ÚLTIMA ACTUALIZACIÓN DEL DOCUMENTO: 26/12/2024

UNIVERSIDAD DE ALICANTE | SERVICIO DE INFORMÁTICA

VALIDANDO MODELO

MVC .NET CORE

Servicio de informática | Universidad de Alicante
03690 | Campus de Sant Vicent del Raspeig | España
https://si.ua.es/

CONTENIDO

CONTROL DEL DOCUMENTO

INFORMACIÓN GENERAL

Título	Validando todos los modelos que recibimos en aplicaciones MVC .NET Core
Creado por	Andrés Vallés
Revisado por	
Lista de distribución	
Nombre fichero	

HISTÓRICO DE REVISIONES

Versión	Fecha	Autor	Observaciones
001	23/12/2024	Andrés Vallés	

INTRODUCCIÓN

Uno de los mayores errores que podemos cometer es que creamos que validar la información en el cliente es suficiente para tener aplicaciones seguras.

La parte cliente es muy vulnerable y debemos SIEMPRE volver a validar que la información llega correctamente en la parte servidor. La mayoría de los ataques que reciben nuestras aplicaciones son para encontrar vulnerabilidades en los campos que recibimos.

DATAANNOTATIONS

Las dataannotations son la forma más sencilla de poner reglas a cada uno de los campos de nuestro modelo para facilitar su validación.

Con la forma de trabajar actual, las anotaciones las ponemos en nuestro propio modelo, antes de cada uno de los campos.

DISPLAYNAME O DISPLAY

Podemos indicar el nombre de los campos. Por defecto coge los mismo que lo de la base de datos o modelo

```
1  [DisplayName("Nombre Campo")]
2  [Display(Name = "Nombre Campo")]
3  [Display(Name = "tituloLabel", ResourceType = typeof(Resources.modelos))]
```

REQUIRED

Indicamos que el campo es obligatorio

```
1  [Required]
2  public string Email { get; set; }
```

STRINGLENGTH

Indicamos el número máximo de caracteres que se puede escribir en ese campo. Es una forma de evitar el mensaje de desbordamiento que se produce en el momento que sobrepasamos el tamaño permitido.

```
1  [StringLength(200)]
2  public string Email { get; set; }
```

REGULAREXPRESSION

En caso de que queramos que la información que se introduzca cumpla unos criterios, podemos usar las expresiones regulares. Por ejemplo, si queremos que el correo sea de la Universidad de Alicante usaríamos.

```
1  [RegularExpression(@"^[0-9a-zA-Z]([-\\.\\w]*[0-9a-zA-Z])*@(ua)\\.(es)$")]
2  public string Email { get; set; }
```

RANGE

Limitamos un valor mínimo y máximo para un determinado valor. Si queremos que solo se puedan seleccionar entre 2 y 10 asignaturas usaríamos.

```
1  [Range(2, 10)]
2  public int Asignaturas { get; set; }
```

OCULTAR CAMPOS

En muchas ocasiones queremos que ciertos campos aparezcan ocultos al usuario

```
1  [HiddenInput(DisplayValue = false)]
```

PERSONALIZACIÓN DE MENSAJES

Aunque los mensajes de error son bastantes explícitos, lo más normal es que los personalicemos. La forma de hacerlo es incluir dentro del propio atributo con ErrorMessage = "Texto del mensaje". Se separa con una , (coma).

```
[RegularExpression(@"^[0-9a-zA-Z]([-\\.\\w]*[0-9a-zA-Z])*@(ua)\\.(es)$",
ErrorMessage = "El formato del correo no se corresponde con uno de la Universidad de Alicante.")]
public string Email { get; set; }
```

Si queremos mostrar dentro del mensaje alguno de los valores por los que se personaliza (por ejemplo, el tamaño máximo) aparezca sin tener que introducirlo de nuevo, le hacemos referencia con {posición donde se encuentra el valor}.

```
[StringLength(200, ErrorMessage = "El nombre no puede ser mayor de {0} caracteres.")]
public string NombreCompleto { get; set; }
```

PERSONALIZACIÓN DE LOS MENSAJES POR IDIOMA

Ya vimos como personalizar las descripciones en diferentes idiomas con los ficheros de recursos. Para hacerlo con los mensajes de error podemos aplicar la misma técnica, indicamos en ErrorMessageResourceType = typeof([carpeta de recursos].[nombre fichero de recursos sin extensión]) y en ErrorMessageResourceName = “[Nombre de la clave dentro del fichero de recursos]”.

```
[RegularExpression(@"^[0-9a-zA-Z]([-\\.\\w]*[0-9a-zA-Z])*@(ua)\\.(es)$",
ErrorMessageResourceType = typeof(Resources.Alumno),
ErrorMessageResourceName = "ErrorFormatoCorreoUA")]
[Required(ErrorMessageResourceType = typeof(Resources.Resources),
ErrorMessageResourceName = "PasswordRequired")]
public string Email { get; set; }
```

TIPOS DE DATOS

ASP.NET MVC dispone un atributo para indicar el formato del campo DataType. Los valores que puede tomar son:

- DateTime. Muestra la fecha y la hora de un valor de tipo DateTime
- Date. Muestra la parte de la fecha de un valor de tipo DateTime
- Time. Muestra la parte de la hora de un valor de tipo DateTime
- Text. Muestra un campo de texto sencillo
- MultilineText. Muestra el campo como un TextArea
- Password. Campo para escribir contraseñas. Todo lo que escribamos aparecerán como asteriscos.

- Url. Muestra el valor como una dirección URL.
- EmailAddress. Muestra el valor como una dirección de correo electrónico.
- PhoneNumber. Muestra el valor como un número de teléfono.
- Currency. Muestra a cantidad como moneda del país. En nuestro caso pospone el símbolo del euro.
- ImageUrl. Muestra el valor como una dirección a una imagen.

La forma de usarlo es como un atributo más del campo.

```
1  [DataType(DataType.Password)]
2  public string Password { get; set; }
```

Lo que nos llama la atención cuando comencemos a probarlo, es que la mayoría no tienen ningún efecto en la visualización.

NOVEDADES EN CORE 8

MINLENGTH Y MAXLENGTH PARA COLECCIONES

Nos permite indicar el número de elementos mínimo y máximo de una variable que permite una colección.

Por ejemplo, si queremos que tenga al menos un perfil, pero no más de tres.

```
1  [MinLength(1, ErrorMessage = "Una persona debe tener al menos un perfil")]
2  [MaxLength(3, ErrorMessage = "Una persona no puede tener más de tres perfiles")]
3  public List<ClasePerfiles> Perfiles { get; set; } = new List<ClasePerfiles>();
```

FILEEXTENSIONS

Podemos limitar las extensiones de los ficheros que queremos recibir en variables del tipo IFormFile, que reciben ficheros.

Por ejemplo, si queremos que la foto la recibamos con las extensiones jpg, png o pdf.

```
1    [FileExtensions(Extensions = "jpg,png,pdf", ErrorMessage = "La foto debe tener el formato
     .jpg, .png o .pdf.")]
2    public IFormFile? Foto { get; set; }
```

VALIDACIÓNCREDITCARD, EMAILADDRESS Y PHONENUMBER

Validar tarjetas de crédito, correos electrónico y números de teléfono.

```
1    [EmailAddress]
2    public string EmailPersonal { get; set; } = "";
```

CREAR TU PROPIO ATRIBUTO

Explicar el del correo de la UA, con el código o con la ClaseSeguridad

Aunque con la validación con expresiones regulares se nos abren muchas posibilidades, sí que es posible que nos interese hacer validaciones más complejas o poder reutilizarlas de forma muy cómoda entre proyectos.

El proceso es sencillo:

- Creamos una clase que herede de ValidationAttribute
- Definimos la función IsValid con la lógica que queremos validar (recibe en value el valor que queremos validar)
- Si necesitamos pasar parámetros, tenemos que crear un constructor que lo reciba y lo almacene en una variable local.

VALIDACIÓN CORREO UA

En este caso solo validamos el valor, y lo comparamos con una expresión regular. En caso de que no se corresponda, damos un mensaje de error personalizado.

```
1    public class CorreoUAAttribute : ValidationAttribute
2    {
3        protected override ValidationResult IsValid(object value, ValidationContext
     validationContext)
```

```
4       {
5           if (value is string correo)
6           {
7               if (!string.IsNullOrEmpty(correo)) {
8                   string pattern = @"^[\w\.-]+@([\w-]+\.)*ua\.es$";
9                   bool isMatch = Regex.IsMatch(correo, pattern);
10
11                  if (isMatch) {
12                      return ValidationResult.Success;
13                  }
14              }
15          }
16
17          return new ValidationResult($"El campo {validationContext.DisplayName} no es un
    correo de la UA.");
18      }
19  }
```

VALIDACIÓN CON PARÁMETROS

En este caso pasamos un parámetro, numeroMaxAlumnos, que usaremos posteriormente en la validación.

```
1   public class NumeroMaxAlumnosAtributo : ValidationAttribute
2   {
3       public NumeroMaxAlumnosAtributo(int numeroMaxAlumnos)
4       : base("{0} tiene demasiados alumnos")
5       {
6           _numeroMaxAlumnos = numeroMaxAlumnos;
7       }
8       protected override ValidationResult IsValid(
9       object value, ValidationContext validationContext)
10
11      {
12          if (value != null)
13          {
14              int numeroMaxAlumnos;
15              if (Int32.TryParse(value.ToString(), out numeroMaxAlumnos))
16              {
17                  if (numeroMaxAlumnos > _numeroMaxAlumnos)
18                  {
19                      var errorMessage = FormatErrorMessage(validationContext.DisplayName);
20                      return new ValidationResult(errorMessage);
21                  }
22              }
23          }
24          return ValidationResult.Success;
25      }
26      private readonly int _numeroMaxAlumnos;
27  }
```

PERSONALIZAR LOS ERRORES

Por defecto, .NET Core hace las validaciones antes de llegar a nuestro controlador y muestra el error.

Nosotros necesitamos personalizarlo para que el cliente Vue pueda recibirlo, interpretarlo y adaptarlo a mensajes visibles.

Para ello debemos desbloquear esa validación que hace .NET Core, en program.cs

```
builder.Services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressModelStateInvalidFilter = true;
});
```

Esto nos permite gestionar los errores desde los controladores

Por defecto con

```
if (!ModelState.IsValid)
{
  return BadRequest(ModelState);
}
```

Si se produce un error, obtenemos la información con este formato

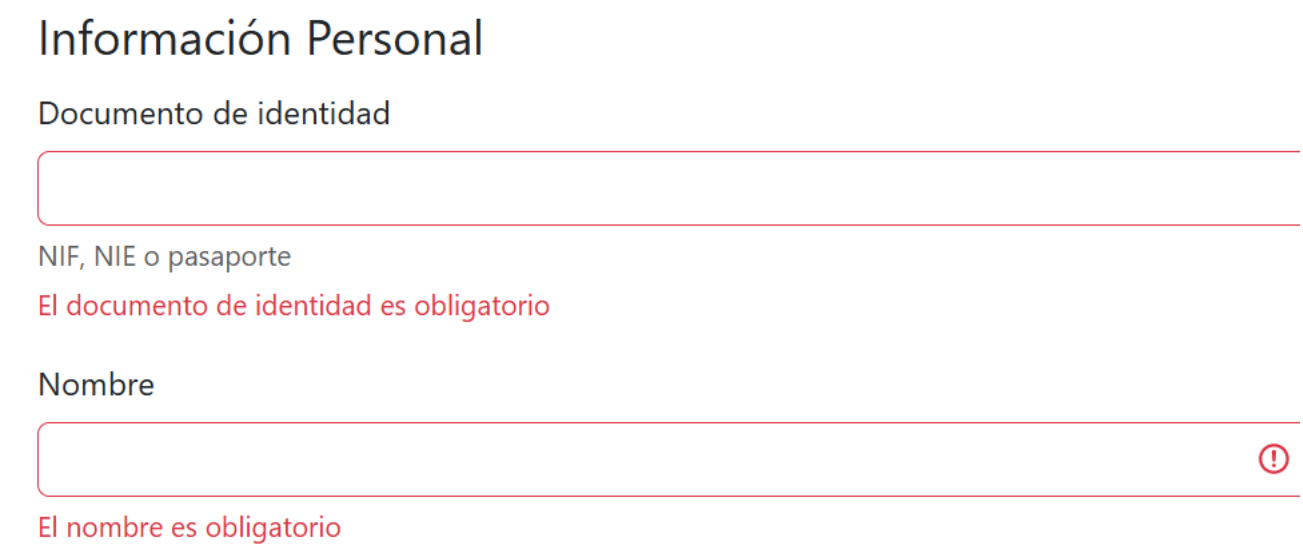
```
{
  "emailUA": [
    "El campo EmailUA no es un correo de la UA."
  ]
}
```

En el documento APIs en Core - Crear APIs.pdf se explica como se personalizan los mensajes de error para que desde Vue los podamos integrar y visualizar.

INTEGRACIÓN CON EL CLIENTE

En el documento Aplicaciones Vue 3 y TypeScript - Uso de APIs.pdf se explica como gestionar los errores recibidos para mostrarlos:

En los campos implicados



O como notificaciones Toast