Habla de 2 arquitecturas básicas:

En la primera el usuario hace peticiones al servidor, y éste le responde al usuario.

En la segunda, gran parte de las cosas suceden del lado del cliente. También están las peticiones que se hacen al servidor.

Blazor trabaja del lado del cliente, permite hacer llamados al servidor.

Web Assembly transforma C# en un lenguaje intermedio, el cual puede ser ejecutado en el navegador (Chrome, Edge, explorer, Firefox).

Mono web assembly es runtime

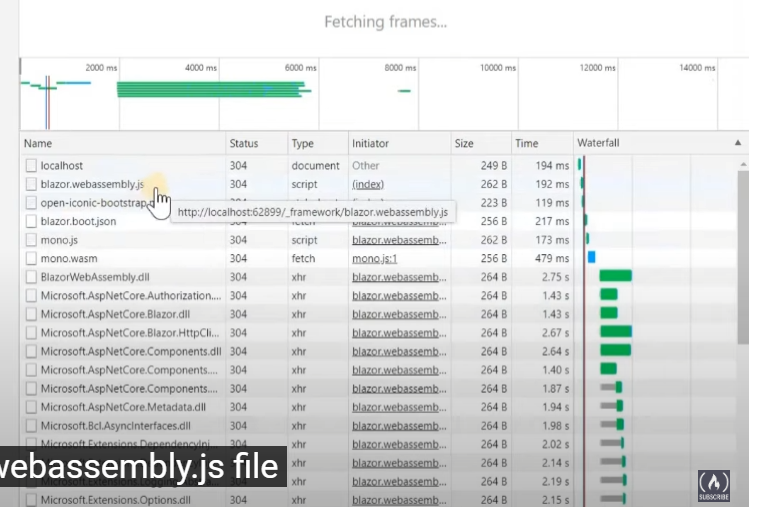
Blazor genera html

2 modelos de hospedaje

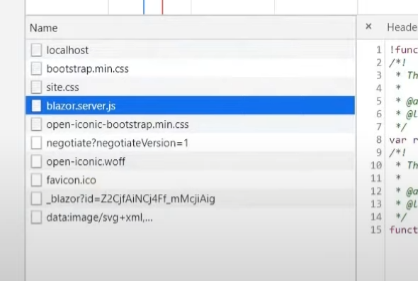
Puede todo estar del lado del cliente. Se descarga todo al principio. Pros y contras. No hay que hacer llamados a APIs. Se hace lento al tener que procesar muchos datos. Lenta carga inicial.

Server Side. Rápida carga inicial. No anda si no hay señal.

Ahí nos muestra lo que se descarga:

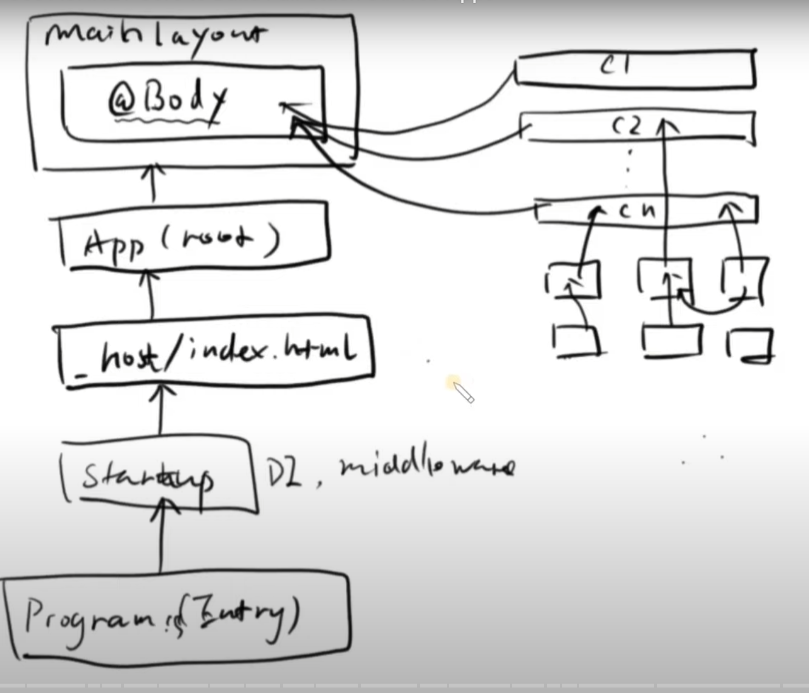


Esto que sigue es lo que se descarga con Server Side, se nota que es mucho menos:



StartUp no existe mas.

Host es para server side. Index.html es para client side.



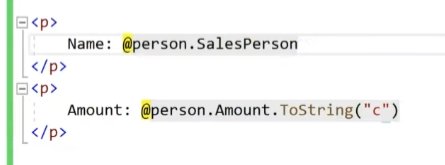
Data Binding

Sin data Binding tendría que manualmente establecerse qué variable se dispone en la UI.

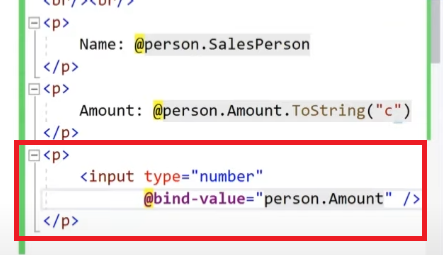
Se une (bind) un elemento con la variable.

Habla de One Way Data Binding y Two Ways.

Esta es la One Way. Desde ya, en otro lugar crea el objeto persona, con sus propiedades.



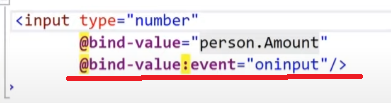
Esa es la otra forma:



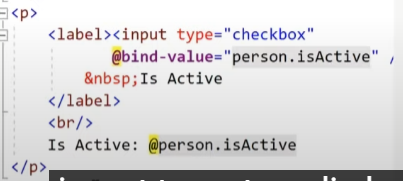
Ahí le agrego uno, y ya se muestra también en el Amount de arriba.



Agregando esa línea, hace que la modificación se vea apenas él la introduce. Antes era necesario tabular o darle enter.



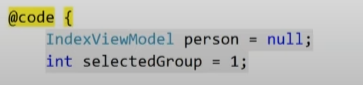
Checkbox:





RadioButton

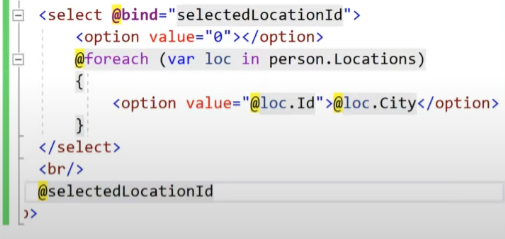






Este lo que hace es que al tocarse un radiobutton se muestra el selectedGroup (numerito) debajo. Le falta código para hacerlo interesante.

DropDownList



El option value 0, es lo que se va a mostrar inicialmente, que sería nada.

El @selectedlocationId de abajo es lo que se verá cuando se seleccione algo:



List y table lo hace one way, sin bind. Luego lo cambia.

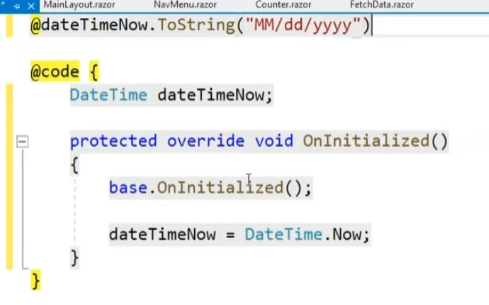
Sale medio cortadín.

Hace gran parte one way y la ciudad de la table 2 way.



Crear components

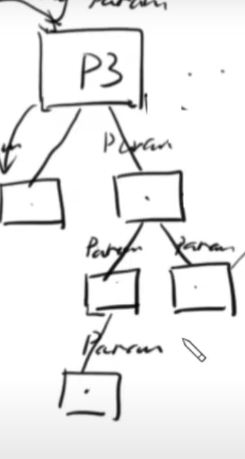
Así crea uno que muestre la hora actual



También se pueden usar páginas como componentes.

Comunicación entre componentes

Dice que la comunicación entre una página y sos componentes se hace a través de ***parámetros*** ***regulares***. En tanto que la comunicación entre las páginas se hace a través de ***Routing*** ***Parameters***.

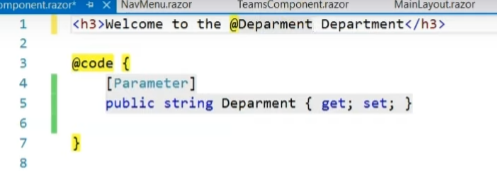


Para que llegue un parámetro desde arriba de todo hasta abajo, puede apelarse a los parámetros regulares, pero afirma, que también están los ***cascading*** ***parameters***.

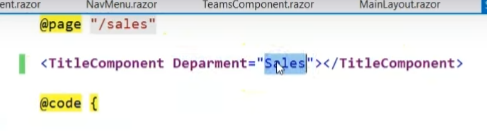
Cuando un hijo le cuenta algo al padre (componente a página, por ejemplo). Se llama ***Event*** ***Callback***.

Parámetro regular

Este es el componente, el hijo. Aquí se le va a mandar info, y se va a mostrar en la línea de arriba.

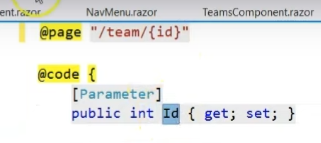


Lo de abajo es el padre. “Sales” es el parámetro que se le pasa a Department.

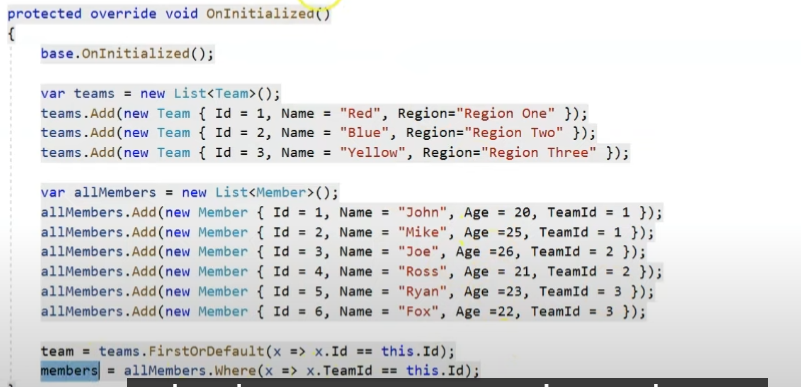


Cascading Parameter

En el página a la que se le manda el parámetro tiene:

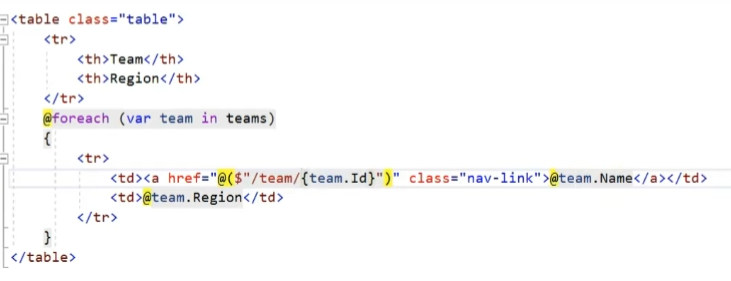


En realidad eso tiraría error, porque al pasarse Id se lo toma como string y la propiedad es un entero. Se soluciona de 2 formas: o poniendo la propiedad como string o poniendo: @page “/team/{id:int}”

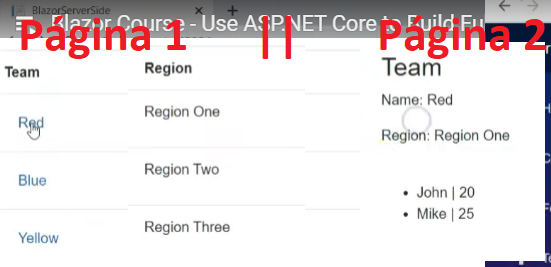


Ahí mismo hace un par de búsquedas sobre su “BBDD”, en función del Id que recibe como parámetro.

Así se le envía desde el padre el team.id



Esa es la página 1, y luego la 2.



Comunicación entre componentes con EventCallBack

El hijo le cuenta algo importante al padre.

Declara una variable en el componente padre, que recibirá:  
Manejo un tipo de dato extraño, propio del WeatherForeCast

private WeatherForecast[]? forecasts;

En el componente hijo declara una propiedad de tipo EventCallBack, al cual lo establece como parámetro. Y en el método que hace que suceda lo importante, invoca al evento

[Parameter]

public EventCallback<WeatherForecast[]> SettearValores { get; set; }

public void SumaUno()

{

foreach (var item in forecasts)

{

item.TemperatureC += 1;

}

SettearValores.InvokeAsync(forecasts);

}

Se toca de Vuelta el padre:

<**FetchData** **SettearValores**="SetearValoresEnPadre"></**FetchData**>

@code {

private WeatherForecast[]? \_forecasts;

private void SetearValoresEnPadre(WeatherForecast[]? forecasts)

{

\_forecasts = forecasts;

}

}

SettearValores es el nombre del evento y SetearValoresEnPadre debe ser igual que el método.

Y ahí ya está, se envía el resultado novedoso al padre, quien puede decidir qué hacer al respeto.

Dice que el EventCallBack se parece mucho a un delegado pero no lo es. Dice que se lo puede invocar, esté o no lleno(¿registrado?).

Comunicación entre componentes By referencing Child Components

Crea un componente hijo. Allí declara un atributo booleano (false) y establece un if en el html para que solo se muestre su contenido en caso de ser true.

Asimismo, en el hijo establece un método público que haga el cambio de estado del booleano+StateHasChanged. Este método será llamado por el padre.

Los componentes se renderizan en 4 tipos de situaciones:

1- Cuando el componente es creado.

2- Cuando cambia el valor de algún parámetro.

3- Cuando hay un evento handler.

4- Cuando el desarrollador se lo ordena: StateHasChanged();

Componente padre:

<button class="btn btn-primary" @onclick="(()=>{

hijo.Mostrattore();

})">Mostrar hijo</button>

<**HijoComponent** @ref="hijo"></**HijoComponent**>

@code{

private HijoComponent hijo;

}

Crea un botón que llama al método del hijo. En el code se crea una instancia del hijo, a la vez que se establece la referencia al mismo en HijoComponent.

Templated Components



Render Fragment

Copia algo así:

<div class="mb-3">

<label for="exampleFormControlInput1" class="form-label">Email address</label>

<input type="email" class="form-control" id="exampleFormControlInput1" placeholder="name@example.com">

</div>

Lo cual sirve para ingresar un dato. Afirma que si se quisieran ingresar p ej 10 datos, serían demasiadas líneas de código (en mi caso no tantas) . Por eso hace muchas cosas.

Crea un componente en donde pega eso y luego lo edita:

<h3>NuevoComponent</h3>

<div class="mb-3">

<label for="exampleFormControlInput1" class="form-label"> @Label</label>

@Control

</div>

@code {

[Parameter]

public string Label { get; set; }

[Parameter]

public RenderFragment Control { get; set; }

}

Ese control hace acordar al @body del MainLayout.

En la página pone esto:

<**NuevoComponent** **Label**="Name">

<**Control**>

<input type="text" class="form-control" id="exampleFormControlInput1" placeholder="Name">

</**Control**>

</**NuevoComponent**>

<**NuevoComponent** **Label**="Email">

<**Control**>

<input type="email" class="form-control" id="exampleFormControlInput1" placeholder="name@example.com">

</**Control**>

</**NuevoComponent**>

<**NuevoComponent** **Label**="Department">

<**Control**>

<input type="text" class="form-control" id="exampleFormControlInput1" placeholder="Department">

</**Control**>

</**NuevoComponent**>

Siendo unas pocas líneas nada mas

Generic Typed Items List

Se vincula con el cuarto de los temas presentados de los templates.

En la página, pone una clase de empleado. En OnInitialized crea una lista y le mete 3 empleados. Luego hace esto:

<**GridAutoComponent** Items="empleados"></**GridAutoComponent**>

Lo pasa por parámetro.Los recibe un parámetro llamado Items

Además:

Agrega componente automaticGridComponent. Parámetro. Agrega arriba algo de typeParam.

Usa Reflection. @Sysyem.Reflection

OnParameterSet, Evento que se ejecuta después de que los parámetros se establecen y antes de que se renderiza el componente.

@typeparam TItem

@using System.Reflection

@if (memberInfo != null && memberInfo.Length > 0)

{

<table class="table">

<thead>

<tr>

@foreach (var member in memberInfo)

{

if (member.MemberType == MemberTypes.Property)

{

<th>

@member.Name

</th>

}

}

</tr>

</thead>

<tbody>

@foreach (var item in Items)

{

<tr>

@foreach (var member in memberInfo)

{

@if (member.MemberType == MemberTypes.Property)

{

<td>

@item.GetType().GetProperty(member.Name).GetValue(item)

</td>

}

}

</tr>

}

</tbody>

</table>

}

@code {

[Parameter]

public List<TItem>Items { get; set; }

Aquí se recibe el listado que se le manda.

MemberInfo[] memberInfo;

Esto de arriba viene del Reflection

protected override void OnParametersSet()

{

base.OnParametersSet();

if (Items!=null && Items.Count > 0)

{

Type typeOfItem = this.Items[0].GetType();

memberInfo = typeOfItem.GetMembers();

}

}

Luego hace modificaciones para que no se muestre el Id.

Le pasa otro parámetro mas. En padre:

<**GridAutoComponent** **Items**="empleados" **ColumnsToExcludeCSV**="Id"></**GridAutoComponent**>

En componente:

[Parameter]

public string ColumnsToExcludeCSV { get; set; }

Luego lo utiliza como filtro en el table (en thead y en tbody):

@foreach (var member in memberInfo)

{

@if (member.MemberType == MemberTypes.Property &&

!ColumnsToExcludeCSV.Contains(member.Name, StringComparison.OrdinalIgnoreCase))

{

¡¡Modificar clases del html!!

Se manda y recibe parámetro y allí donde se define la clase se pone:

<table class="@($"table {CssClass}")">

Generic Typed RenderFragment

Dice -creo- que hay 2 formas típicas de esto.

La primera es como repeater.

Desde la página se hace lo mismo que con lo anterior. Crea una clase, luego una lista con 3 objetos de esa clase y se la manda por parámetro al componente. Agrega si algo, en lo que se manda por parámetro para establecer el template.

<**RepeaterComponent** **Items**="customers">

<**Row** **Context**="costumer">

<br />

<div><b>@costumer.CompanyName </b></div>

<div class="ms-5">@costumer.ContactName</div>

<div class="ms-5">@costumer.PhoneNumber</div>

@\*Agrega un margen a la izquierda y al ppio un renglon en blanco\*@

</**Row**>

</**RepeaterComponent**>

El componente queda con poco código. Básicamente recibe los parámetros y los muestra:

@typeparam TItem

@foreach (var item in Items)

{

@Row(item)

}

@code {

[Parameter]

public List<TItem> Items { get; set; }

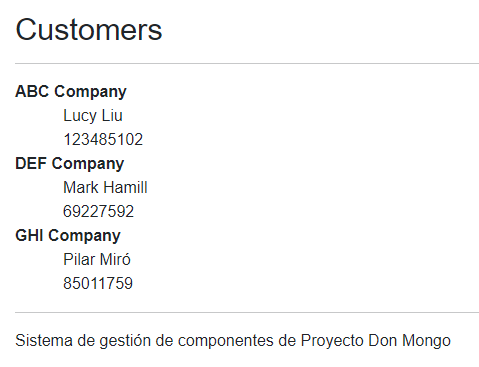
[Parameter]

public RenderFragment<TItem> Row { get; set; }

}

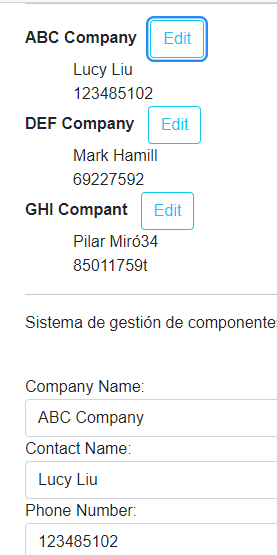


Le agrega luego header y footer.



Ahora estaría yendo hacia lo que llamó la segunda función de este template, algo así como Edit Form…

Hace miles de cosas:



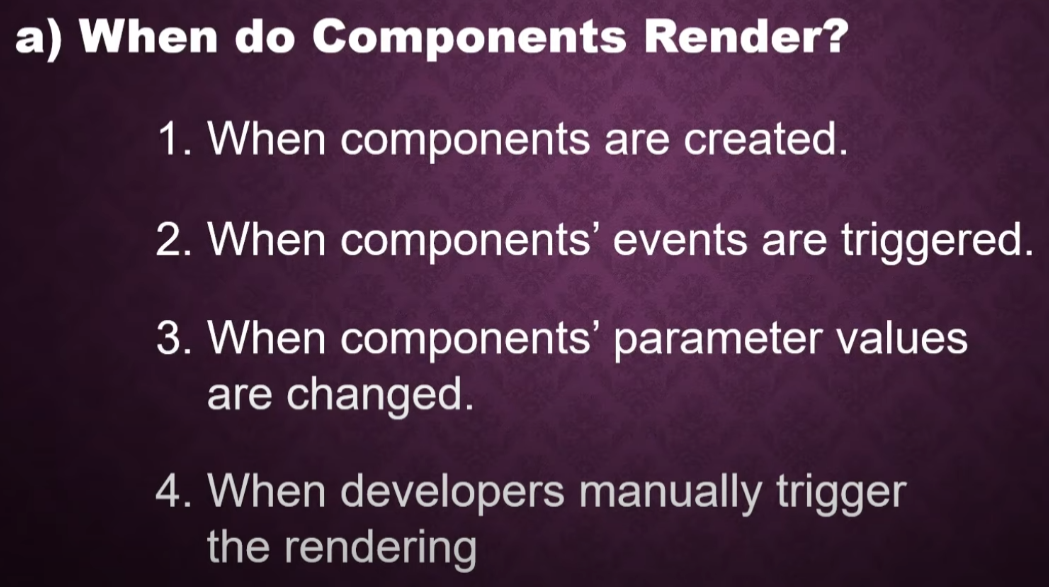
Así queda. Si tocas Edit, aparecen los datos abajo y al cambiarlos, se cambian también arriba.

LifeCycle Events:

When is Rendering Triggered

El primero refiere a cuando se crea, se refresca la página o se navega fuera y se vuelve.

Del segundo aclara que los eventos que renderizan son los eventos UI y tiene que suceder dentro del componente en cuestión., con la excepción de los template components.



Event Sequence

Los divide en 2 categorías.

La primera tiene que ver con los eventos que se llevan a cabo cuando la página se renderiza.

-SetParameterSetAsync

-OnInitialized

-OnParameterSet

Dice que estos eventos son ejecutados 2 veces. No entendí por qué. Eso sucede con Blazor Server no con WebAssembly, porque no hay Signal R.

Este también es invocado, luego de los otros 3.

-OnAfterRender.

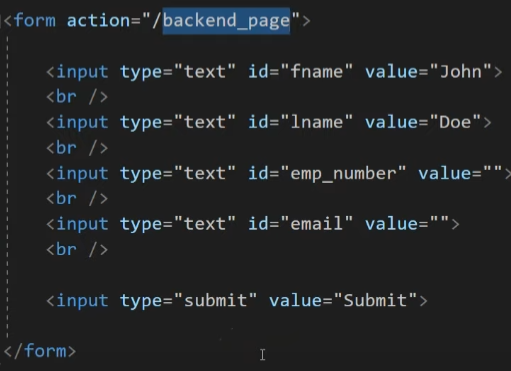
Ahora habla de los eventos invocados luego de que la página se renderizó.

Por ejemplo, luego de tocar un botón de la página, se invoca el anterior, así como el que sigue. Se re-renderiza el componente, no así el child-component.

-ShouldRender

Forms and validations

Así se hace con html. Pero aclara que Blazor es Simple Page Application, por lo que no se hace postback al tocar el botón que hace submit



En blazor se usa EditForm. Se especifica un método que se ejecutará. Además, está atado a un modelo. Están dentro los bind-value así como el tipo de input del que se trata.

Posteriormente agrega que pueden usarse los dataAnottations.

Aclara que el botón no debe ser tipo submit, sino tipo button.

@using System.ComponentModel.DataAnnotations;

<**EditForm** **Model**="person" **OnInvalidSubmit**="ValidFormSubmitted" >

<**DataAnnotationsValidator**></**DataAnnotationsValidator**>

<**ValidationSummary**></**ValidationSummary**>

@\*Esta segunda línea agrega los msjitos de invalidación\*@

<label>

First Name:<br />

<**InputText** id="fname" @bind-Value="person.FirstName"></**InputText**>

</label>

<br />

<label>

Last Name: <br />

<**InputText** id="lname" @bind-Value="person.LastName"></**InputText**>

</label>

<br />

<label>

Employee Number: <br />

<**InputNumber** id="empNumber" @bind-Value="person.EmployeeNumber"></**InputNumber**>

</label>

<br />

<label>

Email Adress:<br />

<**InputText** id="email" @bind-Value="person.Email"></**InputText**>

</label>

<br />

<input type="submit" value="Submit" />

<button type="button">Enviar data</button>

</**EditForm**>

@if (personita != null)

{

<p><b>@personita.FirstName @personita.LastName</b></p>

<p>@personita.EmployeeNumber</p>

<p>@personita.Email</p>

}

@code{

Person person = new Person();

Person personita;

private void ValidFormSubmitted()

{

personita = person;

}

class Person{

[Required]

[StringLength(50)]

public string FirstName {get;set;}

[Required]

[StringLength(50)]

public string LastName { get; set; }

[Required]

[Range(1,1000)]

public int EmployeeNumber { get; set; }

[Required]

[EmailAddress]

public string Email { get; set; }

}

}

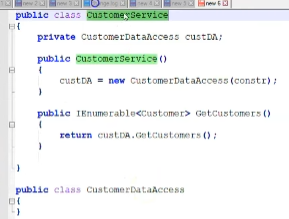
\*\*\*REVISAD\*\*\*

Muestra alguna falencia al tocar el botonito.

Dependency Injection

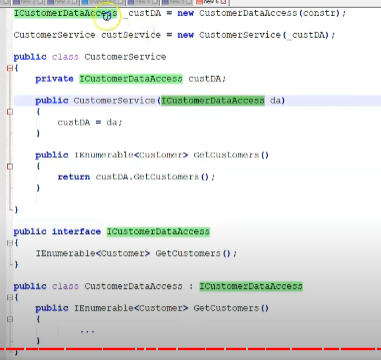
Ejemplifica

Tenés 2 clases CustomerService y CustomerDataAccess. La primera quiere acceder a la segunda para obtener su data. Por eso se la hace depender de la segunda.



Ahí se ve cómo se declara y luego en el constructor crea una instancia de DAL en customerService. Eso está altamente acoplado.

Dependency Inversion. Se depende de la función, no de la clase. Y la función es abstracta.



Ahí se ve como CustomerDAL implementa ICostumerDAL. Ésta última es la que se le pasa por parámetro a CustomerService por medio del constructor.

Hay 3 formas: transient, scoped y singleton.

Con transient, cada vez que, por ejemplo, abrimos una página, se crea una instancia de las clases que se inyectan.

Con Singleton, se crea una sola instancia, y cada vez que se abre una página se trae esa misma y única instancia.

Scoped. Hace que el servicio viva tanto como la conexión. Es decir, si te vas de una página y volvés, todo será igual (como con singleton), pero si refrescás la página, se crea una nueva instancia de lo inyectado.

Todo lo anterior vale para Blazor Server Side.

Con WebAssembly:

Transiend trabaja igual.

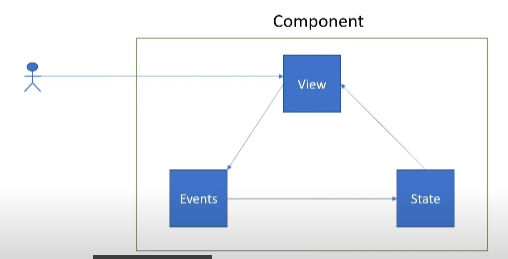
Singleton, al refrescar se crea una nueva instancia.

Scoped. Trabaja igual, e igual que singleton de webAssembly.

Esto es porque se descargó toda el código al navegador. Todos los Dlls. Program Cs corre en el navegador, por eso las clases no son destruidas y creadas de vuelta.

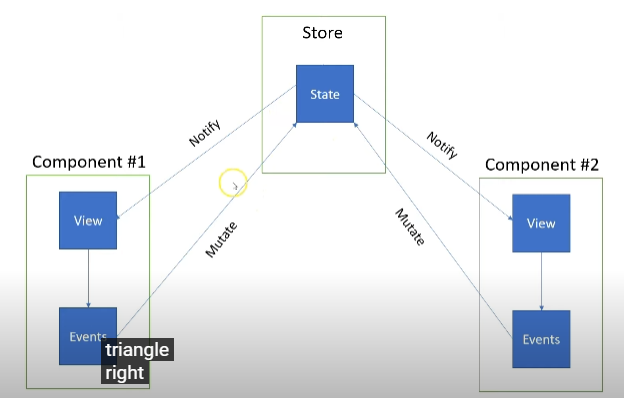
State Management with Flux in C#

Flux sería una arquitectura.

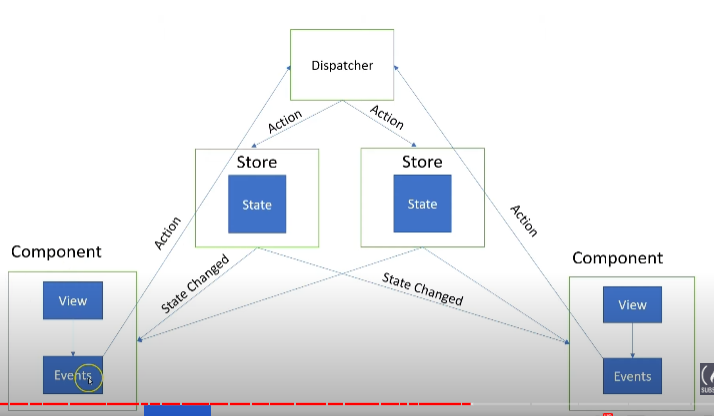


Eso vale tanto para Blazor, React, Angular, etc.

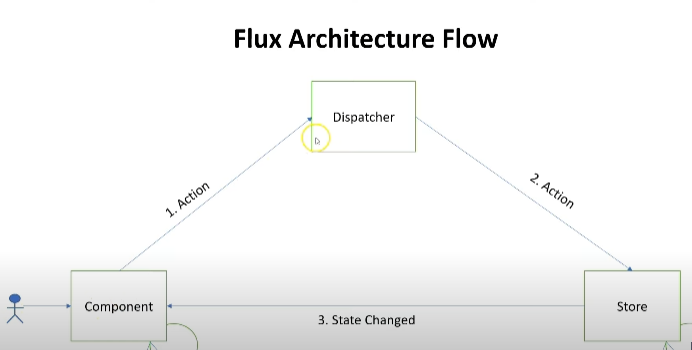
Compartir el estado entre componentes complica las cosas.



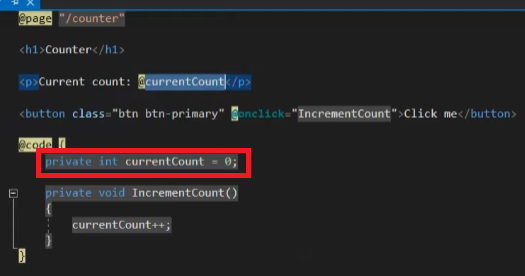
Este es un patrón observer. Todo lo que hay en Store (Subject) lo ven los componentes (observers).



Ahí agrega el dispatcher. Que es quien modifica los stores (state). Es un observador de 2 niveles. Entiendo que el dispatcher es un observer de observers.



Flux simplificaría lo anterior.



Ese atributo es un local state.

Crea carpeta Store y allí otra CounterStoreFolder. Ahí una clase CounterStore. Así queda:

public class CounterState

{

public int Count { get; }

public CounterState(int count)

{

Count = count;

}

}

public class CounterStore

{

private CounterState \_state;

public CounterStore()

{

\_state = new CounterState(0);

}

public CounterState GetState()

{

return \_state;

}

public void IncrementCount()

{

var count=this.\_state.Count;

count++;

this.\_state=new CounterState(count);

BroadcastStateChange();

}

public void DecrementCount()

{

var count = this.\_state.Count;

count--;

this.\_state = new CounterState(count);

BroadcastStateChange();

}

///////////////////////

//Acá entra el patrón Observer

private Action \_listeners;

public void AddStateChangeListener(Action listener)

{

\_listeners += listener;

}

public void RemoveStateChangeListener(Action listener)

{

\_listeners = listener;

}

private void BroadcastStateChange()

{

\_listeners.Invoke();

}

}

En program.Cs inyecciona la dependencia:

builder.Services.AddScoped<CounterStore>();

En la página inject CounterStore. Ahí mismo o en imports va el using.

Elimina el atributo y usa el counterStore:

@page "/counter"

@inject CounterStore counterStore

<**PageTitle**>Counter</**PageTitle**>

<h1>Counter</h1>

@\*Acá se modificó el atributo por el counterstore.getState()\*@

<p role="status">Current count: @counterStore.GetState().Count</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {

//Acá se eliminó el atributo que recogía el conteo

protected override void OnInitialized()

{

base.OnInitialized();

//acá se suscribe al observer.

counterStore.AddStateChangeListener(() => { StateHasChanged(); });

}

//Acá se usan los métodos para incrementar y reducir el contador:

private void IncrementCount()

{

counterStore.IncrementCount();

}

private void DecrementCount()

{

counterStore.DecrementCount();

}

}

Ahora toca el NavMenu. Le inyecta el CounterStore. Lo suscribe al observer (en el OnInitialized) y luego hace esto:

<div class="nav-item px-3">

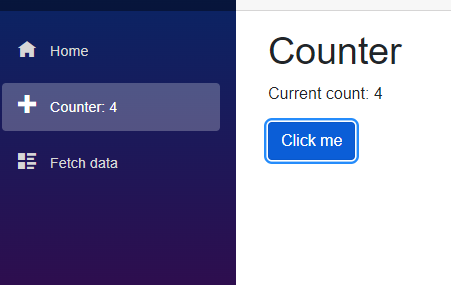
<**NavLink** class="nav-link" href="counter">

<span class="oi oi-plus" aria-hidden="true"></span> Counter: @counterStore.GetState().Count

</**NavLink**>

</div>

Hasta acá es un observer de un solo nivel.



La información se genera clickeando en el componente counter, se guarda en CounterStore y luego tanto counter como NavMenu acceden a la misma.

Como están las cosas, con Blazor Server y Scoped. Si navego por la página la información persiste. Si refresco, la info se pierde; parecería que esto es un error.

Toquetea mucho el código:

@page "/counter"

@implements IDisposable

@inject CounterStore counterStore

<**PageTitle**>Counter</**PageTitle**>

<h1>Counter</h1>

@\*Acá se modificó el atributo por el counterstore.getState()\*@

<p role="status">Current count: @counterStore.GetState().Count</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {

//Acá se eliminó el atributo que recogía el conteo

protected override void OnInitialized()

{

base.OnInitialized();

//acá se suscribe al observer.

counterStore.AddStateChangeListener(() => { UpdateView(); });

}

//Acá se usan los métodos para incrementar y reducir el contador:

private void IncrementCount()

{

counterStore.IncrementCount();

}

private void DecrementCount()

{

counterStore.DecrementCount();

}

private void UpdateView()

{

StateHasChanged();

}

public void Dispose()

{

counterStore.RemoveStateChangeListener(UpdateView);

}

}

Aclara que va a implementar Dispatch.

Agrega clase en Store IncrementAction, otra DecrementAction, ambas implementan la interfaz IAction. También crea otra ActionDispatcher.

En esta última codea esto:

public class ActionDispatcher

{

private Action<IAction> \_registeredActionHandlers;

public void Subscript(Action <IAction> actionHandler)

{

\_registeredActionHandlers += actionHandler;

}

public void UnSubscript(Action<IAction> actionHandler)

{

\_registeredActionHandlers -= actionHandler;

}

public void Dispatch (IAction action)

{

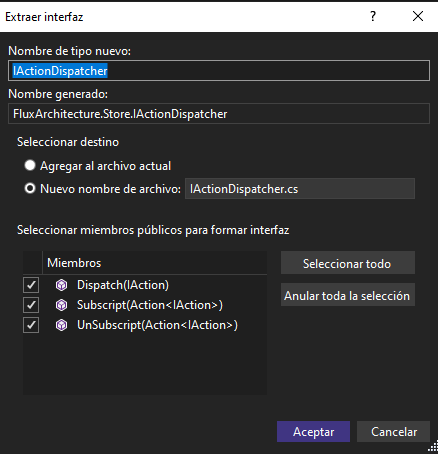
\_registeredActionHandlers?.Invoke(action);

}

}

Esta clase debería implementar una interfaz, que aún no existe. Para eso, hay una forma fácil de hacerlo con Visual Studio.

Edit=>Refactor=>ExtractInterface



public interface IActionDispatcher

{

void Dispatch(IAction action);

void Subscript(Action<IAction> actionHandler);

void UnSubscript(Action<IAction> actionHandler);

}

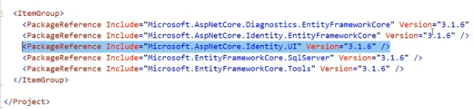
Program.CSS addscoped

Ahora le llega el turno al Store y agrega el ActionDispatcher al ctor. Lo suscribe.

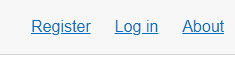
El Counter por su parte, no se comunica directamente con el Store, sino con el dispatcher.

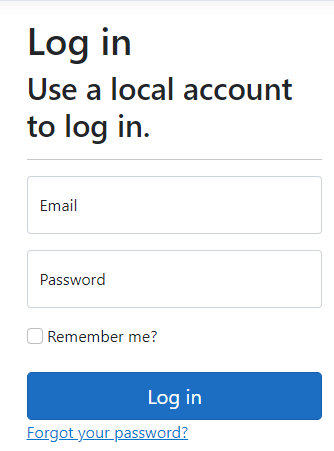
Authentication

Autenticación de campo=>Cuentas individuales.



Paquetes NuGet. Ya vienen instalados.





Todo eso ya aparece así. Tirará error porque no está hecha la migración.

"DefaultConnection": "Server=(local);Database=aspnet-Authentication-9a84cfec-4190-492e-9103-44fa0caa2f2b;Trusted\_Connection=True;MultipleActiveResultSets=true;TrustServerCertificate=True;"

Tuve que cambiar appSettings para poder poner update-database.

pablokrh@gmail.com

Pp2023\_kk

Para requerir el logueo a la hora de utilizar una página:

@attribute [Authorize]

Con solo esa línea ya deniega el acceso:



Pero, de esta otra forma, no está toda la página bloqueada, sino que el navegador puede ver lo que uno le permite.

@page "/counter"

<**PageTitle**>Counter</**PageTitle**>

<h1>Counter</h1>

<**AuthorizeView**>

<**Authorized**>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

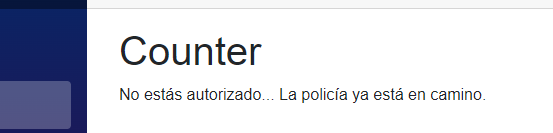
</**Authorized**>

<**NotAuthorized**>

No estás autorizado... La policía ya está en camino.

</**NotAuthorized**>

</**AuthorizeView**>



También se puede solamente poner:

@attribute [Authorize]

En la página, y en app.razor poner:

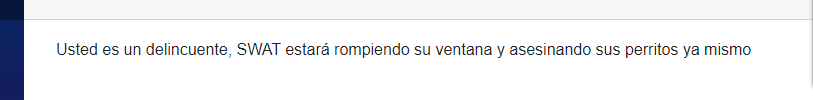
<**AuthorizeRouteView** **RouteData**="@routeData" **DefaultLayout**="@typeof(MainLayout)">

<**NotAuthorized**>

Usted es un delincuente, SWAT estará rompiendo su ventana y asesinando sus perritos ya mismo

</**NotAuthorized**>

</**AuthorizeRouteView**>



Utilizar para la App la información del usuario:

@if (authState.User.Identity.IsAuthenticated != false)

{

<p>

Name:@authState.User.Identity.Name

</p>

@foreach (var claim in authState.User.Claims)

{

<p>

Claim Type: @claim.Type

</p>

<p>

Claim Value: @claim.Value

</p>

}

}

@code{

[CascadingParameter]

private Task<AuthenticationState>\_authState{ get; set; }

private AuthenticationState authState;

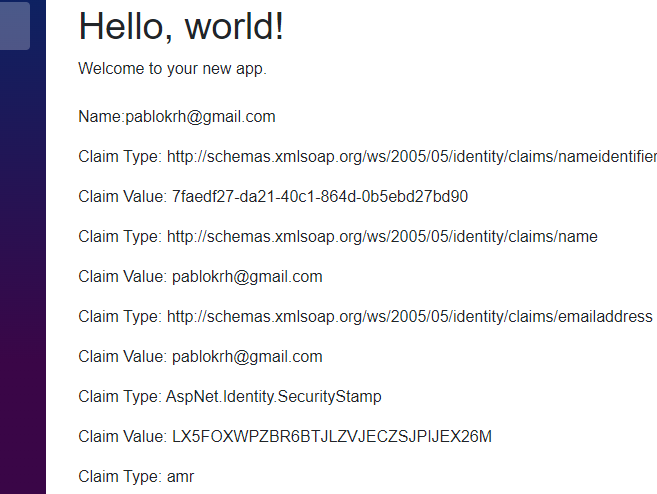
protected override async Task OnInitializedAsync()

{

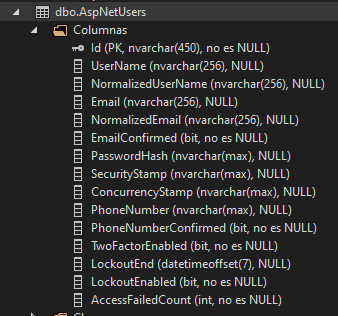
authState = await \_authState;

}

}



Ahora busca customizar algo:



Quiere agregarle a dicha tabla una columna mas, con los datos de la escuela. Crea una clase en la carpeta Data/Migrations y la hace heredera de IdentityUser.

public class AspNetUser : IdentityUser

{

public string School { get; set; }

}

Luego en ApplicationDbContext, modifica lo amarilleado:

public class ApplicationDbContext : IdentityDbContext<AspNetUser>

Program.Cs

En vez de esta línea:

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)

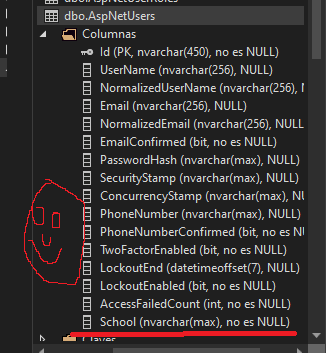
Pone esto otro:

builder.Services.AddIdentity<AspNetUser, IdentityRole>(options => options.SignIn.RequireConfirmedAccount = true)

Migrazzione:

add-migration extended-AspNetUsers

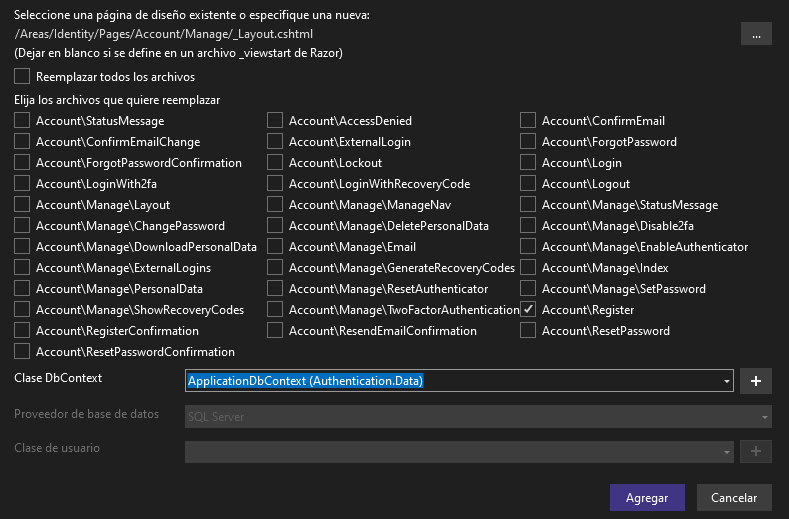
Update-Database



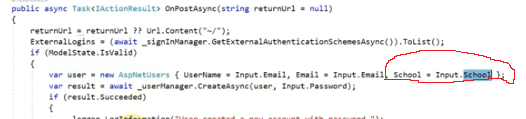
Customizar otra cosa:

Click en el Proyecto=> Add=>Nuevo element con scaffold=>Identity

Luego elige sobreescribir solamente el registro y se elige el contexto.



Tras esto uno tiene acceso a un archivo razor con el registro. Toca también el Cs.



Yo no lo tengo así.

Busca luego en toda la Solución donde diga IdentityUser y lo reemplaza por AspNetUser

Toca también el program Cs.

builder.Services.AddIdentity<AspNetUser, IdentityRole>(options => options.SignIn.RequireConfirmedAccount = true)

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultUI()

.AddDefaultTokenProviders();

Aborto aborto

6:01:41 /

DataGrid: Columns Configuration

Crea DataGridComponent.Razor.

Le pasa un parámetro genérico <TItem>.

Crea una carpeta Configuration, para configurar el datagrid. El nombre de la columna puede ser diferente del caption de la columna, por esto, hace esto:

new ConfigurationDG{DataField="EmploymentDate", Caption="Employment Date"

public class ConfigurationDG

{

public ConfigurationDG()

{

this.DataType = DataType.NotSet;

this.Alignment = Alignment.NotSet;

}

public string DataField { get; set; }

public string Caption { get; set; }

public string Format { get; set; }

public DataType DataType { get; set; }

public Alignment Alignment { get; set; }

}

Alignment y DataType son 2 enums, definidos en otro lado.

En el HTML pone esto, para clarificar:

<table class="table">

<thead>

<tr>

@if (ConfigurationDG == null || ConfigurationDG.Count <= 0)

{

//create the header with reflection

}

else

{

//Creare the header based on columns definitions

}

</tr>

</thead>

</table>

En index, crea un modelo (employee), un listado de objetos de tal modelo, y lo inicializa. También crea un listado de configuracionesdeColumna.

Luego, vuelve al componente y pone:

@foreach (var item in ConfigurationDG)

{

<th>@item.Caption</th>

}

}

</tr>

</thead>

<tbody>

@foreach (var item in DataItems)

{

<tr>

@foreach (var column in ConfigurationDG)

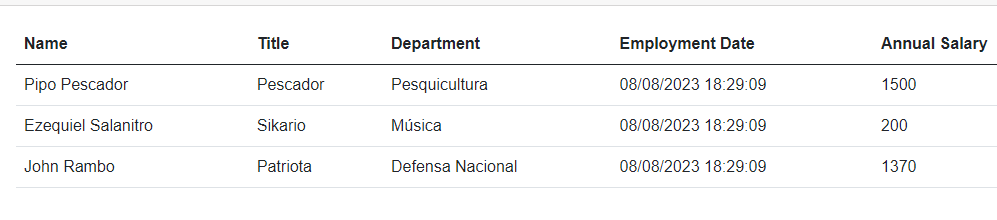
{

<td>@typeof(TItem).GetProperty(column.DataField).GetValue(item)</td>

}

</tr>

}



Ahí ya estaría, pero hace 2 retoques:

Toca el alineamiento del texto:

@foreach (var column in ConfigurationDG)

{

@if (column.Alignment != Alignment.NotSet)

{

<td align="@column.Alignment.ToString().ToLower()">@typeof(TItem).GetProperty(column.DataField).GetValue(item)</td>

}

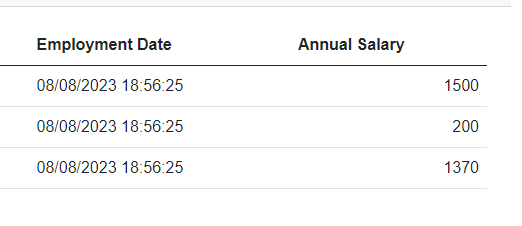
else

{

<td>@typeof(TItem).GetProperty(column.DataField).GetValue(item)</td>

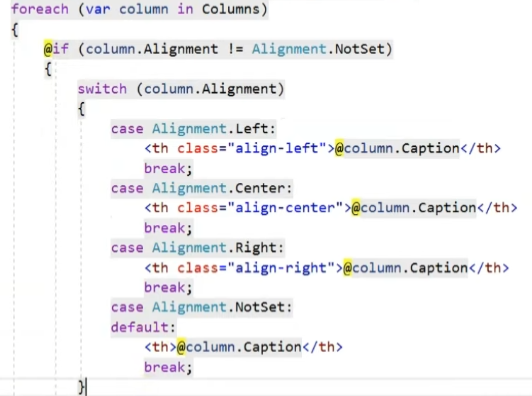
}

}

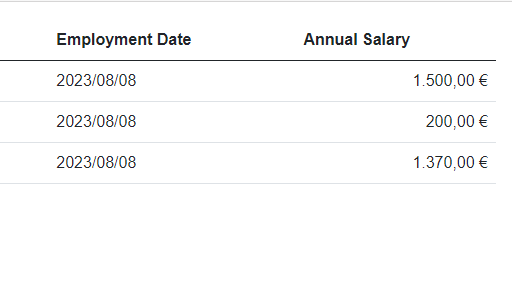


Como se ve los valores del salario se alinearon a la derecha. Lo mismo hizo con el título (Yo no copié el código al DocX), sin embargo no sucedió eso. Por eso, creó un archivo Css para dar el estilo

Prefiero darlo por fracaso y no hacer esto:



Así le quedó. Esas clases fueron creadas por él.

Luego toca el formato de los strings printeados:  
 @foreach (var column in ConfigurationDG)

{

@if (!string.IsNullOrWhiteSpace(column.Format) && column.DataType != DataType.NotSet)

{

string value = string.Empty;

switch (column.DataType)

{

case DataType.Date:

case DataType.DateTime:

value = DateTime.Parse(typeof(TItem).GetProperty(column.DataField).GetValue(item).ToString()).ToString(column.Format);

break;

case DataType.Boolean:

value = bool.Parse(typeof(TItem).GetProperty(column.DataField).GetValue(item).ToString()).ToString();

break;

case DataType.Currency:

value = Math.Round(double.Parse(typeof(TItem).GetProperty(column.DataField).GetValue(item).ToString()),2).ToString("c");

break;

default:

break;

}

@if (column.Alignment != Alignment.NotSet)

{

<td align="@column.Alignment.ToString().ToLower()">@value</td>

}

else

{

<td>@value</td>

}

}

else

{

@if (column.Alignment != Alignment.NotSet)

{

<td align="@column.Alignment.ToString().ToLower()">@typeof(TItem).GetProperty(column.DataField).GetValue(item)</td>

}

else

{

<td>@typeof(TItem).GetProperty(column.DataField).GetValue(item)</td>

}

}

}

Es largo, hay un primer gran if, para el caso en que se haya definido un formato. Ahí se ejecuta un switch que va fijando el string.

DataGrid: Paging

Agrega una clase PagingConfig en Configuration.

public class PaginConfig

{

public bool Enabled { get; set; }

public int PageSize { get; set; }

public bool CustomPager { get; set; }

public int NumOfItemsToSkip (int pageNumber)

{

if (Enabled)

{

return (pageNumber - 1)\*PageSize;

}

//Acá hay un else implícito:

return 0;

}

public int NumOfItemsToTake (int totalItems)

{

if (Enabled)

{

return PageSize;

}

return totalItems;

}

//IR A LA PRÓXIMA O ANTERIOR PÁGINA

public int PrevPageNumbe (int currentPageNumber)

{

if (currentPageNumber > 1)

{

return (currentPageNumber - 1);

}

else return 1;

}

public int NextPageNumbe(int currentPageNumber, int totalItemsCount)

{

if (currentPageNumber < MaxPageNumber(totalItemsCount))

{

return (currentPageNumber + 1);

}

else return currentPageNumber;

}

public int MaxPageNumber(int totalItemsCount)

{

int maxPageNumber;

double numberOfPages = (double)totalItemsCount / PageSize;

if (numberOfPages ==Math.Floor(numberOfPages))

{

maxPageNumber = (int)numberOfPages;

}

else

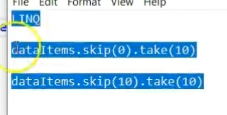
{

maxPageNumber = (int)numberOfPages + 1;

}

return maxPageNumber;

}



Señala que se estipula que hayan 10 registros por página. En la primera línea se muestran sus primeros 10, y en la segunda línea se muestran los segundos 10.

Componente

Se le pasa por parámetro el paging desde el padre.

[Parameter]

public PaginConfig Paging { get; set; }

También crea como parámetro el número de página actual, siendo por default 0.

[Parameter]

public int CurrentPageNumber { get; set; } = 1;

Luego va al foreach que muestra los datos, no el encabezado, del DG y hace esto:

@foreach (var item in DataItems.Skip(Paging.NumOfItemsToSkip(CurrentPageNumber))

.Take(Paging.NumOfItemsToTake(DataItems.Count)))

Utiliza Bulma para hacer la botonera, yo lo hará precariamente. Agrega bool CustomPager a la clase que configura la paginación.

En el componente agrega otro parámetro que es un renderFragment

[Parameter]

public RenderFragment CustomPager { get; set; }

<div>

@if (Paging!=null && Paging.Enabled)

{

@if (Paging.CustomPager)

{

@CustomPager

}

else

{

<button class="btn btn-primary" @onclick="GoToPrevPage">Previous</button>

<button class="btn btn-primary" @onclick="GoToNextPage">Next</button>

}

}

Agrega los 2 métodos pasamanos

public void GoToPrevPage()

{

CurrentPageNumber = Paging.PrevPageNumber(CurrentPageNumber);

}

public void GoToNextPage()

{

CurrentPageNumber = Paging.NextPageNumber(CurrentPageNumber, DataItems.Count);

}

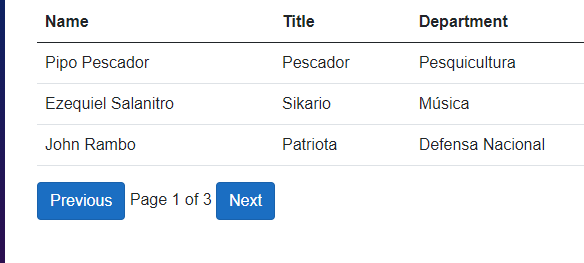
Suma esto al padre, cuando crea el componente

**Paging** =@(new PaginConfig {

Enabled=true,

CustomPager=false,

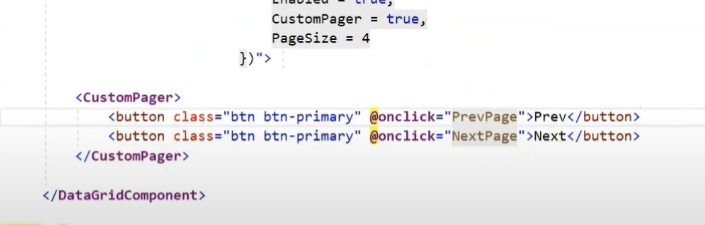
PageSize=3



Agrega eso a la botonera

<span>Page @CurrentPageNumber of @Paging.MaxPageNumber(DataItems.Count)</span>

Ahí pone el customPage en true. Señala que es algo que se le daría a un desarrollador, no a un usuario. Mas adelante codea los botones, mas adelante pone los métodos correspondientes.



DataGrid: Sorting Columns

W3Schools busca las flechitas (arrow) y copia el código Css.

.arrow {

border: solid black;

border-width: 0 3px 3px 0;

display: inline-block;

padding: 3px;

}

.right {

transform: rotate(-45deg);

-webkit-transform: rotate(-45deg);

}

.left {

transform: rotate(135deg);

-webkit-transform: rotate(135deg);

}

.up {

transform: rotate(-135deg);

-webkit-transform: rotate(-135deg);

}

.down {

transform: rotate(45deg);

-webkit-transform: rotate(45deg);

}

Crea Css en la carpeta Css. En el Host.html se agrega la referencia.

Funca perfecto. Los números no los ordena bien.

Tuve que desactivar en el Parente (index) la paginación.