Demo :

1. Attribute Binding
2. Conditional Attributes
3. Event Handling
4. Component Parameter
5. Event Callback
6. Cascading Values
7. Using @Key
8. Attribute Splatting
9. Referencing with @ref
10. Working With Chained Binds
11. Templated Components
12. Using Form Components
13. Lifecycle Methods
14. Creating Blazor ClassLibraries

**Demo : Attribute Binding**

* Bind to an HTML attribute

1. Add Class attributes in site.css

.red-background {

background: red;

color: white;

}

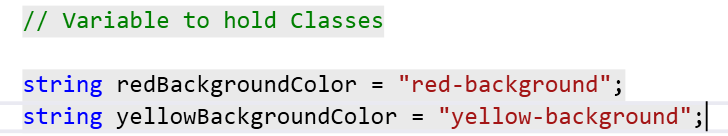
.yellow-background {

background: yellow;

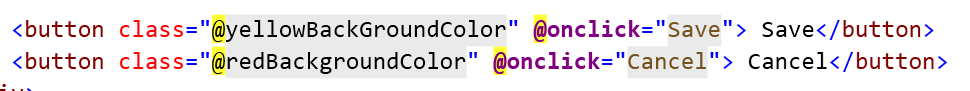
color: black;

}

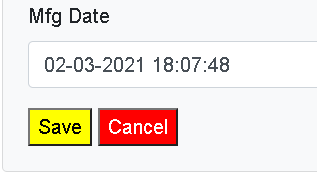
1. Add Variable to hold classes.



1. Apply it on HTML control as attribute



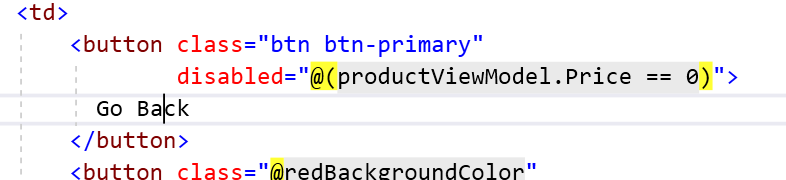
1. Check the Output



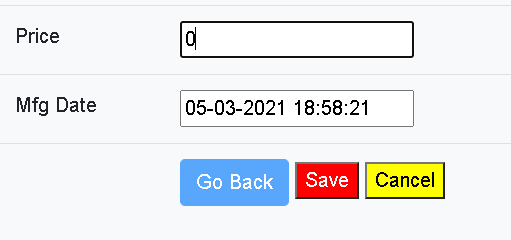
**Demo : Conditional Attribute**

* Use to control the browser by adding some attributes.

1. Add Conditional Attribute



1. Pass 0 as Price

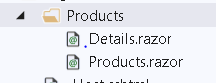


**Event Handling**

* HTML element attribute named @on**{EVENT}** with a delegate typed.
* Event Handler Can be Asynchronous and return a Task.
* To Pass a parameter you can use Lambda Expression.

**Demo : How to Perform Event Handling**

1. Add Products.Razor Component



1. Add a Add Product and Search button

<div class="card bg-light mt-3" style="width:30rem ">

<div class="card-body">

<button class="btn btn-primary" **@onclick**="AddProduct"> Add Product</button>

<span>

Search Product <input type="text" **@bind**="ProductName" />

<button class="btn btn-warning"

**@onclick**="()=>SearchProduct(ProductName)">

Search

</button>

</span>

</div>

</div>

1. Add a code for the same

@code {

public void AddProduct()

{

Console.WriteLine("Add Product Function Called");

}

public void SearchProduct(string productName)

{

Console.WriteLine(ProductName);

}

//Hold ProductName Entered

public string ProductName { get; set; }

}

1. Check the Output



Communication Between Components

Component Parameter attribute

* This is required for components that want to expose their public properties to the parent component.
* Its use to pass data from Parent to child components.
* @ChildContent will hold content and needs to be type of RenderFragment because this is way Blazor engine passes it.

1. Parent to Child : Data Transfer

Pass

Parameter

Child

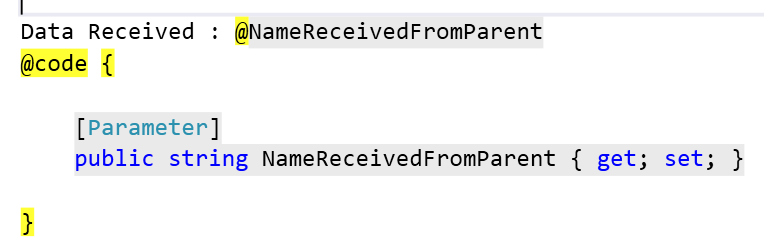
Parent

Demo : How to Pass Parameter from Parent to Child

1. Add Demo folder and add Parent , Child and GrandChild component



1. Go Child Parameter and add parameter property



1. Go to Parent , add variable and send it to Child .

@page "/parent"

<div class="card bg-light mt-3" style="width:40rem ">

<div class="card-body">

<h3>ParentComponent</h3>

Send Data to Child : <input **@bind**="NameSendToChild"

**@bind:event**="oninput" />

<div class="card bg-light mt-3" style="width:30rem ">

<div class="card-body">

<**ChildComponent** **NameReceivedFromParent**="@NameSendToChild">

</**ChildComponent**>

</div>

</div>

</div>

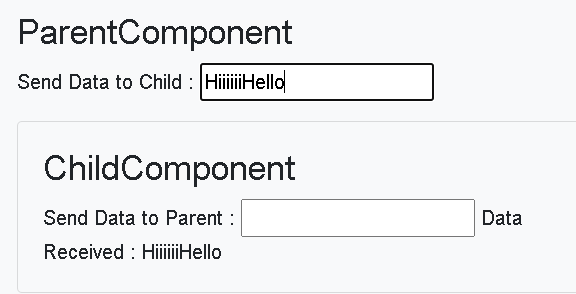
</div>

@code {

public string NameSendToChild { get; set; }

}

1. Check the Output



1. EventCallback

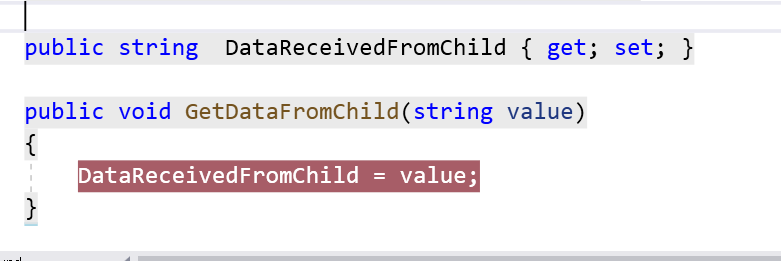
* Use to inform Parent about something has happened

Demo : EventCallback

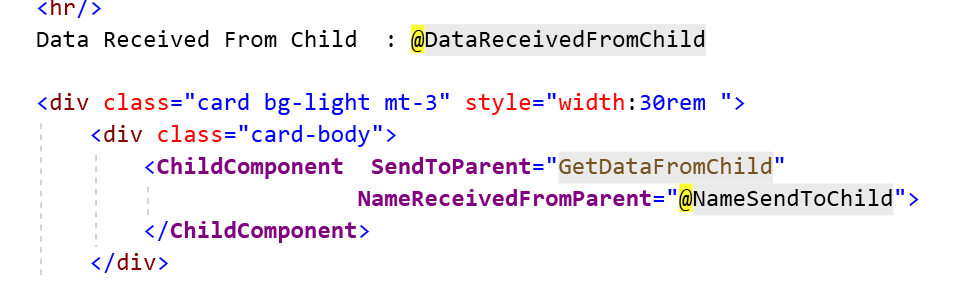
1. Add a code in child component



1. Add Code in Parent Component



1. Display data



1. Check the Output

**Chained Binds**

Two Way Data Binding : Between Parent And child Component

* It allows us to update the child component values from a parent component and vice versa.
* In order to do that we need to use @bind-{Child Property}
* We need to create EventCallback to update the value of Parent Component.

1. Add a Code in ChainedBindChild.razor

<h3>ChainedBindChild</h3>

<h3> @MainTitle</h3>

<input

**@bind**="MainTitle"

**@bind:event**="oninput"

**@onkeyup**="ModifyMainTitle"

/>

@code {

[Parameter]

public string MainTitle { get; set; }

[Parameter]

public EventCallback<string> MainTitleChanged { get; set; }

private async Task ModifyMainTitle()

{

await MainTitleChanged.InvokeAsync(MainTitle);

}

}

1. Add a code in ChainedBindParent.razor

@page "/chainedparent"

<h3>ChainedBindParent</h3>

<h3> @Title</h3>

<input **@bind**="Title"

**@bind:event**="oninput" />

<**ChainedBindChild**

**@bind-MainTitle:event**="MainTitleChanged"

**@bind-MainTitle**="Title" />

@code {

public string Title { get; set; } = "Two Way Data Binding";

}

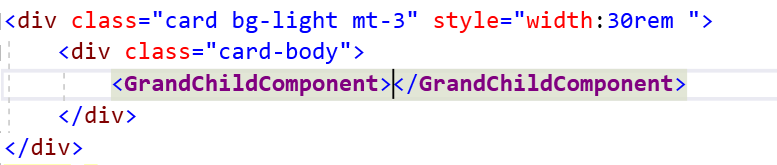
1. Check the Output

Cascading Values

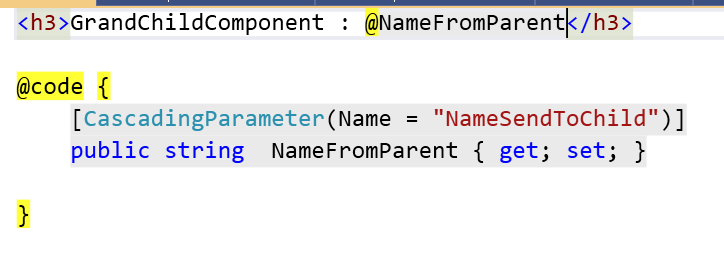
* Its use to pass values from Parent to component to any leaf node

**Demo : Cascading Values**

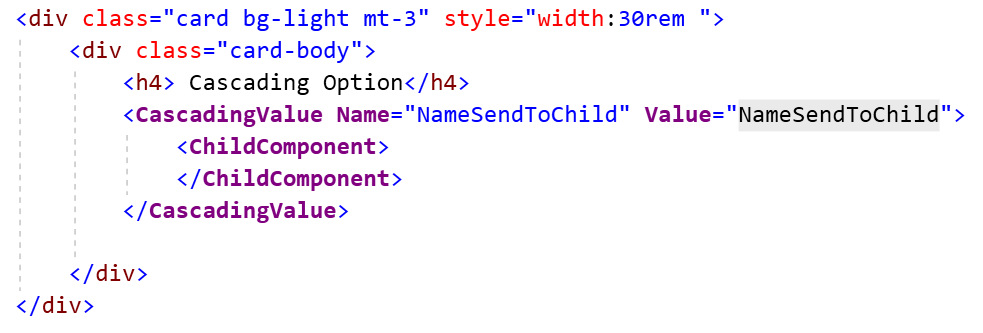
1. Add a GrandChildComponent in Child Component



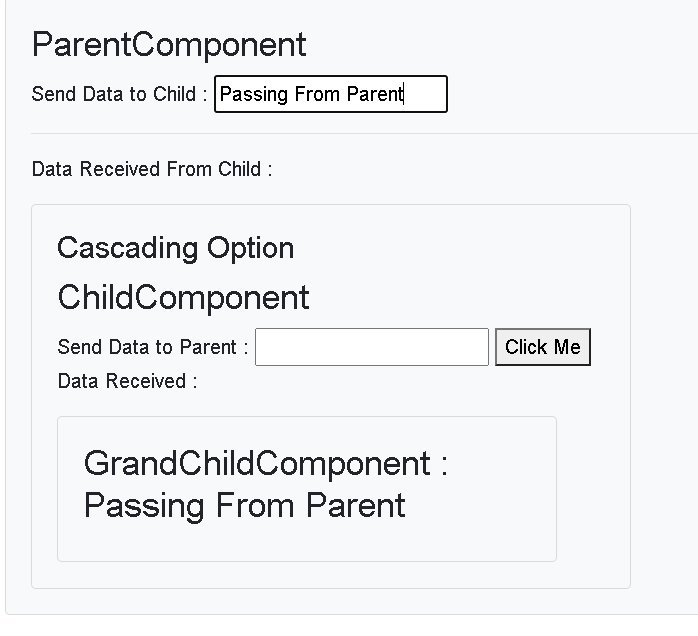
1. Pass Data using Cascading Valus option to descendent and Lead one.



1. Pass the data using Cascading option



1. Check the Output



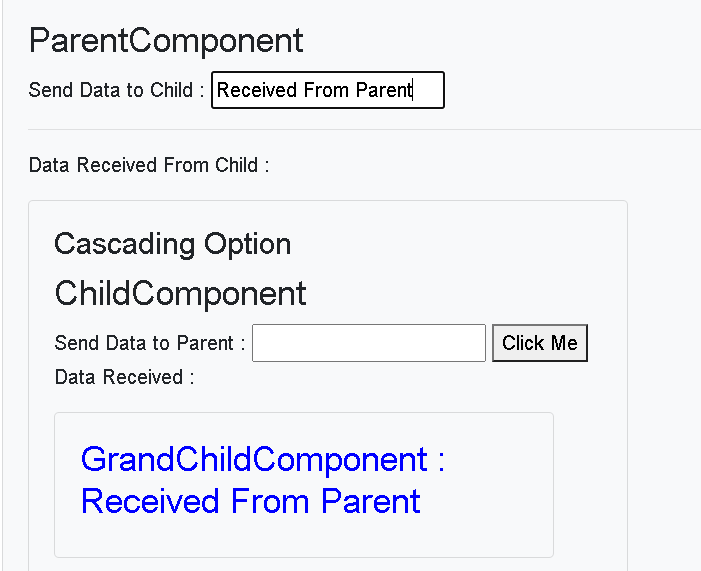
1. To Pass multiple parameters



1. Pass this TitleColor from the Parent



1. Check the Output



1. @ref
2. Add Modal Child Component and add following code

<h3>ModelChild</h3>

@code {

public void ModalChildFunction()

{

Console.WriteLine("Modal Child Function Called");

}

}

1. Call this function from Parent

private ModelChild dialog;

public void ParentFunction()

{

dialog.ModalChildFunction();

}

1. Add a code to call the function from Child

<**ModelChild** **@ref**="dialog"></**ModelChild**>

Communication between 2 Components

1. Add a Code in Parent Component

@inject NavigationManager Navigate

<h4> SetParameterAsync : </h4>

<label> FirstName</label>

<input **@bind**="FirstName" />

<button **@onclick**="GotoSearchPage"> Go</button>

1. Add a Method for the same

@code {

string FirstName = "Sachin";

public void GotoSearchPage()

{

Navigate.NavigateTo($"parent2/{FirstName}");

}

1. Go to Parent2 and add a code to read data from Route Parameters

@page "/parent2/{FirstName}"

<h3>Parent2 : @FirstName </h3>

@code {

[Parameter]

public string FirstName { get; set; }

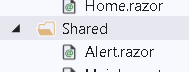
}

1. Check the Output

**Child Content**

* It holds the Content
* Type of RenderFragment

1. Add Alert.Razor



1. Add Following code into it

<h3>Alert</h3>

<div class="alert alert-secondary alert-dismissable fade show mt-4"

role="alert">

@ChildContent

</div>

@code {

[Parameter]

public RenderFragment ChildContent { get; set; }

}

1. Call Child Component and pass HTML that needs to be rendered in Home.razor

<**Alert**>

<span class="oi oi-check mr-2" aria-hidden="true"></span>

<strong>Data is Deleted Successfully</strong>

</**Alert**>

1. Check the Output
2. Add a Show Variable and Dismiss function

@code {

[Parameter]

public bool Show { get; set; }

public void Dismiss() => Show = false;

1. Add Conditional Code to show the data

@if (Show)

{

<div class="alert alert-secondary alert-dismissable fade show mt-4"

role="alert">

@ChildContent

<button type="button" class="close" data-dismiss="alert"

aria-label="Close" **@onclick**="@Dismiss">

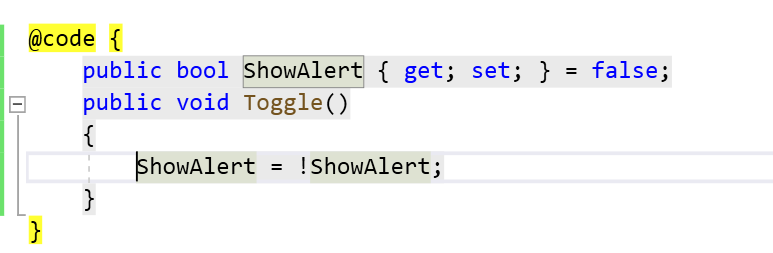
<span aria-hidden="true">&times;</span>

</button>

</div>

}

1. Add Showalert variable and Method to toggle the value in home.razor.



1. Pass this variable to Alert component

<button **@onclick**="Toggle">Toggle</button>

<**Alert** **Show**="ShowAlert">

<span class="oi oi-check mr-2" aria-hidden="true"></span>

<strong>Data is Deleted Successfully</strong>

</**Alert**>

1. Check the Output

Using @Key

* While rendering a list of element or components , blazor diff algorithm must decide which of the previous elements or components can be retained .
* Normally it works fine but some time when collection content changed by insertion, deletion or during re-ordering entries.
* This might cause more complex rerendering than expected
* The mapping process can be controller with @ key directive attribute
* @key causes diffing algorithm to guarantee preservation of elements based on the key value.
* use [@key](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-5.0#key) whenever a list is rendered
* Use primary key kind of thing for the key.

1. Add New Component



1. Create an List Of Product and array it



1. Display data

@page "/list"

<ul>

@foreach (var item in productViewModels)

{

<li **@key**="@item.ProductID">@item.Name</li>

}

</ul>

1. Check the Output

======================================================

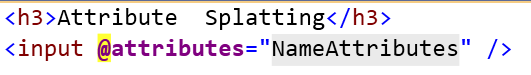
Attribute Splatting

* Useful when defining component that produces a markup element that support variety of customizations.
* Attributes can be captured in dictionary and then splatted on to element when the component is rendered using @attributes Razor Directive

1. Add New Component Splatting
2. Add following code into that



1. Apply attributes to component



1. Check the Output

**Templated Component**

Demo : Template Component

1. Add ListviewTemplate.razor

@\*Directive to specify type parameters\*@

@typeparam TItem

<ul>

@foreach (var item in Items)

{

@ItemTemplate(item)

}

</ul>

@code {

[Parameter]

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

[Parameter]

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

}

1. Use this in Product.razor

<**testblazorserver.Templates.ListViewTemplate** **Items**="products">

<**ItemTemplate** **Context**="product">

<li>@product.Name</li>

</**ItemTemplate**>

</**testblazorserver.Templates.ListViewTemplate**>

1. Check the Output
2. Add TableViewTemplate.razor

@typeparam TItem

<table class="table">

<thead> <tr> @TableHeader</tr></thead>

<tbody>

@foreach (var item in Items)

{

<tr>@RowTemplate(item)</tr>

}

</tbody>

</table>

@code {

[Parameter]

public RenderFragment TableHeader { get; set; }

[Parameter]

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

[Parameter]

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

}

1. Add this in product.razor

<**testblazorserver.Templates.TableViewTemplate** **Items**="products">

<**TableHeader**>

<th>Name</th>

<th>Description</th>

<th>Price</th>

<th>Mfg Date</th>

<th>Category</th>

<th></th>

</**TableHeader**>

<**RowTemplate**>

<td>@context.Name</td>

<td>@context.Description</td>

<td>@context.Price</td>

<td>@context.MfgDate</td>

<td>@context.Category</td>

</**RowTemplate**>

</**testblazorserver.Templates.TableViewTemplate**>

1. Check the output

**Form Components**

Using Blazor Form Components

* Provides built in components that are used to render form elements.
* Ensures that server side component properties are updated after user interaction and integrating validation.

1. EditForm

* Renders Form Element
* Wired up with data validation

1. InputText

* Renders an Input Element and bound to C# string property

1. InputCheckbox
2. InputDate
3. InputNumber
4. InputTextArea

Note : EditForm Component must be used for any of the other components to work.

Demo : How to use EditFormComponent

1. Add NewProduct.razor component
2. Add New Object

@code {

// Create an object

ProductViewModel ProductInfo = new ProductViewModel

{

ProductID = 1,

Name = "IPhone",

Category = "Electronics",

Description = "IPhone",

MfgDate = DateTime.Now,

Price = 20.3M

};

}

1. Display data using binding

<div class="card bg-light mt-3" style="width:30rem ">

<div class="card-body">

<**EditForm** **Model**="ProductInfo" >

<div class="form-group">

<label>Product ID</label>

<**InputNumber** class="form-control"

**@bind-Value**="ProductInfo.ProductID" />

</div>

<div class="form-group">

<label>Name</label>

<**InputText** class="form-control" **@bind-Value**="ProductInfo.Name" />

</div>

<div class="form-group">

<label>Price</label>

<**InputNumber** class="form-control" **@bind-Value**="ProductInfo.Price" />

</div>

<div class="form-group">

<label>Description</label>

<**InputText** class="form-control" **@bind-Value**="ProductInfo.Description" />

</div>

<div class="form-group">

<label>Mfg Date</label>

<**InputDate** class="form-control" **@bind-Value**="ProductInfo.MfgDate" />

</div>

</**EditForm**>

@\*<button class="btn btn-primary" @onclick="SaveProduct"> Save</button>\*@

</div>

</div>

Validating Form Data

* Blazor provides components that perform validation using the standard attributes

1. DataAnnotationsValidator

* Integrates Validation attributes applied to the model class

1. Validation Message : display validation error message for a single property.
2. Validation Summary : Error message for the entire model object

Validation components generate elements assigned to classes

1. Validation errors :
2. Validation-message
3. The ValidationMessage attribute is configured using the For attribute, which receives a function that returns the property the component represents.

Demo

1. Add Validation classes in Site.css

.validation-errors {

background-color: rgb(220, 53, 69);

color: white;

padding: 8px;

text-align: center;

font-size: 16px;

font-weight: 500;

}

div.validation-message {

color: rgb(220, 53, 69);

font-weight: 500

}

.modified.valid {

border: solid 3px rgb(40, 167, 69);

}

.modified.invalid {

border: solid 3px rgb(220, 53, 69);

}

1. Add Validation code : Integration with html

<**EditForm** **Model**="ProductInfo" **OnValidSubmit**="SaveProduct"

>

<**DataAnnotationsValidator** />

@\*<ValidationSummary/>\*@

<div class="form-group">

<label>Product ID</label>

<**ValidationMessage** **For**="@(()=>ProductInfo.ProductID)" />

<**InputNumber** class="form-control"

**@bind-Value**="ProductInfo.ProductID" />

</div>

<div class="form-group">

<label>Name</label>

<**ValidationMessage** **For**="@(()=>ProductInfo.Name)" />

<**InputText** class="form-control" **@bind-Value**="ProductInfo.Name" />

</div>

<div class="form-group">

<label>Price</label>

<**ValidationMessage** **For**="@(()=>ProductInfo.Price)" />

<**InputNumber** class="form-control" **@bind-Value**="ProductInfo.Price" />

</div>

<div class="form-group">

<label>Description</label>

<**ValidationMessage** **For**="@(()=>ProductInfo.Description)" />

<**InputText** class="form-control" **@bind-Value**="ProductInfo.Description" />

</div>

<div class="form-group">

<label>Mfg Date</label>

<**ValidationMessage** **For**="@(()=>ProductInfo.MfgDate)" />

<**InputDate** class="form-control" **@bind-Value**="ProductInfo.MfgDate" />

</div>

<button class="btn btn-primary" type="submit"> Save</button>

</**EditForm**>

1. Add annotations

public class ProductViewModel

{

[Required]

public long? ProductID { get; set; }

[Required]

public string Name { get; set; }

[Required]

public string Description { get; set; }

[Required]

public decimal? Price { get; set; }

[Required]

public string Category { get; set; }

public DateTime MfgDate { get; set; }

}

1. Check the Output

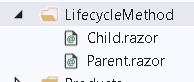
Lifecycle Methods

1. SetParameterAsync

* First method to invoked
* Called before any parameter value is assigned to their respective properties.
* Also called each time new or updated parameters are received
* Useful for Parameter validation

Demo :

1. Add New Component Parent and child



1. Add Code to child component

<h3>Child</h3>

<h4> SetParameter Async</h4>

<p> @FirstNameFromParent</p>

@code {

[Parameter]

public string FirstNameFromParent { get; set; }

public override Task SetParametersAsync(ParameterView parameters)

{

foreach (var item in parameters)

{

Console.WriteLine(item.Name + " " + item.Value);

}

return base.SetParametersAsync(parameters);

}

}

1. Pass Data from Parent Component

@page "/parent"

<h3>Parent</h3>

<h4> SetParameterAsync : </h4>

<label> FirstName</label>

<input **@bind**="FirstName" />

<**Child** **FirstNameFromParent**="@FirstName">

</**Child**>

@code {

string FirstName = "Sachin";

}

1. OnInit or OnInitAsync

* This methods execute when the component is ready to start
* In this , initial parameters are available in their respective properties.
* Invoked once after OnInit is finished. Any operations that require the component to rerender should be placed here.

Demo : OnInit and OnInitAsync

1. Add Title variale in child component and methods

// Initialization

string Title = String.Empty;

// OnInit

protected override void OnInitialized()

{

Title = "Title is Empty";

base.OnInitialized();

}

protected override async Task OnInitializedAsync()

{

await Task.Delay(3000);

Title = "Update Title";

}

1. Check the Output .. Data will update after 3 sec
2. OnParameterSet and OnParameterSetAsync

* By the time this methods are called , all parameters have been assigned to their respective properties.
* They both get executed for the first time on initialiation and each time parameter values updated by the parent.
* Invoked After SetParametersAsync and OnInit are finished and then each time new or updated parameters are received.

1. Go to child component and add following code

// OnParameter Set

[Parameter]

public int RandNumber { get; set; } = 0;

int count = 0;

int randNumWithDelay = 0;

protected override void OnParametersSet()

{

count++;

}

protected override async Task OnParametersSetAsync()

{

await Task.Delay(2000);

randNumWithDelay = RandNumber;

//return base.OnParametersSetAsync();

}

1. Display data using table in child component

<h4> On Parameter Set</h4>

<table class="table">

<tr>

<th> Parameter Set call</th>

<th>Random No</th>

<th> RandomNo with Delay</th>

</tr>

<tr>

<td>@count</td>

<td>@RandNumber</td>

<td>@randNumWithDelay</td>

</tr>

</table>

1. Generate Random number and pass it to Child from Parent . Make the following code changes in Parent.

<**Child**

**FirstNameFromParent**="@FirstName"

**RandNumber**="@randomNumber">

</**Child**>

<button **@onclick**="RandomNumberGen" class="btn btn-primary">

Generate Random Number</button>

@code {

string FirstName = "Sachin";

// OnParameter Set

int randomNumber = 0;

Random random = new Random();

public void RandomNumberGen()

{

randomNumber = random.Next(100);

}

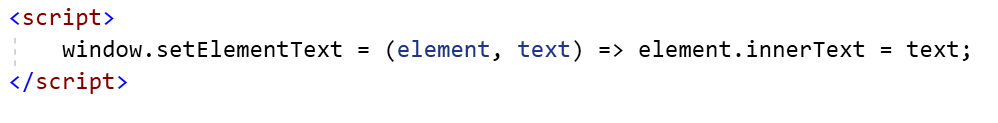
}

1. Check the Output:

OnAfterRender and OnAfterRenderAsync

* These Methods are essential when you need to perform additional initialization steps using rendered content . such as initializaing JS libraries.
* Use this stage to perform additional initialization steps using the rendered content, such as activating third-party JavaScript libraries that operate on the rendered DOM elements.

1. Add JS code in \_Host.cshtml



1. Add New Component and add a code to call this JS library function

@page "/micro-modal"

@using Microsoft.JSInterop

@inject IJSRuntime JSRuntime

<h2>Micro Modal</h2>

<div **@ref**="divElement"> Text During Render</div>

@code {

int count = 0;

private ElementReference divElement;

protected override void OnAfterRender(bool firstRender)

{

count++;

Console.WriteLine($"OnAfterRender execution count: {count}");

}

protected override async Task OnAfterRenderAsync(bool firstRender)

{

await Task.Delay(3000);

await JSRuntime.InvokeVoidAsync(

"setElementText", divElement, "Text after render");

}

}

ShouldRender

* Method returns a Boolean value
* If the value is true , the UI is re-rendered
* Its used to suppress subsequent rendering of the component.

For Example.

1. We have simple counter component is used to trigger UI refresh when the value count changes
2. When the ShouldRender method gets invoked , we can use suppress further renderting.
3. Add following code in Parent.razor

// Should Render

bool shouldRender = true;

int count = 0;

public void TurnOff()

{

shouldRender = false;

}

public void TurnOn()

{

shouldRender = true;

}

public void IncrementCount()

{

count++;

}

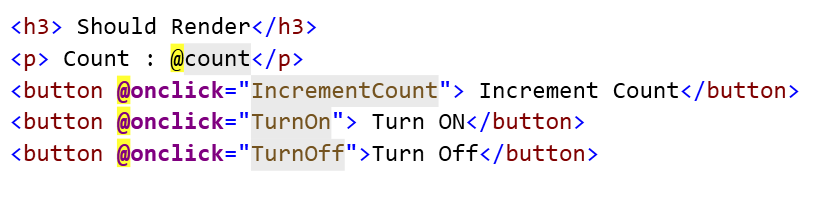
protected override bool ShouldRender()

{

return shouldRender;

}

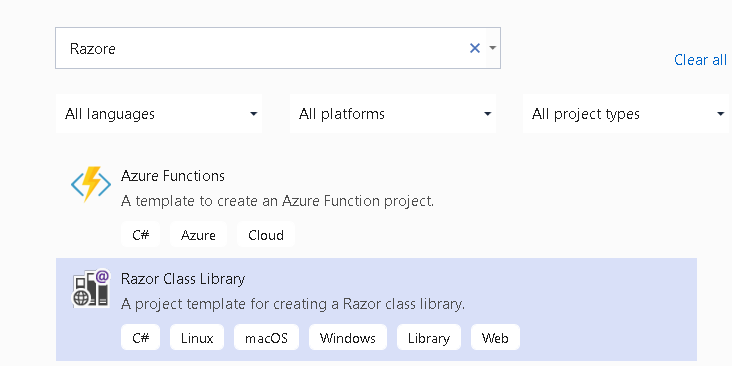
1. Add a Display logic



1. Check the Output . updates will work only if ShouldRender is On

Building Razor Library

1. Create new Project



1. Add a Ref to Current Project where you to want to use it



1. Use the Component



1. Check the Output

**Building Project**

1. Add New Folder Data
2. Add IProductRepository Interface to it

public interface IProductRepository

{

Task<List<Product>> GetProducts();

Task<Product> GetProduct(long? id);

Task<List<Product>> GetByCategories(string Category);

Task<Product> GetProductByName(string productName);

Task UpdateProduct(Product product);

}

1. Add ProductDBContext

public class ProductDBContext:DbContext

{

public ProductDBContext(DbContextOptions options) :base(options)

{

}

public DbSet<Product> Products { get; set; }

}

1. Add ConnectionString

"ConnectionStrings": {

"ProductConnection": "Data Source=SWAPNIL-PC\\SQLEXPRESS;Initial Catalog=BOATest;Integrated Security=true"

}

1. Implement Interface in ProductRepository

public class ProductRepository : IProductRepository

{

private readonly ProductDBContext \_context;

public ProductRepository(ProductDBContext context)

{

\_context = context;

}

public async Task UpdateProduct(Product product)

{

\_context.Products.Update(product);

await \_context.SaveChangesAsync();

}

public Task<List<Product>> GetByCategories(string Category)

{

var query = from data in \_context.Products

where data.Category == Category

select data;

List<Product> products = new List<Product>();

foreach (var item in query)

{

products.Add(item);

}

return Task.FromResult(products);

}

public async Task<Product> GetProduct(long? id)

{

return await \_context.Products.FindAsync(id);

}

public async Task<Product> GetProductByName(string productName)

{

return await Task

.FromResult(\_context.Products.

Single(x => x.Name == productName));

}

public Task<List<Product>> GetProducts()

{

return Task.FromResult(\_context.Products.ToList());

}

}

1. Configure Service with DbContext

services.AddDbContext<ProductDBContext>(options =>

{

options.UseSqlServer(Configuration["ConnectionStrings:ProductConnection"]);

});

services.AddScoped<IProductRepository, ProductRepository>();

1. Inject it into Products.Razor

@page "/products"

@inject IProductRepository \_repository

1. Add code in the @code section of products.razor

public List<Product> products { get; set; }

protected override async Task OnInitializedAsync()

{

await this.GetProducts();

}

public async Task GetProducts()

{

products =await \_repository.GetProducts();

}

1. Display data in products.razor

<table class="table table-bordered">

<thead>

<tr>

<th>Name</th>

<th>Price</th>

<th>Category</th>

</tr>

</thead>

<tbody>

@foreach (var product in products)

{

<tr>

<td>@product.Name</td>

<td>@product.Price</td>

<td>@product.Category</td>

</tr>

}

</tbody>

</table>

1. Check the Output

How to Create Details Component that is accepting parameters.

1. Add New Razor Component



1. Add a Following code in ProductDetails.Razor

<h3>Product Details</h3>

<table class="table table-bordered">

<tr>

<td>ID</td>

<td>@Product.ProductID</td>

</tr>

<tr>

<td>Name</td>

<td>@Product.Name </td>

</tr>

<tr>

<td>Price</td>

<td>@Product.Price </td>

</tr>

<tr>

<td>Category</td>

<td>@Product.Category </td>

</tr>

<tr>

<td>Description</td>

<td>@Product.Description </td>

</tr>

</table>

@code {

[Parameter]

public Product Product { get; set; }

}

1. Go to Products.razor and add the code

public Product product { get; set; }

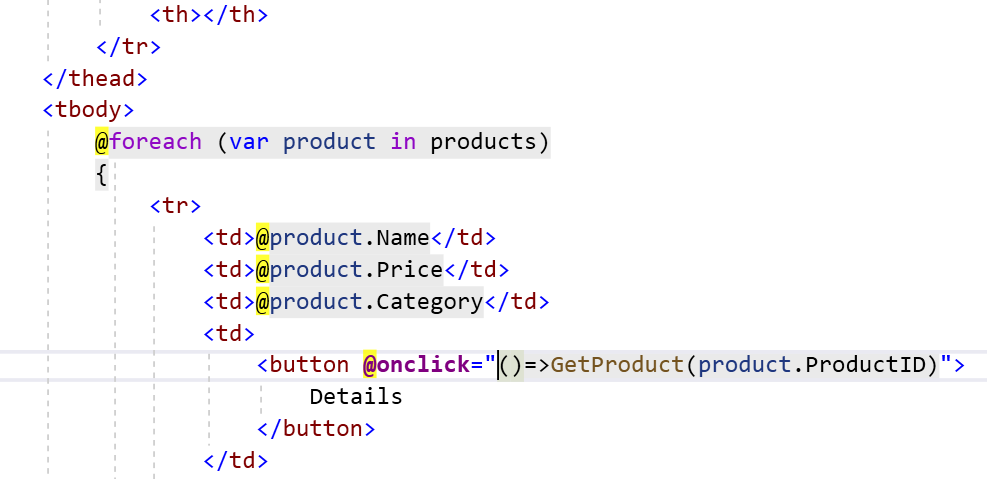
public async Task GetProduct(long? id)

{

product = await \_repository.GetProduct(id);

}

1. Add a Event code to call above functionality



1. Add a code to display productdetails component when the product is not null.



1. Check the Output

How to use Routing to Search and Display the data.

Problem : We want to pass a data that want to

1. Add a code to Search by category

public string SelectedCategory { get; set; }

public void GotoSearchPage()

{

if (SelectedCategory != null)

{

NavigationManager.NavigateTo($"search/{SelectedCategory}");

}

}

1. Add a Navigation Manager in products.razor

@inject NavigationManager NavigationManager

1. Iterate over Categories and add button

<select **@bind**="SelectedCategory">

<option value="">Select</option>

@foreach (var p in products)

{

<option value="@p.Category">@p.Category</option>

}

</select>

<button **@onclick**="GotoSearchPage">Search</button>

1. Add a code for the searching by category
2. Modify IProductRepository

Task<List<Product>> GetByCategories(string Category);

1. Implement IProductRepository

public Task<List<Product>> GetByCategories(string Category)

{

var query = from data in \_context.Products

where data.Category == Category

select data;

List<Product> products = new List<Product>();

foreach (var item in query)

{

products.Add(item);

}

return Task.FromResult(products);

}

1. Use it in Search Page

@code {

[Parameter]

public string Category { get; set; }

public List<Product> products { get; set; }

protected override async Task OnInitializedAsync()

{

products = await \_repository.GetByCategories(Category);

}

public void GoBack()

{

NavigationManager.NavigateTo("products");

}

}

1. Display the data

@page "/search/{Category}"

@inject IProductRepository \_repository

@inject NavigationManager NavigationManager

@if (!String.IsNullOrEmpty(Category))

{

<table class="table table-bordered">

<thead> <tr> <td>Name</td><td>Price</td><td> Description</td></tr></thead>

<tbody>

@foreach (var p in products)

{

<tr><td>@p.Name</td><td>@p.Price</td><td>@p.Category</td></tr>

}

</tbody>

</table>

<button **@onclick**="GoBack"> Go Back</button>

}

else

{

<h4>Category Not Provided</h4>

}

1. Check the Output