**1) Verdadero o falso ASP.NET MVC. Justifique en los casos que sea falso: a) Todas las clases que representan al Model, deben terminar con la palabra “Model” por convención. b) El modo de enrutamiento de la aplicación se define en el archivo appsettings.json. c) Un middleware es configurado en el método “Configure” de la clase “Startup”. Solo puede existir un middleware por aplicación Web. d) El controlador ejecuta lógica pero el Modelo es el responsable de elegir la vista que será procesada y devuelta a cliente.**

**a) Falso**

No es obligatorio que todas las clases que representan al **Model** terminen con la palabra "Model" en ASP.NET MVC. Aunque es una **convención** común en algunos proyectos para hacer el código más legible (por ejemplo, UserModel), **no es un requisito** del framework. Las clases del modelo pueden llamarse de cualquier forma que tenga sentido para la aplicación.

**b) Falso**

El modo de enrutamiento de la aplicación **no** se define en el archivo appsettings.json. El enrutamiento en **ASP.NET MVC** se configura principalmente en el archivo Startup.cs, en el método Configure, usando **UseRouting()** y **UseEndpoints()** para definir las rutas. El archivo appsettings.json se usa principalmente para configurar opciones como cadenas de conexión, variables de configuración, etc., pero no el enrutamiento.

**c) Falso**

Si bien es cierto que los **middlewares** son configurados en el método Configure de la clase Startup, **puede haber múltiples middlewares** en una aplicación web, no solo uno. De hecho, una típica aplicación ASP.NET Core puede tener varios middlewares para manejar cosas como la autenticación, el manejo de errores, el enrutamiento, etc. Los middlewares se agregan en una **pipeline**, donde cada middleware puede decidir pasar el control al siguiente.

**d) Falso**

En ASP.NET MVC, el **controlador** es el responsable de ejecutar la lógica de la aplicación y de **elegir la vista** que será procesada y devuelta al cliente, no el Modelo. El Modelo se encarga de gestionar los datos (lógica de negocio, acceso a bases de datos, etc.), pero el **controlador** usa el Modelo y luego selecciona la vista a través del método View().

3) Verdadero o falso ASP.NET - MVC. Justifique en los casos que sea falso:

1. a) Respecto a las variables de sesión en ASP.NET CORE, existe un mecanismo automático para informar al código de la aplicación (servidor) que el navegador se ha cerrado.
2. b) El tag helper <partial name="\_Vista” /> es la forma tradicional de renderizar una vista parcial.
3. c) En ASP.NET CORE el uso de variables de sesión está activado por defecto; solo es necesario utilizar el objeto o clase: “HttpContext.Session”.
4. d) ViewBag es similar a ViewData, pero se diferencian en su “tiempo de vida”, así ViewData permite mantener variables y su contenido después de una redirección, pero ViewBag no.

**a) Falso**

En **ASP.NET Core**, **no existe un mecanismo automático** para informar al código de la aplicación (servidor) cuando el navegador se ha cerrado. Las **variables de sesión** no se destruyen inmediatamente cuando el navegador se cierra. El estado de la sesión es mantenido en el servidor y depende del tiempo de vida configurado en el servidor (por ejemplo, tiempo de inactividad o expiración de la sesión). El servidor no recibe ninguna señal directa cuando el usuario cierra el navegador.

**b) Falso**

El **tag helper** para renderizar una vista parcial en **ASP.NET Core** es:

html

Copiar código

<partial name="\_Vista" />

Pero **no es la forma "tradicional"** de renderizar vistas parciales. En versiones anteriores de **ASP.NET MVC** (antes de ASP.NET Core), la forma tradicional era usar:

csharp

Copiar código

@Html.Partial("\_Vista")

El tag helper es un enfoque más moderno introducido en **ASP.NET Core**, pero el uso de @Html.Partial es la forma tradicional en las versiones anteriores de **ASP.NET MVC**.

**c) Falso**

En **ASP.NET Core**, el uso de **variables de sesión no está activado por defecto**. Para usar variables de sesión, debes realizar la configuración explícita en Startup.cs agregando el middleware de sesión. Esto incluye los siguientes pasos:

1. Agregar los servicios de sesión en ConfigureServices:

services.AddSession();

1. Configurar el middleware de sesión en Configure:

app.UseSession();

Una vez activado, puedes acceder a las variables de sesión usando HttpContext.Session. Pero **no está activado de manera automática**.

**d) Falso**

**ViewBag** y **ViewData** son muy similares en cuanto a su uso, ya que ambos permiten pasar datos entre el controlador y la vista. **Ambos tienen el mismo "tiempo de vida"** y **ninguno** de ellos sobrevive después de una redirección. Tanto **ViewData** como **ViewBag** son específicos de la **solicitud actual** (request) y no persisten después de una redirección.  
Si necesitas persistir datos entre redirecciones, debes usar **TempData**, que está diseñada específicamente para eso.

**Resumen:**

* **a)** Falso: No hay un mecanismo automático para saber que el navegador se ha cerrado.
* **b)** Falso: El tag helper <partial> es moderno, no la forma "tradicional".
* **c)** Falso: El uso de sesiones no está activado por defecto.
* **d)** Falso: Ni **ViewBag** ni **ViewData** sobreviven a redirecciones.

4) Se sabe que la siguiente Vista NO es un layout y tiene errores. Asuma que existe una clase “Cliente” con una propiedad pública “RazonSocial” con espacio de nombres: “PrimerParcialEjercicios.Models” y que el controller y la acción son correctos. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ @Model PrimerParcialEjercicios.Models.Cliente

@{ Layout = "Bienvenidxs al primer parcial de Web 3"; }

@RenderBody()

{ <h2> Bienvenido @model.RazonSocial a la evaluación</h2> }

La vista tiene varios errores conceptuales y sintácticos. A continuación te explico los problemas y cómo corregirlos:

**Errores y correcciones:**

**1. Declaración de @model incorrecta:**

La declaración del modelo en Razor debe usar la palabra clave **@model** (en minúsculas) en lugar de @Model (mayúsculas). También es incorrecto escribir @Model PrimerParcialEjercicios.Models.Cliente en la misma línea. Lo correcto sería:

**Error:**

@Model PrimerParcialEjercicios.Models.Cliente

**Corrección:**

@model PrimerParcialEjercicios.Models.Cliente

**Explicación**: @model define el tipo de datos que la vista espera recibir, en este caso, una instancia de Cliente del espacio de nombres PrimerParcialEjercicios.Models.

**2. Layout asignado incorrectamente:**

El valor de **Layout** debe ser la ruta o el nombre del archivo de diseño que usará la vista, no una cadena arbitraria. En el código dado, parece que se está intentando asignar un mensaje de bienvenida a Layout, lo cual es incorrecto. Layout debe referirse a la ruta del archivo de diseño si se está utilizando uno.

**Error:**

@{ Layout = "Bienvenidxs al primer parcial de Web 3"; }

**Corrección:** Si no hay un Layout (porque no es una vista de diseño), simplemente se puede eliminar o, si debería estar asignado a un diseño específico, debe hacer referencia al archivo correcto, por ejemplo:

@{ Layout = null; } // Si no se usa un layout.

O si se necesita un layout:

@{ Layout = "\_Layout"; } // Nombre del archivo Layout, si existiera uno.

**3. Uso incorrecto de @RenderBody():**

La función **@RenderBody()** solo debe aparecer en los **Layouts**, ya que indica el lugar donde se va a renderizar el contenido de una vista secundaria dentro del layout. Dado que la vista dada no es un layout, no se debe usar @RenderBody().

**Error:**

@RenderBody()

**Corrección:** Simplemente elimina esta línea, ya que no tiene sentido en una vista que no sea un layout.

**4. Acceso incorrecto a la propiedad del modelo:**

El acceso a la propiedad del modelo debe hacerse a través de **@Model** (con mayúscula). El código usa @model.RazonSocial (en minúsculas), lo cual es incorrecto porque @model define el tipo de datos, mientras que @Model hace referencia a la instancia del modelo.

**Error:**

<h2> Bienvenido @model.RazonSocial a la evaluación</h2>

**Corrección:**

<h2> Bienvenido @Model.RazonSocial a la evaluación</h2>

**Vista corregida:**

@model PrimerParcialEjercicios.Models.Cliente

@{ Layout = null; } <!-- O referenciar un layout si existe -->

<h2> Bienvenido @Model.RazonSocial a la evaluación</h2>

**Explicación:**

1. **@model PrimerParcialEjercicios.Models.Cliente**: Indica que la vista recibirá un objeto de tipo Cliente.
2. **Layout = null**: Indica que la vista no está utilizando un layout. Si es necesario, puedes reemplazar null por el nombre del layout adecuado.
3. **@Model.RazonSocial**: Muestra el valor de la propiedad RazonSocial del modelo Cliente que recibe la vista.

Esta versión corregida debería funcionar correctamente si se pasa un objeto Cliente con la propiedad RazonSocial desde el controlador a la vista.

5) Indique V o F. Justifique si es falso

a) En una aplicación ASP.NET MVC el runtime (Framework o Core) debe estar instalado en el server y se recomienda que también este instalado en el cliente para evitar problemas de compatibilidad de versiones.

b) Un Assembly (.dll / .exe) es la unidad mínima de ejecución cuyo código MSIL es creado por el CLR.

c) .Net Core tiene similitudes a .Net Framework y fue escrito sobre la base de .Net Framework.

d) En .Net para castear un string a un int se puede usar la forma de conversión: int i = (int) mystring;

**a) Falso**

En una aplicación **ASP.NET MVC**, el **runtime** (ya sea **.NET Framework** o **.NET Core**) **debe estar instalado solo en el servidor**, no en el cliente. Las aplicaciones web se ejecutan en el servidor y entregan HTML, CSS y JavaScript al cliente (el navegador). El cliente no necesita tener instalado .NET, ya que solo interactúa con la aplicación web a través del navegador. Instalar el runtime en el cliente no tiene sentido ni es necesario para las aplicaciones web.

**b) Falso**

Un **Assembly** (un archivo **.dll** o **.exe**) es la **unidad mínima de implementación** y contiene código **MSIL** (Microsoft Intermediate Language), pero **el código MSIL no es creado por el CLR**, sino que es **ejecutado por el CLR**. El código MSIL es generado por el compilador de .NET (C#, VB.NET, etc.) y luego es compilado en tiempo de ejecución a código nativo por el **CLR** (Common Language Runtime). Así que la afirmación confunde el rol del compilador y el CLR.

**c) Falso**

**.NET Core** no fue escrito sobre la base de **.NET Framework**. Aunque **.NET Core** y **.NET Framework** comparten algunas similitudes en la sintaxis y en el uso de bibliotecas, **.NET Core fue escrito desde cero** para ser **multiplataforma** y **más modular**. Es una reimplementación moderna de la plataforma .NET que incluye mejoras en rendimiento y portabilidad, pero no se basa directamente en **.NET Framework**.

**d) Falso**

En **.NET**, no puedes castear directamente un string a int usando un casting explícito como int i = (int)mystring;. Esto lanzaría un error de compilación, ya que el casting solo funciona entre tipos compatibles. Para convertir un string a int, se debe usar una de las siguientes formas:

* Usar int.Parse():

int i = int.Parse(mystring);

* Usar int.TryParse() si deseas manejar la conversión de manera segura sin lanzar excepciones:

int i;

bool success = int.TryParse(mystring, out i);

Estas son las formas correctas de convertir un string a int en **.NET**.

**Resumen:**

* **a)** Falso: El runtime solo debe estar instalado en el servidor.
* **b)** Falso: El MSIL es generado por el compilador, no por el CLR.
* **c)** Falso: .NET Core fue escrito desde cero, no basado en .NET Framework.
* **d)** Falso: El casting directo (int) no es válido para conversiones de string a int.