3) Dada la siguiente acción:

Public ActionResult.Inicio()

{

ViewBag.Message = “Seguramente estudiaron, mostrame en una vista”;

Return View();

}

Codifique la vista razor para que muestre el mensaje en un div (<div> mensaje </div>)

🡪Para mostrar el mensaje en una vista Razor utilizando el método Inicio que proporcionaste, debes crear un archivo de vista (por ejemplo, Inicio.cshtml) y utilizar el ViewBag para acceder al mensaje. Aquí te muestro cómo podrías codificar esa vista:

**Código de la Vista Razor (Inicio.cshtml)**

@{

ViewBag.Title = "Inicio";

}

<h2>Inicio</h2>

<div>

@ViewBag.Message

</div>

**Desglose del Código**

1. **@{ ViewBag.Title = "Inicio"; }**: Esto establece el título de la página en "Inicio". Esto es opcional y puede omitirse si no es necesario.
2. **<h2>Inicio</h2>**: Se muestra un encabezado que indica que estás en la página de inicio.
3. **<div> @ViewBag.Message </div>**: Aquí se utiliza el ViewBag para mostrar el mensaje que se configuró en el controlador. El contenido de ViewBag.Message se insertará dentro del <div>.

**Integración**

1. **Controlador**: Asegúrate de que el método de acción Inicio está en un controlador adecuado (por ejemplo, HomeController).
2. **Navegación**: Asegúrate de tener configuradas las rutas en tu aplicación para que cuando navegues a la acción Inicio, se muestre esta vista.

**Resultado Esperado**

Cuando accedas a la acción Inicio, deberías ver la siguiente salida en el navegador:

Inicio

Seguramente estudiaron, mostrame en una vista

4) Se desea visualizar y resumir en una misma vista las ventas de los clientes, las compras de los proveedores y los sueldos de los empleados en un determinado mes. Asumiedo que la lógica de negocio de la aplicación existe y es correcta, ¿Cómo considera que podría mejorar (refactorear) el código del controller y consecuentemente el código de su vista? Explique y-o ejemplifique.

public ActionResult VerResumen(int.mes)

{

ViewBag.VentasClientes = ClientesServicio.Ventas(mes);

ViewBag.ComprasProveedores = ProveedoresServicio.Compras(mes);

ViewBag.SueldosEmpleados = EmpleadosServicio.Sueldos(mes);

Return View();

}

🡪Para mejorar (refactorizar) el código del controlador VerResumen y su vista asociada, hay varias estrategias que se pueden implementar. A continuación, se describen algunas de estas mejoras, junto con ejemplos.

**Problemas Identificados**

1. **Uso de ViewBag**: Aunque ViewBag es útil para pasar datos de manera rápida entre el controlador y la vista, no es el enfoque más robusto ni el más claro. No se proporciona una verificación de tipo y puede resultar en errores difíciles de depurar.
2. **Falta de un modelo fuerte**: La vista no tiene un modelo fuerte que pueda ser utilizado para representar los datos. Esto puede complicar el mantenimiento y la legibilidad del código.
3. **Separación de preocupaciones**: La lógica de negocio y la presentación de datos pueden estar mejor organizadas utilizando un modelo de vista (ViewModel).

**Propuesta de Refactorización**

1. **Crear un ViewModel**: Define un modelo que contenga todas las propiedades necesarias para la vista.
2. **Usar el modelo en la vista**: Esto proporciona un tipo seguro y mejora la claridad de la vista.
3. **Optimizar la lógica del controlador**: Mantener el controlador limpio y enfocado en la lógica de procesamiento.

**Ejemplo de Refactorización**

**Paso 1: Crear un ViewModel**

Define un modelo que contenga las propiedades que necesitas para la vista:

public class ResumenViewModel

{

public decimal VentasClientes { get; set; }

public decimal ComprasProveedores { get; set; }

public decimal SueldosEmpleados { get; set; }

}

**Paso 2: Modificar el Controlador**

Ajusta el controlador para utilizar el nuevo modelo:

public ActionResult VerResumen(int mes)

{

var resumen = new ResumenViewModel

{

VentasClientes = ClientesServicio.Ventas(mes),

ComprasProveedores = ProveedoresServicio.Compras(mes),

SueldosEmpleados = EmpleadosServicio.Sueldos(mes)

};

return View(resumen);

}

**Paso 3: Modificar la Vista**

Ahora, ajusta la vista para utilizar el modelo fuerte:

@model TuNamespace.ResumenViewModel

@{

ViewBag.Title = "Resumen de Ventas, Compras y Sueldos";

}

<h2>Resumen de Ventas, Compras y Sueldos para el Mes: @Model.Mes</h2>

<div>

<h3>Ventas de Clientes</h3>

<p>@Model.VentasClientes</p>

<h3>Compras de Proveedores</h3>

<p>@Model.ComprasProveedores</p>

<h3>Sueldos de Empleados</h3>

<p>@Model.SueldosEmpleados</p>

</div>

**Beneficios de la Refactorización**

1. **Tipo Seguro**: Al usar un ViewModel, se obtiene un código más seguro en tiempo de compilación, lo que reduce el riesgo de errores en tiempo de ejecución.
2. **Mejor Organización**: La lógica de negocio y la presentación están claramente separadas. Esto hace que el mantenimiento del código sea más sencillo y entendible.
3. **Facilidad de Pruebas**: Los ViewModels son más fáciles de probar, ya que se pueden instanciar con datos simulados y se puede verificar la funcionalidad de la vista de manera más sencilla.
4. **Extensibilidad**: Si en el futuro necesitas agregar más datos a la vista, solo tienes que agregar propiedades al ViewModel y ajustar el controlador en consecuencia.

**Conclusión**

Refactorizar el controlador y la vista de esta manera no solo mejora la calidad del código, sino que también facilita el mantenimiento y la extensión de la aplicación en el futuro. Es una buena práctica utilizar ViewModels en aplicaciones ASP.NET MVC para garantizar que los datos sean gestionados de forma clara y segura.

5) Se tiene un controller calculadora cuyas acciones son las siguientes:

public class CalculadoraConroller: Controller

{

public ActionResult Sumar(int x, int y)

{

ViewBag.Suma = x + y;

Return View();

}

public ActionResult Restar(int x, int y)

{

ViewBag.Resta = x - y;

Return View();

}

Se quiere utilizar la calculadora por medio del siguiente patron de urls.ej./Calculadora/Sumar/10/20

Donde el resultado esperado seria: 30

1. Indique si la configuración de enrutamiento por default de asp.net mvc reconocería como valida ede tipo de urls. Justifique.
2. B) indique cual/es de las siguientes entradas de patrones de url podría agregarse a la tabla de ….
3. url: “(controller)/(action)/(int x)/(int y)”,
4. url: “(controller)/action)/x/y”,
5. url: “(controller)/action)/(idx)/(idy)”,
6. url: “(controller)/action)/(x)/(y)”,

🡪Para abordar el problema planteado sobre el enrutamiento en ASP.NET MVC para la calculadora, analicemos cada parte.

**a) Reconocimiento de la URL por el Enrutamiento por Defecto**

La configuración de enrutamiento por defecto en ASP.NET MVC es la siguiente:

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

En esta configuración, la ruta espera un controlador, una acción y un parámetro opcional (id). Sin embargo, en este caso, la acción de la calculadora (Sumar o Restar) espera dos parámetros enteros (x y y).

**Evaluación de la URL Calculadora/Sumar/10/20:**

* **Controlador:** Calculadora
* **Acción:** Sumar
* **Parámetro 1 (x):** 10
* **Parámetro 2 (y):** 20

**Justificación**

Por defecto, el enrutamiento de ASP.NET MVC no reconoce dos parámetros adicionales después de action con la configuración actual, ya que solo está buscando un parámetro opcional (id). Por lo tanto, **la URL Calculadora/Sumar/10/20 no sería válida** a menos que se modifique la configuración del enrutamiento para que reconozca dos parámetros.

**b) Modificación de la Tabla de Enrutamiento**

Para que las URL como Calculadora/Sumar/10/20 funcionen, es necesario agregar una entrada de patrón a la tabla de rutas. A continuación se analizan las opciones proporcionadas:

1. **a)** url: “(controller)/(action)/(int x)/(int y)”
   * **Validación:** Esta entrada es **correcta**. Define explícitamente dos parámetros (int x y int y), lo que permite que la URL sea reconocida adecuadamente.
2. **b)** url: “(controller)/action)/x/y”
   * **Validación:** Esta entrada tiene un error de sintaxis (un paréntesis de cierre extra) y es **inválida**.
3. **c)** url: “(controller)/action)/(idx)/(idy)”
   * **Validación:** Esta entrada también tiene un error de sintaxis (un paréntesis de cierre extra) y es **inválida**. Sin embargo, si se ajusta a "(controller)/(action)/(int idx)/(int idy)", sería válida, aunque idx y idy no son los nombres de los parámetros esperados (x y y).
4. **d)** url: “(controller)/action)/(x)/(y)”
   * **Validación:** Esta entrada también tiene un error de sintaxis (un paréntesis de cierre extra) y es **inválida**. Pero si se ajusta a "(controller)/(action)/(int x)/(int y)", sería válida, similar a la opción a).

**Resumen**

* **URL válida:** Calculadora/Sumar/10/20 **no es reconocida** con la configuración por defecto.
* **Opción válida para enrutamiento:** **a)** url: “(controller)/(action)/(int x)/(int y)” sería la única opción que funcionaría, siempre que se defina correctamente en la configuración de rutas de la aplicación.

**Modificación de la Tabla de Rutas**

Para implementar la opción válida, se puede añadir la siguiente línea en el archivo de configuración de rutas (normalmente en RouteConfig.cs):

routes.MapRoute(

name: "Calculadora",

url: "Calculadora/{action}/{x}/{y}",

defaults: new { controller = "Calculadora", action = "Sumar" }

);

Esto permitirá que las URLs como Calculadora/Sumar/10/20 y Calculadora/Restar/20/10 sean reconocidas y procesadas correctamente.

6) Existe la necesidad de registrar – dar de alta un nuevo empleado. Considere que existe una clase Empleado, con dos propiedades string Nombre y Apellido.

Suponiendo que existe un controller con la siguiente acción, la cual grabaria el empleado en cuestión:

{HttpPost}

public ActionResult RegistrarEmpleado(Empleado em)

{

//Lógica para grabar al empleado: GrabarEmpleado(em);

}

Escriba los controles html inputs o helpers necesarios para el correcto funcionamiento del ejemplo:

<form action=”/Clientes/RegistrarEmpleado” method=”post”>

<input type=”submit” value=”Guardar”/>

</form>

🡪Para registrar un nuevo empleado en tu aplicación ASP.NET MVC, es necesario construir un formulario que contenga los campos necesarios para capturar las propiedades de la clase Empleado, que son Nombre y Apellido. Además, aseguraremos que los campos estén vinculados correctamente al modelo para que se envíen al controlador en la acción RegistrarEmpleado.

**Ejemplo de Formulario HTML utilizando Helpers de ASP.NET MVC**

A continuación se muestra un ejemplo completo de un formulario utilizando **HTML helpers** en ASP.NET MVC:

@model TuNamespace.Empleado <!-- Reemplaza con el namespace correcto de tu modelo -->

<form action="/Clientes/RegistrarEmpleado" method="post">

<div>

<label for="Nombre">Nombre:</label>

@Html.TextBoxFor(m => m.Nombre, new { @id = "Nombre", @required = "required" })

</div>

<div>

<label for="Apellido">Apellido:</label>

@Html.TextBoxFor(m => m.Apellido, new { @id = "Apellido", @required = "required" })

</div>

<div>

<input type="submit" value="Guardar" />

</div>

</form>

**Desglose del Código**

1. **Modelo**: @model TuNamespace.Empleado:
   * Asegúrate de reemplazar TuNamespace con el espacio de nombres real de tu clase Empleado. Esto permite que el formulario esté vinculado al modelo de datos.
2. **Campo de Nombre**:

<label for="Nombre">Nombre:</label>

@Html.TextBoxFor(m => m.Nombre, new { @id = "Nombre", @required = "required" })

* + Se utiliza @Html.TextBoxFor para crear un campo de entrada que se enlaza automáticamente a la propiedad Nombre del modelo Empleado. El atributo @required asegura que el campo sea obligatorio.

1. **Campo de Apellido**:

<label for="Apellido">Apellido:</label>

@Html.TextBoxFor(m => m.Apellido, new { @id = "Apellido", @required = "required" })

* + Similar al campo de nombre, este campo se enlaza a la propiedad Apellido del modelo y también es obligatorio.

1. **Botón de Envío**:

<input type="submit" value="Guardar" />

* + Un botón de tipo submit que envía el formulario al controlador.

**Consideraciones Adicionales**

* **Validación**: Si estás utilizando validaciones en tu modelo (como atributos de anotación de datos), considera incluir un sistema para mostrar mensajes de error en la vista, usando Html.ValidationMessageFor en cada campo.
* **Estilo y Diseño**: Dependiendo de tus necesidades, puedes agregar clases CSS o estilos para mejorar la presentación del formulario.

**Ejemplo Mejorado con Validación**

Si deseas incluir mensajes de validación, el formulario podría verse así:

@model TuNamespace.Empleado

<form action="/Clientes/RegistrarEmpleado" method="post">

<div>

<label for="Nombre">Nombre:</label>

@Html.TextBoxFor(m => m.Nombre, new { @id = "Nombre", @required = "required" })

@Html.ValidationMessageFor(m => m.Nombre)

</div>

<div>

<label for="Apellido">Apellido:</label>

@Html.TextBoxFor(m => m.Apellido, new { @id = "Apellido", @required = "required" })

@Html.ValidationMessageFor(m => m.Apellido)

</div>

<div>

<input type="submit" value="Guardar" />

</div>

</form>

Aquí, @Html.ValidationMessageFor se utiliza para mostrar mensajes de error si la validación del modelo falla al intentar enviar el formulario. Esto proporciona una mejor experiencia de usuario y garantiza que los datos se ingresen correctamente.

@HTML.TEXTBOXFOR (X=>X.NOMBRE)

@HTML.TEXTBOXFOR(X=>X.APELLIDO)

7) Verdadero o falso ASP.NET – MVC. Justifique en los casos que sea falso:

a) El modo de almacenamiento de Session StateServer guarda el estado de sesión en memoria RAM dentro del mismo proceso asp.net usado por la aplicación web.

**Falso**: El modo de almacenamiento SessionStateServer guarda el estado de sesión en un proceso independiente, específicamente en un servicio de estado de sesión que puede estar en el mismo servidor o en otro servidor, pero no dentro del mismo proceso de la aplicación ASP.NET. Esto permite que las sesiones sean compartidas entre múltiples aplicaciones o instancias de una aplicación.

b) En Asp.Net MVC, la sesión se destruye en el momento en que el usuario cierra el navegador.

**Falso:** La sesión no se destruye inmediatamente cuando el usuario cierra el navegador. En ASP.NET MVC, la sesión puede continuar activa hasta que expire el tiempo de inactividad definido en la configuración de la aplicación (por defecto, 20 minutos). El estado de la sesión se mantiene en el servidor hasta que se alcance el tiempo de expiración o hasta que el desarrollador lo destruye explícitamente.

c) ModelState.isValid es “false” si y solo si todas las validaciones de la vista dan “false” (invalidas).

**Falso**: ModelState.IsValid es false si hay al menos una validación que falla. No requiere que todas las validaciones sean falsas, solo una. Si, por ejemplo, hay un campo que no cumple con los requisitos de validación (como un campo requerido que está vacío), ModelState.IsValid se evaluará como false.

d) En el binding manual no es obligatorio tener en cuenta el atributo name de los controles de la vista, en ese caso se debe usar FormCollection en la acción.

**Falso:** En el binding manual, el atributo name de los controles de la vista es **necesario** si se desea que el modelo se vincule correctamente. Si usas FormCollection, puedes acceder a los valores por su nombre, pero para que el modelo se vincule automáticamente, los nombres deben coincidir con las propiedades del modelo. Por lo tanto, es crucial que el nombre sea igual al de la propiedad correspondiente para que el modelo se cree correctamente.

e) ViewBag incorpora variables dinámicas que si están mal escritas producen errores en tiempo de compilación.

**Falso:** ViewBag es un tipo dinámico en C#. Si se escribe mal una variable de ViewBag, no se producirá un error en tiempo de compilación; en su lugar, se producirá un error en tiempo de ejecución cuando se intente acceder a esa variable. Esto se debe a que ViewBag utiliza la reflexión para acceder a sus propiedades, lo que significa que los errores relacionados con nombres mal escritos solo se detectan cuando se ejecuta el código, no durante la compilación.

8) SOLID: Explique y diferencie “S” y “O”.

9) indique v o f.

a) int.parse sirve para castear un string a un entero, pero en el caso de querer castear un string a un long deberia castealo con long n = (strng) "10000000000"

**a) Falso**

La afirmación es incorrecta por dos razones:

1. **int.Parse sí sirve para convertir (no "castear") un string a un int**. La conversión de un string a un tipo numérico no se considera un **cast** (que solo funciona entre tipos relacionados), sino una **conversión**. Por ejemplo, para convertir un string a un int, se utiliza:

int number = int.Parse("123");

1. **No puedes castear un string a un long usando (long) "10000000000".** Esto no es válido en C#. El casting explícito como (long) solo funciona entre tipos numéricos compatibles. Para convertir un string a un long, debes usar **long.Parse** o **long.TryParse**:
   * Usar long.Parse() para una conversión directa:

long number = long.Parse("10000000000");

* + Usar long.TryParse() para una conversión segura sin lanzar excepciones:

long number;

bool success = long.TryParse("10000000000", out number);

1. **Conclusión:** No puedes usar (long) para convertir un string a long. Debes usar long.Parse() o long.TryParse().

b) en una aplicacion ASP .NET MVC el framework y visual studio deben estar instalados en el server para su correcta ejecucion.

**Falso**

Para que una aplicación ASP.NET MVC funcione correctamente en un servidor, **no es necesario que Visual Studio esté instalado en el servidor**, ni tampoco el **framework de desarrollo completo** (por ejemplo, Visual Studio). Solo es necesario que el **runtime** adecuado de .NET esté instalado en el servidor.

**Detalles:**

* **Visual Studio**: Es un **entorno de desarrollo** (IDE) que se utiliza para escribir, compilar y depurar el código durante la fase de desarrollo, pero **no es necesario** en el servidor donde se va a desplegar la aplicación.
* **Framework o Runtime**:
  + Si la aplicación es una **aplicación ASP.NET MVC basada en .NET Framework**, el servidor debe tener el **.NET Framework** instalado.
  + Si la aplicación es una **aplicación ASP.NET Core**, el servidor debe tener el **runtime de .NET Core** o **.NET 5+** instalado, dependiendo de la versión utilizada en el proyecto.

El **runtime** es lo único que se necesita en el servidor para ejecutar la aplicación, ya que es responsable de compilar el código intermedio (IL) y ejecutar la aplicación.

**Conclusión:**

Solo el **runtime** de .NET debe estar instalado en el servidor, no Visual Studio ni el framework completo de desarrollo.

c) Las vistas y partial views son vistas, pero las partials views no deben usar un layout.

**Verdadero**

Las **partial views** en **ASP.NET MVC** y **ASP.NET Core** **no deben** usar un **layout**. Esto se debe a que una **partial view** está diseñada para ser una parte de otra vista más grande, y no es una página completa por sí sola.

**Explicación:**

* **Vistas completas** pueden tener un **layout** (diseño compartido), que proporciona una estructura común para la página, como encabezados, menús o pies de página.
* **Partial views**, en cambio, son fragmentos reutilizables de HTML y lógica Razor que se **insertan** dentro de otras vistas. Dado que se espera que formen parte de una vista más grande, **no deben incluir un layout**, ya que sería redundante y podría causar problemas de visualización.

Por ejemplo, si una partial view incluyera un layout, este podría duplicar secciones como el encabezado o el pie de página dentro de una vista principal que ya los tiene.

**Conclusión:**

Es correcto decir que las **partial views** no deben usar un layout, ya que están destinadas a ser reutilizadas dentro de otras vistas que ya gestionan el layout completo.

d) para poder usar un layout, desde la vista que lo implementa se debe explicitar la linea de codigo @RenderBody()

**Falso**

En **ASP.NET MVC** y **ASP.NET Core**, la línea de código **@RenderBody()** **no debe estar en la vista que utiliza el layout**, sino en el **archivo del layout** mismo.

**Explicación:**

* **@RenderBody()** es un marcador de posición que se utiliza **en el layout** para indicar dónde se debe insertar el contenido de la vista que utiliza dicho layout.
* Desde una **vista** que utiliza un layout, **no es necesario ni correcto** incluir @RenderBody(). En su lugar, solo necesitas indicar qué layout utilizar mediante la propiedad **@Layout**. Por ejemplo:

@{ Layout = "\_Layout"; }

* El **archivo de layout** es el que contiene la llamada a @RenderBody(), que define dónde se mostrará el contenido principal de las vistas que usan ese layout.

**Ejemplo de uso correcto:**

1. **Archivo de Layout (\_Layout.cshtml):**

<html>

<body>

<header>

<!-- Encabezado -->

</header>

<main>

@RenderBody() <!-- Aquí se renderiza el contenido de la vista -->

</main>

<footer>

<!-- Pie de página -->

</footer>

</body>

</html>

1. **Vista que usa el Layout (Index.cshtml):**

@{ Layout = "\_Layout"; }

<h1>Bienvenido a la página principal</h1>

<p>Este es el contenido específico de la vista Index.</p>

**Conclusión:**

La afirmación es falsa porque **@RenderBody() se utiliza en el archivo de layout** y **no debe ser incluido** en las vistas que implementan dicho layout. Las vistas simplemente indican qué layout usar mediante la propiedad @Layout.

e) En .Net, out y ref no tienen ninguna diferencia en el pasaje de parametros.

**Falso**

En **.NET**, **out** y **ref** son modificadores de parámetros que permiten pasar variables por referencia, pero **tienen diferencias importantes** en su comportamiento y requisitos de uso.

**Diferencias entre out y ref:**

1. **ref (referencia existente)**:
   * El parámetro que se pasa con **ref** **debe estar inicializado** antes de pasarlo al método.
   * El método puede leer y modificar el valor de la variable.
   * **Uso de ref**: El método que recibe el parámetro tiene acceso al valor inicial, lo modifica si es necesario, y esos cambios se reflejan en la variable original del llamador.

**Ejemplo:**

void ModificarValor(ref int valor) {

valor = valor + 10;

}

int numero = 5;

ModificarValor(ref numero);

Console.WriteLine(numero); // Output: 15

1. **out (solo salida)**:
   * El parámetro que se pasa con **out** **no necesita estar inicializado** antes de pasarlo al método.
   * Es **obligatorio** que el método asigne un valor al parámetro antes de salir del método.
   * **Uso de out**: El método no puede leer el valor inicial de la variable (si es que lo tuviera), pero debe asignar un nuevo valor antes de que el método retorne.

**Ejemplo:**

void AsignarValor(out int valor) {

valor = 20; // Obligatorio asignar un valor

}

int numero; // No necesita estar inicializado

AsignarValor(out numero);

Console.WriteLine(numero); // Output: 20

**Resumen de las diferencias:**

* **ref**: El valor debe estar **inicializado antes** de pasarlo al método.
* **out**: El valor **no necesita estar inicializado**, pero el método **debe asignarle un valor** antes de que termine.

**Conclusión:**

La afirmación es falsa, ya que **out y ref tienen diferencias importantes** en cómo manejan la inicialización y el uso del valor pasado al método.

10) Indique que atributos de validación – DataAnnotations usaría para validar que la Edad de una Persona (Escriba la clase persona y los dataannotations)

a) Sea de ingreso obligatorio y el mensaje de error sea “La edad es requerida”

b) Este en un rango de 1 a 100 y el mensaje de error sea “La edad ingresada debe ser entre 1 y 100”.