Blazor Server

Component Model

2023

Component life cycle methods

- The fundamental building blocks of Blazor applications are components.
- Components can also contain other components. They encapsulate any data that a piece of UI requires to function.
- The data a component contains is more commonly referred to as its state.
- Methods on a component define its logic. These methods manipulate and control that state via the handling of events.

Structuring Components

- Single File
  - When using a single file approach, unsurprisingly, all markup and logic for a component is defined in a single file
  - The primary advantage of this approach is that it allows you to work with everything in one place.

```
@page "/"
@inject HttpClient Http
@inject NavigationManager NavigationManager

<PageTitle>Index</PageTitle>

@if (_cards == null)
{
(...)

@code {
(...)
}
```

Markup

Logic

Structuring Components

- When building applications, it's easy to create very large components that are doing lots of things.
- However, just like when creating regular C# classes, you should try to keep your components focused, with a single purpose.
- The logic in a component should be logic that operates over the markup and drives the function of the component.

Structuring Components

- Partial class
  - Another approach is to split the markup and logic of a component into two separate files
  - The markup of the component is kept in the .razor file, and the logic is added to a C# class

```
@page "/"
@if (_cards == null) {
    <p>Loading cards...</p>
} else {
  <div class="grid">
  @foreach (var trail in _trails)
     <Card card=”card" />
  }
</div>
}
```

HomePage.razor

```
using Microsoft.AspNetCore.Components;
using System.Net.Http.Json;

namespace app.Features.Home;

public partial class HomePage{

   private IEnumerable<Card>? _cards;
   [Inject]
    public HttpClient Http { get; set; } = default!

   protected override async Task OnInitializedAsync() {
     (...)
   }
}
```

HomePage.razor.cs

Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

Component life cycle methods

- Components in Blazor have a life cycle: they're created, they exist for a period, and then they're destroyed.
- Depending on what an application is doing, it may need to perform actions at certain points during this life cycle.
- For example, load initial data for the component to display when it is first created, or update the UI when a parameter has a certain value from the parent.
- Blazor supports this by giving us access to the component life cycle at specific points, which are:
  - OnInitialized/OnInitializedAsync
  - OnParametersSet/OnParametersSetAsync
  - OnAfterRender/OnAfterRenderAsync

Component life cycle methods

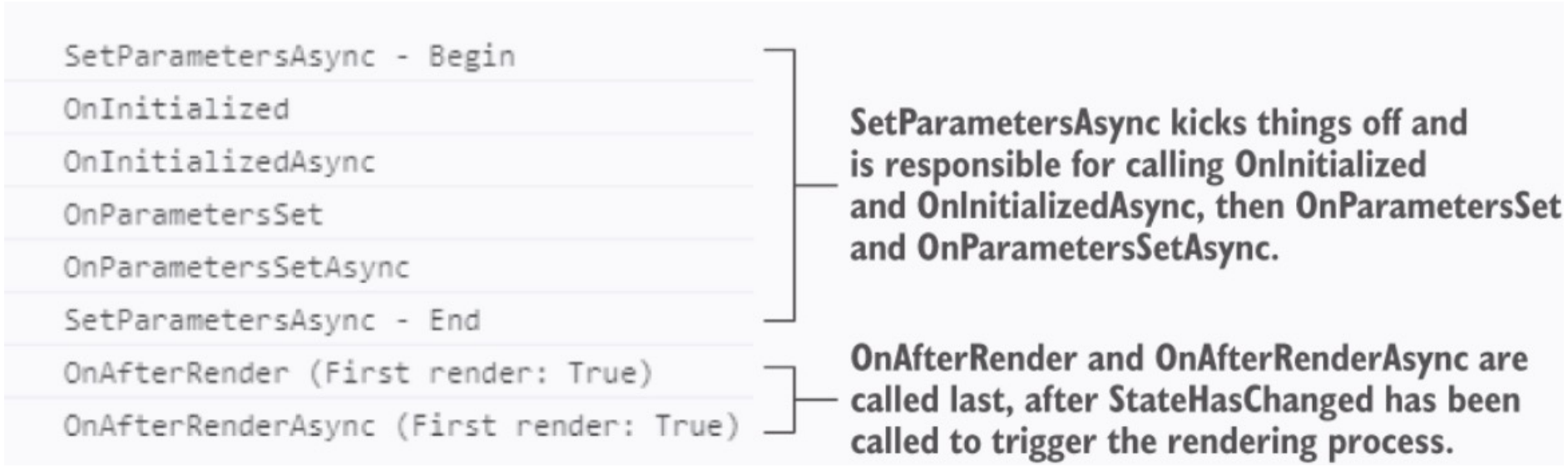
- SetParametersAsync is not a life cycle method that is often used by developers. Commonly, it is just OnInitialized, OnParametersSet, and OnAfterRender.
- During the first render, the component hasn’t been initialized.
- This means that OnInitialized and OnInitializedAsync will be called first, these are like constructors for your component.

Component life cycle methods

- Once the OnInitialized methods have run, OnParametersSet and On-ParametersSetAsync are called.
- These methods allow developers to perform actions whenever a component's parameters change. In the case of a first render, the component's parameters have been set to their initial values
- The final methods to run are OnAfterRender and OnAfterRenderAsync.
- These methods both take a Boolean value indicating if this is the first time the component has been rendered.
- The primary use of the OnAfterRender methods is to perform JavaScript interop and other DOM-related operations, such as setting the focus on an element.

Component life cycle methods

- Each method has a synchronous and asynchronous version.
- The synchronous version is always called before the asynchronous version.



Component life cycle methods – The First Render

- During the first render, all the component's life cycle methods will be called. During subsequent renders, only a subset of the methods will run
- The process starts with SetParametersAsync being called.
- This is the only life cycle method that requires us to call the base method; if we don't, then the component will fail to load.

Component life cycle methods – The First Render

- This is because the base method does two essential things:
  - Sets the values for any parameters the component defines—This happens both the first time the component is rendered and whenever parameters could have changed.
  - Calls the correct life cycle methods—This depends on whether the component is running for the first time or not.

Component life cycle methods – The life cycle with async

- One key point about the render we just covered is that it ran synchronously.
- In the Lifecycle component, there are no awaited calls in any of the async life cycle methods, meaning each method ran in sequence.
- However, when async calls are added, then things look a bit different.

Component life cycle methods – The life cycle with async

```
SetParametersAsync - Begin
OnInitialized
OnInitializedAsync - Begin
OnAfterRender (First render: True)
OnAfterRenderAsync (First render: True)
OnInitializedAsync - End
OnParametersSet
OnParametersSetAsync
OnAfterRender (First render: False)
OnAfterRenderAsync (First render: False)
SetParametersAsync - End
```

When awaiting an async call, StateHasChanged is invoked, triggering the render process. This allows the UI to be updated with the results of any synchronous code that has run up to this point.

Component life cycle methods – The life cycle with async

- While Blazor was awaiting the async call, the component was rendered.
- It was then rendered a second time after the OnParametersSet methods, as before.
- This is because Blazor checks to see if an awaitable task is returned from OnInitializedAsync.
- If there is, it calls StateHasChanged to render the component with the results of any of the synchronous code that has been run so far, while awaiting the completion of the task.
- This behavior is also true for async calls made in OnParametersSetAsync

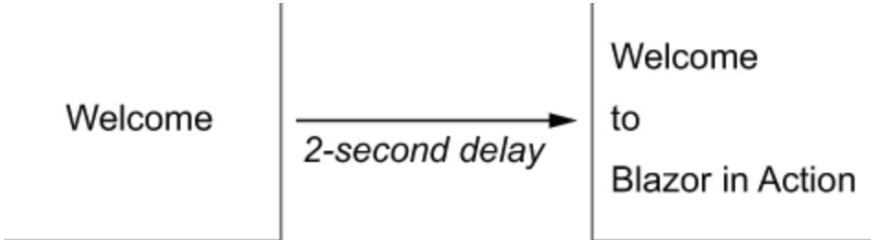
Component life cycle methods – The life cycle with async

- When dealing with multiple asynchronous calls, rendering may not behave quite as you'd expect;

```
@foreach (var word in _greeting)
{
    <p>@word</p>
}
@code {
    List<string> _greeting = new List<string>();
    protected override async Task OnInitializedAsync()
    {
        _greeting.Add("Welcome");
        await Task.Delay(1000);
        _greeting.Add("to");
        await Task.Delay(1000);
        greeting.Add("Blazor in Action");
    }
}
```

This component is simulating making multiple asynchronous calls in its On-InitializedAsync life cycle method

What happens is: the word Welcome is displayed, then after 2 seconds the words to Blazor in Action are added



Component life cycle methods – The life cycle with async

- The code up to the first awaited method is executed, and, as we just learned, a call is made to StateHasChanged at this point to render the results of any synchronous code while awaiting that task.
- There is no way for it to know that it should render after we add to the greeting list. Instead, the code continues to execute until the end of the method, and at this point, Blazor can perform a new render of the component
- If we want the UI to update after each word is added to the list, then we must manually call StateHasChanged to inform Blazor that the UI should be updated:

Component life cycle methods – The life cycle with async

- By telling Blazor when the UI needs to be updated, we've achieved the desired result of the words being rendered as they're added to the list.

```
(...)
await Task.Delay(1000);
  _greeting.Add("to");
  StateHasChanged();
(...)
```

Component life cycle methods – The extra life cycle method

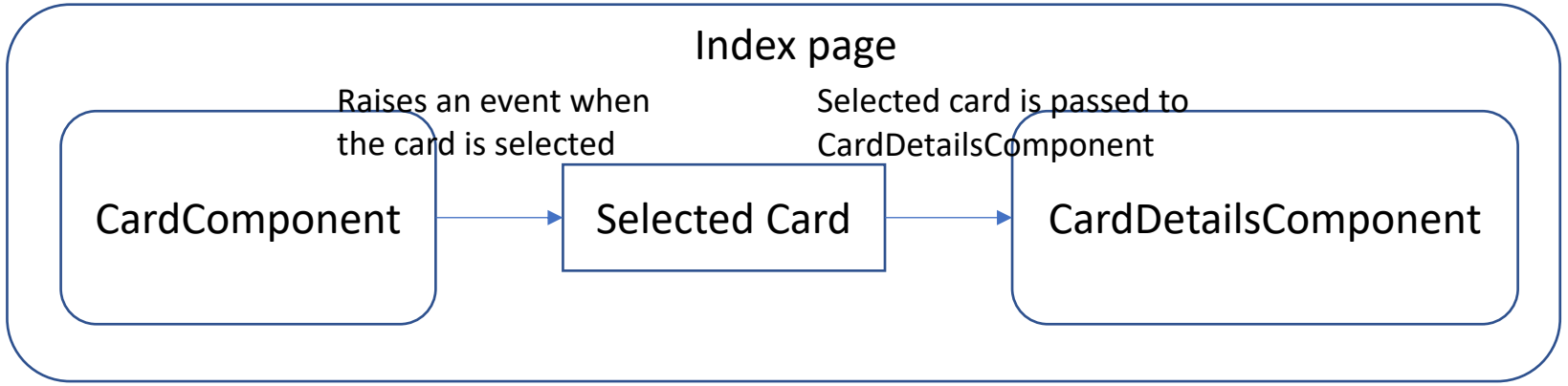
- There is another life cycle method that we can use, but this one is optional and it's not built in to the ComponentBase class: Dispose.
- This method is used for the same purposes in Blazor as in other C# applications: to clean up resources.
- This method is essential when creating components that subscribe to events, as failing to unsubscribe from events before a component is destroyed will cause a memory leak.
- To see the effect of this new life cycle method, we need to navigate away from the component. This will remove it from the DOM and invoke the Dispose method.

Working with parent and child components

- Components will sometimes need to communicate with each other, such as for passing data and firing and handling events.
- In Blazor, we achieve this using component parameters.
- Component parameters are declared on a child component, which forms that component's API.
- A parent component can then pass data to the child using that API. But component parameters can also be used to define events on the child that the parent can handle.
- This allows data to be passed from the child back up to the parent.

Working with parent and child components

- We'll add a View button to the Card that, when clicked, will slide open a drawer on the right side of the application.
- This drawer will display more detailed information about the selected card.
- For this to work, we need to have three different components communicate and pass data



Working with parent and child components

- The Index component will coordinate the operation.
- It will handle any On-Selected events from the Card component.
- When an OnSelected event is raised, the Index component will record the selected card and pass it into the CardDetails component.
- Inside the CardDetails component, whenever the card value changes, it will trigger the drawer to activate and slide into view.

Passing values from a parent to a child

- The CardDetails component, will display the selected card, which is passed in via a component parameter.
- We're going to create this component in the Home feature folder.

When a new card is passed in, the _isOpen field is set to true. This triggers the logic at the top of the component to render the slide CSS class.

```
<div class="drawer-wrapper @(_isOpen ? "slide" : "")">
    <div class="drawer-mask"></div>
    <div class="drawer">
      @if (_activeCard is not null)
      {
         <div class="drawer-content">
            <img src="@_activeCard.Image" />
            <div class="trail-details">
            <h3>@_activeCard.Name</h3>
(...)
```

Blazor uses Parameter attribute to find component parameters during the execution of the SetParametersAsync

```
@code {
    private bool _isOpen;
    private Card? _activeCard;
    [Parameter, EditorRequired]
    public Card? card { get; set; }

    protected override void OnParametersSet(){
      if (card != null){
        _activeCard = card;
        _isOpen = true;
      }
    }
    private void Close(){
       _activeCard = null;
        _isOpen = false;
    }
}
```

https://gist.github.com/brunobmo/9f5809b2b3554b619793194cbf572da0

Passing values from a parent to a child

- We’re using the OnParametersSet life cycle method to trigger the drawer sliding into view.
- As we learned earlier, this life cycle method is run every time the component’s parameters change.
- This makes it perfect for our scenario, as we can use it to trigger opening the drawer.

Passing values from a parent to a child

- In the app.css file (found in the wwwroot > css folder), we need to add the styles shown in the following listing to the bottom of the file.

```
.drawer {
    display: flex;
    flex-direction: column;
    position: fixed;
    top: 0;
    right: 0;
    bottom: 0;
    width: 35em;
    overflow-y: auto;
    overflow-x: hidden;
    background-color: white;
    border-left: 0.063em solid gray;
    z-index: 100;
    transform: translateX(110%);
    transition: transform 0.3s ease, width 0.3s ease;
}
.drawer-wrapper.slide > .drawer {
    transform: translateX(0);
}
(...)
```

The translateX function positions the drawer off the right-hand side of the screen by 110% of its width, making it appear closed

When the .slide class is applied in the CardDetails component, translateX is used again to position the drawer into view.

https://gist.github.com/brunobmo/a232f97008225b8db5268292c8b85f0e

Updating the Index component

- To pass the card into the CardDetails component, we use attributes when defining the component in the parent.
- The parent for us is the Index component.
- The following listing shows the HomePage component updated with the CardDetails component.

Updating the Index component

```
@page "/"
@inject HttpClient Http
@inject NavigationManager NavigationManager

<PageTitle>Index</PageTitle>

@if (_cards == null){
  <p>Loading cards...</p>
}
else{
  <CardDetails card="_selectedCard" />
  <div class="grid">
    @foreach (var card in _cards){
     <CardComponent card="card"
OnSelected="HandleTrailSelected"/>
    }
  </div>
}
```

Data is passed to component parameters using attributes on the element.

In the Index component, we have defined a field called _selectedCard, which will store the selected card.

```
@code {
    private IEnumerable<Card>? _cards;
    private Card? _selectedCard;
    protected override async Task OnInitializedAsync(){
      try{
        string domainName = NavigationManager.Uri;
        _cards = await
Http.GetFromJsonAsync<IEnumerable<Card>>(domainName +
"cards/cards.json");
      }
      catch (HttpRequestException ex){
        Console.WriteLine($"There was a problem loading cards
data: {ex.Message}");
      }
    }

    private void HandleCardSelected(Card card){
      _selectedCard = card;
      StateHasChanged();
    }
}
```

https://gist.github.com/brunobmo/2895041707eead3e8802702d80c06f43

Passing data from a child to a parent

- In order to see something happen onscreen, we need to select a card to display.
- This can be done by passing that information up from the Card component to the Index component.
- To do this, we will use component parameters to define an event on the Card.
- This event will pass the selected card; the Index component can then handle this event and pass the card to the CardDetails component to display.

Passing data from a child to a parent

- The following listing shows the updated code for the Card component.

```
<div class="card shadow" style="width: 18rem;">
(...)
     <button class="btn btn-primary" title="View" @onclick="@(()
=> OnSelected?.Invoke(card))">
     <i class="bi bi-binoculars"></i>
     </button>
   </div>
</div>



@code {
   [Parameter, EditorRequired]
   public Card card { get; set; } = default!;

   [Parameter, EditorRequired]
   public Action<Card> OnSelected { get; set; }
}
```

The delegate is invoked, passing in the current card.

Events are defined using delegate types of either Action or Func.

https://gist.github.com/brunobmo/812b44eed49b504397b9aa4770cc676d

A delegate is a type that defines a method signature and can be used to reference methods that match that signature. It allows methods to be treated as objects, which can be passed as arguments, assigned to variables, or returned from other methods.

Handling DOM Events

- Blazor has its own events system that wraps the standard DOM events, allowing us to work with them natively, in C#.
- To handle an event, we use the following syntax on an element: @onEVENTNAME="Handler"
- EVENTNAME is the name of the event you wish to handle, and HANDLER is the name of the method that will be invoked to handle the event.
- Blazor will also pass an event argument to the Handler method, which is appropriate for the event.

Handling DOM Events

- For example, if we wanted to handle the keydown event on an input, we could do so like this: `<input @onkeydown="HandleKeydown" />`
- Then in the code block, we can define the Handler like this:

```
private void HandleKeydown(KeyboardEventArgs args)
{
    Console.WriteLine(args.Key);
}
```

For a full list of event arguments and the events they are for, check out the official docs at http://mng.bz/M5Wo

Passing data from a child to a parent

- With the Card updated, all that's left to do is handle the event in the Index component.
- First, we need to add a method to the code block, which will be called whenever an event is raised:

```
private void HandleCardSelected(Card card)
{
  _selectedCard = card;
  StateHasChanged();
}
```

This method accepts the selected card and assigns it to the _selectedCard field.
However, in order to see anything happen, we must call StateHasChanged. This lets Blazor know that we need the UI to update.

EventCallback

- There is another way.
- We can use a different type to define our event on the Card called EventCallback.
- By using this type for our event, Blazor will automatically call StateHasChanged on the component that handles the event, removing the need to manually call it.

```
<div class="card" style="width: 18rem;">
    <img src="@Card.Image" class="card-img-top" alt="@Card.Name">
    <div class="card-body">
       // other markup omitted for brevity
       <button class="btn btn-primary"
            @onclick="@(async () => await OnSelected.InvokeAsync(Card))">View</button>
    </div>
</div>
@code {
  [Parameter, EditorRequired]
  public Card Card { get; set; } = default!;

  [Parameter]
  public EventCallback<Card> OnSelected { get; set; }
}
```

When using EventCallback, a null check is not required. It also supports async handlers; therefore, we must invoke the event asynchronously.

The component parameter is now typed as EventCallback<Trail>

Passing data from a child to a parent

- The final update is to assign the HandleCardSelected method to the OnSelected event in the Index component.
- We do this the same way we did to pass the selected card into the CardDetails component—using attributes:

```
(...)
<CardDetails card="_selectedCard" />
<div class="grid">
   @foreach (var card in _cards){
      <CardComponent card="card"  OnSelected="HandleCardSelected" />
   }
</div>
(...)
```

Resume

- Components can be structured in different ways: they can be defined in a single file containing both markup and logic, or they can be defined in two different files, one containing markup and the other the logic
- Components have multiple life cycle methods that can be hooked into to perform actions at defined points on that life cycle.
- The three commonly used life cycle methods are OnInitialized/OnInitializedAsync, OnParametersSet/OnParametersSetAsync, and OnAfterRender/OnAfterRenderAsync.
- The OnInitialized/OnInitializedAsync method runs only once in the lifetime of a component. The other methods can run multiple times.

Resume

- Parent components can pass data into child components using component parameters.
- Component parameters form the public API of a component.
- Events can be defined as component parameters, which allow data to be passed from child to parent.
- A parent handling a child component's event, defined as either Action/Action<T> or Func/Func<T>, must call StateHasChanged manually to trigger any required UI updates.
- Child components can define their events using the type EventCallback or EventCallback<T>, which will automatically call StateHasChanged on the parent component once the handler has been run

Blazor Server

Component Model

2023