18. Agregar funcionalidad de paginación a la página Index de Categorías

Para agregar paginación a la página Index de categorías, vamos a crear una clase `PaginatedList` que utilice instrucciones `Skip` y `Take` para filtrar datos en el servidor en lugar de siempre recuperar todas las filas de la tabla. A continuación, realizará cambios adicionales en el método Index y agregará botones de paginación a la vista `Index`.

En la carpeta de proyecto crear `Paginacion.cs` y, a continuación, reemplace el código de plantilla con el código siguiente:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

public class Paginacion<T> : List<T>
{
    public int PageIndex { get; private set; }
    public int TotalPages { get; private set; }
    public int TotalR { get; private set; }

    public Paginacion(List<T> items, int count, int pageIndex, int pageSize)
    {
        PageIndex = pageIndex;
        TotalPages = (int)Math.Ceiling(count / (double)pageSize);
        TotalR=count;


        this.AddRange(items);
    }

    public bool HasPreviousPage
    {
        get
        {
            return (PageIndex > 1);
        }
    }

    public bool HasNextPage
    {
        get
        {
            return (PageIndex < TotalPages);
        }
    }

    public static async Task<Paginacion<T>> CreateAsync(IQueryable<T> source, int
pageIndex, int pageSize)
```

```
    {
        var count = await source.CountAsync();
        var items = await source.Skip((pageIndex - 1) *
pageSize).Take(pageSize).ToListAsync();
        return new Paginacion<T>(items, count, pageIndex, pageSize);
    }
}
```

El método `CreateAsync` en este código toma el tamaño de página y el número de página y aplica las instrucciones de `Skip` y `Take` apropiadas al `IQueryable`. Cuando se llama a `ToListAsync` en `IQueryable`, devolverá una lista que contenga sólo la página solicitada. Las propiedades `HasPreviousPage` y `HasNextPage` se pueden utilizar para activar o desactivar los botones de paginación Anterior y Siguiente.

Se utiliza un método `CreateAsync` en lugar de un constructor para crear el objeto `Paginacion<T>` porque los constructores no pueden ejecutar código asíncrono.

Agregar funcionalidad de paginación al método Index

Agregar funcionalidad de paginación al método Index

En *CategoriasController.cs*, reemplace el método de índice con el código siguiente.

```
public async Task<IActionResult> Index(string sortOrder,
            string currentFilter,
            string searchString,
            int? page)
        {
            ViewData["CurrentSort"] = sortOrder;
            ViewData["NombreSortParm"] = String.IsNullOrEmpty(sortOrder) ? "nombre_desc"
: "";
            ViewData["DescripcionSortParm"] = sortOrder == "descripcion_asc" ?
"descripcion_desc" : "descripcion_asc";


            if (searchString != null)
            {
                page = 1;
            }
            else
            {
                searchString = currentFilter;
            }
            ViewData["CurrentFilter"] = searchString;

            var categorias = from s in _context.Categorias
                             select s;
            if (!String.IsNullOrEmpty(searchString))
            {
```

```
                categorias = categorias.Where(s => s.Nombre.Contains(searchString)
                                       || s.Descripcion.Contains(searchString));
            }

            switch (sortOrder)
            {
                case "nombre_desc":
                    categorias = categorias.OrderByDescending(s => s.Nombre);
                    break;
                case "descripcion_asc":
                    categorias = categorias.OrderBy(s => s.Descripcion);
                    break;
                case "descripcion_desc":
                    categorias = categorias.OrderByDescending(s => s.Descripcion);
                    break;
                default:
                    categorias = categorias.OrderBy(s => s.Nombre);
                    break;
            }
            //return View(await categorias.AsNoTracking().ToListAsync());
            //return View(await _context.Categorias.ToListAsync());
            int pageSize = 3;
            return View(await
Paginacion<Categoria>.CreateAsync(categorias.AsNoTracking(), page ?? 1, pageSize));
        }
```

Este código agrega un parámetro de número de página (page), un parámetro de orden de clasificación actual (sortOrder) y un parámetro de filtro actual (currentFilter) a la firma de método.

```
public async Task<IActionResult> Index(string sortOrder,
            string currentFilter,
            string searchString,
            int? page)
```

La primera vez que se muestra la página o si el usuario no ha hecho clic en un enlace de paginación o clasificación, todos los parámetros serán nulos. Si se hace clic en un enlace de paginación, la variable de página contendrá el número de página que se mostrará.

El elemento `ViewData` denominado CurrentSort proporciona la vista con el orden actual, ya que debe incluirse en los enlaces de paginación para mantener el orden de clasificación durante la paginación.

El elemento `ViewData` denominado CurrentFilter proporciona la vista con la cadena de filtro actual. Este valor debe incluirse en los enlaces de paginación para mantener la

configuración del filtro durante la paginación y debe restaurarse en el cuadro de texto cuando se vuelva a mostrar la página.

Si la cadena de búsqueda se cambia durante la paginación, la página debe restablecerse a 1, ya que el nuevo filtro puede dar lugar a que aparezcan datos diferentes. La cadena de búsqueda se cambia cuando se introduce un valor en el cuadro de texto y se presiona el botón Enviar. En ese caso, el parámetro `searchString` no es null.

```
if (searchString != null)
{
    page = 1;
}
else
{
    searchString = currentFilter;
}
```

Al final del método `Index`, el método `Paginacion.CreateAsync` convierte la consulta de la categoria en una sola página de categorías en un tipo de colección que admita la paginación. Esa única página de categorías se pasa a la vista.

```
return View(await Paginacion<Categoria>.CreateAsync(categorias.AsNoTracking(), page ?? 1,
pageSize));
```

El método `Paginacion.CreateAsync` toma un número de página. Los dos signos de interrogación representan el operador nulo-coalescente. El operador de coagulación nula define un valor predeterminado para un tipo anulable; La expresión `(page ?? 1)` significa devolver el valor de `page` si tiene un valor, o devolver 1 si `page` es nula.

Agregar vinculos de paginación a la vista Index de categorías

En *Views /Categoria/Index.cshtml*, reemplace el código existente con el código siguiente. Los cambios se resaltan.

```
@model Paginacion<Sistema.Models.Categoria>
@{
    ViewData["Title"] = "Index";
}
```

```
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Filtro: <input type="text" name="searchString"
value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Buscar" class="btn btn-default" /> |
            <a asp-action="Index">Todos los registros</a> | Registros mostrados
@Model.TotalR
        </p>
    </div>
</form>
<table class="table">
    <thead>
        <tr>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["NombreSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Nombre</a>
                </th>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["DescripcionSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Descripcion</a>
                </th>
                <th>
                    Estado
                </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Nombre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Descripcion)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Estado)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.CategoriaID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.CategoriaID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.CategoriaID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

```
@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-page="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
    Previous
</a>
@for (int i = 1; i <= Model.TotalPages; i++)
{
    <a asp-action="Index"
       asp-route-sortOrder="@ViewData["CurrentSort"]"
       asp-route-page="@i"
       asp-route-currentFilter="@ViewData["CurrentFilter"]">
        <button class="btn btn-default">@i</button>
    </a>
}

<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-page="@(Model.PageIndex + 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @nextDisabled">
    Next
</a>
```