



**Escuela Politécnica Superior de
Alicante**

Diseño de aplicaciones web

**Taller de ASP.net
CORE II**

Contenidos

- Acceso a Bds / Ingeniería inversa
 - Creación de un BDContext
- Listados con Scaffolding
 - Detalle con Scaffolding
 - Patrones de ordenación, filtrado
- Operaciones del CRUD: añadir y eliminar datos.
- Ejercicios
- Referencias

La base de datos

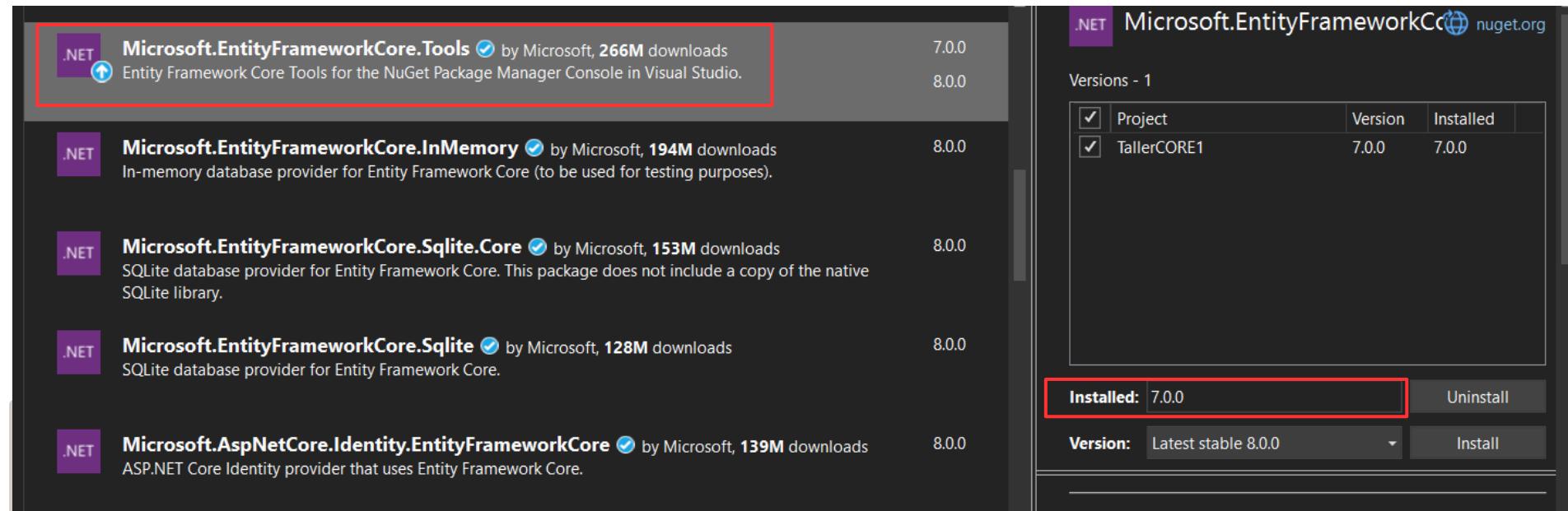
- Para acceder a una base de datos, usaremos modelos creados a partir del ORM Entity Framework (EF).
- Existen dos formas de vincular una BD a nuestro proyecto:
 - Crear el esquema mediante código y subirlo al SGBD con migraciones (<https://docs.microsoft.com/es-es/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli>)
 - Ingeniería inversa: apto para cuando el esquema y sus tablas YA existen en el SGBD y lo tenemos que vincular/importar a nuestro proyecto
 - En este caso se crean con EF los modelos necesarios vinculados a las tablas/objetos de la BD.

La base de datos

- En nuestro caso vamos a usar Ingeniería Inversa y para ello:
 - Crearemos una BD en SQL-Server Express con dos tablas:
 - Productos (id PK, nombre, descripcion, precio, categoria FK)
 - Categorías (id PK, nombre)
 - Crearemos un login para acceder al SQL-Server
 - Añadiremos unos datos de ejemplo (varias categorías, varios productos, al menos dos de la misma categoría)
- Podeis obtener un script de creación de esta BD en el moodle.

Ingeniería Inversa

- Vamos a partir desde el proyecto de la sesión anterior.
- Necesitamos instalar el paquete de NuGet 'Microsoft.EntityFrameworkCore.Tools', la versión 7.0 es compatible con nuestro proyecto (.net 6.0):
 - Herramientas > Administrador de paquetes NuGet > Administrar paquetes NuGet para la solución



DbContext y modelos del EF

- También tenemos que instalar el paquete de NuGet 'Microsoft.EntityFrameworkCore.SqlServer', en la misma versión que el paquete 'Tools'.

The screenshot shows the NuGet Package Manager interface. On the left, a list of packages is displayed:

- Microsoft.EntityFrameworkCore.SqlServer** by Microsoft, 409M downloads. This package is highlighted with a red box.
- Microsoft.EntityFrameworkCore.Design** by Microsoft, 393M downloads. Shared design-time components for Entity Framework Core tools.
- Microsoft.EntityFrameworkCore.Tools** by Microsoft, 266M downloads. Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.
- Microsoft.EntityFrameworkCore.InMemory** by Microsoft, 194M downloads. In-memory database provider for Entity Framework Core (to be used for testing purposes).
- Microsoft.EntityFrameworkCore.Sqlite.Core** by Microsoft, 153M downloads. SQLite database provider for Entity Framework Core. This package does not include a copy of the native SQLite library.

On the right, the details for the selected package (**Microsoft.EntityFrameworkCore.SqlServer**) are shown:

Versions - 1

Project	Version	Installed
TallerCORE1	7.0.0	7.0.0

Installed: 7.0.0 **Uninstall**

Version: Latest stable 8.0.0 **Install**

DbContext y modelos del EF

- Ahora disponemos de las herramientas para crear Bds (con migraciones o ingeniería inversa) desde nuestro proyecto (desde la consola 'Packet manager console').
- Comandos disponibles:
 - Add-Migration
 - Bundle-Migration
 - Drop-Database
 - Get-DbContext
 - Get-Migration
 - Optimize-DbContext
 - Remove-Migration
 - **Scaffold-DbContext**
 - Script-DbContext
 - Script-Migration
 - Update-Database

Scaffold-DbContext

- Usaremos la herramienta 'Scaffold-DbContext' para crear los modelos basados en las tablas de nuestra BD:

```
Scaffold-DbContext "Server=SERVIDOR;
```

```
Database=NUESTRABD; Integrated Security=False;
```

```
Persist Security Info=False; User ID=USUARIO;
```

```
Password=#####; Encrypt=False"
```

```
Microsoft.EntityFrameworkCore.SqlServer
```

```
-Tables Categorias, Productos
```

```
-OutputDir Models
```

```
-Context NOMBREDECLASE
```

- Servidor:

Máster SQL-Server: (local)

Web SQL-Server Express: (local)\sqlexpress

Scaffold-DbContext

- Tras ejecutar el comando 'Scaffold-DbContext' obtendremos varios modelos nuevos:
 - Una clase DBContext con la configuración de conexión y otras inicializaciones
 - Una clase con un modelo basado en cada una de las tablas que hayamos indicado en el parámetro 'Tables': Producto y Categoría
- Además, en el fichero 'Program.cs', tenemos que registrar el nuevo DBContext creado:

```
// Add services to the container.  
builder.Services.AddRazorPages();  
  
builder.Services.AddDbContext<TallerCORE1.Models.TallerDAWDbContext>();  
  
var app = builder.Build();
```

Modelos generados

- Vemos que en ambas clases, al estar relacionadas, se han creado atributos para 'navegar' por la relación en ambos sentidos:

```
public partial class Producto
{
    public decimal Id { get; set; }

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

    public string? Descripcion { get; set; }

    public decimal Precio { get; set; }

    public decimal Categoria { get; set; }

    public virtual Categoria? CategoriaNavigation { get; set; } = null!;
}

public partial class Categoria
{
    public decimal Id { get; set; }

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

    public virtual ICollection<Producto> Productos { get; } = new List<Prod
```

En la clase 'Producto' tenemos un objeto 'Categoria' llamado 'CategoriaNavigation'

Para evitar problemas cuando no se instancie (por ejemplo a la hora de añadir una nueva tupla) lo haremos nutable (Categoria?)

En la clase 'Categoria' tenemos una colección de objetos 'Producto' llamada 'Productos'.

DataAnnotations

- Vamos a usar los atributos del componente 'Data Annotations' que nos permite añadir información sobre:
 - Aspecto
 - Comportamiento
 - semántica
 - y validación de cada campo del modelo.
- Esto nos permitirá completar y afinar más a la hora de generar las pantallas con scaffolding
 - y también a la hora de usar los helpers de visualización y de formularios.

DataAnnotations

- Atributos utilizados:
 - Display: permite indicar textos para miembros de los modelos.
 - Description: descripción del campo
 - Name: nombre del campo en la interfaz
 - Shortname: nombre para columnas en listados
 - Prompt: aviso o info al usuario (placeholder?)
 - Required: para campos obligatorios
 - Errormessage: indica el mensaje de error
 - Datatype: Asocia un tipo de dato al campo
 - Range: indica los límites del rango de valores de un campo numérico (mínimo, máximo, mensajes de error, etc.)

DataAnnotations

- Vamos a añadir a los campos del modelo 'Producto' las siguientes anotaciones o atributos:

```
[Display(Description = "Código del producto",
    Name = "Código: ",
    Prompt = "Código del producto",
    ShortName = "Cód.")]
[Required(ErrorMessage = "Este campo es obligatorio")]
0 referencias
public decimal Id { get; set; }

[Display(Description = "Nombre del producto",
    Name = "Nombre: ",
    Prompt = "Nombre del producto",
    ShortName = "Nombre")]
[Required(ErrorMessage = "Este campo es obligatorio")]
0 referencias
public string Nombre { get; set; }

[Display(Description = "Descripción del producto",
    Name = "Descripción: ",
    Prompt = "Descripción del producto",
    ShortName = "Descr.")]
[Required(ErrorMessage = "Este campo es obligatorio")]
0 referencias
public string Descripcion { get; set; }
```

```
[Display(Description = "Categoría o familia",
    Name = "Categoría: ",
    Prompt = "Categoría o familia",
    ShortName = "Cat.")]
[Required(ErrorMessage = "Este campo es obligatorio")]
0 referencias
public decimal Categoría { get; set; }

[Display(Description = "Categoría o familia",
    Name = "Categoría: ",
    Prompt = "Categoría o familia",
    ShortName = "Cat.")]
0 referencias
public virtual Categoría CategoríaNavigation { get; set; }

[Display(Description = "Precio del producto",
    Name = "Precio: ",
    Prompt = "Precio del producto",
    ShortName = "PVP")]
[Required(ErrorMessage = "El campo '{0}' es obligatorio")]
[DataType(DataType.Currency)]
[Column(TypeName = "decimal(18, 2)")]
[Range(0, Double.MaxValue,
    ErrorMessage = "El valor del campo '{0}' debe ser mayor que {1}")]
0 referencias
public decimal Precio { get; set; }
```

ModelMetadataType

- Puede darse el caso que necesitemos generar de nuevo los modelos importándolos de la BD (porque ésta haya sido modificada o para añadir tablas nuevas).
- El proceso de ingeniería inversa nos sobrescribirá el código de los modelos con la nueva importación y perderemos los atributos de anotaciones de datos.
- Para evitar esto, podemos vincular a cada modelo una clase 'annotations' que defina los mismos atributos que el modelo en el que se basa y que contenga los atributos de anotaciones.
- Despues tendremos que definir esta clase como tipo metadato del modelo: 'ModelMetadataType', y referenciarlo al modelo original.

Y podrémos quitar las anotaciones del modelo original.

ModelMetadataType

- Por tanto, crearemos la clase 'Producto_annotations', que contendrá los mismos atributos que el modelo Producto y sus anotaciones de campos. Y la vinculamos a su clase modelo de esta forma:

```
namespace TallerCORE.Models
{
    [ModelMetadataType(typeof(Producto_annotations))]
    11 referencias
    public partial class Producto
    {
    }

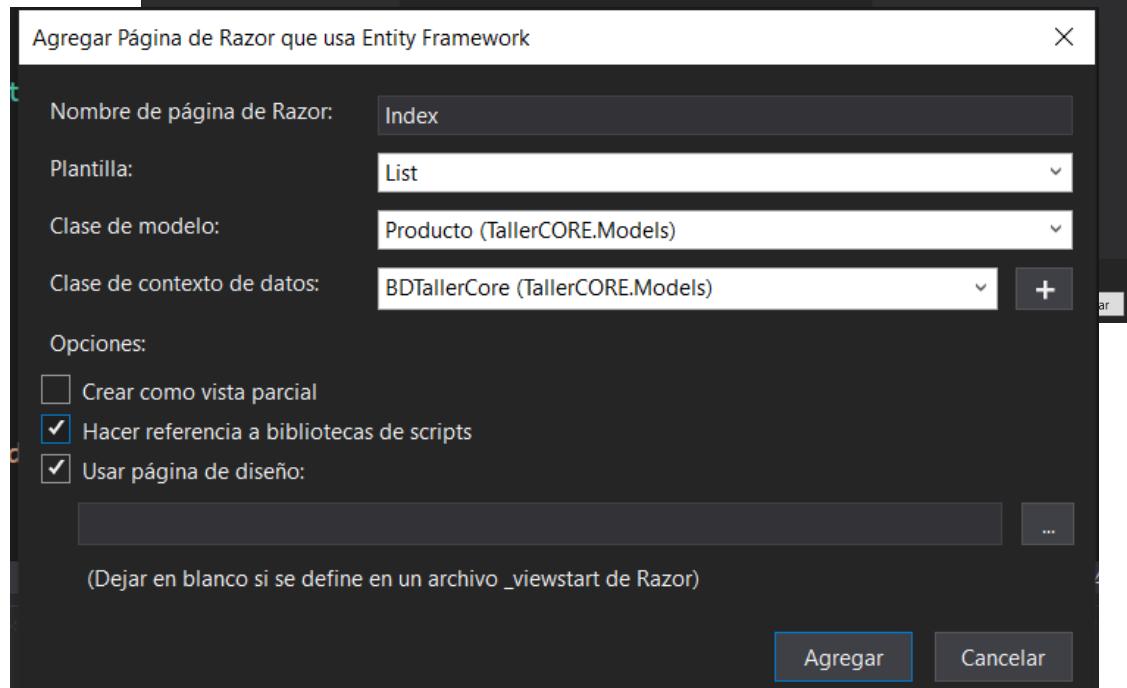
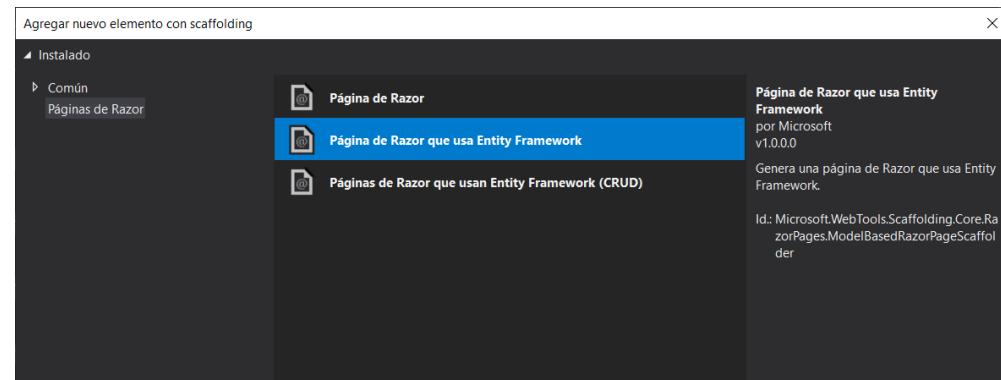
    2 referencias
    public class Producto_annotations
    {
        [Display(Description = "Código del producto",
            Name = "Código: ",
            Prompt = "Código del producto",
            ShortName = "Cód.")]
        [Required(ErrorMessage = "Este campo es obligatorio")]
        0 referencias
        public decimal Id { get; set; }
    }
}
```

Listado/detalle con Scaffolding

- Una vez preparados los modelos, vamos a diseñar nuestro catálogo.
- Primero añadiremos una nueva carpeta 'Catalogo' dentro de 'Pages'.
- Después, crearemos dos páginas nuevas con scaffolding:
 - Página 'Catalogo', que consistirá en un listado de productos (nombre, precio, categoría) al que le añadiremos posibilidad de ordenar por nombre, categoría y precio y filtrar por categoría.
 - Página 'Detalle', donde podremos acceder a una ficha de un producto con toda su información.
- Para poder acceder, añadiremos en el menú del sitio una opción nueva que apunte a la página del listado de productos.

Listado/detalle con Scaffolding

- Creamos la nueva página, dentro de la carpeta 'Catalogo':
 - la llamaremos 'Index' (nombre por defecto).
 - Será una página 'razor page' que usa Entity Framework.
 - Usaremos la plantilla de listado 'List' .
 - Basada en el modelo 'Producto'.
 - Y en el contexto 'DBTallerCore'.



Listado/detalle con Scaffolding

- El código de la clase PageModel es:

```
namespace TallerCORE1.Pages.Catalogo
{
    public class IndexModel : PageModel
    {
        private readonly TallerCORE1.Models.TallerDAWDBContext _context;

        public IndexModel(TallerCORE1.Models.TallerDAWDBContext context)
        {
            _context = context;
        }

        public IList<Producto> Producto { get; set; } = default!;

        public async Task OnGetAsync()
        {
            if (_context.Productos != null)
            {
                Producto = await _context.Productos
                    .Include(p => p.CategoriaNavigation).ToListAsync();
            }
        }
    }
}
```

Podemos observar:

- Se declara una variable de tipo DBContext
- El constructor recibe como argumento el contexto de la BD del proyecto.
- Se define una lista de objetos 'Producto'
- La cual en el método 'OnGetAsync' se puebla con datos obtenidos de la BD.

- En el caso de la vista 'index.cshtml', se trata de una tabla cuyo contenido se pinta recorriendo con un foreach cada dato recibido.

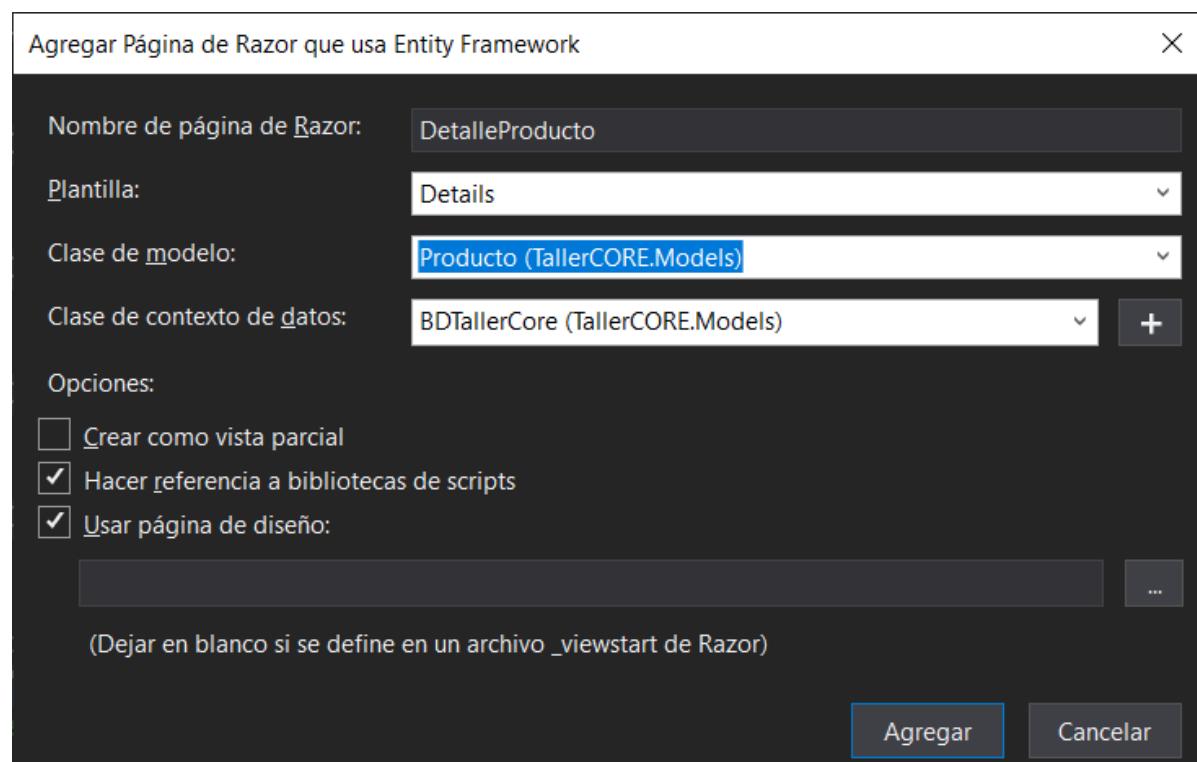
Listado/detalle con Scaffolding

- Vamos a adaptar un poco el código generado para acercarlo al propósito de nuestro sitio web. Para ello:
 - Cambiaremos los títulos por uno más adecuado.
 - Quitamos el link de crear nuevo.
 - Añadiremos estilos a la tabla: striped, bordered, hover, etc.
 - Ocultamos la columna de 'Descripción', la dejaremos para el detalle del producto.
 - Ajustaremos los precios a la derecha y centraremos los nombres de las categorías.
 - Ocultaremos los enlaces de 'Edit' y 'Delete'. Y el enlace de Detalle lo modificaremos:

```
<a asp-page="./DetalleProducto" asp-route-id="@item.Id"  
class="btn btn-info">Detalles</a>
```

Listado/detalle con Scaffolding

- Ahora, vamos a añadir la página de detalle del producto:
 - Nombre: 'DetalleProducto'
 - Tipo: 'razor page que usa Entity Framework'.
 - Plantilla 'Details' .
 - Basada en el modelo 'Producto'.
 - Y en el contexto 'DBTallerCore'.



Listado/detalle con Scaffolding

- Echamos un vistazo al código:

```
public async Task<IActionResult> OnGetAsync(decimal? id)
{
    if (id == null || _context.Productos == null)
    {
        return NotFound();
    }

    var producto = await _context.Productos.FirstOrDefaultAsync(m => m.Id == id);
    if (producto == null)
    {
        return NotFound();
    }
    else
    {
        Producto = producto;
    }
    return Page();
}
```

- La función 'OnGetAsync' recibe un 'id' como argumento.
- Obtiene el dato a partir de 'Linq'.
- Si no existe el argumento 'id' o no encuentra nada en la BD, envia una respuesta http 404 (Resource not found).

Listado/detalle con Scaffolding

- Vamos ahora a adaptar la página 'DetalleProducto' a nuestro proyecto. Cambios de la vista:
 - Cambiamos los títulos del HTML y del H1 (en éste pondremos el nombre del producto)
 - Quitamos el subtítulo, el H4.
 - Comentamos el enlace al 'Edit'
 - Modificamos el enlace de volver al listado con el texto correspondiente y el enlace correcto y estilos de botón.
 - Añadiremos un 'if' de forma que si el modelo viene vacío (=null) mostramos un mensaje de error adecuado y en otro caso mostramos el H1 y la información del detalle:

```
if (Model.Producto != null)
{
    <h1>@Html.DisplayFor(model => model.Producto.Nombre)</h1>

    <div>
        ...
    </div>
} else
{
    <h1>Atención!</h1>
    <div class="alert alert-danger">No se han encontrado datos</div>
}
```

Listado/detalle con Scaffolding

- En el 'code behind' de la página de detalle, tendremos que modificar el método 'OnGetAsync' para que en lugar de devolver una respuesta HTTP 404, devuelva un código 200 (OK) pero con el dato del producto a null.

```
public async Task<IActionResult> OnGetAsync(decimal? id)
{
    if (id == null)
    {
        //return NotFound();
        Producto = null;
    }

    Producto = await _context.Productos
        .Include(p => p.CategoríaNavigation).FirstOrDefaultAsync(m => m.Id == id);

    /*if (Producto == null)
    {
        //return NotFound();
    }*/
    return Page();
}
```

Ordenación del listado de productos

- Vamos a ordenar la lista por tres campos: nombre, precio y categoría.
- Primero modificaremos la vista creando enlaces en la cabecera de las columnas, para enviar los criterios de ordenación:

```
<table class="table table-bordered table-striped table-hover">
    <thead>
        <tr>
            <th>
                <a asp-page=".~/Index" asp-route-o="N" asp-route-d="@Model.dir">
                    @Html.DisplayNameFor(model => model.Producto[0].Nombre)
                </a>
            </th>
            <!--<th>
                @Html.DisplayNameFor(model => model.Producto[0].Descripcion)
            </th>-->
            <th>
                <a asp-page=".~/Index" asp-route-o="P" asp-route-d="@Model.dir">
                    @Html.DisplayNameFor(model => model.Producto[0].Precio)
                </a>
            </th>
            <th>
                <a asp-page=".~/Index" asp-route-o="C" asp-route-d="@Model.dir">
                    @Html.DisplayNameFor(model => model.Producto[0].CategoriaNavigation)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
```

Ordenación del listado de productos

- En el back-end, debemos declarar dos campos: 'orden' y 'dir' (dirección de la ordenación), además los inicializaremos en el constructor:

```
public IndexModel(TallerCORE.Models.BDTallerCore context)
{
    _context = context;
    orden = "N";
    dir = "ASC";
}

6 referencias
public IList<Producto> Producto { get;set; }

1 referencia
public string orden { get; set; }

5 referencias
public string dir { get; set; }
```

Ordenación del listado de productos

- Y la función 'OnGetAsync', quedará así:

```
public async Task OnGetAsync(string o="N", string d="ASC")
{
    IQueryable<Producto> list = from p in _context.Productos select p;

    Expression<Func<Producto, object>> expresion=(x=>x.Nombre);
    switch (o)
    {
        case "N":
            expresion = (x => x.Nombre);
            break;
        case "P":
            expresion = (x => x.Precio);
            break;
        case "C":
            expresion = (x => x.CategoríaNavigation.Nombre);
            break;
    }

    if (d == "ASC")
    {
        list = list.OrderBy(expresion);
    }
    else
    {
        list = list.OrderByDescending(expresion);
    }
    dir = (d == "ASC" ? "DESC" : "ASC");

    Producto = await list.AsNoTracking().Include(p => p.CategoríaNavigation).ToListAsync();
}
```

Observaciones:

- Añadimos dos parámetros nuevos al método.
- Declaramos una expresión (reflexion expression) para decidir el campo con el que ordenar
- Según la dirección usaremos un método de ordenación diferente y alternamos su valor.

Filtrar el listado por categoría

- Para añadir un filtro por categoría a nuestro catálogo, debemos modificar el ViewModel de la siguiente forma:
 - Añadiremos un nuevo campo 'cat' y lo inicializaremos
 - Añadiremos un 'SelectListItem' para obtener las categorías de la BD y poblar el desplegable del filtro.

```
public IndexModel(TallerCORE.Models.BDTallerCore context)
{
    _context = context;
    orden = "N";
    dir = "ASC";
    cat = "";
}

7 referencias
public IList<Producto> Producto { get; set; }

4 referencias
public IList<SelectListItem> Categorias { get; set; }

2 referencias
public string orden { get; set; }

6 referencias
public string dir { get; set; }

6 referencias
public string cat { get; set; }
```

Filtrar el listado por categoría

- La función 'OnGetAsync' tendrá un nuevo argumento 'c' y realizará el filtrado de la siguiente forma:

```
public async Task OnGetAsync(string o="N", string d="ASC", string c="")  
{  
    /*Producto = await _context.Productos  
     .Include(p => p.CategoríaNavigation).ToListAsync();*/  
    IQueryable<Producto> list = null;  
    if (String.IsNullOrEmpty(c))  
    {  
        list = from p in _context.Productos select p;  
    } else  
    {  
        try  
        {  
            decimal idCat = Convert.ToDecimal(c);  
            list = from p in _context.Productos where p.Categoría == idCat select p;  
        } catch  
        {  
            list = from p in _context.Productos select p;  
        }  
        cat = c;  
    }  
}
```

Filtrar el listado por categoría

- La función 'OnGetAsync' ademas, al final, creara los items de la lista de items para el desplegable, así:

```
Producto = await list.AsNoTracking().Include(p => p.CategoríaNavigation).ToListAsync();

var cats = await _context.Categorías.OrderBy(c=>c.Nombre).ToListAsync();

Categorías = new List<SelectListItem>();

Categorías.Add(new SelectListItem { Text = "Todas", Value = "" });
foreach (Categoría aux in cats)
{
    Categorías.Add(new SelectListItem { Text = aux.Nombre, Value = aux.Id.ToString() });
}

}
```

Filtrar el listado por categoría

- En la vista realizaremos tres cambios.
- Primero, incluiremos un formulario con el desplegable para seleccionar la categoría por la que filtrar:

```
<form asp-page=".Index" asp-route-o="@Model.orden" asp-route-d="@Model.dir" method="get" class="form">
    <div class="form-group row">
        <label for="cat" class="form-label col-md-1 offset-8">Categoría:</label>
        <div class="col-md-3">
            <select name="c" id="c" onchange="this.form.submit()" asp-for="cat"
                    asp-items="@Model.Categorias" class="form-select"></select>
        </div>
    </div>
</form>
```

Filtrar el listado por categoría

- Segundo, cambiaremos los enlaces de las cabeceras de las columnas de la tabla, para incluir el dato del filtro:

```
<table class="table table-bordered table-striped table-hover">
    <thead>
        <tr>
            <th>
                <a asp-page=".Index" asp-route-o="N" asp-route-d="@Model.dir" asp-route-c="@Model.cat">
                    @Html.DisplayNameFor(model => model.Producto[0].Nombre)
                </a>
            </th>
            <!--<th>
                @Html.DisplayNameFor(model => model.Producto[0].Descripcion)
            </th>-->
            <th>
                <a asp-page=".Index" asp-route-o="P" asp-route-d="@Model.dir" asp-route-c="@Model.cat">
                    @Html.DisplayNameFor(model => model.Producto[0].Precio)
                </a>
            </th>
            <th>
                <a asp-page=".Index" asp-route-o="C" asp-route-d="@Model.dir" asp-route-c="@Model.cat">
                    @Html.DisplayNameFor(model => model.Producto[0].CategoriaNavigation)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
```

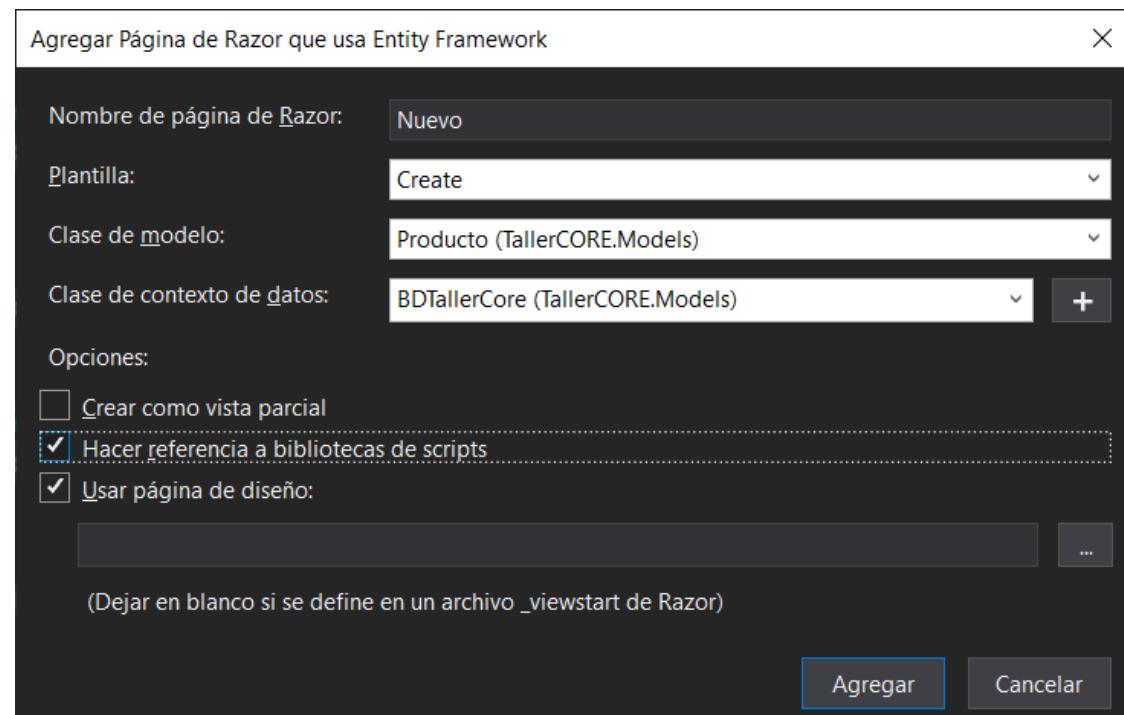
Filtrar el listado por categoría

- Tercero, incluiremos un 'if' para mostrar un mensaje de error en caso de que no encontremos productos de la categoría seleccionada:

```
@if (Model.Producto.Count > 0)
{
    <table class="table table-bordered table-striped table-hover">
        ...
    </table>
} else
{
    <div class="alert alert-danger">No se han encontrado productos</div>
}
```

Añadir un nuevo producto

- Añadimos una nueva vista/página para la acción de añadir un nuevo producto:
 - Nombre: 'Nuevo'
 - Tipo: 'razor page que usa Entity Framework'.
 - Plantilla de 'Create' .
 - Basada en el modelo 'Producto'.
 - Y en el contexto 'DBTallerCore'.



Añadir un nuevo producto

- Este es el código del back-end:

```
public IActionResult OnGet()
{
    ViewData["Categoria"] = new SelectList(_context.Categorias, "Id", "Id");
    return Page();
}

[BindProperty]
public Producto Producto { get; set; } = default!;

// To protect from overposting attacks, see https://aka.ms/RazorPagesCRUD
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid || _context.Productos == null || Producto == null)
    {
        return Page();
    }

    _context.Productos.Add(Producto);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Tenemos:

- Enviamos una SelectList al formulario en el método 'OnGet'.
- Tenemos un miembro 'Producto' que esta enlazado a la vista con 'BindProperty'
- El método 'OnPostAsync' valida el modelo con las annotations y si es correcto, añade el dato a la BD y redirige al listado.

Añadir un nuevo producto

- Reactivamos (o añadimos) el botón 'nuevo' en el listado

```
<h1>Catálogo de productos</h1>

<p>
    <a asp-page="Nuevo" class="btn btn-primary">Nuevo</a>
</p>
```

- Adaptamos la vista cambiando los títulos y subtítulos y textos de botones. Convertir el campo 'Descripción' en un 'textarea'.

```
@page
@model TallerCORE.NuevoModel

@{
    ViewData["Title"] = "Añadir un nuevo producto";
}
```

```
<h1>Añadir un nuevo producto</h1>
<h4>Rellena el siguiente formulario</h4>
<hr />
```

```
<div class="form-group">
    <div class="btn-group">
        <input type="submit" value="Añadir" class="btn btn-primary" />
        <a class="btn btn-warning" asp-page="Index">Volver al listado</a>
    </div>
</div>
```

Añadir un nuevo producto

- Podemos añadir una opción 'fake' al desplegable de categorías con un mensaje al usuario, para ello, tenemos que modificar el método 'OnGet':

```
public IActionResult OnGet()
{
    var listCats = (from c in _context.Categorias
                    select new SelectListItem { Text = c.Nombre, Value = c.Id.ToString() })
                    .ToList<SelectListItem>();
    listCats.Insert(0, new SelectListItem { Text = "seleccione categoría", Value = "" });
    ViewData["Categoria"] = listCats;
    //ViewData["Categoria"] = new SelectList(_context.Categorias, "Id", "Nombre");
    return Page();
}
```

- Y, en la vista, añadimos el campo con el mensaje de validación/error:

```
<div class="form-group">
    <label asp-for="Producto.Categoría" class="control-label"></label>
    <select asp-for="Producto.Categoría" class ="form-select" asp-items="ViewBag.Categoría"></select>
    <span asp-validation-for="Producto.Categoría" class="text-danger"></span>
</div>
```

Añadir un nuevo producto

- Si el separador de decimales nos da problemas, tenemos que comprobar que la web y la BD 'hablen' el mismo idioma.
- Podemos añadir en el fichero 'Program.cs' la inicialización de la configuración regional en el formato 'es-ES':

```
var app = builder.Build();

var cultureInfo = new CultureInfo("es-ES");
cultureInfo.NumberFormat.NumberDecimalSeparator = ".";

CultureInfo.DefaultThreadCurrentCulture = cultureInfo;
CultureInfo.DefaultThreadCurrentUICulture = cultureInfo;

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
```

Añadir un nuevo producto

- Si aún así el validador de cliente (JqueryValidator) no nos lo permite, podemos 'puentearlo' con javascript:

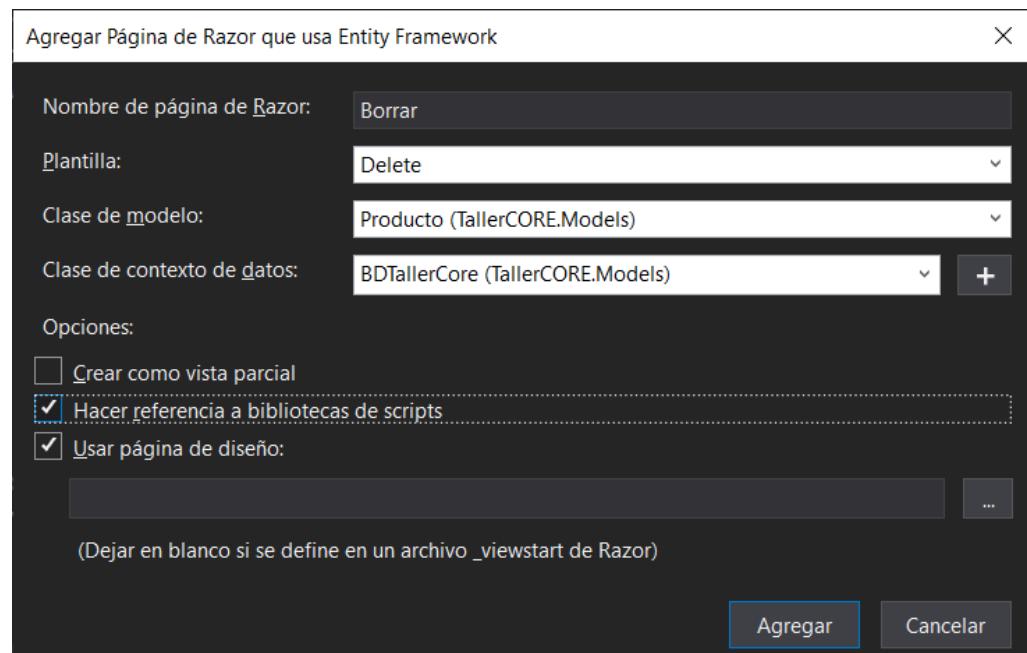
```
<div class="form-group">
    <label asp-for="Producto.Precio" class="control-label"></label>
    <input asp-for="Producto.Precio" class="form-control" onblur="cambioComa()" />
    <span id="error_precio" asp-validation-for="Producto.Precio" class="text-danger"></span>
</div>
```

- Y añadimos un scriptlet para cambiar el separador de decimales y refrescar la validez del campo:

```
@section Scripts {
    <script type="text/javascript">
        function cambioComa() {
            var valor = $("#Producto_Precio").val();
            $("#Producto_Precio").val(valor.replace(',', '.'));
            $('#Producto_Precio').valid();
        }
    </script>
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
```

Eliminar un producto existente

- Añadimos una nueva vista o página para la acción de eliminar un producto:
 - Nombre: 'Borrar'
 - Tipo: 'razor page que usa Entity Framework'.
 - Plantilla de 'Delete' .
 - Basada en el modelo 'Producto'.
 - Y en el contexto 'DBTallerCore'.



Eliminar un producto existente

- En el listado, en la última columna, añadiremos el botón que permita acceder a eliminar el producto:

```
<td>
    <div class="btn-group">
        <!--<a asp-page=".Edit" asp-route-id="@item.Id">Edit</a>-->
        <a asp-page=".DetalleProducto" asp-route-id="@item.Id" class="btn btn-info">Detalles</a>
        <a asp-page=".Borrar" asp-route-id="@item.Id" class="btn btn-danger">Eliminar</a>
    </div>
</td>
```

Eliminar un producto existente

- En el 'code behind' tenemos que realizar unas adaptaciones para evitar respuestas 'NotFound' y capturar los posibles errores para avisar al usuario.

```
[BindProperty]
public Producto Producto { get; set; } = default;

public async Task<IActionResult> OnGetAsync(decimal? id)
{
    if (id == null || _context.Productos == null)
    {
        //return NotFound();
        TempData["error"] = "Error al eliminar un producto: No hay dato";
        return RedirectToPage("./Index");
    }

    var producto = await _context.Productos.FirstOrDefaultAsync(m => m.Id == id);

    if (producto == null)
    {
        //return NotFound();
        TempData["error"] = "Error al eliminar un producto: Producto no encontrado";
        return RedirectToPage("./Index");
    }
    else
    {
        Producto = producto;
    }
    return Page();
}
```

Observaciones:

- Tenemos un miembro 'Producto' que esta enlazado a la vista con 'BindProperty'
- El método 'OnGetAsync' si encuentra algún error, almacena en el 'TempData' un mensaje e invoca la página del listado.
- En caso de ir todo bien, mostrará la vista con el producto que vamos a borrar

Eliminar un producto existente

- En el 'code behind', la función 'OnPostAsync' si encuentra algún error tambien se redirigirá al listado con un aviso del fallo y si el borrado funciona correctamente, enviará un texto informativo.

```
public async Task<IActionResult> OnPostAsync(decimal? id)
{
    if (id == null)
    {
        //return NotFound();
        TempData["error"] = "Error al eliminar un producto: No hay dato";
        return RedirectToPage("./Index");
    }

    Producto = await _context.Productos.FindAsync(id);

    if (Producto != null)
    {
        _context.Productos.Remove(Producto);
        await _context.SaveChangesAsync();
        TempData["ok"] = "Producto eliminado correctamente";
    } else
    {
        TempData["error"] = "Error al eliminar un producto: Producto no encontrado";
    }

    return RedirectToPage("./Index");
}
```

Eliminar un producto existente

- Ahora debemos añadir en el listado (página 'Index') el código necesario para gestionar los mensajes que pueden llegarnos desde el TempData:

```
<h1>Catálogo de productos</h1>

@if (TempData["error"] != null && TempData["error"].ToString() != "")
{
    <div id="msgError" class="alert alert-danger">@TempData["error"].ToString()</div>
}

@if (TempData["ok"] != null && TempData["ok"].ToString() != "")
{
    <div id="msgOk" class="alert alert-info">@TempData["ok"].ToString()</div>
}

<p>
    <a asp-page="Nuevo" class="btn btn-primary">Nuevo</a>
</p>
```

Eliminar un producto existente

- Y, para que se oculten automáticamente estos mensajes, añadiremos un timeout en la sección de scripts de la página del listado:

```
@section Scripts {
    <script type="text/javascript">
        $(document).ready(function () {
            setTimeout(function () {
                $('#msgError').hide();
                $('#msgOk').hide();
            }, 5000);
        });
    </script>
}
```

Eliminar un producto existente

- Finalmente, adaptaremos la página previa al borrado al formato de nuestro proyecto, cambiando títulos, textos y botones:

```
@model TallerCORE.BorrarModel

@{
    ViewData["Title"] = "Eliminar producto del catálogo";
}

<h1>Eliminar producto del catálogo</h1>

<h3>¿Está seguro que desea eliminar el producto '@Model.Producto.Nombre'?</h3>
<div>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Producto.Nombre)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Producto.Nombre)
        </dd>
    </dl>
    <form method="post">
        <input type="hidden" asp-for="Producto.Id" />
        <div class="btn-group">
            <input type="submit" value="Eliminar" class="btn btn-danger" />
            <a class="btn btn-primary" asp-page="./Index">Volver al listado</a>
        </div>
    </form>
</div>
```

Ejercicio

- Modifica el listado para que se pueda ver la descripción del producto debajo del nombre hasta un máximo de 100 caracteres (y si se pasa el ratón por encima, se pueda ver la descripción completa, con un tooltip)
- Añade un nuevo campo a la tabla productos: **especificaciones**, de tipo texto (varchar de 2000).
 - Inserta datos de prueba, a ser posible textos largos (usa el generador de 'loren ipsum': <https://www.lipsum.com/>)
 - Refactoriza el proyecto para que se pueda visualizar en el detalle de producto.
 - Convierte la celda de las especificaciones en 'colapsable' de forma que funcione con el efecto 'acordeon' y su contenido pueda ocultarse o verse con un botón (lupa, ojo, signo '+').

Ejercicio

- Añade una nueva tabla: Marcas
 - Estructura:
 - id (numérico, PK), nombre (varchar(100), not null). url(varchar(100), nullable).
 - Y añade un nuevo campo a la tabla productos: 'marca' que sea una clave ajena al 'id' de la tabla 'marcas'.
 - Inserta unos cuantos datos en la nueva tabla, y vincula una marca a todos los productos de existentes en la BD.
 - Refactoriza el proyecto para poder filtrar en el catálogo los productos por marcas, además de por categorías.
 - En el detalle del producto, añade su marca y un enlace a su web (campo URL).

Ejercicio

- Completa el CRUD añadiendo la opción de modificar un producto.
- Implementa las operaciones de añadir y eliminar de forma asíncrona con llamadas AJAX.
 - Haz los métodos con manejadores con nombre en la misma página.
 - Usa un modal para incrustar el formulario de añadir
 - Usa un segundo modal para solicitar confirmación al usuario antes de eliminar el producto.

Referencias

- Tutorial sobre ASP.net CORE:
 - <https://docs.microsoft.com/es-es/aspnet/core/data/ef-rp/intro?view=aspnetcore-6.0&tabs=visual-studio>
- Comandos de las herramientas EF para CORE:
 - <https://docs.microsoft.com/en-us/ef/core/cli/powershell>
- Sobre atributos integrados:
 - <https://docs.microsoft.com/es-es/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-6.0>
- Acceso a datos y scaffolding:
 - <https://docs.microsoft.com/es-es/ef/core/managing-schemas/scaffolding?tabs=vs>

Referencias

- Atributos disponibles para las anotaciones de datos:
 - <https://docs.microsoft.com/es-es/dotnet/api/system.componentmodel.dataannotations?view=net-6.0>
- Validación de modelos:
 - <https://docs.microsoft.com/es-es/aspnet/core/mvc/models/validation?view=aspnetcore-6.0>
- Razor Pages con EF Core en ASP.NET Core: Ordenación, filtrado y paginación
 - <https://docs.microsoft.com/es-es/aspnet/core/data/ef-rp/sort-filter-page?view=aspnetcore-6.0>