🟦 Blazor — Fundamentals

Introduction to Microsoft’s modern web framework

Agenda

1. **What is Blazor**
2. **Hosting models** — Server (SignalR), WebAssembly (WASM)
3. **What is WebAssembly (WASM)**
4. **S-Expression format**
5. **Hello World in WebAssembly**

1️⃣ What is Blazor?

Blazor is a **.NET web framework** for building **interactive client-side web UIs** using **C# instead of JavaScript**.

Key idea:

> "Write C#, run in the browser."

How Blazor works

Blazor apps are built from **components**:

- `.razor` files (HTML + C#)
- Each component has:
  - UI markup
  - State
  - Event handlers

Example:

```
<h3>Counter: @count</h3>
<button @onclick="Increment">Add</button>

@code {
    int count = 0;
    void Increment() => count++;
}
```

Why Blazor?

✅ C# full-stack development

✅ Reuse .NET libraries

✅ Strong typing & tooling

✅ Integration with existing .NET ecosystem

✅ Runs either **server-side** or **client-side (WebAssembly)**

2️⃣ Blazor Hosting Models

Blazor supports multiple **hosting models** that define *where* the app executes and *how* it communicates with the browser.

🔹 Blazor Server (SignalR)

Execution: on the server

Communication: via **SignalR** WebSocket connection

DOM updates: streamed to the browser

```
[ Browser ]
    ▲     │
    │     │  SignalR (diff updates)
    ▼     │
[ Blazor Server (.NET) ]
```

Pros:

- Small client download
- Centralized logic and security
- Realtime updates

Cons:

🔶 Blazor WebAssembly (Client-side)

Execution: in the browser

Runtime: .NET IL compiled to **WebAssembly**

```
[ Browser ]
   ├── .NET runtime (WASM)
   ├── App DLLs
   └── Runs entirely on client
```

Pros:

- Offline support
- Scalable (no server state)
- Client-side performance

Cons:

- Larger download

🧩 Summary of Hosting Models

Model	Runs On	Communication	Offline	Latency
Blazor Server	Server	SignalR	❌	Higher
Blazor WebAssembly	Browser	HTTP (API calls)	✅	Low

3️⃣ What is WebAssembly (WASM)

WebAssembly (WASM) is a **low-level binary format** that runs at near-native speed inside modern browsers.

- Runs in a **sandboxed environment**
- Designed for **performance-critical** applications
- Can be compiled from many languages:
  - C/C++
  - Rust
  - Go
  - C# (via .NET)

WASM in the browser stack

```
+------------------------+
|   JavaScript           |
|   WebAssembly (WASM)   |  ←  Blazor runs here
+------------------------+
|   Browser Engine       |
|   (V8, SpiderMonkey)   |
+------------------------+
|   Operating System     |
+------------------------+
```

➡️ Think of it as a **portable, sandboxed CPU** inside the browser.

Key features of WebAssembly

- Binary instruction format
- Safe and portable
- Stack-based virtual machine
- Designed for compilation
- Supported by all major browsers

4️⃣ S-Expression Format

WebAssembly text format (.wat) uses **S-Expressions** (like Lisp).

They are **parenthesized expressions** representing instructions and structure.

Example:

```
(module
  (func $add (param $x i32) (param $y i32) (result i32)
    local.get $x
    local.get $y
    i32.add)
  (export "add" (func $add))
)
```

Breakdown

Element	Meaning
`(module …)`	Defines a WASM module
`(func $add …)`	Declares a function
`(param $x i32)`	Function parameter
`(i32.add)`	Adds two 32-bit integers
`(export "add")`	Makes function callable from outside

5️⃣ Hello World in WebAssembly

A minimal WebAssembly example in **text format (.wat)**:

```
(module
  (import "env" "print" (func $print (param i32)))
  (memory (export "memory") 1)
  (data (i32.const 0) "Hello, World!\00")

  (func (export "main")
    i32.const 0
    call $print)
)
```

This program:

1. Imports a JS function called `print`
2. Stores “Hello, World!” in memory
3. Calls `print(0)` — JS prints from memory

JavaScript integration

```javascript
const importObject = {
  env: {
    print: (ptr) => {
      const bytes = new Uint8Array(memory.buffer, ptr);
      const text = new TextDecoder('utf8').decode(bytes);
      console.log(text);
    }
  }
};

const response = await fetch('hello.wasm');
const bytes = await response.arrayBuffer();
const { instance } = await WebAssembly.instantiate(bytes, importObject);

instance.exports.main();
```

🚀 Blazor and WASM Together

Blazor uses WebAssembly to:

- Run the **.NET runtime** in the browser
- Load C# assemblies dynamically
- Execute IL code as native WebAssembly instructions

Result:

> ".NET running **natively inside your browser**, without plugins."

🧭 Summary

Concept	Description
Blazor	C#-based web framework
Server Model	Runs on server via SignalR
WASM Model	Runs client-side in browser
WebAssembly	Binary format for web execution
S-Expression	Human-readable WASM text format

💡 Key takeaway

Blazor merges **modern .NET** with **modern browsers**:

- Write C# → Run anywhere
- Powered by **WebAssembly**
- No JavaScript required
- Full control, full performance