**Web Socket**

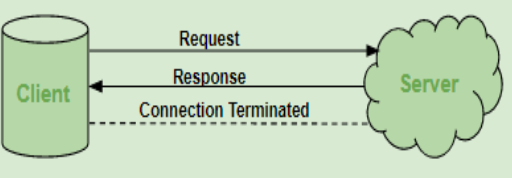
HTTP and WebSocket both are communication protocols used in client-server communication.

**HTTP protocol:** HTTP is unidirectional where the client sends the request and the server sends the response. Let’s take an example when a user sends a request to the server this request goes in the form of HTTP or HTTPS, after receiving a request server sends the response to the client, each request is associated with a corresponding response, after sending the response the connection gets closed, each HTTP or HTTPS request establishes the new connection to the server every time and after getting the response the connection gets terminated by itself.

HTTP is stateless protocol runs on the top of TCP which is a connection-oriented protocol it guarantees the delivery of data packet transfer using the three-way handshaking methods and re-transmit the lost packets.

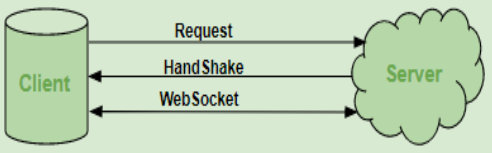
HTTP can run on the top of any reliable connection-oriented protocols such as TCP. When a client sends HTTP request to the server, a TCP connection is open between the client and server and after getting the response the TCP connection gets terminated, each HTTP request opens separate TCP connection to the server, for e.g. if client send 10 requests to the server the 10 separate TCP connection will be opened. and get closed after getting the response/fallback.

HTTP message information encoded in ASCII, each HTTP request message is composed HTTP protocol version(HTTP/1.1, HTTP/2), HTTP methods (GET/POST etc.), HTTP headers (content type, content length), host information, etc. and the body which contains the actual message which is being transferred to the server. HTTP headers varied from 200 bytes to 2 KB in size, the common size of HTTP header is 700-800 bytes. When web application uses more cookies and other tools at client-side that expand storage features of agent it reduces the HTTP header payload.



**WebSocket**: WebSocket is a bidirectional protocol that is used in the same scenario of client-server communication, unlike HTTP it starts from ws:// or wss:// (for security). It is a stateful protocol, which means the connection between client and server will keep alive until it is terminated by either party (client or server). after closing the connection by either of the client and server, the connection is terminated from both the end.

Let’s take an example of client-server communication, there is client which is a web browser and a server, whenever we initiate the connection between client and server, the client-server made the handshaking and decide to create a new connection and this connection will keep alive until terminated by any of them. When the connection is established and alive the communication takes place using the same connection channel until it is terminated.



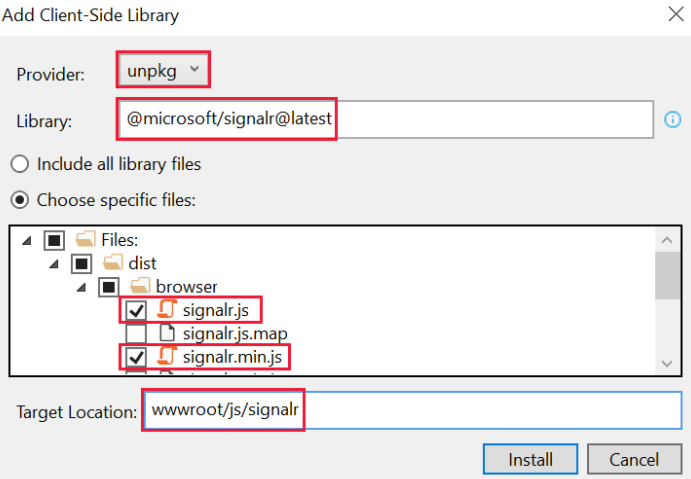
**SignalR**

SignalR is a free and open-source software library for Microsoft ASP.NET that allows server code to send asynchronous notifications to client-side web applications.

SignalR is framework to communicate in real-time without polling (keep asking if there is any data to send) between server and client (browsers). Web socket is the main protocol to use in SignalR when web socket is available. If web socket is not available, then SignalR will use a different way to push data specially from server to the client.

Let’s realize SignalR in the MVC pattern. We will use Library Manager (LibMan) to get the client library from unpkg. unpkg is a content delivery network (CDN) that can deliver anything found in npm, the Node.js package manager.

* In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.
* In the **Add Client-Side Library** dialog, for **Provider** select **unpkg**.
* For **Library**, enter @microsoft/signalr@latest.
* Select **Choose specific files**, expand the dist/browser folder, and select signalr.js and signalr.min.js.
* Set **Target Location** to wwwroot/js/signalr/
* Select **Install**

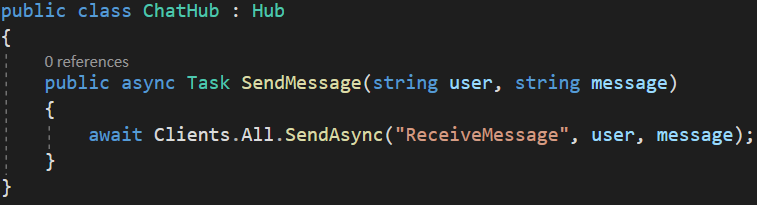


We will need just 2 js files.

The link 🡪 <https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-5.0&tabs=visual-studio>

After this we create a hubs folder inside which we will have a ChatHub class which will inherit from Hub class. A hub is a class that serves as a high-level pipeline that handles client-server communication. The Hub class manages connections, groups, and messaging.

We will send requests to the SendMessage method which is in ChatHub class.



We also need to add an endpoint for the ChatHub 🡪



We will also need to add a particular service for the SignalR 🡪



Then in the page where we have chatting, for instance, we will need to add the libraries that we installed 🡪

@section Scripts{

<script src="~/lib/microsoft/signalr/dist/browser/signalr.min.js"></script>

<script src="~/lib/microsoft/signalr/dist/browser/signalr.js"></script>

<script **asp-append-version**="true">

"use strict";

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

//Disable 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();

});</script>

}