

BLAZOR

.NET IN THE BROWSER USING WEB ASSEMBLY

-={ S. KYLE KORNDORFER }=-



OVERVIEW

- What is Web Assembly (wasm)?
 - asmjs, only standardized through W3C
 - Basics of Web Assembly
 - Really simple example
- What is Blazor?
 - Full stack development using .NET

LEADING UP TO WEB ASSEMBLY

- JavaScript is the primary mechanism for making web pages more dynamic/functional
 - Supported natively by all browsers
 - While the language has gotten more expressive and capable, there are plenty of shortcomings
- Plugins to the rescue!?
 - ActiveX, Java, Flash, Silverlight, etc.
 - Insecure mechanisms for loading and executing the plugins (NPAPI is over 25yrs old!)

WHAT IS WEB ASSEMBLY (WASM)?



SO WHAT IS IT?

- Low-level, assembly like language
- Compact binary format
- Runs with near-native performance
- Languages like C/C++, Rust, etc. can target this format
- Runs alongside JavaScript in the same sandboxed VM so JavaScript code can call into wasm modules

GOALS OF WEB ASSEMBLY

- Fast, efficient, & portable
 - Designed against common hardware capabilities across all platforms (including mobile)
- Readable & debuggable
 - Does have a human readable format that can be written/viewed/debugged by hand
 - Not a great debugging experience yet
- Secure
 - Runs in a safe, sandboxed execution environment with forced same-origin and permission policies
- Don't break the web!
 - Must work with other web technologies and have backwards compatibility

HOW DOES IT WORK?

- The web platform has essentially 2 parts
 - A Virtual Machine that runs the app code (JavaScript)
 - No-compile/optimization; large libraries take time to download/compile/optimize
 - Web APIs that control the browser or device (DOM, CSSOM, WebGL, etc.)
- Wasm extends the VM to be able to execute the low-level assembly format
 - Wraps exported wasm code so JS can invoke it
 - Can import JS code so wasm modules can use existing frameworks/libs

HOW DO I USE WASM?

- Port an existing C/C++ app using [Emscripten](#)
 - [WasmFiddle](#), [WasmFiddle++](#), [WasmExplorer](#)
- Write/generate WebAssembly directly at the assembly level
- Target WebAssembly from a higher-level language (Rust)

SIMPLE WEB ASSEMBLY DEMO



WHAT IS BLAZOR?

.NET IN THE BROWSER

- NOT a compilation target for .NET code
 - Code is still compiled into .NET assemblies/dll's
- MONO runtime compiled into Web Assembly
 - Loads and processes compiled .NET IL code
 - Supports .NET Standard 2.0 (not all API's; no Filesystem, Sockets, etc.)
- Client-side web UI framework (C#/Razor/HTML)
 - similar to Angular, Vue, React, etc.
 - Currently **highly** experimental & unsupported; expect breaking changes!

HOW DOES IT WORK?

- Compile C# & Razor files into .NET assemblies
- .NET runtime and assemblies downloaded by the browser
- JavaScript bootstraps the .NET runtime
- Loads the compiled assemblies
 - DOM manipulation & browser API calls are handled by Blazor runtime through JavaScript Interop
 - C# can call JavaScript & JavaScript can call C#

DEPLOYMENT?

- Standalone app
 - Static, client-side application published in a *dist* folder that can be deployed onto many hosting services list GitHub pages, etc. **No .NET server support required!**
- Full Stack ASP.NET Core App
 - Code can be shared between client & server (models, etc.)
 - Can be deployed anywhere ASP.NET Core apps are supported

CORE CAPABILITIES

- Components
- Layouts
- Routing
- Dependency Injection

BASIC BLAZOR DEMO



COMPONENTS

Building blocks of Blazor

COMPONENTS

- Piece of the UI (page, dialog, form, etc.)
- Can be nested, reused, and shared between projects
- Can be written as either a C# class or a Razor markup file (*.cshtml)
- Can be unit tested without needing a browser DOM
- Compiled into a class with the same name as the file
- Render tree changes are compared against previous & deltas applied

SIMPLE COMPONENT (CSHTML)

```
<div>
  <h2>@Title</h2>
  @BodyContent
  <button onclick=@OnOK>OK</button>
</div>

@functions {
  public string Title { get; set; }
  public RenderFragment BodyContent { get; set; }
  public Action OnOK() { ... }
}
```


COMPONENT PARAMETERS

ParentComponent.cshtml

```
@page "/ParentComponent"  
  
<h1>Parent-child example</h1>  
  
<ChildComponent Title="Panel from Parent">  
    Child content of the child component  
    is supplied by the parent component.  
</ChildComponent>
```

ChildComponent.cshtml

```
<div class="panel panel-success">  
    <div class="panel-heading">@Title</div>  
    <div class="panel-body">  
        @ChildContent  
    </div>  
</div>  
  
@functions {  
    [Parameter]  
    private string Title { get; set; }  
  
    [Parameter]  
    private RenderFragment ChildContent {  
        get; set;  
    }  
}
```

ANATOMY OF A COMPONENT

```
@using System
@page “/User/{id}”
@layout MasterLayout
@inherits BaseComponent
@implements IDisposable
@inject ILogger logger
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
<!-- HTML markup w/ Razor intermixed -->
@functions { ... }
```

DATA BINDING

- Uses the 'bind' or 'bind-*' (child component) attribute
- Done this way, triggers UI render tree to be updated
 - Updates through code requires a manual trigger to update UI

```
<input type="checkbox" id="italicsCheck" bind="@_italicsCheck" />
```

EVENT HANDLING

- Uses 'on<event>' HTML format (onclick, onsubmit, onchange, etc.)
- Some events pass optional event-specific arguments
 - UIEventArgs
 - UIChangeEventArgs
 - UIKeyboardEventArgs
 - UIMouseEventArgs

EVENT HANDLING

```
<button class="btn btn-primary" onclick="@UpdateHeading">  
    Update heading  
</button>
```

```
@functions {  
    void UpdateHeading( UIMouseEventArgs e )  
    {  
        // ...  
    }  
}
```


EVENT HANDLING WITH LAMBIDAS

```
<button onclick="@ (e => Console.WriteLine("Hello, world!"))">  
    Say hello  
</button>
```

COMPONENT REFERENCES

```
<MyLoginDialog ref="loginDialog" ... />
```

```
@functions {  
    MyLoginDialog loginDialog;  
  
    void OnSomething()  
    {  
        loginDialog.Show();  
    }  
}
```

LIFECYCLE METHODS

- `OnInit` / `OnInitAsync`
- `OnParametersSet` / `OnParametersSetAsync`
 - called when component has received parameters from its parent and the values are assigned to properties; executed after `OnInit` during component initialization
- `OnAfterRender` / `OnAfterRenderAsync`
 - called each time after a component has finished rendering; element and component references are populated at this point; use to perform additional initialization steps using the rendered content, such as activating third-party JavaScript libraries that operate on the rendered DOM elements.

LIFECYCLE METHODS

```
protected override async Task OnInitAsync() { await ... }
protected override async Task OnParametersSetAsync() { await ... }
protected override async Task OnAfterRenderAsync() { await ... }

public override void SetParameters( ParameterCollection parameters )
{
    // ...
    base.SetParameters( parameters );
}

protected override bool ShouldRender() { return {true|false}; }
```

'CODE BEHIND' EXPERIENCE

BlazorRocks.cshtml

```
@page "/BlazorRocks"  
@inherits BlazorRocksBase  
  
<h1>@BlazorRocksText</h1>
```

BlazorRocksBase.cs

```
using Microsoft.AspNetCore.Blazor.Components;  
  
public class BlazorRocksBase : BlazorComponent  
{  
    public string BlazorRocksText { get; set; } =  
    "Blazor rocks the browser!";  
}
```


LAYOUTS

Technically, just another Blazor component!

Keepin' it DRY!

WHAT DEFINES A LAYOUT?

- Standard component (with binding, injection, etc.) with 2 differences:
 - Must inherit from `BlazorLayoutComponent`
 - Must have a '@Body' property in the markup where content gets rendered
- Layouts can be nested to multiple levels
- Each folder can have it's own '`_ViewImports.cshtml`'

SIMPLE LAYOUT

```
@inherits BlazorLayoutComponent
```

```
<header>  
    <h1>ERP Master 3000</h1>  
</header>
```

```
<nav>  
    <a href="master-data">Master Data Management</a>  
    <a href="invoicing">Invoicing</a>  
    <a href="accounting">Accounting</a>  
</nav>
```

```
@Body
```

```
<footer>&copy; by @CopyrightMessage</footer>
```

```
@functions {  
    public string CopyrightMessage { get; set; }  
    // ...  
}
```

ROUTING

How do requests make it to the right component?

ROUTING BASICS

- Uses the '@page' directive to define the route path
- Can have multiple '@page' directives defined
- Can have parameters that get mapped to component properties (no optionals yet)

```
@page "/User/{UserId}"  
<h1>Welcome User @UserId!</h1>  
@functions {  
    [Parameter]  
    private Guid UserId { get; set; } = Guid.Empty();  
}
```


CREATING NAVIGATION LINKS

- Use NavLink instead of anchor tags to generate links to other components
- Automatically adds an `'active'` class to the anchor element when the current path matches the link definition
- The `'Match'` attribute controls when this class is applied
 - `'NavLinkMatch.All'` = active when it matches the entire current URL
 - `'NavLinkMatch.Prefix'` = active when it matches any *prefix* of the current URL

```
<NavLink href="/" Match=NavLinkMatch.All>Home</NavLink>
```

DEPENDENCY INJECTION

All your services are belong to us!

MANAGING EXTERNAL DEPENDENCIES

- Based on the same as DI system in ASP.NET Core
 - Register services in `ConfigureServices()` method
- Services can have 3 (really 2 currently) different lifetimes when registered
 - *Singleton* – same service instance for app lifetime
 - *Transient* – new service created each time a component needs it
 - *Scoped* – not implemented yet so behaves like Singleton for now

DEFAULT SERVICES

- Blazor automatically registers 2 services
 - `IUrlHelper` – Helper methods for working with URI's and Navigation (singleton)
 - `HttpClient` – Sending/receiving HTTP requests/responses using the browser to handle the traffic in the background (singleton).
 - `HttpClient.BaseAddress` automatically set to the base URI prefix of the app

HOW?

- Use `@inject {Type} {ReferenceName}` in CSHTML files
 - If using a base class for a component, use the `[Inject]` attribute
 - no need for `@inject` in the child component
- **Example:** `@inject IDataAccess DataRepository`

JAVASCRIPT INTEROP

Leveraging the work of others

JAVASCRIPT FROM .NET CODE

- Use the `IJSRuntime` abstraction available through `JSRuntime.Current`
- Call the `InvokeAsync<T>()` method
 - First argument is the name of the `window` (global) scoped JS function you wish to invoke
 - Additional arguments must be JSON serializable (any number)
 - Return type `<T>` must also be JSON serializable
- Can be done synchronously (not recommended)
 - Cast to `IJSInProcessRuntime` and call `Invoke<T>()` instead

EXAMPLE

- `exampleJsInterop.js`

```
window.exampleJsFunctions = {  
  showPrompt: function(message) {  
    return prompt(  
      message,  
      'Type anything here');  
  }  
};
```

- `exampleJsInterop.cs`

```
public static Task<string> Prompt(  
    string message )  
{  
    return  
        JSRuntime.Current.InvokeAsync<string>(  
            "exampleJsFunctions.showPrompt",  
            message );  
}
```

ELEMENT REFERENCES

- Reference HTML elements in order to invoke actions or manipulate in some way
- Use the `ref` attribute on the HTML element and the `ElementRef` type in C# code
- Do **not** use captured references as a way to populate the DOM
 - Messes with Blazor's declarative rendering model
- Only used for passing the reference through for JS Interop
 - JS code receives the reference as an `HTMLElement` instance

ELEMENT REFERENCE (EXAMPLE I)

```
<input ref="username" ... />
```

```
@functions {  
    ElementRef username;  
}
```


ELEMENT REFERENCE (EXAMPLE 2)

myLib.js

```
window.myLib = {  
  focusElement : function (element) {  
    element.focus();  
  }  
}
```

ExampleComponent.cshtml

```
@using MyLib  
<input ref="username" />  
<button onclick="@SetFocus">Set  
focus</button>  
@functions {  
  ElementRef username;  
  void SetFocus() {  
    username.Focus();  
  }  
}
```

ElementRefExtensions.cs

```
using Microsoft.AspNetCore.Blazor;  
using Microsoft.JSInterop;  
using System.Threading.Tasks;  
  
namespace MyLib  
{  
  public static class MyLibElementRefExtensions  
  {  
    public static Task Focus(this ElementRef elementRef)  
    {  
      return JSRuntime.Current.InvokeAsync<object>(  
        "myLib.focusElement",  
        elementRef );  
    }  
  }  
}
```

INVOKING .NET FROM JAVASCRIPT

- .NET method must be **public, static**, and decorated with the `[JSInvokable]` attribute
- Invoked using `DotNet.invokeMethodAsync`
 - or `DotNet.invokeMethod`; not preferred
- Need to pass the assembly name, method name, and any arguments

EXAMPLE

- JavaScriptInteropable.cs

```
public class JavaScriptInvokable
{
    [JSInvokable]
    public static Task<int[]> ReturnArrayAsync()
    {
        return Task.FromResult(
            new int[] { 1, 2, 3 });
    }
}
```

- JavaScriptInteropable.cs

```
DotNet.invokeMethodAsync(
    assemblyName,
    'ReturnArrayAsync'
).then( data => ... )
```

TODO DEMO

Build a static TODO application

LINKS

- [Webassembly.org](https://webassembly.org)
- [Mozilla Developer site](#) on Web Assembly
- [Blazor.net](https://blazor.net)
- [LinkedIn Learning](#) course (free for a limited time)
- [Learn Blazor](#)
- [.NET Conf 2018 on Blazor](#)