200 XP

Share data in Blazor applications

7 minutes

Blazor includes several ways to share information between components. You can use component parameters or cascading parameters to send values from a parent component to a child component. The AppState pattern is another approach you can use to store values and access them from any component in the application.

Suppose you're working on the new pizza delivery website. Multiple pizzas should be displayed on the home page in the same way. You want to display the pizzas by rendering a child component for each pizza. Now, you want to pass an ID to that child component that determines the pizza it will display. You also want to store and display a value on multiple components that shows the total number of pizzas you've sold today.

In this unit, you'll learn three different techniques you can use to share values between two or more Blazor components.

Sharing information with other components by using component parameters

In a Blazor web app, each component renders a portion of HTML. Some components render a complete page but others render smaller fragments of markup, such as a table, a form, or a single control. If your component renders only a section of markup, you must use it as a child component within a parent component. Your child component can also be parent to other, smaller components that render within it. Child components are also known as nested components.

In this hierarchy of parent and child components, you can share information between them by using *component parameters*. Define these parameters on child components, and then set their values in the parent. For example, if you have a child component that displays pizza photos, you could use a component parameter to pass the pizza ID. The child component looks up the pizza from the ID and obtains pictures and other data. If you want to display many different pizzas, you can use this child component multiple times on the same parent page, passing a different ID to each child.

You start by defining the component parameter in the child component. It's defined as a C# public property and decorated with the `[Parameter]` attribute:

razor

```razor
<h2>New Pizza: @PizzaName</h2>

<p>@PizzaDescription</p>

@code {
    [Parameter]
    public string PizzaName { get; set; }

    [Parameter]
    public string PizzaDescription { get; set; } = "The best pizza you've ever tasted."
}
```

Because the component parameters are members of the child component, you can render them in your HTML by using Blazor's reserved `@` symbol, followed by their name. Also, the preceding code specifies a default value for the `PizzaDescription`

parameter. This value is rendered if the parent component doesn't pass a value. Otherwise, it's overridden by the value passed from the parent.

You can also use custom classes in your project as component parameters. Consider this class that describes a topping:

C#

```
public class PizzaTopping
{
    public string Name { get; set; }
    public string Ingredients { get; set; }
}
```

You can use that as a component parameter in the same way as a parameter value to access individual properties of the class by using dot syntax:

razor

```
<h2>New Topping: @Topping.Name</h2>

<p>Ingredients: @Topping.Ingredients</p>

@code {
    [Parameter]
    public PizzaTopping Topping { get; set; }
}
```

In the parent component, you set parameter values by using attributes of the child component's tags. You set simple components directly. With a parameter based on a custom class, you use inline C# code to create a new instance of that class and set its values:

razor

```
@page "/pizzas-toppings"

<h1>Our Latest Pizzas and Topping</h1>

<Pizza PizzaName="Hawaiian" PizzaDescription="The one with pineapple" />

<PizzaTopping Topping="@(new PizzaTopping() { Name = "Chilli Sauce", Ingredients = "Three kinds of chilli." })"
/>
```

Share information by using cascading parameters

Component parameters work well when you want to pass a value to the immediate child of a component. Things become awkward when you have a deep hierarchy with children of children and so on. Component parameters aren't automatically passed to grandchild components from ancestor components or further down the hierarchy. To handle this problem elegantly, Blazor includes cascading parameters. When you set the value of a cascading parameter in a component, its value is automatically available to all descendant components to any depth.

In the parent component, using the `<CascadingValue>` tag specifies the information that will cascade to all descendants. This tag is implemented as a built-in Blazor component. Any component that's rendered within that tag is able to access the value.

razor

```razor
@page "/specialoffers"

<h1>Special Offers</h1>

<CascadingValue Name="DealName" Value="Throwback Thursday">
    <!-- Any descendant component rendered here will be able to access the cascading value. -->
</CascadingValue>
```

In the descendant components, you can access the cascading value by using component members and decorating them with the `[CascadingParameter]` attribute.

razor

```razor
<h2>Deal: @DealName</h2>

@code {
    [CascadingParameter(Name="DealName")]
    private string DealName { get; set; }
}
```

So in this example, the `<h2>` tag has the content `Deal: Throwback Thursday` because that cascading value was set by an ancestor component.

> **Note**
>
> As for component parameters, you can pass objects as cascading parameters if you have more complex requirements.

In the preceding example, the cascading value is identified by the `Name` attribute in the parent, which is matched to the `Name` value in the `[CascadingParameter]` attribute. You can optionally omit these names, in which case the attributes are matched by type. Omitting the name works well when you have only one parameter of that type. If you want to cascade two different string values, you must use parameter names to avoid any ambiguity.

Share information by using AppState

Another approach to sharing information between different components is to use the AppState pattern. You create a class that defines the properties you want to store and register it as a scoped service. In any component where you want to set or use the AppState values, you inject the service and then you can access its properties. Unlike component parameters and cascading parameters, values in AppState are available to all components in the application, even components that aren't children of the component that stored the value.

As an example, consider this class that stores a value about sales:

C#

```csharp
public class PizzaSalesState
{
    public int PizzasSoldToday { get; set; }
}
```

You would add the class as a scoped service in the **Program.cs** file:

C#

```
...
// Add services to the container
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();

// Add the AppState class
builder.Services.AddScoped<PizzaSalesState>();
...
```

Now, in any component where you want to set or retrieve AppState values, you can inject the class and then access properties:

razor

```
@page "/"
@inject PizzaSalesState SalesState

<h1>Welcome to Blazing Pizzas</h1>

<p>Today, we've sold this many pizzas: @SalesState.PizzasSoldToday</p>

<button @onclick="IncrementSales">Buy a Pizza</button>

@code {
    private void IncrementSales()
    {
        SalesState.PizzasSoldToday++;
    }
}
```

> ⓘ Note
>
> This code implements a counter that increments when the user selects a button, much like the example in the Blazor Tutorial - Build your first Blazor app. The difference is that in this case, because we've stored the counter's value in an AppState scoped service, the count persists across page loads and can be seen by other users.

In the next unit, you'll try it yourself!

Check your knowledge

1. Blazor components can receive input by using two types of parameters. What are they? *

- ○ Parameter and DescendingParameter
- ○ Parameter and RequiredParameter
- ◉ Parameter and CascadingParameter

✔ **Parameters pass data directly into a component, and CascadingParameter passes data indirectly into a child component.**

2. AppState can be registered with the service locator by using which of these statements? *

- ( ) `builder.Services.AddAppState()`
- ( ) `builder.Services.AddServerSideBlazor()`
- (•) `builder.Services.AddScoped<PizzaSalesState>()`

✔ This method registers the PizzaSalesState object as a scoped service that can be used within Blazor pages and components.

Next unit: Exercise - Share data in Blazor applications

Continue >