Part 1: quick intro to ASP.NET Core Blazor

ASP.NET Core Blazor is a framework for building web UI with .NET. These UI elements are referred to as components in Blazor apps. These components are written in Razor syntax, which is a combination of HTML and C# code. They have the .razor file extension. Blazor has many built-in components and supports custom component. However, instead of talking about the components, the presentation will be focus on some of the techniques used to create the components and connect them to each other.

Note for self: part 2 requires a prototype that implements the following features:

* One way data binding
* Two-way data binding (via the modifier)
* Data binding with <select> element
* Binding with component parameters

Part 2: data binding

Data binding is a technique that allows the developer to bind the data from a data source, like a database, to an UI element for display, for example, a table in HTML. In Blazor, data binding is a built-In feature. This allows the us the developer to separate the concerns for the data source and the UI, results in a simpler and more maintainable application. For the most basic one-way data binding, simply prefix a data item with the @ character in a Razor component, like this:

The time is @DateTime.Now.

In this small example, the value can change dynamically, but it cannot be modified by the user directly. Therefore, one-way data binding’s usefulness is limited. A far more common way of data binding is two-way data binding. In Blazor, it is achieved by using the @bind:get and @bind:set modifier.

[INSERT PROTOTYPE SCREENSHOT HERE]

The @bind: set modifier allows us to control the underlying value, whereas the @bind:get binds the value to the corresponding UI element.

By its very definition, two-way binding involves user input. However, what if the user provides an unparsable value to a databound item? For example, the user provides a decimal value to an integer item. Depends on your implementation of the input textbox, how Blazor will resolve this situation varies. By default, meaning if you did not specify on which event should the binding applies, the value will be reverted after the user changes the focus. However, if you have bound the event to “oninput”, the user will not be able to type illegal characters into the input textbox. In our example, this means the user will not be able type the “.” character.

When binding a date, you can also provide a format string using the @bind: format modifier. As of the writing, this is the only format string supported by Blazor. This means, if you want to use format string on other data types, such as decimal, you need to create a custom C# get and set accessors.

Data binding also supports multiple option selection with <select> elements. However, there are some limitations to this. First, the value must be bound to an array type. Second, there is no way to bind the <select> element options to C# object null values, since HTML attributes can’t have null values. If the user selects an option with no value, this value will be treated as the text content of that option.

You can also bind the property of a child component to its parent component. This scenario is also known as chained bind, since multiple levels of binding occur simultaneously. To implement chained binding, an event handler is needed for updating the child component’s property from the parent component. In the parent component, we can set up chained binding using the @bind-{PROPERTY} syntax. You can’t implement chained bind using the @bind syntax in the child component, that is why the event handler is necessary.

As we all know, Blazor components can be nested. We can bind properties through any number of nested components, but you must respect the one-way flow of data. First, the change notifications flow up the hierarchy; second, the new parameter values flow down the hierarchy. To avoid confusion about what state must be updated, it is recommended to store the underlying data in the parent component.

Note for self: this part requires a prototype with the following features:

* Component parameter
* Component parameter overwritten by parent component(?)

Part 3: Component parameters

Since we have just talked about parent and child components and how to bind data through them as a way of passing data between them, now is a good time to talk about component parameters. Component parameters is one of the several ways in Blazor used to pass data between components. To start off, mark the property that you want to use as a parameter with the [Parameter] attribute, so that it will accessible to the parent component. Then, in order to give the parameter a value, reference the child component and the parameter in the parameter component.

This process sounds straight forward, but we need to keep in mind that we can accidently overwriting parameters and its set accessor, which can result in unintended behavior from the component. The following practices may result in the child’s component parameter being overwritten:

* Allowing the child component to write to the parameter directly
* The child component in question has one or more data-bound parameters
* Re-rendering the parent component, which will result in the child’s parameter being overwritten and loses its current state.

As a general guide line, do not create components that directly write to their own parameters after the component is rendered for the first time. You can provide default values to these parameters.

Note for self: this part requires a prototype that implement the following features:

* A single pair of Cascading value and parameter
* Multiple cascading values and parameters of the same type

Part 4: cascading values and parameters

Cascading is a convenient way to flow data down a component hierarchy from a parent component to any number of child components. It is very similar to component parameter, but do not confuse this with component parameters, cascading values and parameters don’t require an attribute assignment for each descendent component where the data is consumed. For example, let’s say you have a nested component with a length of 4 and you want to pass some data from the parent component all the way to the last child component. If you are using component parameter, you have to have a reference of that parameter in every single component which very inefficient. However, if we are using cascading value and parameter, only the one component that is going to consume the data needs to have a reference of it.

To start cascading, first wrap value and parameters you want to cascade in the <CascadingValue> element. For any child components that want to use this value, declare a cascading parameter using the [CascadingParameter] attribute. By default, the matching between the value and the parameter is done by type. If you need to cascade multiple values of the same type, you need to provide a unique name string to each <CascadingValue> elements and their corresponding [CascadingParameter] attributes.

You can also pass data across a component hierarchy using this feature by turning a component into a cascading value.

Part 5: Demo

3 demos, with features listed above

This concludes my presentation for today. Thank you for your time and please feel free to ask any questions.