Template engines (NVelocity, Spark, …).
  - Formatos de salida (images, RSS, JSON, …).
  - Pueden definirse **vistas Mock** para testing.
- El controlador ofrece datos a la Vista
  - Datos Loosely typed o Strongly Typed ☺.
- Puedes ser tipadas o No tipadas.

Vistas

Pueden ser Tipadas:

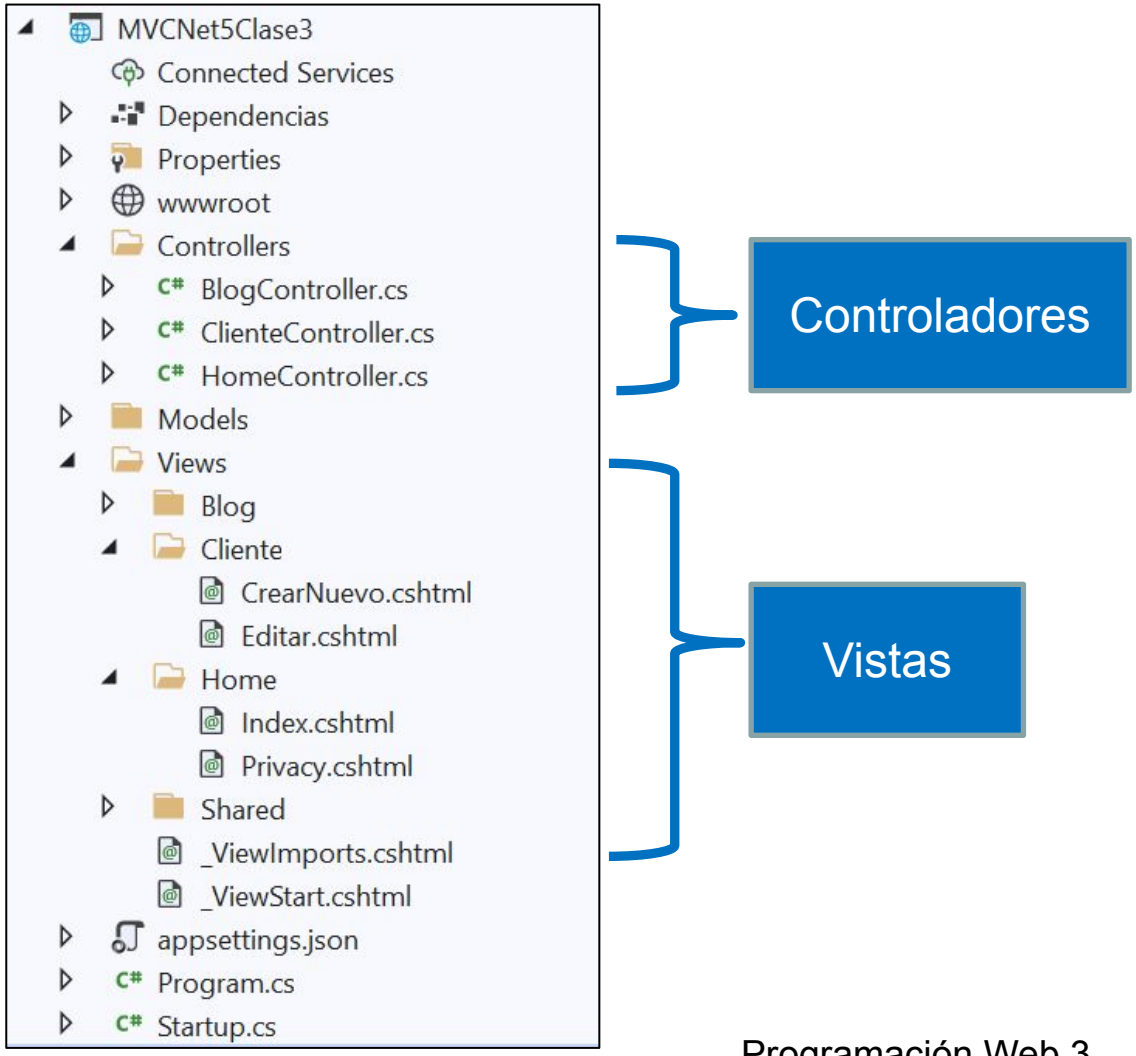
Controller Empleados, en respuesta al request: **/Empleados/Mostrar/10**

```
public IActionResult Mostrar(int id)
{
    Empleado emp = empleadoRepository.GetEmpleadoById(id);
    return View(emp);
}
```

Vista Mostrar:

```
@model MVCFirstApp.Models.Empleado

@{
    Layout = null;
}

<h2>Empleado: @Model.Nombre @Model.Apellido</h2>
```

Estructura en Visual studio



Model Binding

Enlace de elementos HTML a propiedades de un objeto (incluyendo su creación)

Model Binding

Por defecto las acciones en el Controlador responden al verbo **GET**, para capturar el verbo **POST**, se debe anteponer en la acción el atributo [**HttpPost**]

```
[HttpPost]
public ActionResult CrearNuevo(Empleado em)
{
    if (ModelState.IsValid)
    {
        empleadoRepository.Grabar(em);
                return RedirectToAction("Lista");
        }
        else
                 return View(em);
 }
```

Binding Manual (1)

En este caso, el objeto del dominio se crea y se carga manualmente a partir del formulario (**IFormCollection**) recibido por parámetro o por el objeto **Request**.

```
[HttpPost]
public ActionResult Create(IFormCollection form)
{
    ClienteVM cliente = new ClienteVM();

    cliente.RazonSocial = form["RazonSocial"];
    cliente.Direccion = form["Direccion"];

    SaveCliente(cliente);

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

```
[HttpPost]
public ActionResult CrearCliente()
{
    ClienteVM cliente = new ClienteVM();

    cliente.RazonSocial = Request.Form["RazonSocial"];
    cliente.Direccion = Request.Form["Direccion"];

    SaveCliente(cliente);

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

Binding Manual (2)

Para ello se deben respetar los valores del atributo **name** de cada input del formulario.

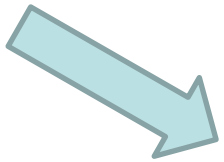
```
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>Crear Cliente</title>
</head>
<body>
<div>
<form action="/Clientes/Create" method="post">
    Nombre: <input type="text" id="txtNombre" name="Nombre" value="" /><br />
    Dirección: <input type="text" id="txtDireccion" name="Direccion" value="" /><br />

    <input type="submit" name="BtnSave" value="Grabar Cliente" />
</form>
</div>
</body>
</html>
```

Binding Automático (1)

A diferencia del manual, el objeto del dominio ya viene cargado automáticamente.

```
[HttpPost]
public ActionResult CrearNuevo(Empleado em)
{
    if (ModelState.IsValid)
    {
        empleadoRepository.Grabar(em);
        return RedirectToAction("Lista");
    }
    else
        return View(em);
 }
```

Binding Automático (2)



El objeto de la acción que atiende o responde la solicitud del submit de la vista, es **mapeado automáticamente** con los controles a través del atributo **name** de los mismos. De esta forma, para que el mapeo sea correcto, cada nombre de la propiedad de la clase debe coincidir con el nombre del atributo o propiedad **name** de cada control.

```
namespace MVCFirstApp.Models
{
    public class Empleado
    {
        public string Nombre { get; set; }
        public string Apellido { get; set; }
        public int Edad { get; set; }
        public double Sueldo { get; set; }
    }
}
```

```
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>Crear Empleado</title>
</head>
<body>
<div>
<form action="/Empleados/CrearNuevo" method="post">
       Nombre: <input type="text" id="txtNombre" name="Nombre" value="" /><br />
       Apellido: <input type="text" id="txtApellido" name="Apellido" value="" /><br />
       Sueldo: <input type="text" id="txtSueldo" name="Sueldo" value="" /><br />
       Edad: <input type="text" id="txtEdad" name="Edad" value="" /><br />

       <input type="submit" name="BtnSave" value="Grabar Empleado" />
</form>
</div>
</body>
</html>
```

Y donde esta la M en ASP.NET MVC ?

ASP.NET MVC no provee una infraestructura en particular obligatoria para el modelo pero existen una gran cantidad de opciones:

- Business Layer / Repositories
- Entity Framework (Business - DAL)
- Nhibernate (Business - DAL)
- Subsonic
- L2SQL

y muchas otras……

Desarrollador de Aplicaciones Web
Programación Web III

Departamento de Ingeniería e Investigaciones Tecnológicas

Muchas gracias

Ing. Mariano Juiz
Ing. Matias Paz Wasiuchnik
Ing. Pablo Nicolás Sanchez