

1. Overview of ASP.NET Core SignalR

What is SignalR?

ASP.NET Core SignalR is a **real-time communication framework**.

It allows:

- Server → Client push
- Client → Server calls
- Bi-directional, low-latency messaging

Traditional HTTP vs SignalR

HTTP	SignalR
Request–Response	Persistent connection
Client must poll(Client must refresh regularly)	Server pushes instantly
Stateless	Stateful connection

Transport mechanisms (automatic fallback)

1. **WebSockets** (preferred)
2. **Server-Sent Events**
3. **Long Polling**

Core Concepts

Concept	Meaning
Hub	Central class for client–server communication
Client	Browser / Blazor / JS / Mobile
Method Invocation	Client calls server & vice-versa
ConnectionId	Unique per connected client
Groups	Broadcast to subsets of users

2. ASP.NET Core Blazor + SignalR (Big Picture)

Why SignalR fits Blazor perfectly

- Blazor UI updates are **state-driven**
- SignalR triggers **UI refresh automatically**
- No manual JavaScript polling

Blazor hosting models

Model	SignalR usage
Blazor Server	Uses SignalR internally already
Blazor WebAssembly	Uses SignalR explicitly (client → API)

◆ 3. .NET API (Browser / Backend)

We need:

- A **Web API**
- A **SignalR Hub**
- Endpoint mapping

◆ 4. Create the Solution (Step-by-Step)

Step 1: Create the Backend API

```
dotnet new webapi -n ChatApi
```

```
cd ChatApi
```

Step 2: Create Blazor WebAssembly App

```
dotnet new blazorwasm -n ChatClient
```

Solution structure:

```
ChatSolution
```

```
|  
|--- ChatApi    (SignalR Hub)  
|--- ChatClient (Blazor UI)
```

◆ 5. Add the SignalR Client Library

Backend (API)

Already included:

```
<PackageReference Include="Microsoft.AspNetCore.SignalR" />
```

Blazor WebAssembly

```
dotnet add ChatClient package Microsoft.AspNetCore.SignalR.Client
```

◆ 6. Add a SignalR Hub (Backend)

📁 ChatApi/Hubs/ChatHub.cs

```
using Microsoft.AspNetCore.SignalR;
```

```
namespace ChatApi.Hubs;
```

```
public class ChatHub : Hub  
{  
    public async Task SendMessage(string user, string message)  
    {  
        // Broadcast to ALL connected clients  
        await Clients.All.SendAsync("ReceiveMessage", user, message);  
    }  
}
```

Explain to students

- Hub = communication gateway
 - SendMessage = callable from clients
 - Clients.All = broadcast
-

◆ 7. Add SignalR Services & Endpoint

📁 ChatApi/Program.cs

```
using ChatApi.Hubs;
```

```
var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddControllers();
```

```
builder.Services.AddSignalR();
```

```
builder.Services.AddCors(options =>
```

```
{
```

```
    options.AddDefaultPolicy(policy =>
```

```
{
```

```
    policy
```

```
        .AllowAnyHeader()
```

```
        .AllowAnyMethod()
```

```
        .AllowCredentials()
```

```
        .SetIsOriginAllowed(_ => true);
```

```
});
```

```
});
```

```
var app = builder.Build();
```

```
app.UseCors();  
  
app.MapControllers();  
app.MapHub<ChatHub>("/chathub");  
  
app.Run();
```

Key teaching points

- AddSignalR() registers services
 - MapHub<ChatHub>("/chathub") exposes endpoint
 - CORS is **mandatory** for WASM
-

◆ 8. Create Razor Chat Component (Blazor)

📁 ChatClient/Pages/Chat.razor

```
@page "/chat"  
  
@implements IDisposable  
  
<h3>SignalR Chat</h3>
```

```
<input @bind="user" placeholder="Your name" />  
  
<input @bind="message" placeholder="Message" />  
  
<button @onclick="Send">Send</button>
```

```
<ul>  
  @foreach (var msg in messages)  
  {
```

```
<li>@msg</li>
}
</ul>
```

◆ 9. Chat Component Code-Behind

Chat.razor.cs

```
using Microsoft.AspNetCore.SignalR.Client;
```

```
public partial class Chat
{
    private HubConnection? hubConnection;
    private string user = string.Empty;
    private string message = string.Empty;
    private List<string> messages = new();
```

```
protected override async Task OnInitializedAsync()
{
    hubConnection = new HubConnectionBuilder()
        .WithUrl("https://localhost:5001/chathub")
        .WithAutomaticReconnect()
        .Build();
```

```
hubConnection.On<string, string>("ReceiveMessage", (user, msg) =>
{
    messages.Add($"{user}: {msg}");
    InvokeAsync(StateHasChanged);
```

```
});

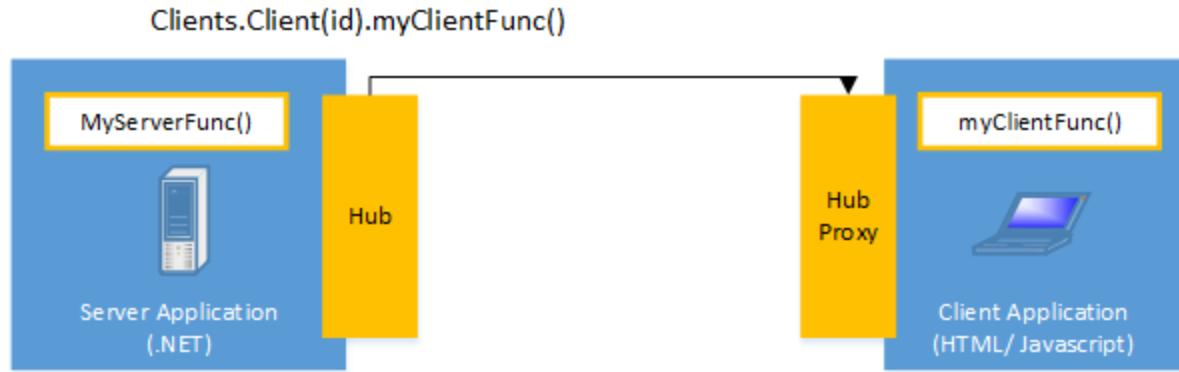
await hubConnection.StartAsync();

}

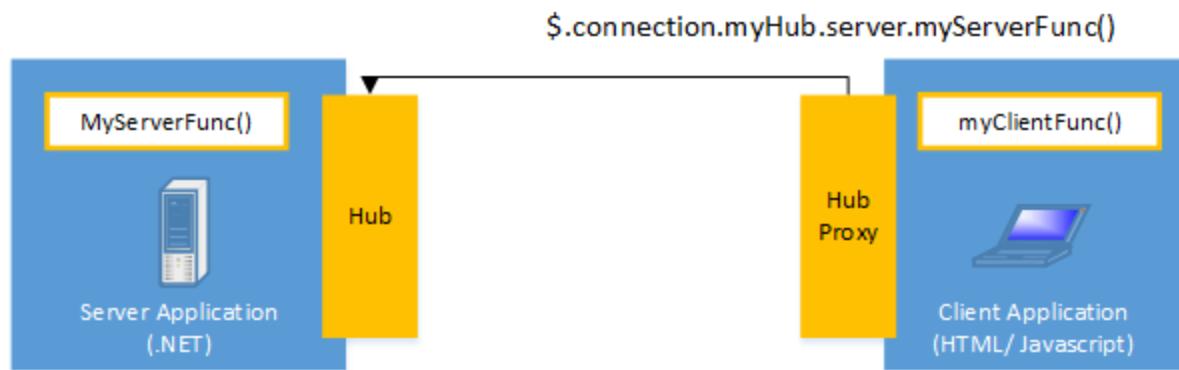
private async Task Send()
{
    if (hubConnection is not null)
    {
        await hubConnection.SendAsync("SendMessage", user, message);
        message = string.Empty;
    }
}

public async ValueTask DisposeAsync()
{
    if (hubConnection is not null)
    {
        await hubConnection.DisposeAsync();
    }
}
```

◆ **10. Explain the Flow (Very Important for Students)**



Server invocation of client method
`myClientFunc()`



Client invocation of server method
`MyServerFunc()`

Step-by-step message flow

1. Blazor client connects to /chathub
2. User clicks **Send**
3. SendMessage() called on Hub
4. Hub broadcasts via Clients.All
5. All clients receive ReceiveMessage
6. UI updates automatically

◆ 11. Run the Demo

Start API

```
dotnet run --project ChatApi
```

Start Blazor

```
dotnet run --project ChatClient
```

Open **two browser windows** → real-time chat ⚡

◆ 12. Common Errors & Fixes

Error	Cause	Fix
404 /chathub	Endpoint not mapped	MapHub missing
CORS error	WASM cross-origin	AllowCredentials()
Messages not updating	UI not refreshed	InvokeAsync(StateHasChanged)
Disconnects	Network drops	WithAutomaticReconnect()

◆ 13. Real-World Use Cases

- Live chat
- Notifications
- Stock prices
- Dashboards
- Multiplayer games
- Collaboration tools