

`private IHostingEnvironment environment;` Esta línea declara una variable privada llamada `environment` del tipo `IHostingEnvironment`. `IHostingEnvironment` es una interfaz proporcionada por ASP.NET Core que proporciona información sobre el entorno de hospedaje en el que se ejecuta la aplicación, como la ruta de contenido raíz,

`private IConfiguration configuration;` Esta línea declara una variable privada llamada `configuration` del tipo `IConfiguration`. `IConfiguration` es una interfaz proporcionada por ASP.NET Core que representa una colección de pares clave-valor utilizados para almacenar la configuración de la aplicación

Esta es la firma del constructor del controlador. Toma dos parámetros, uno del tipo `IHostingEnvironment` llamado `_environment` y otro del tipo `IConfiguration` llamado `_configuration`

`environment = _environment;` Asigna el valor del parámetro `_environment` al miembro `environment` del controlador. Esto permite que el controlador acceda a la información del entorno de hospedaje.

`configuration = _configuration;` Asigna el valor del parámetro `_configuration` al miembro `configuration` del controlador. Esto permite que el controlador acceda a la configuración de la aplicación

```
private IHostingEnvironment environment;
private IConfiguration configuration;

public UsuarioController(IHostingEnvironment _environment, IConfiguration _configuration)
{
    environment = _environment;
    configuration = _configuration;
}
```

Al entrar a la página de los usuario se utiliza el método GetAll que devuelve un ActionResult. En este caso, devuelve una vista que contiene datos obtenidos de una API.

El método está decorado con el atributo [HttpGet], lo que indica que está destinado a manejar solicitudes HTTP GET

```
[HttpGet]
public ActionResult GetAll()
{
    ML.Usuario resultUsuario = new ML.Usuario();
    resultUsuario.Vendedor = new Vendedor();

    resultUsuario.Usuarios = new List<object>();

    using (var client = new HttpClient())
    {
        string urlApi = configuration["urlWebApi"];
        client.BaseAddress = new Uri(urlApi);

        var responseTask = client.GetAsync("Usuario/GetAll/" + resultUsuario);
        responseTask.Wait();

        var result = responseTask.Result;

        if (result.IsSuccessStatusCode)
        {
            var readTask = result.Content.ReadAsAsync<ML.Result>();
            readTask.Wait();

            foreach (var resultItem in readTask.Result.Objects)
            {
                ML.Usuario ResultItemList = Newtonsoft.Json.JsonConvert.DeserializeObject<ML.Usuario>(resultItem);
                resultUsuario.Usuarios.Add(ResultItemList);
            }
        }
    }

    return View(resultUsuario);
}
```

Se crea una nueva instancia de ML.Usuario y se asigna a la variable resultUsuario. También se crea una nueva instancia de Vendedor y se asigna a la propiedad resultUsuario.Vendedor. Se crea una nueva lista de object y se asigna a la propiedad resultUsuario.Usuarios. Está destinada a almacenar una colección de objetos de usuario obtenidos de la API.

```
public class Usuario
{
    public int IdUsuario { get; set; }

    public string Username { get; set; } = null!;

    public string Password { get; set; } = null!;

    public bool Estatus { get; set; }

    public ML.Vendedor? Vendedor { get; set; }

    public ML.Rol? Rol { get; set; }

    public List<object>? Usuarios { get; set; }
}
```

```
public class Vendedor
{
    public int IdVendedor { get; set; }

    public string Nombre { get; set; } = null!;

    public string ApellidoPaterno { get; set; } = null!;

    public string ApellidoMaterno { get; set; } = null!;

    public string Curp { get; set; } = null!;

    public string? Rfc { get; set; }

    public string? Foto { get; set; }

    public string Email { get; set; } = null!;

    public string Celular { get; set; } = null!;

    public List<object>? Vendedores { get; set; }
}
```

Luego, el método configura un HttpClient para comunicarse con una API remota. Lee la URL base de la API desde un archivo de configuración

Luego, el método envía una solicitud GET asíncrona a la API usando HttpClient.GetAsync(). La URL de la solicitud se construye con "Usuario/GetAll/" concatenado con resultUsuario. Esta concatenación podría ser incorrecta porque resultUsuario es un objeto, no una cadena. Podría generar una URL que no funcione como se espera.

El método espera sincrónicamente a que se reciba la respuesta HTTP utilizando responseTask.Wait()

```
[Route("api/Usuario/GetAll/{usuario}")]
[HttpGet]
public IActionResult GetAll(ML.Usuario usuario)
{
    ML.Result result = BL.Usuario.GetAll(usuario);

    if (resultCorrect)
    {
        return Ok(result);
    }
    else { return NotFound(result); }
}
```

El método está decorado con los atributos `[Route("api/Usuario/GetAll/{usuario}")]` y `[HttpGet]`. Estos atributos indican que este método se debe invocar cuando la ruta de la solicitud coincide con la plantilla `"api/Usuario/GetAll/{usuario}"` y que la solicitud debe ser una solicitud HTTP GET

El método utiliza la capa de negocio (`BL.Usuario`) para obtener datos. Es probable que la capa de negocio se encargue de procesar la solicitud y recuperar los datos necesarios de alguna fuente, como una base de datos o un servicio externo. En este caso, el método llama a `BL.Usuario.GetAll(usuario)` para obtener un objeto `ML.Result` que contiene el resultado de la operación

Si `resultCorrect` es `true`, el método devuelve una respuesta HTTP con el código 200 OK, y el objeto `ML.Result` se serializa automáticamente en el cuerpo de la respuesta. La respuesta contendrá el resultado de la operación exitosa.

Si `resultCorrect` es `false`, el método devuelve una respuesta HTTP con el código 404 Not Found, y el objeto `ML.Result` se serializa en el cuerpo de la respuesta. En este caso, es probable que la respuesta contenga información sobre el error o el resultado no encontrado.

```

public static ML.Result GetAll(ML.Usuario usuario)
{
    ML.Result result = new ML.Result();

    try
    {
        using (DL.BienesRaicesSqlContext cnn = new DL.BienesRaicesSqlContext())
        {
            var query = cnn.Usuarios.FromSqlRaw($"UsuarioGetAll '{usuario.Vendedor.Nombre}'");

            result.Objects = new List<object>();

            if (query != null)
            {
                foreach (var row in query)
                {
                    usuario = new ML.Usuario();
                    usuario.IdUsuario = row.IdUsuario;
                    usuario.Username = row.Username;
                    usuario.Password = row.Password;
                    usuario.Estatus = row.Estatus.Value;

                    usuario.Vendedor = new ML.Vendedor();
                    usuario.Vendedor.IdVendedor = row.IdVendedor;
                    usuario.Vendedor.Nombre = row.Nombre;
                    usuario.Vendedor.ApellidoPaterno = row.ApellidoPaterno;
                    usuario.Vendedor.ApellidoMaterno = row.ApellidoMaterno;
                    usuario.Vendedor.Curp = row.Curp;
                    usuario.Vendedor.Rfc = row.Rfc;
                    usuario.Vendedor.Foto = row.Foto;
                    usuario.Vendedor.Email = row.Email;
                    usuario.Vendedor.Celular = row.Celular;

                    usuario.Rol = new ML.Rol();
                    usuario.Rol.IdRol = row.IdRol;
                    usuario.Rol.Nombre = row.NombreRol;
                }
            }
        }
    }
}

```

El método recibe un parámetro ML.Usuario usuario, que representa un objeto de usuario que se utilizará como filtro para obtener información relacionada con usuarios y vendedores.

Se crea una nueva instancia de ML.Result llamada result. ML.Result es una clase que parece ser utilizada para almacenar resultados y errores que se devolverán desde la capa de acceso a datos a la capa de negocio.

```

public class Result
{
    public bool Correct { get; set; }

    public string ErrorMessage { get; set; } = string.Empty;

    public Exception? Ex { get; set; }

    public object? Object { get; set; }

    public List<object>? Objects { get; set; }
}

```

Se inicia un bloque try, lo que significa que el código dentro de este bloque se ejecutará y se capturará cualquier excepción que ocurra.

Dentro del bloque try, se crea una nueva instancia de DL.BienesRaicesSqlContext. Esto sugiere que hay un contexto de base de datos que se utilizará para interactuar con la base de

datos. El código utiliza `using` para garantizar que los recursos se liberen correctamente después de su uso.

Se ejecuta una consulta SQL a través de `cnn.Usuarios.FromSqlRaw(...)`. Parece que se utiliza una consulta SQL parametrizada para recuperar información de usuarios y vendedores de la base de datos

Se inicializa una lista de objetos vacía `result.Objects`, que se utilizará para almacenar los resultados de la consulta.

Si `query` no es nulo, significa que se recuperaron datos de la base de datos. El código procede a recorrer los resultados en el bucle `foreach`.

Dentro del bucle `foreach`, se crea un nuevo objeto `ML.Usuario` llamado `usuario` (este es un objeto diferente del parámetro `usuario` que se pasó al método). Luego, se asignan valores a las propiedades de `usuario` basándose en los resultados de la consulta (columnas de la tabla de usuarios y vendedores).

Se crea un nuevo objeto `ML.Vendedor` y se asignan valores a sus propiedades basándose en los resultados de la consulta.

Se crea un nuevo objeto `ML.Rol` y se asignan valores a sus propiedades basándose en los resultados de la consulta.

Finalmente, se agrega el objeto `usuario` recién creado (con todos los datos relacionados con el vendedor y el rol) a la lista `result.Objects`.

Si ocurre alguna excepción dentro del bloque `try`, se captura en el bloque `catch`. En este caso, `result.Correct` se establecerá en `false`, y se almacenará la excepción en `result.Ex`. Además, se generará un mensaje de error en `result.ErrorMessage`.

Finalmente, se devuelve el objeto `result`, que contendrá los resultados de la consulta en `result.Objects`, así como información sobre el éxito o el error de la operación. Este objeto se utilizará posteriormente en la capa de negocio para tomar decisiones y proporcionar respuestas apropiadas a las solicitudes del cliente

Una vez que se lee y analiza los datos JSON como `ML.Result`, el método itera a través de su propiedad `Objects`, que parece ser una colección de objetos JSON.

Para cada objeto en la colección, se deserializa los datos JSON en una instancia de `ML.Usuario` utilizando `Newtonsoft.Json.JsonConvert.DeserializeObject<ML.Usuario>(resultItem.ToString())`

El objeto `ML.Usuario` resultante se agrega a la lista `resultUsuario.Usuarios`.

[illegible]

td class="text-center"></td>

crea una celda de tabla que contiene un enlace de edición (<a>) representado por un ícono de lápiz. El enlace apunta a la acción "Form" del controlador "Usuario" y pasa el parámetro idUsuario con el valor del usuario.IdUsuario.

`<td class="visually-hidden">@usuario.Vendedor.IdVendedor</td>` representa una celda de tabla que contiene el `IdVendedor` del usuario vendedor, pero el estilo "visually-hidden" indica que este contenido no será visible, aunque seguirá presente para el uso de tecnologías de asistencia.

<td class="text-center">@usuario.Vendedor.Nombre @usuario.Vendedor.ApellidoPaterno @usuario.Vendedor.ApellidoMaterno</td> crea una celda de tabla que muestra el nombre completo del vendedor (nombre, apellido paterno y apellido materno) centrado en el texto.

`<td class="text-center">@usuario.Vendedor.Curp</td>` crea una celda de tabla que muestra el CURP (Clave Única de Registro de Población) del vendedor.

<td class="text-center">@usuario.Vendedor.Rfc</td> crea una celda de tabla que muestra el RFC (Registro Federal de Contribuyentes) del vendedor.

La siguiente parte del código contiene una condición con un if-else para mostrar la imagen del vendedor. Si `usuario.Vendedor.Foto` es nulo o una cadena vacía, muestra una imagen

predeterminada (guest-user.png). De lo contrario, muestra la imagen del vendedor codificada en base64 usando la etiqueta .

<td class="text-center">@usuario.Vendedor.Email</td> crea una celda de tabla que muestra el correo electrónico del vendedor.

<td class="text-center">@usuario.Vendedor.Celular</td> crea una celda de tabla que muestra el número de celular del vendedor.

<td class="visually-hidden">@usuario.IdUsuario</td> representa una celda de tabla que contiene el IdUsuario, pero el estilo "visually-hidden" indica que este contenido no será visible para el usuario.

<td class="text-center">@usuario.Username</td> crea una celda de tabla que muestra el nombre de usuario (Username) del usuario.

<td class="text-center">@usuario.Password</td> crea una celda de tabla que muestra la contraseña (Password) del usuario.

La siguiente parte del código utiliza un if-else para generar una celda de tabla que contiene un interruptor (<input type="checkbox">) que permite cambiar el estado (Estatus) del usuario. Si el usuario.Estatus es true, se mostrará un interruptor activado (checked), y si es false, se mostrará un interruptor desactivado.


<td class="visually-hidden">@usuario.Rol.IdRol</td> representa una celda de tabla que contiene el IdRol del usuario, pero el estilo "visually-hidden" indica que este contenido no será visible.

<td class="text-center">@usuario.Rol.Nombre</td> crea una celda de tabla que muestra el nombre del rol (Nombre) del usuario.

localhost:64492/Usuario/Form? Swagger UI

Formulario Usuario

Nombre	ApellidoPaterno	ApellidoMaterno
<input type="text"/>	<input type="text"/>	<input type="text"/>
Curp	Rfc	Foto
<input type="text"/>	<input type="text"/>	<input type="button" value="Choose file"/> No file chosen




Email	Celular	Username
<input type="text"/>	<input type="text"/>	<input type="text"/>
Password	Rol	
<input type="text"/>	<input type="button" value="Selecciona una opción"/>	

localhost:64492/Usuario/Form? Swagger UI

Formulario Usuario

Nombre	ApellidoPaterno	ApellidoMaterno
<input type="text" value="Jesica"/>	<input type="text" value="Navarete"/>	<input type="text" value="Rocha"/>
Curp	Rfc	Foto
<input type="text" value="ty65uht6j"/>	<input type="text" value="65u876t5yh"/>	<input type="button" value="Choose file"/> No file chosen



Email	Celular	Username
<input type="text" value="JNavarrete@gmail.com"/>	<input type="text" value="5624568356"/>	<input type="text" value="Jess"/>
Password	Rol	
<input type="text" value="12434"/>	<input type="button" value="Administrador"/>	

```

[HttpGet]
public ActionResult Form(int? idUsuario)
{
    ML.Result resultRol = BL.Rol.GetAll();

    ML.Usuario usuario = new ML.Usuario();
    usuario.Vendedor = new ML.Vendedor();

    usuario.Rol = new ML.Rol();

    if (resultRol.Correct)
    {
        usuario.Rol.Roles = resultRol.Objects;
    }
    if (idUsuario == null)
    {
        return View(usuario);
    }
    else
    {
        ML.Result result = new ML.Result();
        using (var client = new HttpClient())
        {
            string urlApi = configuration["urlWebApi"];
            client.BaseAddress = new Uri(urlApi);

            var responseTask = client.GetAsync("Usuario/GetById/" + idUsuario);
            responseTask.Wait();

            var resultAPI = responseTask.Result;

            if (resultAPI.IsSuccessStatusCode)
            {
                var readTask = resultAPI.Content.ReadAsAsync<ML.Result>();
                readTask.Wait();
                ML.Usuario resultItemList = Newtonsoft.Json.JsonConvert.DeserializeObject<ML.Usuario>(readTask.Result.Object.ToString());
                result.Object = resultItemList;

                usuario = (ML.Usuario)result.Object;
                usuario.Rol.Roles = resultRol.Objects;
            }
        }
        return View(usuario);
    }
}

```

[HttpGet]: Esto es un atributo aplicado al método, indicando que responderá a las solicitudes HTTP GET.

La firma del método es `public ActionResult Form(int? idUsuario)`. Toma un parámetro entero opcional `idUsuario`.

`ML.Result resultRol = BL.Rol.GetAll();`: Esta línea parece estar obteniendo datos desde una capa de Lógica de Negocio (BL). Llama al método `GetAll()` en la clase `Rol` dentro de la BL y asigna el resultado a una variable `resultRol`. Según la convención de nombres, es probable que esté obteniendo una lista de roles.

```

public static ML.Result GetAll()
{
    ML.Result result = new ML.Result();

    try
    {
        using (DL.BienesRaicesSqlContext cnn = new DL.BienesRaicesSqlContext())
        {
            var query = cnn.Rols.FromSqlRaw($"RolGetAll").ToList();

            result.Objects = new List<object>();

            if (query != null)
            {
                foreach (var row in query)
                {
                    ML.Rol rol = new ML.Rol();

                    rol.IdRol = row.IdRol;
                    rol.Nombre = row.Nombre;

                    result.Objects.Add(rol);
                }
                result.Correct = true;
            }
        }
    }
    catch (Exception ex)
    {
        result.Correct = false;
        result.Ex = ex;
        result.ErrorMessage = "An error occurred while inserting the record into the table" + result.Ex;
        //throw;
    }

    return result;
}

```

public static ML.Result GetAll(): El método es público, estático y devuelve un objeto ML.Result. El ML.Result es una estructura de datos que parece ser utilizado para almacenar los resultados de la operación.

Se crea una instancia de ML.Result llamada result para almacenar el resultado de la operación.

```

public class Result
{
    public bool Correct { get; set; }

    public string ErrorMessage { get; set; } = string.Empty;

    public Exception? Ex { get; set; }

    public object? Object { get; set; }

    public List<object>? Objects { get; set; }
}

```

Dentro del bloque try, se utiliza un bloque using para crear una instancia del contexto de la base de datos DL.BienesRaicesSqlContext. El contexto de la base de datos permite interactuar con la base de datos utilizando Entity Framework Core

La variable query almacena los resultados de una consulta a la base de datos. La consulta se realiza utilizando FromSqlRaw para ejecutar un procedimiento almacenado en la base de

datos llamado "RolGetAll". La consulta se ejecuta y los resultados se convierten en una lista de objetos de tipo Rols.

Se inicializa una lista de objetos result.Objects para almacenar los roles recuperados de la base de datos. Parece que el tipo de objeto en esta lista es genérico, lo que permite almacenar cualquier tipo de objeto, pero en este caso, se llenará con objetos de tipo ML.Rol.

El código realiza un bucle foreach para recorrer todos los resultados obtenidos de la consulta y crea instancias de ML.Rol a partir de cada fila del resultado. Luego, estos objetos ML.Rol se agregan a la lista result.Objects

Si la consulta no devuelve resultados (query es nulo), el código no entra en el bucle foreach, y result.Correct se mantiene como falso.

Si la consulta devuelve resultados, result.Correct se establece como verdadero, indicando que la operación se realizó correctamente.

Si ocurre una excepción durante la ejecución del bloque try, el control pasa al bloque catch. En el bloque catch, result.Correct se establece como falso, y se captura la excepción y se asigna a result.Ex.

Finalmente, el método devuelve el objeto result que contiene la lista de roles recuperados de la base de datos y un indicador de si la operación fue exitosa o no

```
public class Usuario
{
    public int IdUsuario { get; set; }

    public string Username { get; set; } = null!;

    public string Password { get; set; } = null!;

    public bool Estatus { get; set; }

    public ML.Vendedor? Vendedor { get; set; }

    public ML.Rol? Rol { get; set; }

    public List<object>? Usuarios { get; set; }
}
```

A continuación, crea una nueva instancia de la clase `ML.Usuario` e inicializa sus propiedades `Vendedor` y `Rol`. Parece que `ML.Usuario` representa un objeto de usuario, y `Vendedor` y `Rol` son propiedades relacionadas con el usuario.

```
public class Usuario
{
    public int IdUsuario { get; set; }

    public string Username { get; set; } = null!;

    public string Password { get; set; } = null!;

    public bool Estatus { get; set; }

    public ML.Vendedor? Vendedor { get; set; }

    public ML.Rol? Rol { get; set; }

    public List<object>? Usuarios { get; set; }
}
```

```
public class Vendedor
{
    public int IdVendedor { get; set; }

    public string Nombre { get; set; } = null!;

    public string ApellidoPaterno { get; set; } = null!;

    public string ApellidoMaterno { get; set; } = null!;

    public string Curp { get; set; } = null!;

    public string? Rfc { get; set; }

    public string? Foto { get; set; }

    public string Email { get; set; } = null!;

    public string Celular { get; set; } = null!;

    public List<object>? Vendedores { get; set; }
}
```

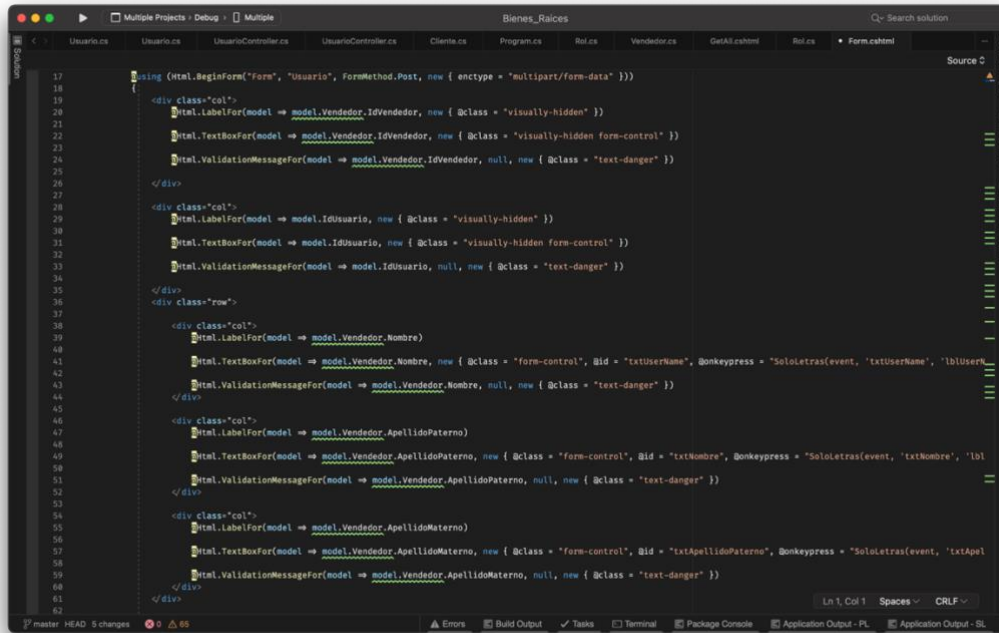
```
public class Rol
{
    public byte IdRol { get; set; }

    public string Nombre { get; set; } = null!;

    public List<object>? Roles { get; set; }
}
```

El código verifica si el resultado de resultRol es correcto (resultRol.Correct). Si es correcto, asigna la lista de roles obtenida en el paso 3 a la propiedad Roles del objeto Rol en usuario.

Luego verifica si el parámetro idUsuario es nulo. Si es nulo, significa que no se proporcionó un ID de usuario específico, y el método devuelve una vista con el objeto usuario (probablemente para mostrar un formulario para crear un nuevo usuario).



```
17 @using (Html.BeginForm("Form", "Usuario", FormMethod.Post, new { enctype = "multipart/form-data" }))
18
19 <div class="col">
20     @Html.LabelFor(model => model.Vendedor.IdVendedor, new { @class = "visually-hidden" })
21     @Html.TextBoxFor(model => model.Vendedor.IdVendedor, new { @class = "visually-hidden form-control" })
22     @Html.ValidationMessageFor(model => model.Vendedor.IdVendedor, null, new { @class = "text-danger" })
23 </div>
24
25 <div class="col">
26     @Html.LabelFor(model => model.IdUsuario, new { @class = "visually-hidden" })
27     @Html.TextBoxFor(model => model.IdUsuario, new { @class = "visually-hidden form-control" })
28     @Html.ValidationMessageFor(model => model.IdUsuario, null, new { @class = "text-danger" })
29 </div>
30 <div class="row">
31 <div class="col">
32     @Html.LabelFor(model => model.Vendedor.Nombre)
33     @Html.TextBoxFor(model => model.Vendedor.Nombre, new { @class = "form-control", @id = "txtNombre", @onkeypress = "SoloLetras(event, 'txtNombre', 'lblUserN"
34     @Html.ValidationMessageFor(model => model.Vendedor.Nombre, null, new { @class = "text-danger" })
35 </div>
36 <div class="col">
37     @Html.LabelFor(model => model.Vendedor.ApellidoPaterno)
38     @Html.TextBoxFor(model => model.Vendedor.ApellidoPaterno, new { @class = "form-control", @id = "txtNombre", @onkeypress = "SoloLetras(event, 'txtNombre', 'bl"
39     @Html.ValidationMessageFor(model => model.Vendedor.ApellidoPaterno, null, new { @class = "text-danger" })
40 </div>
41 <div class="col">
42     @Html.LabelFor(model => model.Vendedor.ApellidoMaterno)
43     @Html.TextBoxFor(model => model.Vendedor.ApellidoMaterno, new { @class = "form-control", @id = "txtApellidoPaterno", @onkeypress = "SoloLetras(event, 'txtApel"
44     @Html.ValidationMessageFor(model => model.Vendedor.ApellidoMaterno, null, new { @class = "text-danger" })
45 </div>
46 </div>
47 </div>
```

@using (Html.BeginForm("Form", "Usuario", FormMethod.Post, new { enctype = "multipart/form-data" })): Este código crea un formulario HTML utilizando el método BeginForm proporcionado por la clase Html. El formulario enviará los datos a la acción Form del controlador Usuario usando el método HTTP POST. El atributo enctype = "multipart/form-data" indica que el formulario puede contener datos binarios, como archivos adjuntos.

El formulario está contenido dentro de un bloque @using que asegura que el formulario se renderice correctamente.

Dentro del formulario, hay varios campos que se corresponden con propiedades del modelo Usuario y Vendedor. Cada campo utiliza las funciones de la clase Html para renderizar los elementos HTML adecuados. Por ejemplo, @Html.LabelFor se utiliza para renderizar etiquetas de etiquetas <label> para cada campo.

@Html.TextBoxFor se utiliza para renderizar campos de entrada <input type="text">, que permiten a los usuarios escribir datos en el formulario. Cada campo TextBoxFor está vinculado a una propiedad específica del modelo mediante expresiones lambda.

@Html.ValidationMessageFor se utiliza para mostrar mensajes de validación asociados con cada campo. Estos mensajes se mostrarán si la validación del modelo falla.

El código incluye un campo de entrada de tipo file para permitir a los usuarios seleccionar una imagen para el vendedor. Se utiliza `@Html.HiddenFor` para almacenar un valor oculto (`model.Vendedor.Foto`) que contendrá la imagen en formato base64 si ya se ha seleccionado una imagen.

`@Html.DropDownListFor` se utiliza para mostrar un menú desplegable (`<select>`) que permite a los usuarios seleccionar un rol para el usuario. El menú desplegable se rellena con opciones obtenidas del modelo `Rol.Roles`.

Al final del formulario, hay un botón de tipo submit que permite enviar los datos del formulario al servidor para su procesamiento. También hay un enlace (`@Html.ActionLink`) que redirige al usuario a la lista de usuarios (`GetAll`) en caso de que quieran cancelar la operación

Si `idUsuario` no es nulo, el método procede a obtener los detalles del usuario desde una API externa usando el ID proporcionado. Configura un `HttpClient` y realiza una solicitud GET asíncrona a la API para obtener los detalles del usuario al agregar el ID del usuario a la dirección de la API.

```
[Route("api/Usuario/GetById/{id}")]
[HttpGet]
public IActionResult GetById(int id)
{
    ML.Result result = BL.Usuario.GetById(id);

    if (resultCorrect)
    {
        return Ok(result);
    }
    else { return NotFound(); }
}
```

`[Route("api/Usuario/GetById/{id}")]`: Este atributo especifica la ruta de acceso a la cual responderá este método de acción. En este caso, el método responderá a las solicitudes GET enviadas a la ruta `"/api/Usuario/GetById/{id}"`, donde `"{id}"` es un marcador de posición para el ID del usuario que se desea obtener.

`[HttpGet]`: Este atributo indica que este método de acción responderá únicamente a solicitudes HTTP GET. Esto significa que el ID del usuario se pasará como parte de la URL.

public IActionResult GetById(int id): El método toma un parámetro int llamado id, que representa el ID del usuario que se desea obtener.

ML.Result result = BL.Usuario.GetById(id);: El método invoca el método estático GetById en la capa de lógica de negocios (BL) para obtener el usuario por el ID proporcionado. El resultado de esta operación se almacena en un objeto de tipo ML.Result.

```
public static ML.Result GetById(int idUsuario)
{
    ML.Result result = new ML.Result();

    try
    {
        using (DL.BienesRaicesSqlContext cnn = new DL.BienesRaicesSqlContext())
        {
            var query = cnn.Usuarios.FromSqlRaw($"UsuarioById {idUsuario}").ToList().FirstOrDefault();

            if (query != null)
            {
                ML.Usuario usuario = new ML.Usuario();

                usuario = new ML.Usuario();
                usuario.IdUsuario = query.IdUsuario;
                usuario.Username = query.Username;
                usuario.Password = query.Password;
                usuario.Estatus = query.Estatus.Value;

                usuario.Vendedor = new ML.Vendedor();
                usuario.Vendedor.IdVendedor = query.IdVendedor;
                usuario.Vendedor.Nombre = query.Nombre;
                usuario.Vendedor.ApellidoPaterno = query.ApellidoPaterno;
                usuario.Vendedor.ApellidoMaterno = query.ApellidoMaterno;
                usuario.Vendedor.Curp = query.Curp;
                usuario.Vendedor.Rfc = query.Rfc;
                usuario.Vendedor.Foto = query.Foto;
                usuario.Vendedor.Email = query.Email;
                usuario.Vendedor.Celular = query.Celular;

                usuario.Rol = new ML.Rol();
                usuario.Rol.IdRol = query.IdRol;
                usuario.Rol.Nombre = query.NombreRol;

                result.Object = usuario;

                result.Correct = true;
            }
        }
    }
    catch (Exception ex)
    {
        result.Correct = false;
    }
}
```

public static ML.Result GetById(int idUsuario): El método es público, estático y devuelve un objeto ML.Result. Toma un parámetro entero idUsuario, que representa el ID del usuario que se desea obtener.

Se crea una instancia de ML.Result llamada result para almacenar el resultado de la operación.

```

public class Result
{
    public bool Correct { get; set; }

    public string ErrorMessage { get; set; } = string.Empty;

    public Exception? Ex { get; set; }

    public object? Object { get; set; }

    public List<object>? Objects { get; set; }
}

```

Dentro del bloque try, se utiliza un bloque using para crear una instancia del contexto de la base de datos DL.BienesRaicesSqlContext. El contexto de la base de datos permite interactuar con la base de datos utilizando Entity Framework Core.

La variable query almacena el resultado de una consulta a la base de datos. La consulta se realiza utilizando FromSqlRaw para ejecutar un procedimiento almacenado en la base de datos llamado "UsuarioById" y se le pasa el parámetro idUsuario. La consulta se ejecuta y se obtiene una lista de objetos Usuarios, y luego se utilizaFirstOrDefault para obtener el primer objeto de la lista.

Si query no es nulo (es decir, se encontró un usuario con el ID proporcionado), el código crea una instancia del objeto ML.Usuario y asigna las propiedades del objeto Usuarios recuperado de la base de datos a las propiedades del objeto ML.Usuario.

Similarmente, se crea una instancia del objeto ML.Vendedor y se asignan las propiedades correspondientes del objeto Usuarios a las propiedades del objeto ML.Vendedor.

También se crea una instancia del objeto ML.Rol y se asignan las propiedades del objeto Usuarios a las propiedades del objeto ML.Rol.

El objeto usuario creado se asigna al objeto result.Object, que es un campo de la estructura ML.Result, para almacenar el usuario recuperado.

Se establece result.Correct en true para indicar que la operación se realizó correctamente.

Si ocurre una excepción durante la ejecución del bloque try, el control pasa al bloque catch. En el bloque catch, result.Correct se establece como false, y se captura la excepción y se asigna a result.Ex. Además, se asigna un mensaje de error al campo result.ErrorMessage

if (result.Correct): El código verifica si la operación de obtener el usuario fue exitosa, comprobando si la propiedad Correct del objeto result es true. Si la operación fue exitosa, se ejecuta el bloque de código dentro de este if.

else { return NotFound(); }: Si la operación de obtener el usuario no fue exitosa (es decir, result.Correct es false), el código devuelve una respuesta HTTP 404 (NotFound) al cliente. Esto significa que el cliente recibirá una respuesta con un código de estado 404, indicando que el recurso solicitado (es decir, el usuario con el ID proporcionado) no fue encontrado.

Después de recibir la respuesta de la API, verifica si la respuesta fue exitosa (resultAPI.IsSuccessStatusCode). Si es exitosa, lee el contenido de la respuesta y lo deserializa a una instancia de ML.Result, luego extrae el objeto de usuario y lo asigna a la variable usuario.

Por último, el método devuelve una vista con el objeto usuario

```

[HttpPost]
public ActionResult Form(ML.Usuario usuario)
{
    IFormFile file = Request.Form.Files["inpImagen"];

    if (file != null)
    {
        usuario.Vendedor.Foto = Convert.ToBase64String(Convert.ToBytes(file));
    }

    if (usuario.IdUsuario == 0)
    {
        //add
        using (var client = new HttpClient())
        {
            string urlApi = configuration["urlWebApi"];
            client.BaseAddress = new Uri(urlApi);

            var postTask = client.PostAsJsonAsync<ML.Usuario>("Usuario/Add/", usuario);

            postTask.Wait();

            var result = postTask.Result;

            if (result.IsSuccessStatusCode)
            {
                ViewBag.Message = "Registro correctamente insertado";
                return PartialView("Modal");
            }
            else
            {
                ViewBag.Message = "Ocurrió un error al Insertar el registro";
                return PartialView("Modal");
            }
        }
    }
}

```

[HttpPost]: Este atributo indica que este método de acción responderá únicamente a solicitudes HTTP POST. Esto significa que se espera que los datos del usuario se envíen en el cuerpo de la solicitud.

public ActionResult Form(ML.Usuario usuario): El método toma un objeto de tipo ML.Usuario como parámetro, que representa los datos del usuario enviados en el cuerpo de la solicitud HTTP.

IFormFile file = Request.Form.Files["inpImagen"];; En esta línea, se obtiene un archivo (IFormFile) del cuerpo de la solicitud HTTP con el nombre "inpImagen". Esta línea asume que el cliente envía un archivo adjunto con el formulario de usuario para actualizar o agregar una imagen para el vendedor. Si no se proporciona un archivo, file será nulo.

Si file no es nulo (es decir, se proporcionó una imagen), el código convierte el contenido del archivo en formato base64 y lo asigna a la propiedad usuario.Vendedor.Foto. Esto sugiere que la imagen del vendedor se está almacenando en formato base64 en la propiedad Foto del objeto Vendedor dentro del objeto Usuario.

```
public static byte[] ConvertToBytes(IFormFile imagen)
{
    using var fileStream = imagen.OpenReadStream();

    byte[] bytes = new byte[fileStream.Length];
    fileStream.Read(bytes, 0, (int)fileStream.Length);

    return bytes;
}
```

`public static byte[] ConvertToBytes(IFormFile imagen)`: El método es público, estático y devuelve un arreglo de bytes (`byte[]`). Toma un parámetro `IFormFile` llamado `imagen`, que representa el archivo adjunto que se desea convertir.

`using var fileStream = imagen.OpenReadStream();`: Aquí, se utiliza un bloque `using` para crear una instancia de `FileStream` a partir del objeto `imagen` utilizando el método `OpenReadStream()`. El bloque `using` garantiza que el `FileStream` se cierre correctamente después de que se complete su uso, lo que es importante para liberar recursos adecuadamente.

`byte[] bytes = new byte[fileStream.Length];`: Se crea un arreglo de bytes (`byte[]`) con un tamaño igual a la longitud del `FileStream`. La longitud del `FileStream` representa el tamaño del archivo adjunto.

`fileStream.Read(bytes, 0, (int)fileStream.Length);`: Se lee el contenido del `FileStream` y se almacena en el arreglo de bytes `bytes`. El método `Read` lee los datos del `FileStream` y los coloca en el arreglo de bytes `bytes`. El segundo argumento (0 en este caso) es el índice de inicio donde se almacenarán los bytes leídos en el arreglo `bytes`, y el tercer argumento es la cantidad de bytes que se leerán (`fileStream.Length` en este caso, ya que queremos leer todo el contenido del archivo).

Finalmente, se devuelve el arreglo de bytes `bytes` que contiene el contenido del archivo adjunto en formato de bytes

La siguiente parte del código maneja dos escenarios diferentes, uno para agregar un nuevo usuario y otro para actualizar un usuario existente.

Si `usuario.IdUsuario == 0`, significa que el ID del usuario es 0, lo que indica que se está agregando un nuevo usuario. En este caso, el código realiza una solicitud HTTP POST a una API web externa para agregar el usuario, utilizando el objeto `HttpClient`.

```

[Route("api/Usuario/Add")]
[HttpPost]
public IActionResult Add([FromBody] ML.Usuario usuario)
{
    ML.Result result = BL.Usuario.Add(usuario);

    if (result.Correct)
    {
        return Ok(result.Objects);
    }
    else { return NotFound(result); }
}

```

[Route("api/Usuario/Add")]: Este atributo especifica la ruta de acceso a la cual responderá este método de acción. En este caso, el método responderá a las solicitudes POST enviadas a la ruta "/api/Usuario/Add".

[HttpPost]: Este atributo indica que este método de acción responderá únicamente a solicitudes HTTP POST. Esto significa que se espera que los datos del usuario se envíen en el cuerpo de la solicitud.

public IActionResult Add([FromBody] ML.Usuario usuario): El método toma un objeto de tipo ML.Usuario como parámetro. La anotación [FromBody] indica que el objeto usuario se deserializará a partir del cuerpo de la solicitud HTTP. Esto significa que se espera que el cliente envíe un objeto JSON en el cuerpo de la solicitud que represente un objeto de tipo ML.Usuario.

ML.Result result = BL.Usuario.Add(usuario); El método invoca el método Add en la capa de lógica de negocios (BL) para agregar el usuario proporcionado al sistema. El resultado de esta operación se almacena en un objeto de tipo ML.Result

```

public static ML.Result Add(ML.Usuario usuario)
{
    ML.Result result = new ML.Result();

    try
    {
        using (DL.BienesRaicesSqlContext cnn = new DL.BienesRaicesSqlContext())
        {
            int query = cnn.Database.ExecuteSqlRaw($"UsuarioAdd '{usuario.Vendedor.Nombre}', '{usuario.Vendedor'");

            if (query > 0)
            {
                result.Correct = true;
            }
        }
    }
    catch (Exception ex)
    {
        result.Correct = false;
        result.Ex = ex;
        result.ErrorMessage = "An error occurred while inserting the record into the table" + result.Ex;
        //throw;
    }

    return result;
}

```

`public static ML.Result Add(ML.Usuario usuario)`: El método es público, estático y devuelve un objeto `ML.Result`. Toma un objeto de tipo `ML.Usuario` llamado `usuario`, que representa el usuario que se desea agregar a la base de datos.

Se crea una instancia de `ML.Result` llamada `result` para almacenar el resultado de la operación.

```
public class Result
{
    public bool Correct { get; set; }

    public string ErrorMessage { get; set; } = string.Empty;

    public Exception? Ex { get; set; }

    public object? Object { get; set; }

    public List<object>? Objects { get; set; }
}
```

Dentro del bloque `try`, se utiliza un bloque `using` para crear una instancia del contexto de la base de datos `DL.BienesRaicesSqlContext`. El contexto de la base de datos permite interactuar con la base de datos utilizando `Entity Framework Core`.

`cnn.Database.ExecuteSqlRaw(...)`: En esta línea, se ejecuta una consulta SQL cruda utilizando el método `ExecuteSqlRaw` del objeto `Database` del contexto de la base de datos. La consulta SQL se compone de una cadena que inserta los valores de las propiedades del objeto `usuario` en la tabla correspondiente. El método `ExecuteSqlRaw` ejecuta la consulta y devuelve el número de filas afectadas.

Si `query > 0`, significa que la consulta fue exitosa y se agregó al menos una fila a la tabla. En este caso, se establece `result.Correct` en `true`, lo que indica que la operación fue exitosa.

Si ocurre una excepción durante la ejecución del bloque `try`, el control pasa al bloque `catch`. En el bloque `catch`, `result.Correct` se establece como `false`, y se captura la excepción y se asigna a `result.Ex`. Además, se asigna un mensaje de error al campo `result.ErrorMessage`.

Finalmente, el método devuelve el objeto `result` que contiene información sobre el resultado de la operación de agregar el usuario

`if (result.Correct)`: El código verifica si la operación de agregar el usuario fue exitosa, comprobando si la propiedad `Correct` del objeto `result` es `true`. Si la operación fue exitosa, se ejecuta el bloque de código dentro de este

`else { return NotFound(result); }`: Si la operación de agregar el usuario no fue exitosa (es decir, `result.Correct` es `false`), el código devuelve una respuesta HTTP 404 (`NotFound`) al

cliente, junto con el objeto result. Esto significa que el cliente recibirá información sobre el error o el resultado de la operación fallida en la respuesta

```
else
{
    //update
    using (var client = new HttpClient())
    {
        string urlApi = configuration["urlWebApi"];
        client.BaseAddress = new Uri(urlApi);

        var postTask = client.PutAsJsonAsync<ML.Usuario>("Usuario/Update/" + usuario.IdUsuario, usuario);
        postTask.Wait();

        var result = postTask.Result;

        if (result.IsSuccessStatusCode)
        {
            ViewBag.Message = "Registro correctamente actualizado";
            return PartialView("Modal");
        }
        else
        {
            ViewBag.Message = "Ocurrió un error al actualizar el actualizado";
            return PartialView("Modal");
        }
    }
}
```

Si usuario.IdUsuario no es 0, significa que el ID del usuario no es 0, lo que indica que se está actualizando un usuario existente. En este caso, el código realiza una solicitud HTTP PUT a la misma API web externa para actualizar el usuario con el ID proporcionado.

El código utiliza HttpClient para enviar los datos del usuario en el cuerpo de la solicitud, ya sea en formato JSON para el POST (PostAsJsonAsync) o en formato JSON para el PUT (PutAsJsonAsync).

```
[Route("api/Usuario/Update/{id}")]
[HttpPut]
public IActionResult Update(int id, [FromBody] ML.Usuario usuario)
{
    ML.Result result = BL.Usuario.Update(usuario);

    if (result.Correct)
    {
        return Ok(result.Objects);
    }
    else { return NotFound(result); }
}

[Route("api/Usuario/GetById/{id}")]
[HttpGet]
```

[Route("api/Usuario/Update/{id}")] : Este atributo especifica la ruta de acceso a la cual responderá este método de acción. En este caso, el método responderá a las solicitudes PUT enviadas a la ruta "/api/Usuario/Update/{id}", donde "{id}" es un marcador de posición para el ID del usuario que se desea actualizar.

[HttpPut]: Este atributo indica que este método de acción responderá únicamente a solicitudes HTTP PUT. Esto significa que el ID del usuario se pasará como parte de la URL y los datos del usuario actualizado se enviarán en el cuerpo de la solicitud HTTP.

public IActionResult Update(int id, [FromBody] ML.Usuario usuario): El método toma dos parámetros: un int llamado id, que representa el ID del usuario que se desea actualizar, y un objeto de tipo ML.Usuario llamado usuario, que representa los datos actualizados del usuario enviados en el cuerpo de la solicitud HTTP.

ML.Result result = BL.Usuario.Update(usuario);: El método invoca el método estático Update en la capa de lógica de negocios (BL) para actualizar el usuario proporcionado. El resultado de esta operación se almacena en un objeto de tipo ML.Result.

```
public static ML.Result Update(ML.Usuario usuario)
{
    ML.Result result = new ML.Result();

    try
    {
        using (DL.BienesRaicesSqlContext cnn = new DL.BienesRaicesSqlContext())
        {
            int query = cnn.Database.ExecuteSqlRaw($"UsuarioUpdate {usuario.Vendedor.IdVendedor}, '{usuario.Vendedor.Nombre}'");

            if (query > 0)
            {
                result.Correct = true;
            }
        }
    }
    catch (Exception ex)
    {
        result.Correct = false;
        result.Ex = ex;
        result.ErrorMessage = "An error occurred while inserting the record into the table" + result.Ex;
        //throw;
    }

    return result;
}
```

public static ML.Result Update(ML.Usuario usuario): El método es público, estático y devuelve un objeto ML.Result. Toma un objeto de tipo ML.Usuario llamado usuario, que representa el usuario que se desea actualizar en la base de datos.

Se crea una instancia de ML.Result llamada result para almacenar el resultado de la operación.

```
public class Result
{
    public bool Correct { get; set; }

    public string ErrorMessage { get; set; } = string.Empty;

    public Exception? Ex { get; set; }

    public object? Object { get; set; }

    public List<object>? Objects { get; set; }
}
```

Dentro del bloque try, se utiliza un bloque using para crear una instancia del contexto de la base de datos DL.BienesRaicesSqlContext. El contexto de la base de datos permite interactuar con la base de datos utilizando Entity Framework Core.

`cnn.Database.ExecuteSqlRaw(...)`: En esta línea, se ejecuta una consulta SQL cruda utilizando el método `ExecuteSqlRaw` del objeto `Database` del contexto de la base de datos. La consulta SQL se compone de una cadena que actualiza los valores de las propiedades del objeto usuario en la tabla correspondiente. El método `ExecuteSqlRaw` ejecuta la consulta y devuelve el número de filas afectadas.

Si `query > 0`, significa que la consulta fue exitosa y se actualizó al menos una fila en la tabla. En este caso, se establece `result.Correct` en `true`, lo que indica que la operación fue exitosa.

Si ocurre una excepción durante la ejecución del bloque try, el control pasa al bloque catch. En el bloque catch, `result.Correct` se establece como `false`, y se captura la excepción y se asigna a `result.Ex`. Además, se asigna un mensaje de error al campo `result.ErrorMessage`

`if (result.Correct)`: El código verifica si la operación de actualizar el usuario fue exitosa, comprobando si la propiedad `Correct` del objeto `result` es `true`. Si la operación fue exitosa, se ejecuta el bloque de código dentro de este `if`.

`else { return NotFound(result); }`: Si la operación de actualizar el usuario no fue exitosa (es decir, `result.Correct` es `false`), el código devuelve una respuesta HTTP 404 (`NotFound`) al cliente, junto con el objeto `result`. Esto significa que el cliente recibirá información sobre el error o el resultado de la operación fallida en la respuesta

Después de realizar la solicitud HTTP, el código verifica si la operación fue exitosa (`result.IsSuccessStatusCode`). Si es exitosa, se muestra un mensaje de éxito utilizando `ViewBag.Message` y devuelve una vista parcial llamada "Modal" utilizando `PartialView("Modal")`. Esta vista parcial probablemente mostrará un mensaje de éxito o error en una ventana emergente (modal).

Si la operación no fue exitosa (por ejemplo, el código de estado HTTP no es 200), también muestra un mensaje de error en `ViewBag.Message` y devuelve la vista parcial "Modal"