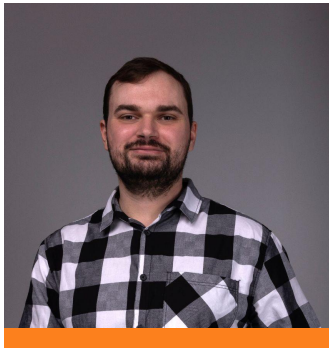
Into The Rabbit Hole of Blazor Wasm Hot-Reload

Andrii Rublov
Software Developer

About me



- **Software Developer** at **JetBrains**

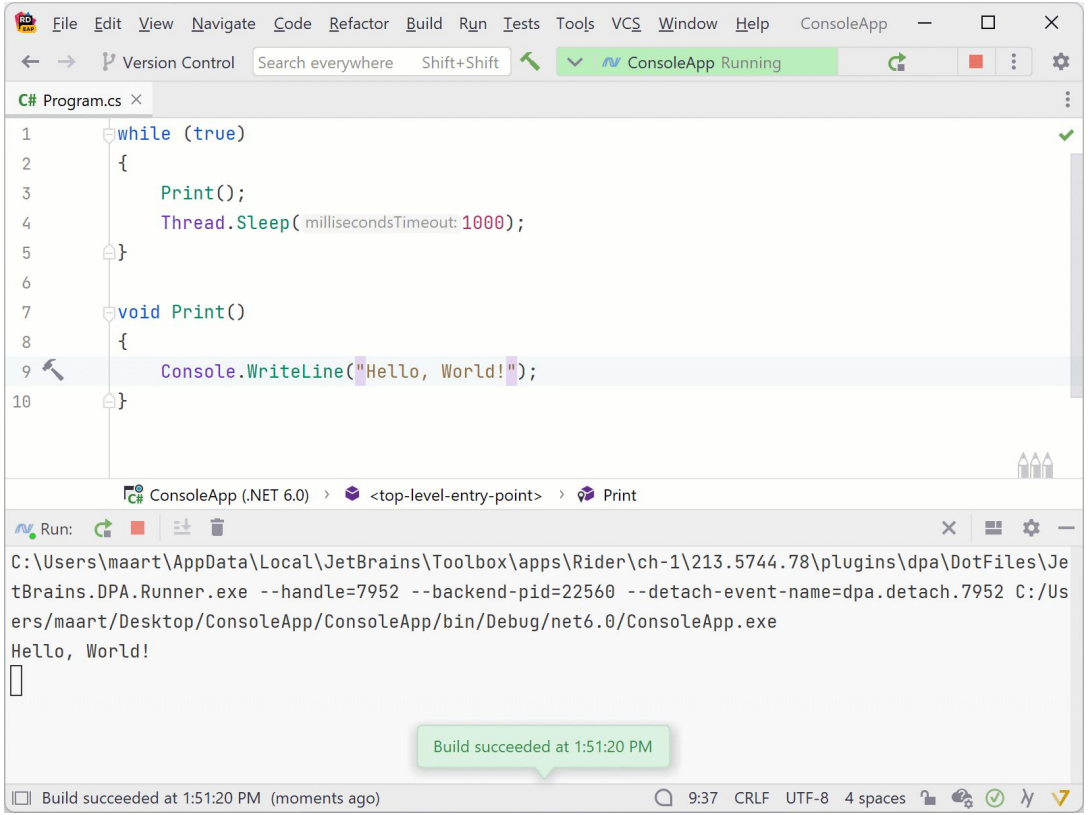
- Mainly working on Rider's
  - Blazor Wasm running, debugging & hot-reload
  - EF / EF Core tooling
  - NuGet integration
  - Run / Debug configurations

- Featured open-source projects
  - **rider-monogame**: MonoGame plugin for JetBrains Rider
  - **rider-efcore**: EF Core plugin for Rider
  - **dotnet-chrome-protocol**: A runtime and schema code generation tools for Chrome DevTools Protocol support in C#/.NET.

linktr.ee/seclerp

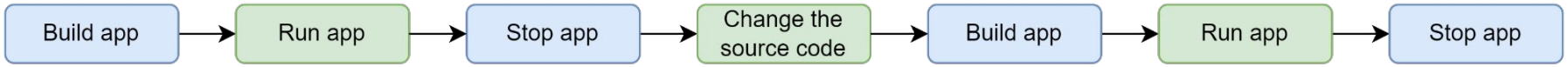
What is “hot-reload”?

A way of changing the running app’s code “on the fly”, i.e. without rebuilding and/or restarting



Why is it needed?

Regular run-change-rerun workflow:

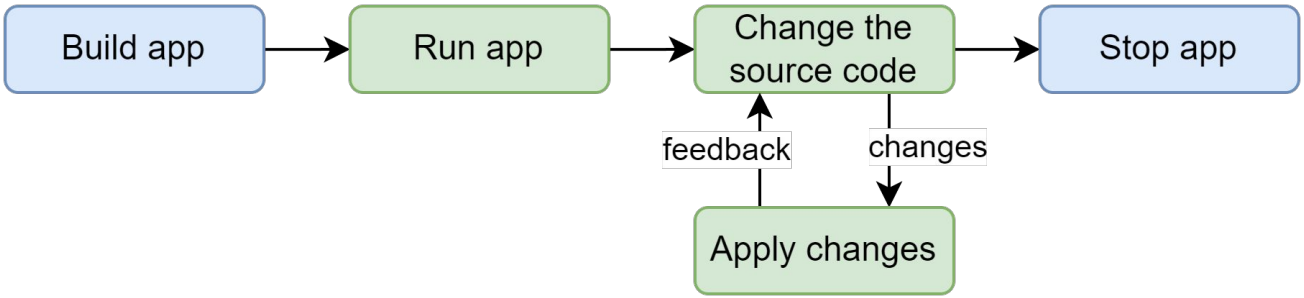


Especially painful feedback loop for UI-involved platforms:

- Mobile apps
- Desktop apps
- Web (frontend) apps

Why is it needed?

Workflow with hot-reload:



Chapter 1:
How hot-reload works in .NET

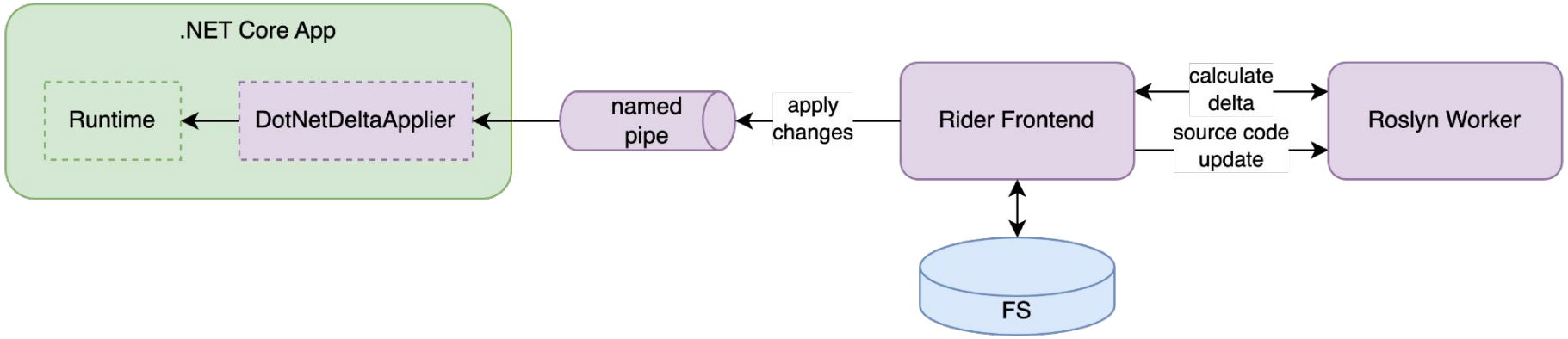
5 types of hot-reload in .NET

- **CoreCLR hot-reload** without debugging (aka “Hot Reload”)
- **CoreCLR hot-reload** when debugging (aka Edit & Continue, EnC)
- **XAML hot-reload for MAUI** (out of talk’s scope)
- **XAML hot-reload for WPF, UWP** (not supported in Rider at the moment)
- **Blazor Wasm hot-reload**

Rider: CoreCLR hot-reload without debugging

- **Source code changes tracking** is based on Roslyn `Workspace` inside **Roslyn Worker**
- **IL & metadata delta calculation** is done by Roslyn infrastructure inside **Roslyn Worker**
- **Delta applying** is done by a `DotNetDeltaApplier` injected into app via **startup hook**

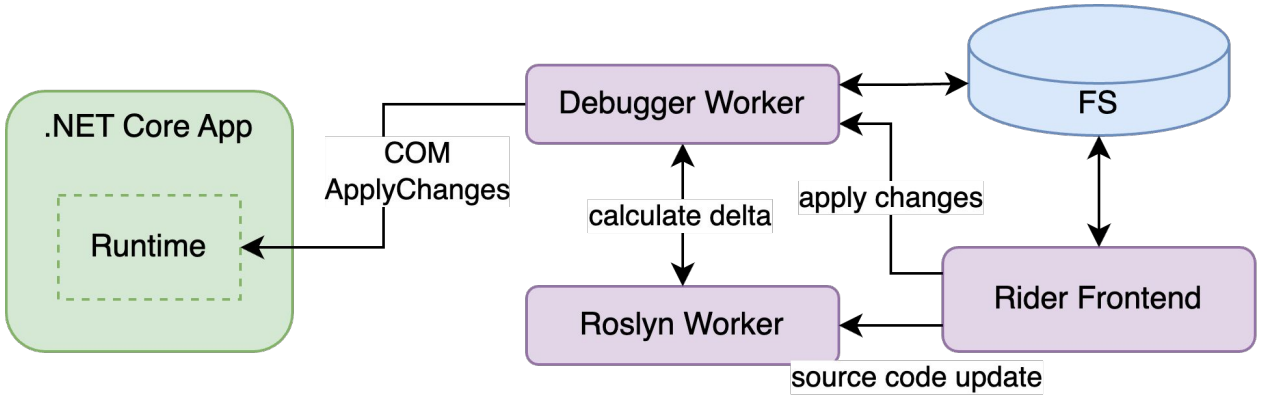
  Communication between the IDE and delta applier is done by **named pipes**



Rider: CoreCLR hot-reload with debugging

- **Source code changes tracking** is based on Roslyn `Workspace` inside **Roslyn Worker**
- **IL & metadata delta calculation** is done by Roslyn infrastructure inside **Roslyn Worker**
- **Delta applying** is done by a special **Debugger Worker process**

  Debugger Worker communicates with the target process via special COM interfaces



Delta

A difference in app's code & metadata mainly consist of 4 parts:

- **IL code diff**
- **Assembly metadata diff**
- **Debugging symbols diff**
- **A set of updated types**

All these parts are kindly calculated by **Roslyn** on request

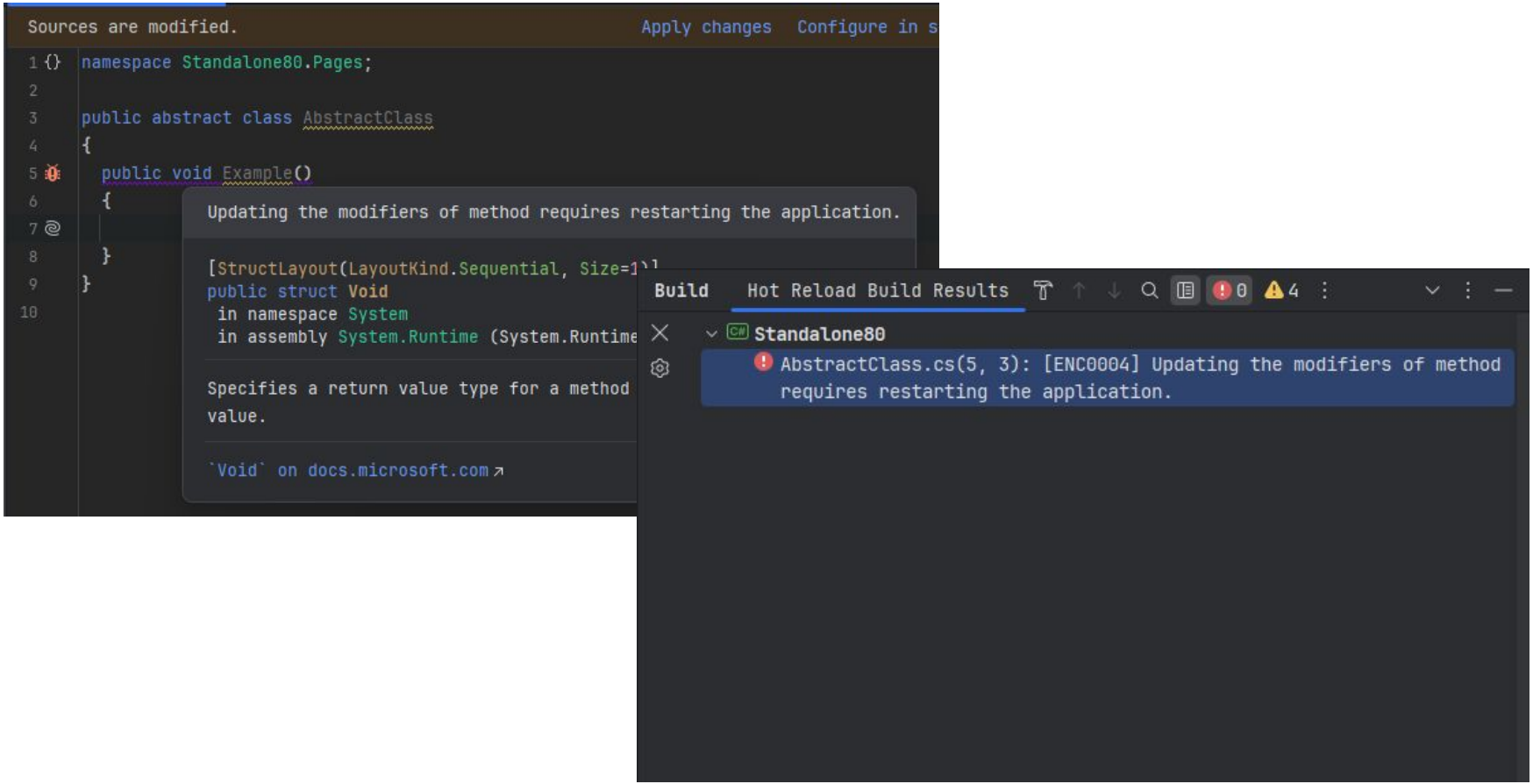
Unsupported changes (“Rude changes”)

At the moment (*.NET 8, September 2024*), such changes are not supported by CoreCLR hot-reload:

- **Adding & modifying `dynamic` objects**
- **Deleting types**
- **Modifying interfaces**
- **Making abstract method non abstract**
- **Adding abstract, virtual or override method**
- **Modifying a type parameter, base type, delegate type**
- **Add destructor to an existing type**

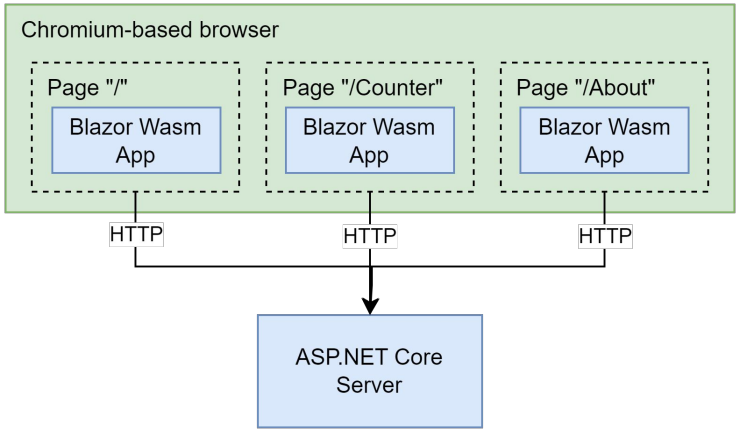
… and more at Supported Edits in Edit & Continue (EnC) and Hot Reload on .NET 8 page

Unsupported changes (“Rude changes”)



Chapter 2:
Specifics of Blazor Wasm hot-reload

Changes should be applied for multiple parties



Code of the app that hosts Blazor Wasm is executed in 2 main sides:

- **Server**: User's ASP.NET Core application* which hosts Blazor Wasm app binaries
- **Client(s)**: One or multiple opened pages with Blazor Wasm apps

* except Blazor Wasm Standalone projects, they use built-in ASP.NET Core-based DevServer that comes with SDK

2 types of runtime to apply changes to

- **CoreCLR**
  - COM interop
  - ICorDebugModule2::ApplyChanges
- **Mono (Wasm-flavoured)**
  - dotnet.js
  - WebSockets-based changes delivery

Page refresh should preserve changes

- When refreshing the existing page, changes made to that moment should be preserved
- Newly opened pages should receive changes as well

Each new or refreshed page = new runtime instance. Which means, any temporary hot reload changes will be lost after refreshing.

Static files updates

Static file updates should be handled separately from regular .NET hot-reload mechanism:

- `.js`
- **`.css`**
- `.razor.js`
- **`.razor.css`**

They are not the part of .NET code at all and only used in resulting HTML pages.

HTTP Caching

- Hot-reload results with new updated assets, which are delivered to the page via HTTP requests
- Caching is very important for production SPAs, especially for “heavy” frameworks like Blazor Wasm
- But it’s a problem when it comes to cleaning old assets and forcing to use new ones

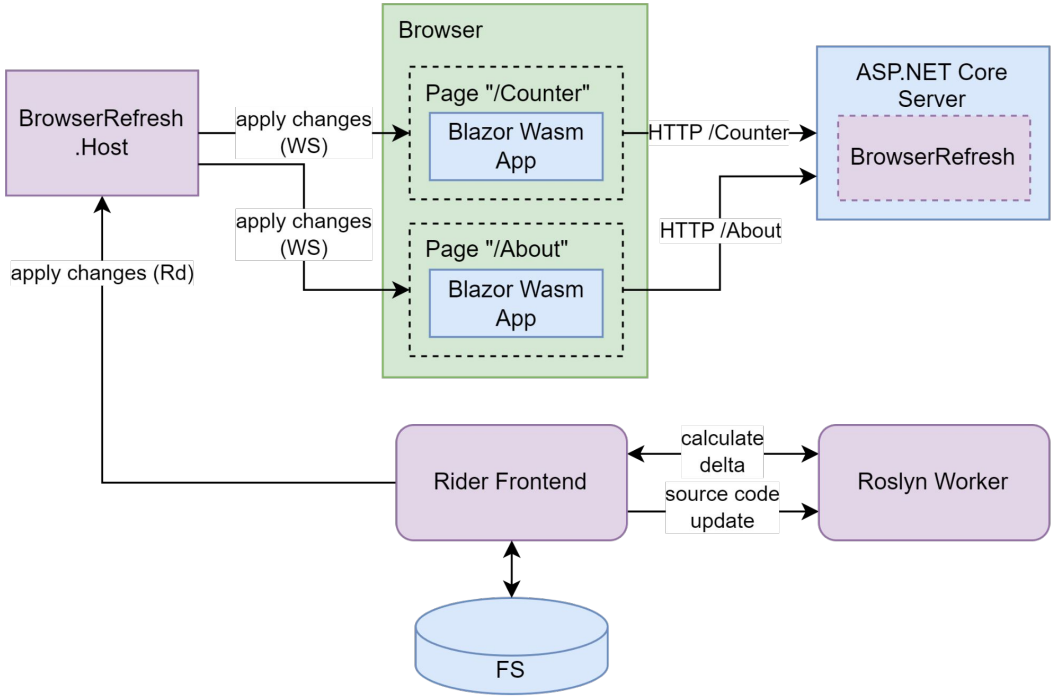
Summary: Specifics of Blazor Wasm hot-reload

- Changes should be applied for both server and all connected clients
- Different changes applying mechanisms for different runtime types
- Changes aggregation and distribution is mandatory for both new clients and refreshed ones
- Static files should be handled in completely different from .NET code manner
- HTTP caching should be bypassed

Chapter 3:
Blazor Wasm hot-reload in Rider

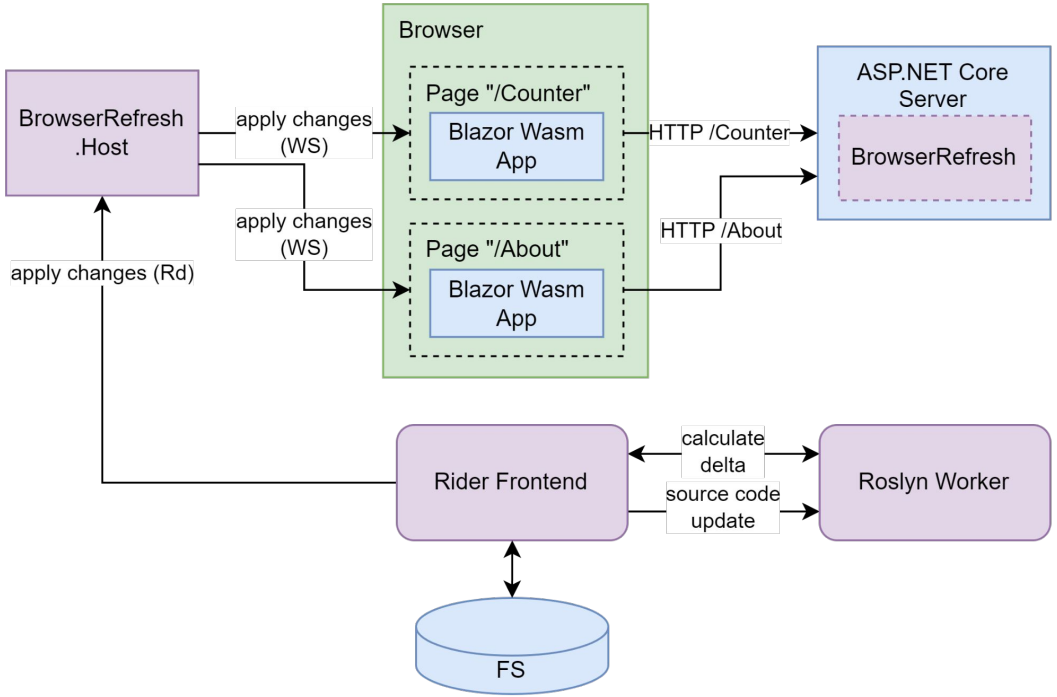
Applying changes to the client code

1. **Rider** starts **BrowserRefresh.Host** - a WebSocket server that distributes changes across all connected clients
2. **Rider** starts the app's server with injected middlewares
3. When browser opens app's page, HTTP request loads modified HTML with injected **blazor-hotreload.js** and **aspnetcore-browser-refresh.js** scripts
4. The script asks server for already applied changes from the past instances, applies them if needed
5. The script establishes the WebSocket connection to the **BrowserRefresh.Host**
6. When user changes source code, **Rider** asks **Roslyn Worker** to calculate delta



Applying changes to the client code

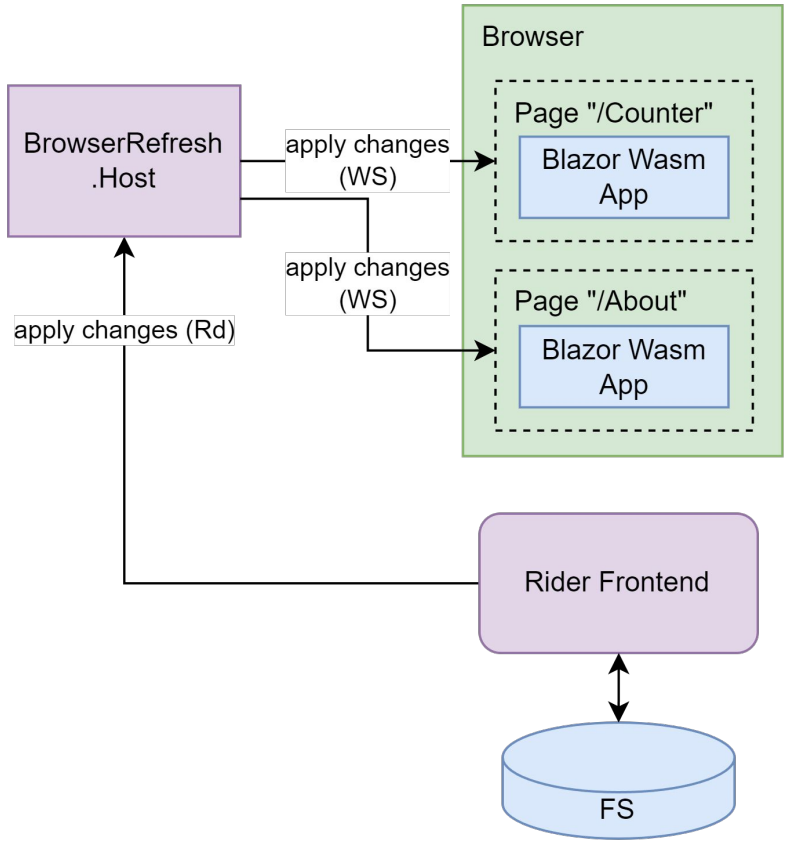
7. **Rider** sends calculated delta to the **BrowserRefresh.Host** using Rd protocol
8. **BrowserRefresh.Host** distributes changes to all connected clients
9. **aspnetcore-browser-refresh.js** applies new changes on the client
10. **aspnetcore-browser-refresh.js** sends a `POST /blazor-hotreload` request to the **BrowserRefresh** middleware to aggregate newly applied change for future clients
11. Client-side hot reload is finished!



BrowserRefresh.Host

BrowserRefresh.Host is actually just a broadcaster.

1. Asks about hot-reload capabilities of each connected client
2. Receives one of the events from Rider:
   - `BroadcastUpdateDelta`
   - `BroadcastUpdateStaticFiles`
   - `BroadcastReload`
   - `BroadcastWait`
   - `NotifyUpdatesApplied`
   - `NotifyDiagnostics`
3. Broadcasts them to all connected WebSocket clients via **BrowserRefresh.Agent**



BrowserRefresh: Injection into user's ASP.NET Core app

2 very useful approaches are used: **startup hooks** and **hosting startup assemblies**.

- **Startup hook** allows to run external code together with a main app

```
DOTNET_STARTUP_HOOKS=...\Microsoft.AspNetCore.Watch.BrowserRefresh.dll
```

- **Hosting startup assemblies** are similar to startup hooks but also allows altering the ASP.NET Core app builder (attaching middleware, startup filters, configuration, etc.)

```
ASPNETCORE_HOSTINGSTARTUPASSEMBLIES=Microsoft.AspNetCore.Watch.BrowserRefresh
```

BrowserRefresh: Server-side infrastructure

- `POST /_framework/blazor-hotreload` submits applied changes from clients
- `GET /_framework/blazor-hotreload` returns all previously applied changes on fresh client
- `POST /_framework/clear-browser-cache` clears local browser caches
- `GET /_framework/aspnetcore-browser-refresh.js`
- `GET /_framework/blazor-hotreload.js`

BrowserRefresh: Client-side infrastructure

Modified `index.html` of your app looks like that:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- ... -->
</head>
<body>
    <div id="app">
        <!-- ... -->
    </div>
    <div id="blazor-error-ui">
        <!-- ... -->
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
<script src="/_framework/aspnetcore-browser-refresh.js"></script></body>
</html>
```

BrowserRefresh: Client-side infrastructure

- **`aspnetcore-browser-refresh.js`**
  - Setups the connection to the **BrowserRefresh.Host** WebSocket server and handles messages from it
  - Applies changes on request
  - Provides server with supported hot-reload capabilities, some sort of versioning
  - Shows nice checkmark at the end

- **`blazor-hotreload.js`**
  - Is loaded by blazor.webassembly.js script on startup
  - Applies delta previously aggregated on the server

BrowserRefresh: aspnetcore-browser-refresh.js

```
const webSocketUrl = '{{hostString}}';
const sharedSecret = await getSecret('{{ServerKey}}');
const connection = await getWebSocket(webSocketUrl);

connection.onmessage = function (message) {
    // …
    const payload = JSON.parse(message.data);
    const action = {
      'UpdateStaticFile': () => updateStaticFile(payload.path),
      'BlazorHotReloadDeltav1': () => applyBlazorDeltas(payload.sharedSecret, payload.deltas, false),
      'BlazorHotReloadDeltav2': () => applyBlazorDeltas(payload.sharedSecret, payload.deltas, true),
      'HotReloadDiagnosticsv1': () => displayDiagnostics(payload.diagnostics),
      'BlazorRequestApplyUpdateCapabilities': () => getBlazorWasmApplyUpdateCapabilities(false),
      'BlazorRequestApplyUpdateCapabilities2': () => getBlazorWasmApplyUpdateCapabilities(true),
      'AspNetCoreHotReloadApplied': () => aspnetCoreHotReloadApplied()
    };

    if (payload.type && action.hasOwnProperty(payload.type)) {
      action[payload.type]();
    } else {
      console.error('Unknown payload:', message.data);
    }
  }
}
```

1

2

BrowserRefresh: aspnetcore-browser-refresh.js

```
async function updateCssByPath(path) {
  const styleElement = document.querySelector(`link[href^="${path}"]`) ||
    document.querySelector(`link[href^="${document.baseURI}${path}"]`);

  // Receive a Clear-site-data header.
  await fetch('/_framework/clear-browser-cache');

  if (!styleElement || !styleElement.parentNode) {
    updateAllLocalCss();
  }

  updateCssElement(styleElement);
}

function updateAllLocalCss() {
  [...document.querySelectorAll('link')]
    .filter(l => l.baseURI == document.baseURI)
    .forEach(e => updateCssElement(e));
}
```

1

```
function updateCssElement(styleElement) {
  const newElement = styleElement.cloneNode();
  const href = styleElement.href;
  newElement.href = href.split('?', 1)[0] +
`?nonce=${Date.now()}`;

  styleElement.loading = true;
  newElement.loading = true;
  newElement.addEventListener('load', function () {
    newElement.loading = false;
    styleElement.remove();
  });

  styleElement.parentNode.insertBefore(newElement,
styleElement.nextSibling);
}
```

2

3

4

BrowserRefresh: aspnetcore-browser-refresh.js

```
async function applyBlazorDeltas(serverSecret, deltas, sendErrorToClient) {
  let applyError = undefined;
  deltas.forEach(d => {
    try {
      window.Blazor._internal.applyHotReload(d.moduleId, d.metadataDelta, d.ilDelta, d.pdbDelta)
    } catch (error) {
      console.warn(error);
      applyError = error;
    }
  });

  try {
    await fetch('/_framework/blazor-hotreload', { method: 'post', headers: { 'content-type': 'application/json' }, body:
JSON.stringify(deltas) });
  } catch (error) {
    console.warn(error);
    applyError = error;
  }

  if (applyError) {
    sendDeltaNotApplied(sendErrorToClient ? applyError : undefined);
  } else {
    sendDeltaApplied();
    notifyHotReloadApplied();
  }
}
```

BrowserRefresh: blazor-hotreload.js

```
export function receiveHotReload() {
  return BINDING.js_to_mono_obj(new Promise((resolve) => receiveHotReloadAsync().then(resolve(0))));
}

export async function receiveHotReloadAsync() {
  const response = await fetch('/_framework/blazor-hotreload');
  if (response.status === 200) {
    const deltas = await response.json();
    if (deltas) {
      try {
        deltas.forEach(d => window.Blazor._internal.applyHotReload(d.moduleId, d.metadataDelta, d.ilDelta));
      } catch (error) {
        console.warn(error);
        return;
      }
    }
  }
}
```

1

2

BrowserRefresh: .NET Client-side Interop

```
namespace Microsoft.AspNetCore.Components.WebAssembly.HotReload;

[EditorBrowsable(EditorBrowsableState.Never)]
public static partial class WebAssemblyHotReload {
    private static HotReloadAgent? _hotReloadAgent;
    private static readonly UpdateDelta[] _updateDeltas = new[] { new UpdateDelta() };

    internal static async Task InitializeAsync() {
        await JSHost.ImportAsync("blazor-hotreload", "/_framework/blazor-hotreload.js");
        await ReceiveHotReloadAsync();
    }

    [JSInvokable(nameof(ApplyHotReloadDelta))]
    public static void ApplyHotReloadDelta(string moduleIdString, byte[] metadataDelta, byte[] ilDelta, byte[] pdbBytes, int[]?
updatedTypes) {
        Interlocked.CompareExchange(ref _hotReloadAgent, new HotReloadAgent(m => Debug.WriteLine(m)), null);

        _updateDeltas[0].ModuleId = Guid.Parse(moduleIdString, CultureInfo.InvariantCulture);
        _updateDeltas[0].MetadataDelta = metadataDelta;
        _updateDeltas[0].ILDelta = ilDelta;
        _updateDeltas[0].PdbBytes = pdbBytes;
        _updateDeltas[0].UpdatedTypes = updatedTypes;

        _hotReloadAgent.ApplyDeltas(_updateDeltas);
    }

    [JSImport("receiveHotReloadAsync","blazor-hotreload")]
    private static partial Task ReceiveHotReloadAsync();
}
```

1

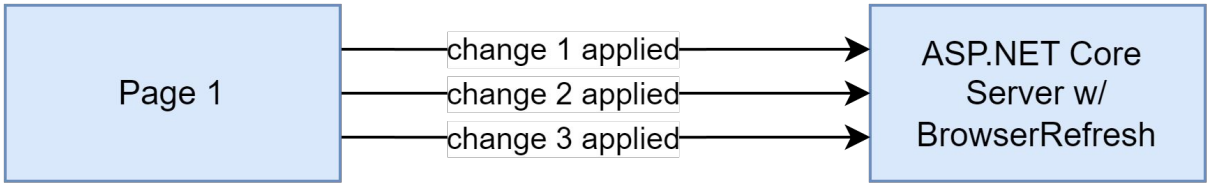
2

3

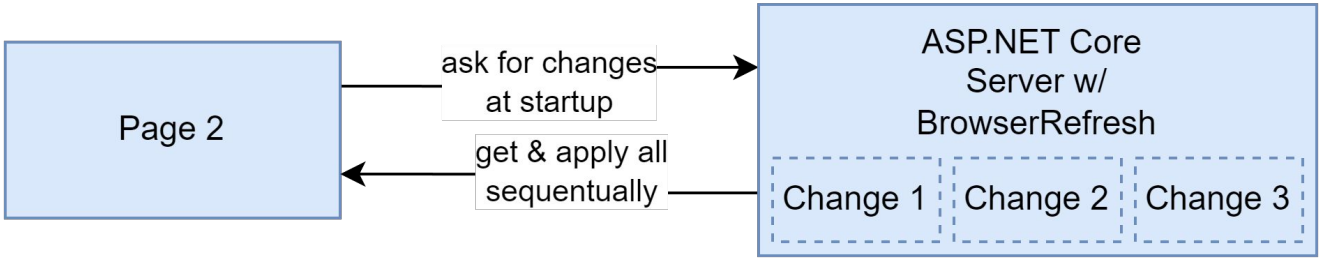
4

BrowserRefresh: Aggregating deltas for new clients

1. Already opened page notifies backend about each applied change



2. Newly opened page (or refreshed one) asks for changes at startup and applies all of them one by one



A Great Journey of changes from IDE to the browser's CLR

1. **Roslyn** calculates changes on request from **IDE**
2. Changes are applied on **ASP.NET Core server** level (regular CoreCLR hot-reload)
3. **IDE** sends changes to the **BrowserRefresh.Host** via Rd
4. **BrowserRefresh.Host** broadcasts changes to each connected client page with the **aspnetcore-browser-refresh** script on board using WebSockets
5. An **aspnetcore-browser-refresh** calls the managed wrapper **WebAssemblyHotReload.ApplyHotReloadDelta** using JS-to-.NET interop
6. **ApplyHotReloadDelta** calls System.Reflection.Metadata.MetadataUpdater.ApplyUpdate and passes changes there
7. After successful apply, **aspnetcore-browser-refresh** sends changes to the ASP.NET Core server for future clients

Summary

- Hot reload is really crucial for UI development
- Blazor Wasm hot-reload consists of 2 stages - server hot-reload and client hot-reload
- Runtimes on new/refreshed pages will receive changes from previous pages's runtimes
- Static file updates are made differently from regular .NET code update
- Server side hot-reload infrastructure is injected into app's server using startup hooks and hosting startup assemblies technic
- Client side hot-reload infrastructure is injected directly into index.html and represented in several JS files with interop calls to .NET side
- Messaging and delta applying between IDE and Blazor client is done indirectly using WebSockets and Rd through BrowserRefresh.Host

Thanks for your attention!

Materials (slides & links):

github.com/seclerp/dotnet-days-24-materials