Lab: ASP.NET Core Microservice - SignalR

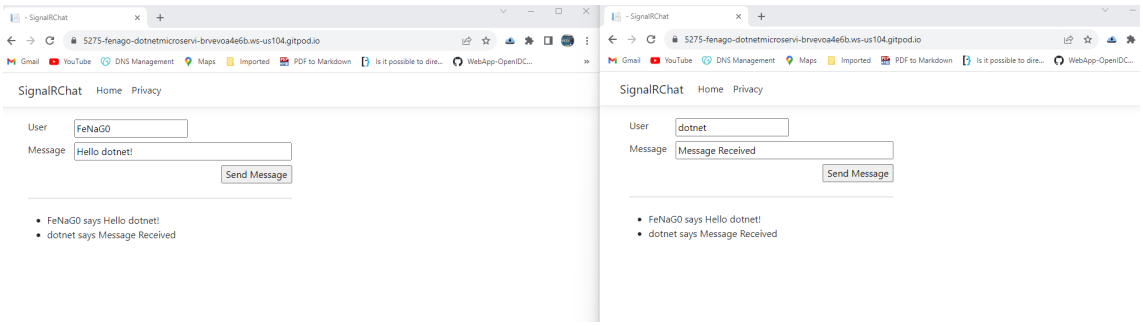
In this lab

1. Prerequisites
2. Create a web app project
3. Add the SignalR client library
4. Create a SignalR hub
5. Configure SignalR
6. Add SignalR client code
7. Run the app

This lab teaches the basics of building a real-time app using SignalR. You learn how to:

- Create a web project.
- Add the SignalR client library.
- Create a SignalR hub.
- Configure the project to use SignalR.
- Add code that sends messages from any client to all connected clients.

At the end, you'll have a working chat app:



Create a web app project

Open the integrated terminal.

Change to the directory (`cd`) that will contain the project.

Run the following commands:

```
dotnet new webapp -o SignalRChat

cd SignalRChat
```

The `dotnet new` command creates a new Razor Pages project in the `SignalRChat` folder.

Opens the SignalRChat folder in the code editor.

Add the SignalR client library

The SignalR server library is included in the ASP.NET Core shared framework. The JavaScript client library isn't automatically included in the project. For this lab, use Library Manager (LibMan) to get the client library from unpkg.

`unpkg` is a fast, global content delivery network for everything on npm.

In the integrated terminal, run the following commands to install LibMan after uninstalling any previous version, if one exists.

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli
```

Navigate to the project folder, which contains the `SignalRChat.csproj` file.

Run the following command to get the SignalR client library by using LibMan. It may take a few seconds before displaying output.

```
libman install @microsoft/signalr@latest -p unpkg -d wwwroot/js/signalr --files
dist/browser/signalr.js
```

Note: Run `export PATH="$PATH:/home/gitpod/.dotnet/tools"` command in the terminal if you get an error.

The parameters specify the following options:

- Use the unpkg provider.
- Copy files to the `wwwroot/js/signalr` destination.
- Copy only the specified files.

The output looks like the following example:

```
wwwroot/js/signalr/dist/browser/signalr.js written to disk
wwwroot/js/signalr/dist/browser/signalr.js written to disk
Installed library "@microsoft/signalr@latest" to "wwwroot/js/signalr"
```

Create a SignalR hub

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

In the SignalRChat project folder, create a `Hubs` folder.

In the `Hubs` folder, create the `ChatHub.cs` file with the following code:

```
using Microsoft.AspNetCore.SignalR;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

The `ChatHub` class inherits from the SignalR [Hub] class. The `Hub` class manages connections, groups, and messaging.

The `SendMessage` method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

Configure SignalR

The SignalR server must be configured to pass SignalR requests to SignalR. Add the following highlighted code to the `Program.cs` file.

```
using SignalRChat.Hubs;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production
scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<ChatHub>("/chatHub");
});

app.Run();
```

The preceding highlighted code adds SignalR to the ASP.NET Core dependency injection and routing systems.

Add SignalR client code

Replace the content in `Pages/Index.cshtml` with the following code:

```
@page
<div class="container">
```

```
    <div class="row p-1">
        <div class="col-1">User</div>
        <div class="col-5"><input type="text" id="userInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-1">Message</div>
        <div class="col-5"><input type="text" class="w-100" id="messageInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-6 text-end">
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row p-1">
        <div class="col-6">
            <hr />
        </div>
    </div>
    <div class="row p-1">
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/js/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

The preceding markup:

- Creates text boxes and a submit button.
- Creates a list with `id="messagesList"` for displaying messages that are received from the SignalR hub.
- Includes script references to SignalR and the `chat.js` app code is created in the next step.

In the `wwwroot/js` folder, create a `chat.js` file with the following code:

```
"use strict";

var connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable the send button until connection is established.
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var li = document.createElement("li");
    document.getElementById("messagesList").appendChild(li);
    // We can assign user-supplied strings to an element's textContent because it
    // is not interpreted as markup. If you're assigning in any other way, you
    // should be aware of possible script injection concerns.
    li.textContent = `${user} says ${message}`;
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
```

```
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function (event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

The preceding JavaScript:

- Creates and starts a connection.
- Adds to the submit button a handler that sends messages to the hub.
- Adds to the connection object a handler that receives messages from the hub and adds them to the list.

Run the app

In your terminal, run the following command:

```
dotnet run
```

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

Choose either browser, enter a name and message, and select the **Send Message** button.

The name and message are displayed on both pages instantly.