educación IT
CODE YOUR FUTURE

Programación Web .NET Core

Módulo 7

Desarrollo de modelos

Desarrollo de modelos

Descripción general del módulo

La mayoría de las aplicaciones web interactúan con varios tipos de datos u objetos. Una aplicación de comercio electrónico, por ejemplo, administra productos, carritos de compras, clientes y pedidos. Una aplicación de redes sociales podría ayudar a administrar usuarios, actualizaciones de estado, comentarios, fotos y videos. Un blog se utiliza para administrar las entradas del blog, comentarios, categorías y etiquetas.

Cuando se escribe una aplicación web *Model-View-Controller (MVC)*, dentro del modelo, se crea un *modelo clase* para cada tipo de objeto. **La clase modelo describe las propiedades de cada tipo de objeto** y puede incluir lógica empresarial que coincida con los procesos empresariales. Por tanto, el **modelo** es un pilar fundamental en una aplicación MVC.

Crear modelos MVC

Crear modelos MVC

Un *modelo MVC* es una colección de clases. Cuando se crea una ***clase de modelo***, se definen las **propiedades y métodos** que son necesarios para el **tipo de objeto** que describe la clase de modelo. Necesitas saber cómo crear y describir modelos, y cómo modificar la forma en que un MVC crea una instancia de clase de modelo cuando ejecuta tu aplicación web.

También es importante saber **cómo los controladores pasan modelos a vistas y cómo las vistas pueden representar los datos almacenados en un modelo en el navegador**.

Desarrollo de modelos

Los programadores deben crear sus modelos en una **carpeta llamada *Models***. Porque MVC se basa en la convención. Recomendamos adherirse a las convenciones.

Veamos las líneas de código de la pantalla siguiente, las cuales ilustran cómo crear una clase modelo, llamada **`Photo`**.

La clase de modelo Photo

```
public class Photo
{
public int PhotoID{ get; set; }
public string Title{ get; set; }
public byte[] PhotoFile { get; set; }
public string Description{ get; set; }
public Date TimeCreatedDate{ get; set; }
public string Owner{ get; set; }
public virtual ICollection<Comment>Comments{ get; set; }
}
```

Observa que **la clase modelo no hereda de ninguna otra clase**. Además, se han creado propiedades públicas en el modelo y se incluye el tipo de dato, como **`int`** o **`string`** en su declaración.

También puedes crear propiedades de solo lectura. La clase **`Photo`** incluye una propiedad **`Comments`**. Esto se declara como una colección de objetos **`Comment`** e implementa un lado de la relación entre **`Photos`** y **`Comments`**.

Las líneas de código de la siguiente pantalla, ilustran cómo se puede implementar la clase de modelo **`Comment`**.

La clase de modelo Comentario

```
public class Comment
{
public int CommentID{ get; set; }
public int PhotoID{ get; set; }
public string UserName{ get; set; }
public string Subject{ get; set; }
public string Body{ get; set; }
public virtual Photo{ get; set; }
}
```

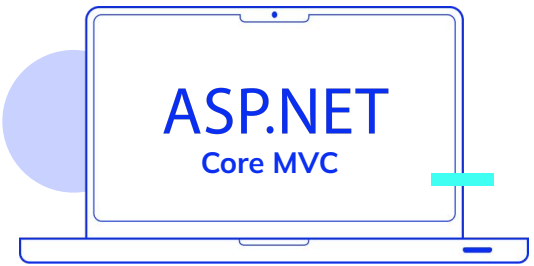
Observa que **la clase `Comment` incluye una propiedad `PhotoID`**. Esta propiedad almacena la identificación de la **`Photo`** que el usuario comentó y vincula el **`Comment`** a una sola **`Photo`**.

La clase incluye una propiedad **`Photo`**, que devuelve el objeto **`Photo`** con el que se relaciona el **`Comment`**. Las propiedades implementan el otro lado de la relación entre **`Photos`** y **`Comments`**.

Pasar modelos a vistas

Pasar modelos a vistas

Cuando una aplicación web **ASP.NET Core MVC** recibe una solicitud de un navegador web, un *objeto instancia* del controlador responde a la solicitud. A continuación, la aplicación web ASP.NET Core MVC determina el método de acción que debe llamarse en el objeto controlador. A menudo, **la acción crea una nueva instancia de una clase modelo, que se puede pasar a una *vista* para mostrar resultados para el usuario**.



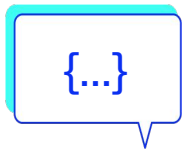
Práctica recomendada: normalmente, la más sencilla

Los objetos pasan del modelo a la vista. Sin embargo, para aplicaciones grandes, esta no es una práctica recomendada. Recomendamos que utilice **`ViewModels`** para separar la presentación del dominio.

Veamos el ejemplo de código siguiente, el cual muestra una clase de modelo simple.

Una clase modelo simple

```
namespace ModelNamespace
{
public class SomeModel
   {
public string Text { get; set; }
   }
}
```



El siguiente código muestra un controlador que crea un **modelo** y lo pasa a una **vista**:

```
public class HomeController :Controller
{
   [Route("Home/Index")]
public IActionResultIndex()
   {
SomeModelmodel = new SomeModel() { Text = "sometext" };
return View(model);
   }
}
```

Puedes utilizar el operador **`Razor@model`** para especificar el tipo de modelo de dominio fuertemente tipado de la vista. Accede al valor de una propiedad en el objeto de dominio, utilizando **`@Model.PropertyName`**.

El siguiente código muestra una vista que recibe un modelo del tipo **SomeModel** y lo usa para generar una respuesta:

Una vista que usa un modelo

```
@model ModelNamespace.SomeModel
@Model.Text
```

Si un usuario solicita la **URL relativa**, la cual es **/Home/Index**, se muestra la siguiente cadena en la pantalla: ***sometext***.

Pasar una colección de elementos

Algunas vistas muestran varias instancias de una clase de modelo. Por ejemplo, se muestra en una página de catálogo de productos, varias instancias de la clase de modelo Producto. En tales casos, el controlador pasa una lista de objetos modelo a la vista, en lugar de un único objeto modelo. Por lo general, recorre los elementos de la lista con un **loop foreach**.

El siguiente código muestra un controlador que crea una **lista de elementos** y los pasa a una **vista**:

```
public class HomeController :Controller
{
  [Route("Home/Index")]
public IActionResultIndex()
  {
SomeModel item1 = new SomeModel() { Text = "firstitem" };
SomeModel item2 = new SomeModel() { Text = "seconditem" };
List<SomeModel>items = new List<SomeModel>() { item1, item2 };
return View(items);
  }
}
```

Una vista que recibe una colección de elementos

```
@modelIEnumerable<ModelNamespace.SomeModel>
@foreach (varitem in @Model)
{
<div>@item.Text</div>
}
```

Si un usuario solicita la URL relativa, **`/Home/Index`**, se mostrará el siguiente texto en el navegador:

```
first item
second ítem
```

Pasar un modelo a una vista diferente

El método **`Controller.View`** está sobrecargado. Los ejemplos anteriores utilizaron una versión de este método que obtuvo solo una instancia de un modelo como parámetro. Cuando usa esta versión de método `Controller.View`, no tiene que especificar el nombre de la vista porque es implícitamente idéntico al nombre de la acción. Sin embargo, puede utilizar otra versión del método `Controller.View` para llamar a una vista cuyo nombre es diferente del nombre de la acción. Cuando usa esta versión, debe especificar explícitamente el nombre de la vista.

El siguiente código muestra un controlador que crea un modelo y lo pasa a una vista. El nombre de la **acción** es **`Index`**, mientras que el nombre de la **vista** es **`Display`**:

Pasar un modelo a la vista de visualización

```
public class HomeController :Controller
{
   [Route("Home/Index")]
public IActionResultIndex()
   {
SomeModelmodel = new SomeModel() { Text = "sometext" };
return View(“Display”, model);
    }
}
```

El siguiente código muestra una vista que recibe un modelo del tipo **SomeModel** y lo utiliza para generar la respuesta:

```
El archivo Display.cshtml
@modelModelNamespace.SomeModel
@Model.Text
```

Si un usuario solicita la URL relativa, la cual es **/Home/Index**, se mostrará la siguiente cadena en la pantalla:

```
some text
```

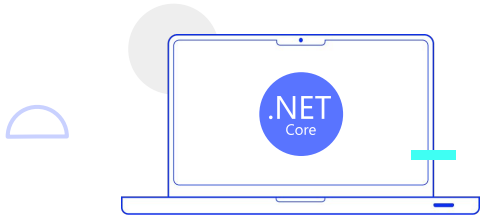
educación IT
CODE YOUR FUTURE

Vinculación de vistas a clases de modelo y visualización de datos

Vinculación de vistas a clases de modelo y visualización de datos

Una aplicación **ASP.NET Core MVC**, utiliza **vistas** para definir la interfaz de usuario. Las vistas están diseñadas para mostrar propiedades de una clase de modelo y se llaman ***vistas fuertemente tipadas***. Puedes vincular las vistas a la clase de modelo. Otras vistas pueden mostrar diferentes propiedades de la clase modelo.

Puede ocurrir que no se use una clase modelo. Estos tipos de vistas se denominan ***vistas dinámicas*** y no se pueden vincular a una clase de modelo específica. Es importante comprender cómo escribir código Razor para ambas vistas.



Vistas fuertemente tipadas

A menudo, **cuando se crea una vista, la acción del controlador pasará un objeto de una clase modelo específica**. Por ejemplo, si estás creando la vista `Display` para la acción `Display` en el controlador `Product`, desde el código de acción, se pasará un objeto `Product` a la vista. En tales casos, puedes crear una ***vista fuertemente tipada***, **que incluye una declaración de la clase modelo**. Cuando declaras la clase de modelo en la vista, Microsoft Visual Studio proporciona **`IntelliSense`** y **comprobación de errores** mientras escribes el código, porque puede comprobar las propiedades de la clase modelo. Esto también simplifica la resolución de problemas de errores en tiempo de ejecución. Por lo tanto, siempre que puedas, crea vistas fuertemente tipadas porque estas funciones adicionales del `IntelliSense` y de verificación de errores pueden ayudarte a reducir errores de codificación. Una vista fuertemente tipada solo funciona con un objeto de la clase modelo en su declaración.

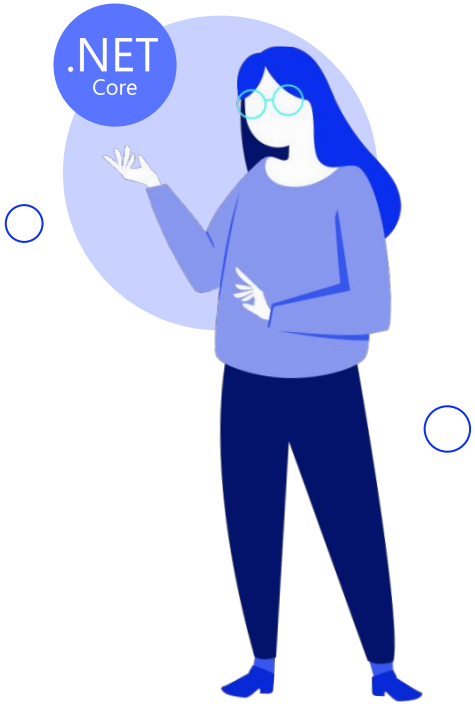
En los archivos de vista, puedes acceder a las propiedades del objeto de modelo con la palabra clave **`Model`**. Para acceder a una propiedad en un modelo, utiliza: **`@Model.PropertyName`**.

Usar vistas dinámicas

Es posible que desees **crear una vista que pueda mostrar más de una clase de modelo**. Por ejemplo, puedes tener una clase de modelo denominada **`Product`** para tus propios productos y otra clase de modelo llamada **`ThirdPartyProduct`** para productos de otros proveedores. Quieres crear una vista que pueda mostrar tus productos y los de terceros. Estas clases de modelo pueden tener algunas propiedades, mientras que otras pueden diferir. Por lo tanto, `Product`, así como `ThirdPartyProduct,` pueden incluir la propiedad **`Price`**, mientras que solo la clase de modelo `ThirdPartyProduct` incluye la propiedad llamada **`Supplier`**. Además, a veces, la acción del controlador no pasa ningún modelo a la vista. En el sitio, la página ***Index*** es el ejemplo más común de esa vista.

En tales casos, puedes crear una ***vista dinámica***. **Una vista dinámica no incluye la declaración `@model` en la parte superior de la página**. Más tarde, puedes optar por agregar la declaración `@model` para cambiar la vista a una vista fuertemente tipada.

Cuando creas vistas dinámicas, Visual Studio no proporciona muchos comentarios ni errores de `IntelliSense` porque no puede comprobar las propiedades de la clase de modelo para verificar el código. En tales escenarios, es tu responsabilidad asegurarte de que se accede sólo a las propiedades que existen. Para acceder a una propiedad que puede existir o no, como la propiedad **`ThirdPartyProduct.Supplier`** en el ejemplo anterior, **comprueba la propiedad para `null` antes de usarla**.



Uso del HTML Helper@Html.EditorForModel

En muchos casos, el **modelo** es la **base para construir la interfaz de usuario**. Una vista puede representar el valor de las propiedades del modelo mediante la directiva **`@Model`**. Sin embargo, existen varios **HTML Helpers** y **Tag Helpers** que puedes utilizar para simplificar el trabajo con los modelos en vistas.

Por ejemplo, el **`@Html.EditorForModel`** El HTML Helper devuelve un elemento de entrada HTML para cada propiedad del modelo.

El siguiente código muestra una clase de modelo simple:

```
namespace ModelNamespace
{
public class Person
   {
public string FirstName{ get; set; }
public string LastName{ get; set; }
   }
}
```

El siguiente código muestra una clase de controlador simple:

```
public class PersonController :Controller
{
[Route("Person/GetName")]
public IActionResultGetName()
{
return View();
}
}
```

El código siguiente, muestra cómo usar el **HTML helper** es **`@html.EditorForModel`** en una vista, para mostrar todas propiedades de un modelo:

```
@model ModelNamespace.Person
<form action="/Person/GetName" method="post">
@Html.EditorForModel()
<input type="submit" value="Submit my name" />
</form>
```

Si un usuario solicita la URL relativa siguiente: **`/Person/GetName`**, se muestra un **formulario** en el navegador. El formulario contiene dos cajas de texto, uno de ellos representa la propiedad **`FirstName`** del modelo y el otro representa la propiedad **`LastName`** del modelo y un botón de envío. Si un usuario ingresa valores en las cajas de texto y hace clic en el botón enviar, se produce una devolución de datos (el modelo) a la acción **`GetName`** en controlador **`PersonController`**.

Uso del asistente de HTML @ Html.BeginForm

En el ejemplo anterior, un formulario se escribió explícitamente en la vista. Al trabajar con ASP.NET Core MVC, el uso del HTML Helper **`@Html.BeginForm`** se considera un mejor enfoque.

El siguiente código muestra cómo usar el HTML Helper **`@Html.BeginForm`** en una vista para representar un formulario implícitamente:

Uso del ayudante de HTML @ html.BeginForm

```
@model ModelNamespace.Person
@using (Html.BeginForm())
{
@Html.EditorForModel()
<input type="submit" value="Submitmyname" />
}
```

educación IT
CODE YOUR FUTURE

¡Sigamos trabajando!