.NET
fwdays

HIDDEN DIFFICULTIES OF DEBUGGER IMPLEMENTATION FOR .NET WASM APPS

Andrii Rublov

JetBrains, Rider Team

About me

- **Software Developer** at **JetBrains**
- Mainly working on Rider's
  - .NET WASM debugging infrastructure
  - EF / EF Core tooling
  - Run / Debug configurations

- Recent open-source projects
  - **wasmer-dotnet**: .NET bindings for Wasmer WebAssembly Runtime
  - **rider-efcore**: EF Core plugin for Rider
  - **rider-monogame**: MonoGame plugin for Rider

- GitHub: **@seclerp**
- Twitter: **@seclerp**
- Blog: **blog.seclerp.me**

Prologue: About .NET WebAssembly

.NET WebAssembly Family

- **Blazor WebAssembly**
  - DevServer-hosted
  - ASP.NET Core-hosted
- **`wasm-experimental` workload**
  - `browser-wasm` RID
  - `console-wasm` RID
- **`wasi-experimental` workload**
  - `browser-wasi` RID
- **`Wasi.Sdk`**
- **NativeAOT-LLVM WASM/WASI**

Chapter 1: .NET WebAssembly App’s Anatomy

.NET WebAssembly App's Anatomy: Blazor

```
> dotnet new blazorwasm
--name BlazorApp1

> cd BlazorApp1/BlazorApp1
```

```
> cat wwwroot/index.html
<!DOCTYPE html>
...
<body>
  <div id="app">
      ...
  </div>
    ...
  <script
src="_framework/blazor.webassembly.js"></script>
</body>
</html>
```

.NET WebAssembly App’s Anatomy: Blazor

```
> dotnet build

> cd
bin/Debug/net7.0/wwwroot/_framework

> ls
blazor.boot.json        ← 1
blazor.webassembly.js   ← 2
BlazorApp1.dll          ← 3
BlazorApp1.pdb
dotnet.wasm             ← 4
mscorlib.dll
...
```

```
> cat blazor.boot.json
{
  ...
  "entryAssembly": "BlazorApp1",
  "resources": {
    "runtime": {
      "dotnet.wasm":
"sha256-6u4NhRISP...",
    },
    "runtimeAssets": {
      "dotnet.wasm": {
        "behavior": "dotnetwasm",
        "hash": "sha256-6u4NhRISP..."
      }
    ...
    }
  }
```

.NET WebAssembly App’s Anatomy: Blazor

```
> cd ../../../../

> dotnet run
```

Process tree:

```
dotnet: run
└── dotnet:
"~\.nuget\packages\microsoft.aspnetcore.components.webassembly.devserver\7.0.5/
tools/blazor-devserver.dll" --applicationpath
"...\BlazorApp1\bin\Debug\net7.0\BlazorApp1.dll"
```

.NET WebAssembly App’s Anatomy: wasm-experimental

```
> dotnet workload install
wasm-tools
wasm-experimental

> dotnet new wasmbrowser
--name WasmApp1

> cd WasmApp1/WasmApp1
```

```
> cat index.html
<!DOCTYPE html>
<html>
<head>
  ...
  <script type='module' src="./main.js"></script>
</head>
<body>
  <span id="out"></span>
</body>
</html>
```

.NET WebAssembly App's Anatomy: wasm-experimental

```
> cat main.js
import { dotnet } from './dotnet.js'

...

await dotnet.run();
```

.NET WebAssembly App’s Anatomy: wasm-experimental

```
> dotnet build

> cd bin/Debug/net7.0/AppBundle

> ls
managed/                      ← 3
dotnet.js                     ← 2
dotnet.js.symbols
dotnet.wasm                   ← 4
index.html
main.js
mono-config.json              ← 1
WasmApp1.runtimeconfig.json ← 1
...
```

```
> cat mono-config.json
{
  "mainAssemblyName": "WasmApp1.dll",
  "assemblyRootFolder": "managed",
  "debugLevel": -1,
  "remoteSources": [],
  ...
  "assetsHash": "sha256-zTDnY1om..."
}
```

.NET WebAssembly App’s Anatomy: wasm-experimental

```
> cat WasmApp1.runtimeconfig.json
{
  "runtimeOptions": {
    "tfm": "net8.0",
    "wasmHostProperties": {
      "perHostConfig": [
        {
          "name": "browser",
          "html-path": "index.html",
          "Host": "browser"
        }
      ],
      "runtimeArgs": [],
      "mainAssembly": "WasmApp1.dll"
    },
  }
}
```

.NET WebAssembly App’s Anatomy: wasm-experimental

```
> cd ../../../../
```

```
> dotnet run
WasmAppHost --runtime-config
bin\Debug\net8.0\browser-wasm\AppBundle\WasmApp1.runtimeconfig.json

App url: ...
```

Process tree:

```
dotnet: run
└── dotnet: exec "C:\Program
Files\dotnet\packs\Microsoft.NET.Runtime.WebAssembly.Sdk\8.0.0-preview.4.23259.
5\WasmAppHost\WasmAppHost.dll" --runtime-config
"D:\Playground\WasmApp1\WasmApp1\bin\Debug\net8.0\browser-wasm\AppBundle\WasmAp
p1.runtimeconfig.json"
```

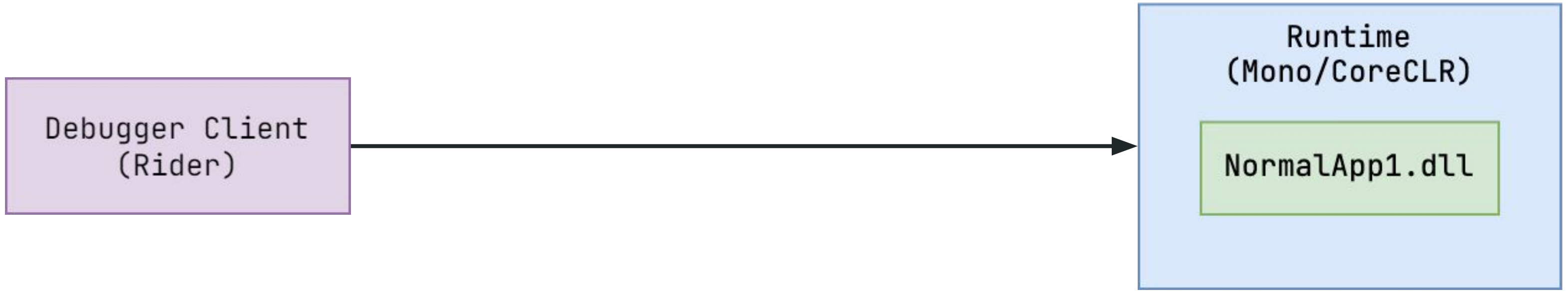
Quick intermediate summary

.NET WebAssembly app:

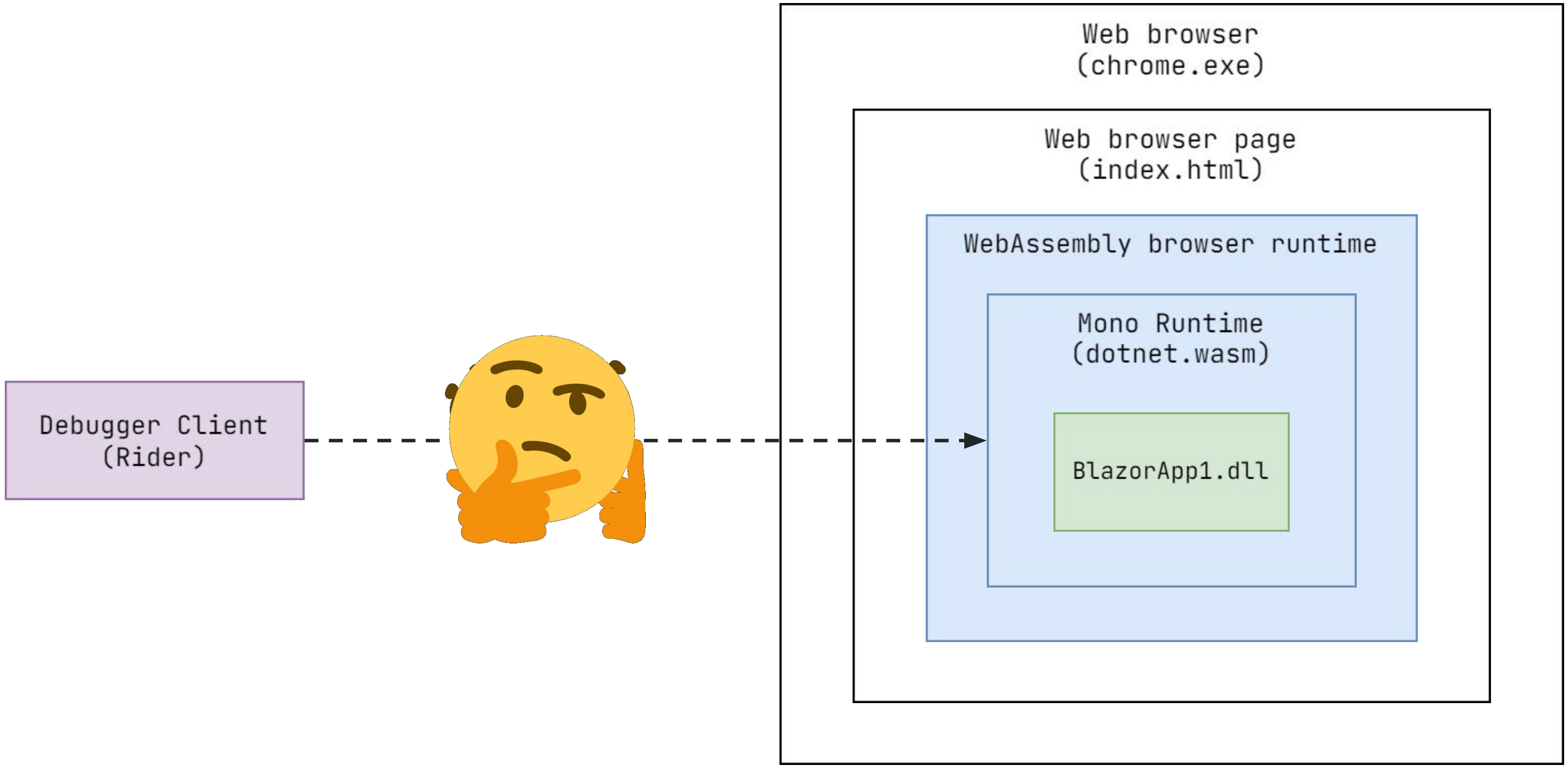
- Runs on Mono runtime*
- Has some JS glue code between the app and runtime
- Has different hosting models and toolchains:
  - DevServer
  - WasmAppHost
  - ASP.NET Core
- **How debugger communicates with the runtime?** 🤔
- **How the runtime communicates with the browser and vice-versa?** 🤔

* Except NativeAOT-LLVM WASM, but it's out-of-scope for this talk, it's very experimental right now

Debugging of regular .NET Apps

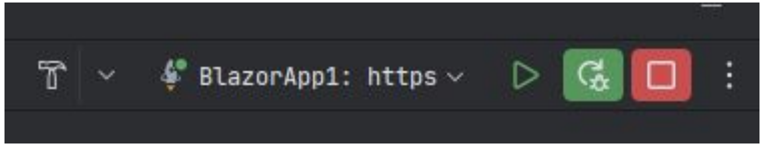


Debugging of .NET WASM Apps



Chapter 2: The Debug Proxy

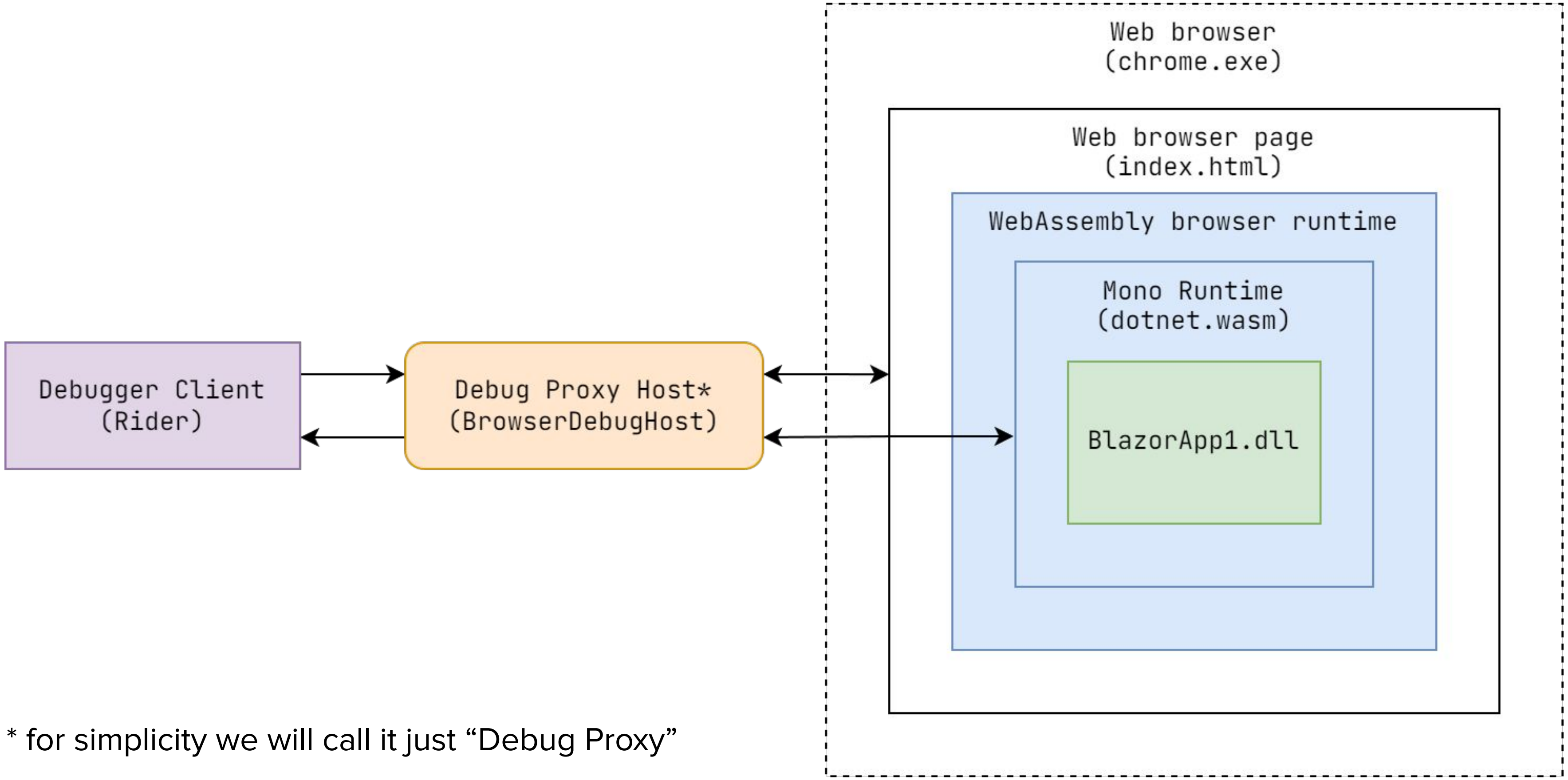
Meet Mono Proxy aka Debug Proxy



Process tree:

```
Rider.Backend.exe: …
└── winpty-agent.exe: …
    └── dotnet:
~/.nuget/packages/microsoft.aspnetcore.components.webassembly.devserver/7.0.5/
tools/blazor-devserver.dll --applicationpath bin\Debug\net7.0\BlazorApp1.dll
        └── dotnet: exec
"~\.nuget\packages\microsoft.aspnetcore.components.webassembly.devserver\7.0.5
\tools\BlazorDebugProxy\BrowserDebugHost.dll" --OwnerPid 16152 --DevToolsUrl
http://127.0.0.1:64069
```

Meet Mono Proxy aka Debug Proxy



* for simplicity we will call it just “Debug Proxy”

Meet Mono Proxy aka Debug Proxy

- **Debugger Client** doesn’t have a direct connection to a **browser page**
- **Debug Proxy** acts a mediator role in communication flow
- **Debug Proxy** proxifies JS app-related events from a **browser page** to a **debugger client** and JS related function calls from the **debugger client** to a **browser page**
- **Debug Proxy** listens for Mono JS events from Mono runtime and controls it’s behavior by calls via dotnet.js API
- **Debug Proxy API** is **not documented** and it’s **quite unstable**

Meet Mono Proxy aka Debug Proxy

Browser Page
Debug Proxy
Debugger Client

How communication between there 3 parts is organized? Which protocol is used? 🤔

Chapter 3: The Protocol

A Handshake Process

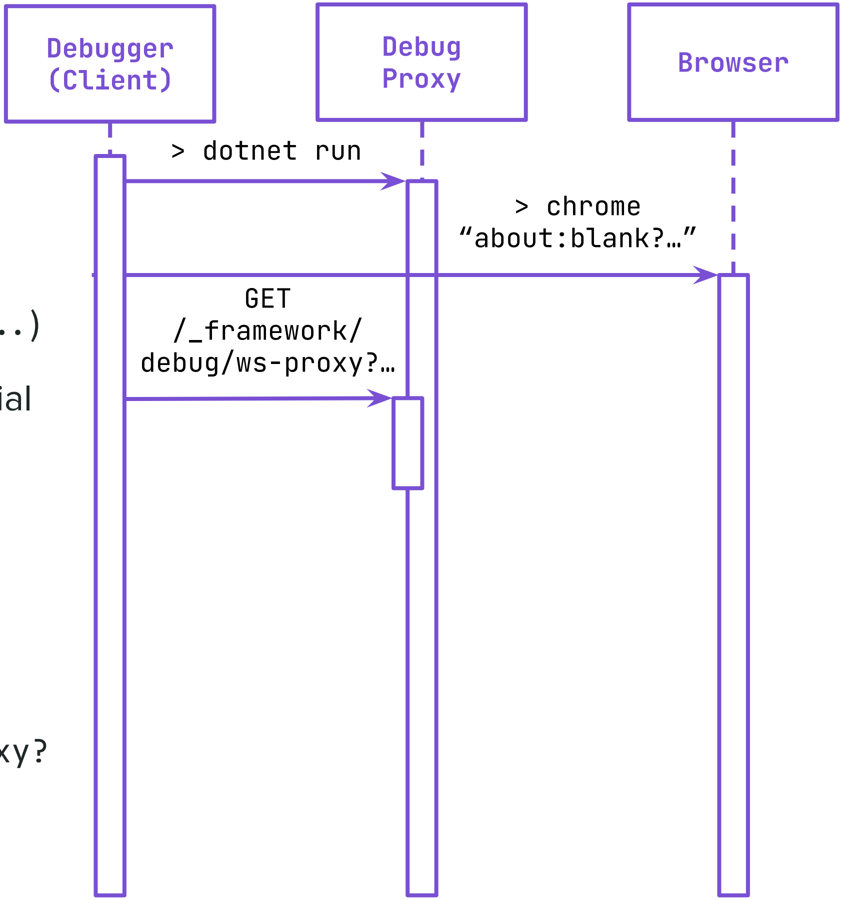
1. **Debugger**: starts a WasmApp1 process (DevServer or ASP.NET Core, depending on the hosting option), **which also starts Debug Proxy as a child process**

2. **Debugger**: starts a compatible **browser** (Chrome/Edge), **with a special placeholder URL** (`about:blank?realUrl=...`)

3. **Browser**: dumps it's debugging **port** and **path** to a special file in user's profile folder.

4. **Debugger**: constructs **debugging endpoint** like that:
`ws://127.0.0.1:{port}/{path}`

5. **Debugger**: sends a **debugging endpoint** to a special private URL called **inspect url**:
```
GET http://localhost:5170/_framework/debug/ws-proxy?
browser=ws://127.0.0.1:{port}/{path}
```



A Handshake Process

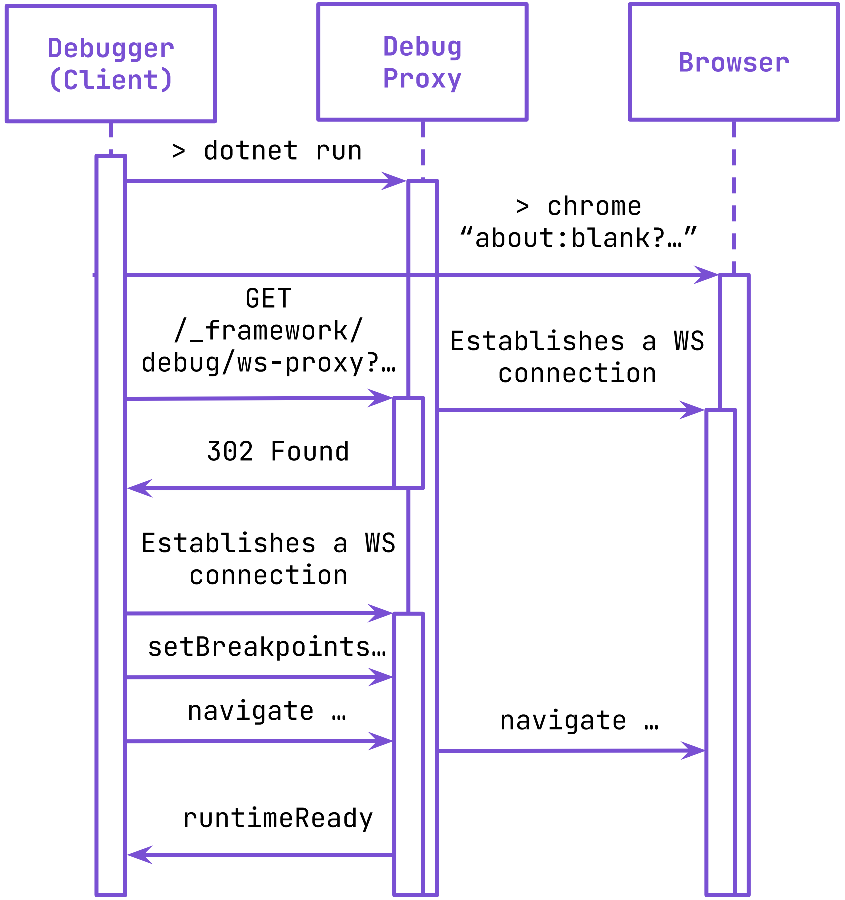
6. **Debug Proxy**: initializes a WebSocket connection to the browser by given **browser debugging endpoint**, returns a new **proxy debugging endpoint** (with similar shape but with a new port) as a 302 Redirect status code

7. **Debugger**: initializes a WebSocket connection to the debug proxy by a received **proxy debugging endpoint**

8. **Debugger**: sends **breakpoint requests** to be resolved by Mono runtime **once ready**

9. **Debugger**: "resolves" a real URL **from a placeholder** URL (`about:blank?…`) and navigates a page to it

10. **Debug Proxy**: sends **special event** indicating that Mono **runtime is ready** to accept .NET specific requests



Quick intermediate summary

- Handshake process is quite complex because of a lot of moving parts

- **A hack with placeholder URLs** exists because we need some controlled time between Debug Proxy initialization and Mono runtime initialization to make preparations (like **setting breakpoints before they will be reached**)

- It’s impossible to run more than 1 instance of Chromium browser under the same user profile folder (because in such a case they will have conflict because of use of identical port)

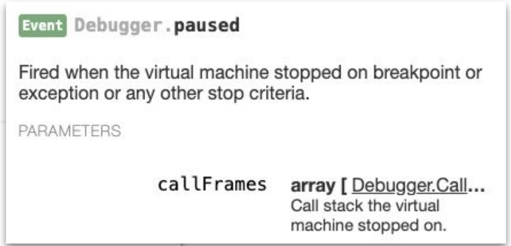
For which communication process WebSocket connections are established?
🤔

CDP aka Chrome DevTools Protocol: Definition

- Defined by a set of modules, “domains”
- Each **domain** may contain:
    - **Types**: transferred data records
    - **Methods**: client-issued calls to the CDP server (browser, Mono Proxy, etc.)

    Events: server-issued notifications to the client

Examples

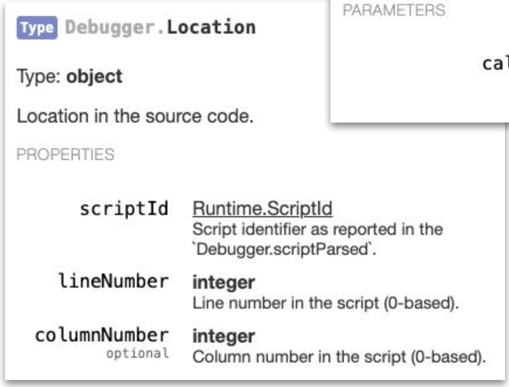


Event Debugger.paused

Fired when the virtual machine stopped on breakpoint or exception or any other stop criteria.

PARAMETERS

callFrames	array [Debugger.Call... Call stack the virtual machine stopped on.



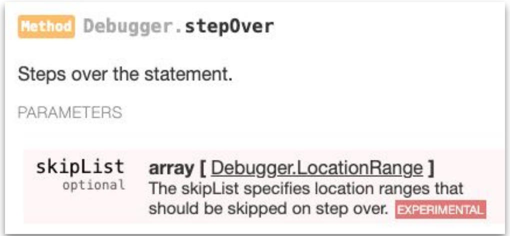
Type Debugger.Location

Type: **object**

Location in the source code.

PROPERTIES

scriptId	Runtime.ScriptId Script identifier as reported in the `Debugger.scriptParsed`.
lineNumber	**integer** Line number in the script (0-based).
columnNumber optional	**integer** Column number in the script (0-based).



Method Debugger.stepOver

Steps over the statement.

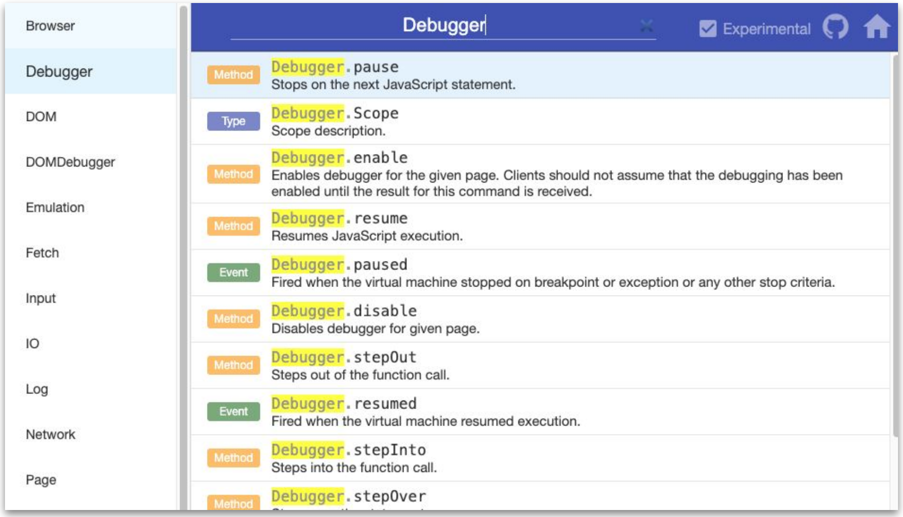
PARAMETERS

skipList optional	**array [** Debugger.LocationRange **]** The skipList specifies location ranges that should be skipped on step over. EXPERIMENTAL

CDP aka Chrome DevTools Protocol: Explore

API Explorer

- Official: chromedevtools.github.io/devtools-protocol
- Alternative: vanilla.aslushnikov.com
- Debug Proxy specific: mono-cdp.seclerp.me



Method Debugger.pause
Stops on the next JavaScript statement.

Type Debugger.Scope
Scope description.

Method Debugger.enable
Enables debugger for the given page. Clients should not assume that the debugging has been enabled until the result for this command is received.

Method Debugger.resume
Resumes JavaScript execution.

Event Debugger.paused
Fired when the virtual machine stopped on breakpoint or exception or any other stop criteria.

Method Debugger.disable
Disables debugger for given page.

Method Debugger.stepOut
Steps out of the function call.

Event Debugger.resumed
Fired when the virtual machine resumed execution.

Method Debugger.stepInto
Steps into the function call.

Method Debugger.stepOver

CDP aka Chrome DevTools Protocol: Transport

- Works over WebSocket
- Based on JSON-RPC 2.0*
- Messages are strictly ordered
- Supports buffering
- Supports 3 types of messages:
    - **Requests** (client → server, ordered)
    - **Responses** (client → server, ordered)
    - **Events** (server → client, unordered)
- Supports sessions (in our case, session per browser tab)

Examples

- Request:
```
{"id":10, "method": "Page.navigate",
"params":{"url":"http://localhost:5170/"
}}
```
- Response:
```
{"id":10, “result”:
{"frameId":"…","loaderId":"…"}}
```
- Event:
```
{"method":
"Network.requestServedFromCache",
"params":{"requestId":"98279.21"}}
```

CDP aka Chrome DevTools Protocol: Targets

- Target defines anything that debugger could be attached to. Examples:
  - **A browser**
  - **A page**
  - A service worker
  - A background page
  - ...
- One target session could be created from another (e.g. **page target** session from **browser target** session)

Chapter 4: Implementation

CDP Client

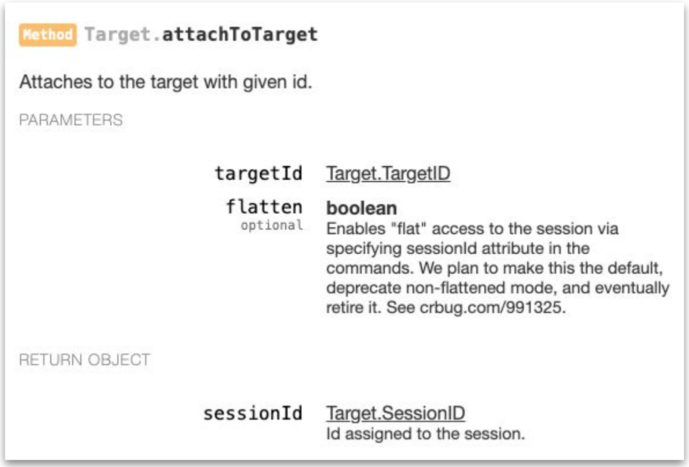
```
// Creating connection
var connection = new DefaultProtocolClient(new Uri("ws://localhost:5151"), logger);
await connection.ConnectAsync(cancellationToken);

// Sending commands
var response = await connection.SendCommandAsync(
    Domains.DotnetDebugger.SetDebuggerProperty(
        JustMyCodeStepping: true
    )
);

// Firing commands (when we're not interested in response)
await connection.FireCommandAsync(Domains.Debugger.StepOut());
```

CDP Client

```
// Listening for events
pageClient.ListenEvent<Domains.Debugger.BreakpointResolved>(async e ⇒
{
    ResolveBreakpoint(e.BreakpointId.Value);
});

// Creating scoped clients (clients for specific sessions)
var scopedClient = connection.CreateScoped(sessionId);
```

Sample: Page connection initialization

```
var result = await connection.SendCommandAsync(
    Domains.Target.AttachToTarget(
        TargetId: placeholderTarget.TargetId,
        // Non-flatten mode will be deprecated in the future
        Flatten: true));

var pageConnection = connection.CreateScoped(result.SessionId.Value);

logger.LogInformation("Initializing debugger-related domains...");


await Task.WhenAll(
    pageConnection.SendCommandAsync(Domains.Debugger.Enable()),
    pageConnection.SendCommandAsync(Domains.Log.Enable()),
    pageConnection.SendCommandAsync(Domains.Runtime.Enable()),
    pageConnection.SendCommandAsync(Domains.Page.Enable()),
    pageConnection.SendCommandAsync(Domains.Network.Enable())
);
```



Method Target.attachToTarget

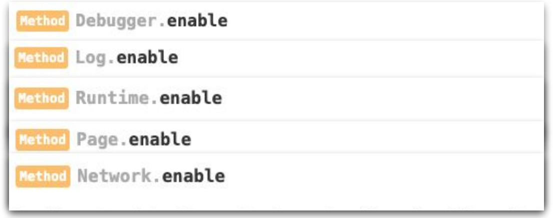
Attaches to the target with given id.

PARAMETERS

targetId	Target.TargetID
flatten optional	**boolean** Enables "flat" access to the session via specifying sessionId attribute in the commands. We plan to make this the default, deprecate non-flattened mode, and eventually retire it. See crbug.com/991325.

RETURN OBJECT

sessionId	Target.SessionID Id assigned to the session.



Method Debugger.enable

Method Log.enable

Method Runtime.enable

Method Page.enable

Method Network.enable

Sending Messages

```
private readonly BlockingCollection<ProtocolRequest<ICommand>> _outgoingMessages = …

public async Task<TResponse> SendCommandAsync<TResponse>(ICommand<TResponse> command,
  string? sessionId = null,
  CancellationToken? token = default) where TResponse : IType
{
  var id = Interlocked.Increment(ref _currentId);
  var resolver = new TaskCompletionSource<JObject>();
  if (_responseResolvers.TryAdd(id, resolver))
  {
    await FireInternalAsync(id, GetMethodName(command.GetType()), command, sessionId);
    var responseRaw = await resolver.Task;
    var response = responseRaw.ToObject...
    return response;
  }
  throw new Exception("Unable to enqueue message to send");
}
```

```
private async Task
StartOutgoingWorker(CancellationToken token)
{
  _logger.LogInformation("Starting outgoing
messages pump...");
  while (!token.IsCancellationRequested)
  {
    var message = _outgoingMessages.Take();
    await ProcessOutgoingRequest(message);
  }
}
```

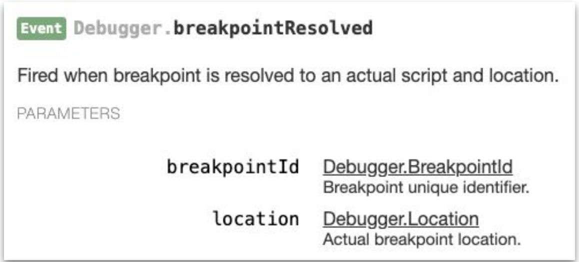
```
private async Task FireInternalAsync(int id, string methodName, ICommand command, string? sessionId)
{
  var request = new ProtocolRequest<ICommand>(id, methodName, command, sessionId);
  if (!_outgoingMessages.TryAdd(request)) throw new Exception("Can't schedule outgoing message for sending.");
}
```

Retrieving Messages

```csharp
private Task ProcessIncoming(string message) ⇒
  DeserializeMessage(message) switch
  {
    ProtocolResponse<JObject> response ⇒ ProcessIncomingResponse(response),
    ProtocolEvent<JObject> @event ⇒ ProcessIncomingEvent(@event),
    _ ⇒ Task.CompletedTask
  };

private async Task ProcessIncomingEvent(ProtocolEvent<JObject> @event)
{
  OnEventReceived?.Invoke(this, @event);
  if (_eventHandlers.TryGetValue(@event.Method, out var handler))
    await handler.Invoke(@event);
}

private async Task ProcessIncomingResponse(ProtocolResponse<JObject> response)
{
  OnResponseReceived?.Invoke(this, response);
  _responseResolvers.TryRemove(response.Id, out var resolver);

  if (response.Error is { } error) resolver?.SetException(new ProtocolErrorException(error));
  if (response.Result is { } result) resolver?.SetResult(result);
}
```

Sample: Set, Remove & Resolve Breakpoints

```
pageClient.ListenEvent<Domains.Debugger.BreakpointResolved>(async e ⇒
{
    ResolveBreakpoint(e.BreakpointId.Value);
});

private void ResolveBreakpoint(string breakpointId)
{
    if (_breakpointsStorage.TryGetBreakEventInfo(breakpointId, out var info))
    {
        var bindingBreakEvent = new WasmBindingBreakEvent(info.BreakEvent, WasmModule.Instance);

        if (!info.AddBindingBreakEvent(bindingBreakEvent))
        {
            _logger.LogInformation($"{bindingBreakEvent} is not added to {info}");
            info.SetStatus(BreakEventStatus.NotBound, $"Could not insert breakpoint {info.BreakEvent}");
        }
        else
            info.SetStatus(BreakEventStatus.Bound, null);
    }
}
```


Event Debugger.breakpointResolved

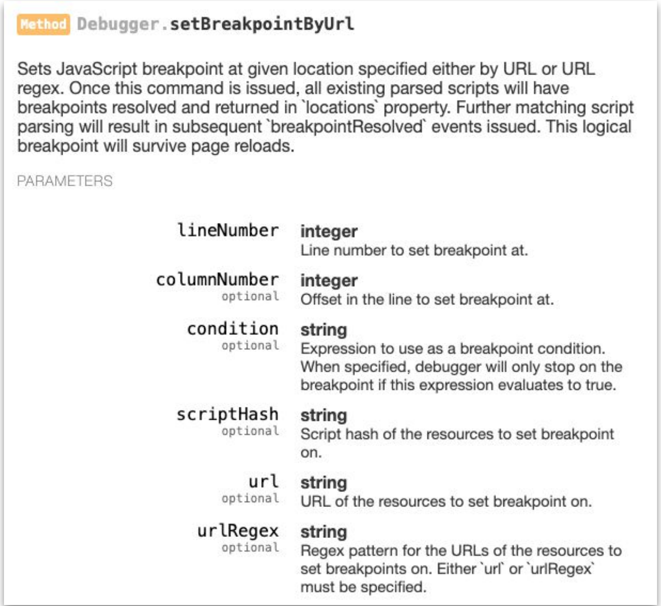
Fired when breakpoint is resolved to an actual script and location.

PARAMETERS

breakpointId	Debugger.BreakpointId Breakpoint unique identifier.
location	Debugger.Location Actual breakpoint location.

Sample: Set, Remove & Resolve Breakpoints

```
async Task HandleAddBreakpointRequest(BreakEventInfo<WasmModule> info)
{
    // ...

    // Map 'C:\Foo\bar' to 'file:///C:/Foo/bar'
    var fileUrl = new Uri(url.Path).ToString();
    var (protocolLine, protocolColumn) =
        WasmProxyLocationMapper.ToMonoProxyUnits(line, column);
    var (breakpointId, locations) = await pageClient.SendCommandAsync(
        Domains.Debugger.SetBreakpointByUrl(
            LineNumber: protocolLine, ColumnNumber: protocolColumn, Url: fileUrl
        ));

    if (!_breakpointsStorage.TryAdd(breakpointId.Value, info))
        _logger.LogWarning("Can't add breakpoint '{Info}' with ID '{BreakpointId}'", info, breakpointId.Value);

    // Check maybe we already know script with resolved script ID
    foreach (var location in locations)
        if (_scriptsStorage.IsLoaded(location.ScriptId.Value))
            ResolveBreakpoint(breakpointId.Value);
}
```



Method Debugger.setBreakpointByUrl

Sets JavaScript breakpoint at given location specified either by URL or URL regex. Once this command is issued, all existing parsed scripts will have breakpoints resolved and returned in `locations` property. Further matching script parsing will result in subsequent `breakpointResolved` events issued. This logical breakpoint will survive page reloads.

PARAMETERS

lineNumber	**integer** Line number to set breakpoint at.
columnNumber optional	**integer** Offset in the line to set breakpoint at.
condition optional	**string** Expression to use as a breakpoint condition. When specified, debugger will only stop on the breakpoint if this expression evaluates to true.
scriptHash optional	**string** Script hash of the resources to set breakpoint on.
url optional	**string** URL of the resources to set breakpoint on.
urlRegex optional	**string** Regex pattern for the URLs of the resources to set breakpoints on. Either `url` or `urlRegex` must be specified.

Bonus: Few words about Hot-Reload

Hot-Reload

- Without debugging: **dotnet watch** and related infrastructure
- With debugging (EnC): Available in **Debug Proxy** since .NET SDK 7 (not yet in Rider 🥵)
- EnC follows the following algorithm:
  1. User pauses execution for some reason (breakpoint, manually, …)
  2. User changes things in code…
  3. User hits Continue or Apply changes
  4. Delta is computed (IL delta, metadata delta and PDB delta)
  5. Debugger sends delta to the runtime (via Debug Proxy or other mechanism)
  6. Runtime applies deltas
  7. Debugger resets breakpoints (as lines in code have possibly been changed)
  8. Debugger resumes execution

Thanks for your attention!

Links

- WebAssembly
  - WebAssembly System Interface: wasi.dev
- Chrome DevTools Protocol
  - API Explorer: chromedevtools.github.io/devtools-protocol
  - API Explorer (alternative): vanilla.aslushnikov.com
  - Mono Extension Explorer: mono-cdp.seclerp.me
- Blog posts
  - The Future of .NET with WASM by Khalid Abuhakmeh
- Videos
  - Blazor United prototype by Steven Sanderson
  - Experiments with the new WASI workload in .NET 8 Preview 4 by Steven Sanderson