# REAL-TIME ASP.NET WITH SignalR

TABLE OF CONTENTS

Introduction 3

Methodology 4

Hub 4

Transports 4

ASP.NET Core Implementation - Server 5

Prerequisites 6

Step 1 - Create a web app project 6

Step 2 - Add the SignalR client library 8

Step 3 - Create a SignalR hub 10

Step 4 - Configure SignalR 11

Step 5 - Add SignalR client code 12

Step 6 - Run the app 14

Angular Implementation - Client 15

Prerequisites 15

Step 1 - Create an angular project 15

Step 2 - Install the required packages 16

Step 3 - Import the required modules 17

Step 4 - Create the components needed 18

Step 5 - Add the code in components 19

Step 6 - Add the routing module code 22

Step 7 - Configure CORS in the server 22

Step 8 - Run the app 23

Conclusion 25

Muthulakshmi M

Logo

Description automatically generated

# Introduction

Today's modern apps are expected to deliver up-to-date information without hitting a refresh button. Adding **real-time functionality** is essential in applications such as dashboards, maps, games and more. Real-time functionality is the ability of server-side code to push content to connected clients as it happens, in real-time. **ASP.NET Core SignalR** is an open-source library that enables real-time web functionality to the applications. With SignalR, clients can also send messages to the server, a capability that demonstrates duplex communication. The following applications are good candidates for SignalR.

* Apps that require high frequency updates from server – gaming, social networks, and maps.
* Dashboard and monitoring apps – company dashboards, instant sales updates, and travel alerts.
* Collaborative apps – Whiteboard apps and team meeting software.
* Apps that require notifications – social networks, email, chat, games, and travel alerts.

The following are some features of SignalR for ASP.NET Core.

* SignalR handles connection management automatically.
* It sends messages to all connected clients simultaneously.
* It sends messages to specific clients or groups of clients.
* It scales to handle increasing traffic.
* It allows client SPAs (Single Page Applications) to react to changes on the server.

# METHODOLOGY

HUB

In SignalR, a hub is used to communicate between clients and servers. A hub is a high-level pipeline that allows a client and a server to call methods on each other. A hub is like as a **proxy** between all connected clients and the server. Each client sends and receives the message from hub. Clients don’t connect to each other directly.

|  |  |
| --- | --- |
| Diagram  Description automatically generated | Diagram  Description automatically generated |
| Client A wants to send message to Client C | Client A wants to send message to all other clients |

TRANSPORTS

SignalR supports the following techniques, or transports, for handling real-time communication:

* WebSockets
* Server-Sent Events
* Long Polling

The order in which the transports are listed here signifies their graceful fallback order. In other words, WebSockets is preferred over Server-Sent Events, and Server-Sent Events is preferred over Long Polling, though any one of them could be used. SignalR automatically chooses the best transport method that's within the capabilities of the server and client.

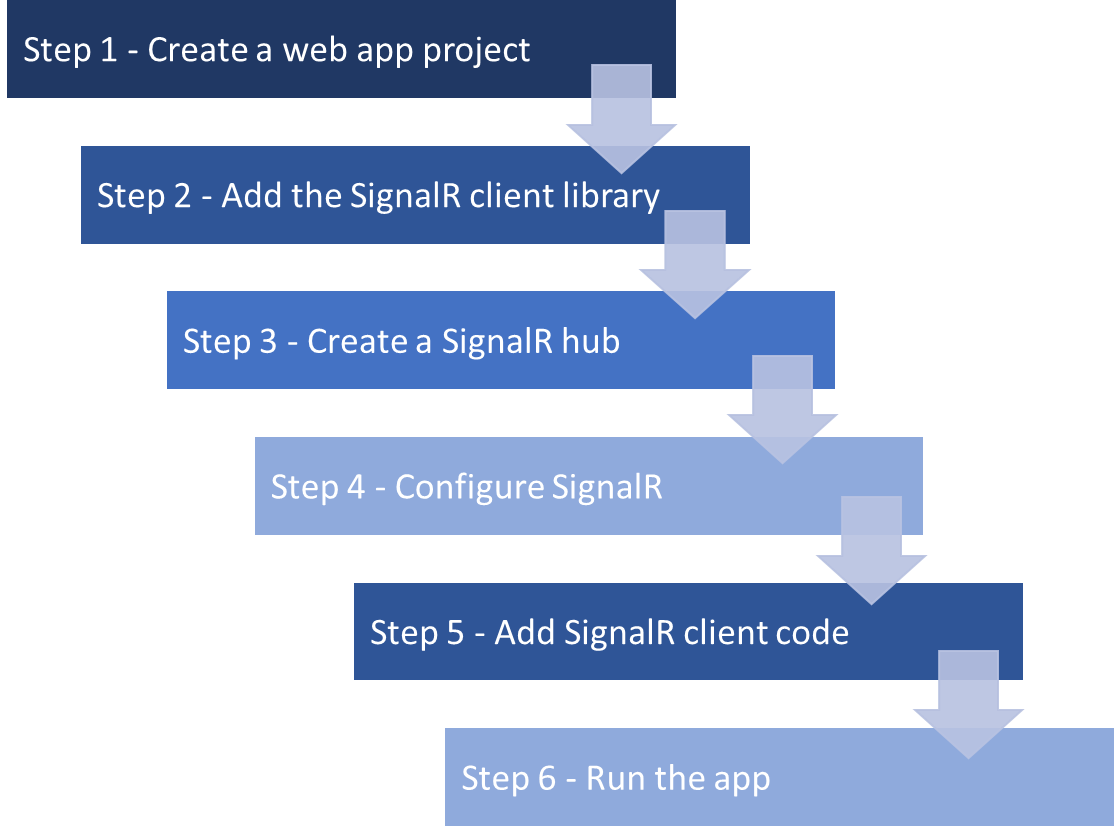
**WebSockets** allow full-duplex communication between the client and the server. First, connection is established using standard HTTP request. Once connected, server can send any number of messages to client. Client can respond back to the server on WebSocket.

Text

Description automatically generated

# ASP.NET Core IMPLEMENTATION - SERVER

The following are the steps to implement a real-time chat application using SignalR and ASP.NET Core. This is the server-side implementation of the SignalR application using ASP.NET Core.



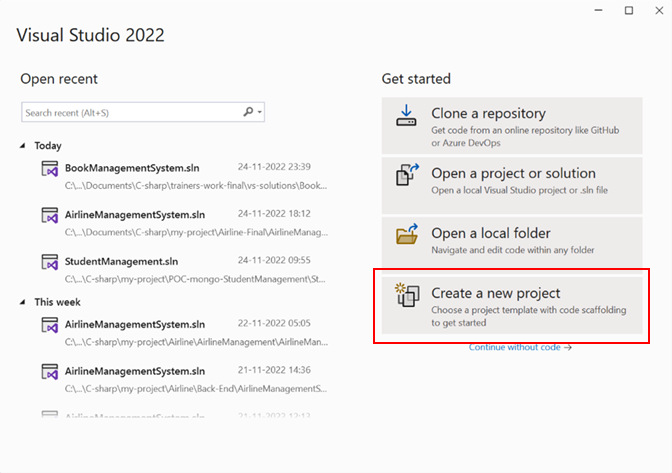
PREREQUISITES

The prerequisite for this application is **Microsoft** **Visual Studio 2022** with **ASP.NET** and **web development** workload.

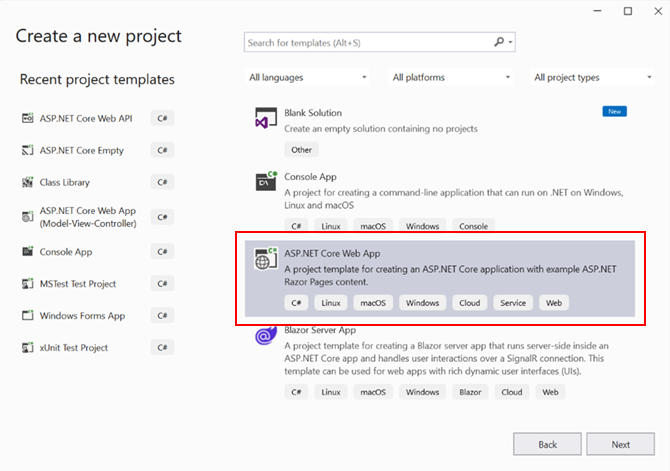


Step 1 - Create a web app project

1. Start Visual Studio 2022 and select **Create a new project**.



1. In the **Create a new project** dialog, select **ASP.NET Core Web App**, and then select **Next**.

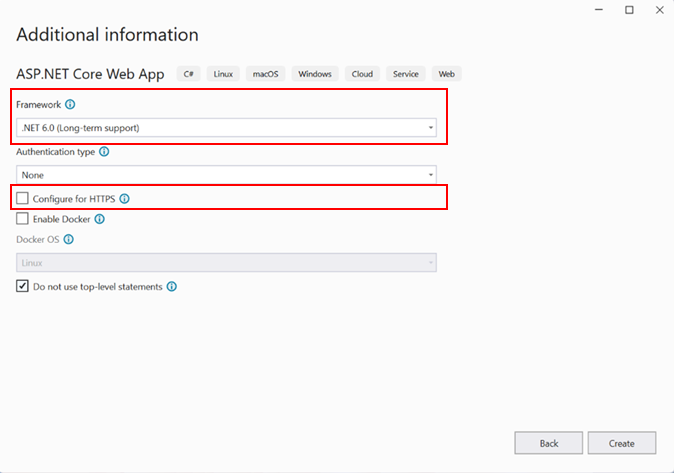


1. In the **Configure your new project** dialog, enter **SignalRChat** for **Project name**. Provide the preferred location for your project. Select **Next.**

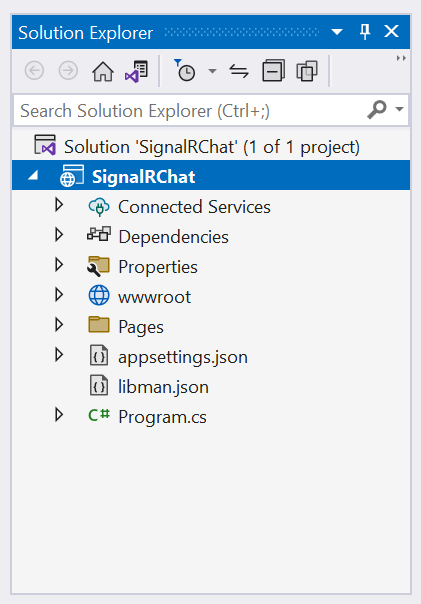
Graphical user interface, text, application, email

Description automatically generated

1. In the **Additional information** dialog, select **.NET 6.0 (Long-term support)**. Unselect Configure for HTTPS. Then select **Create**.



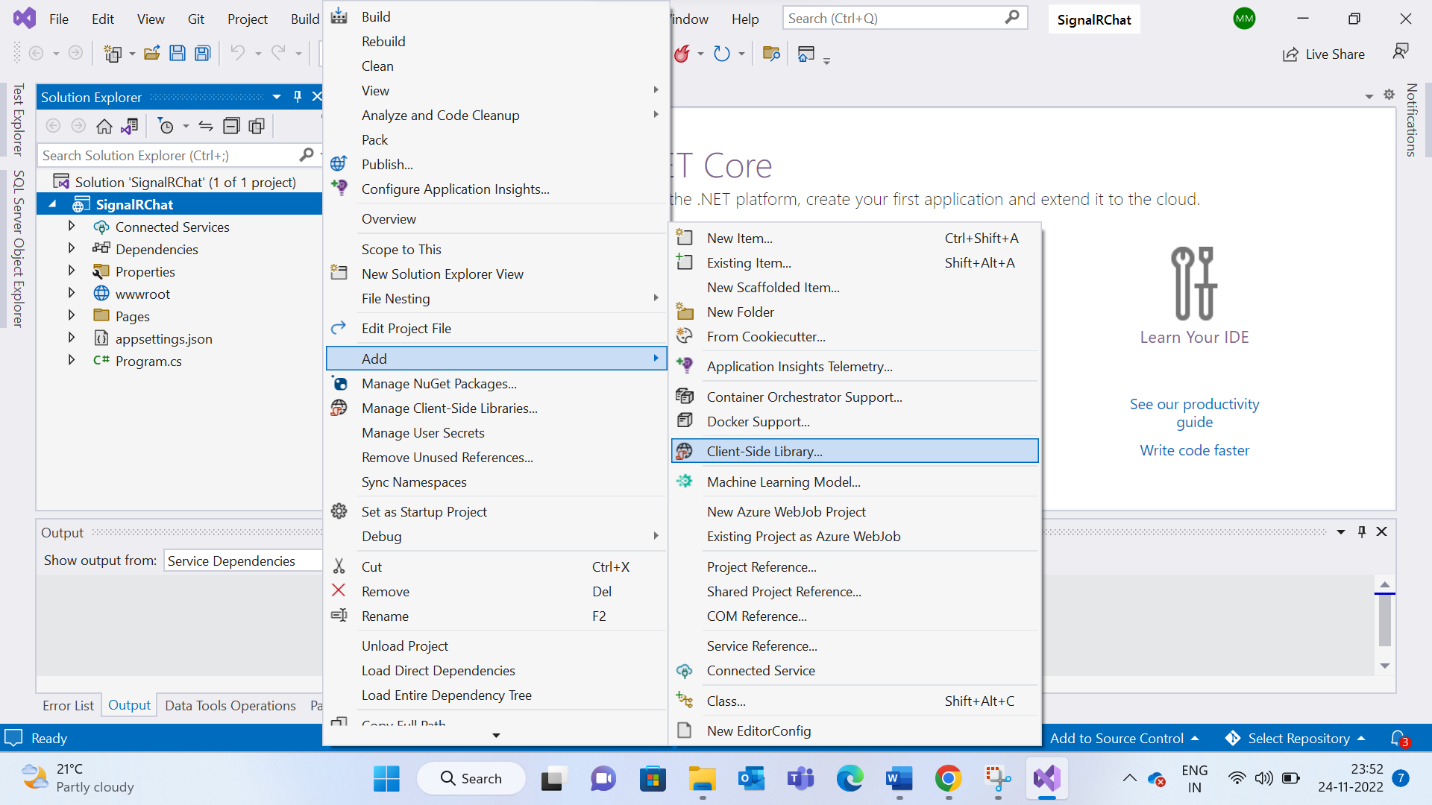
1. An ASP.NET Core Web App project will be created with all the basic files and folders. The folder structure will look like the one shown below.



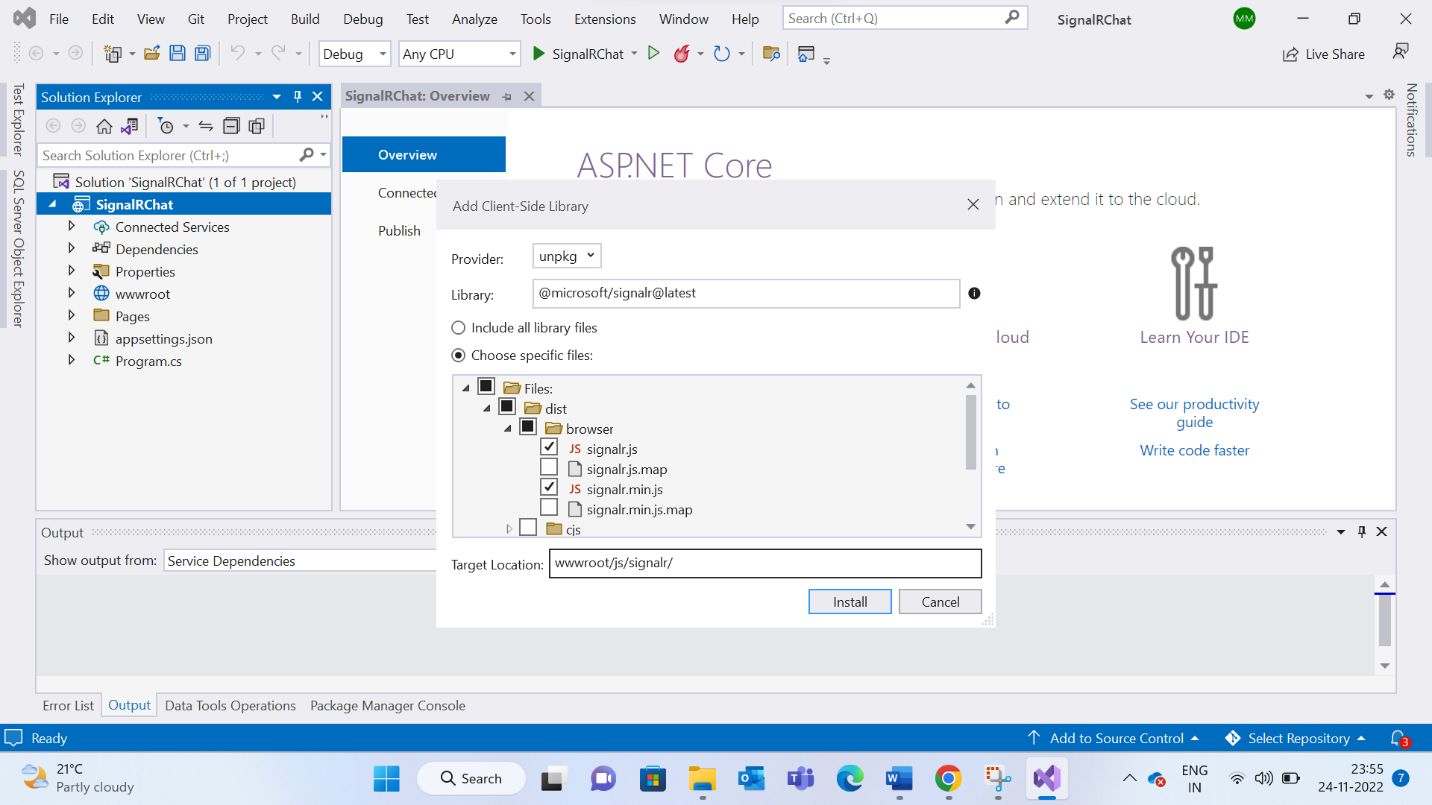
Step 2 - Add the SignalR client library

The SignalR server library is included in the ASP.NET Core shared framework. The JavaScript client library isn't automatically included in the project. Library Manager (LibMan) is used to get the client library from **unpkg**. unpkg is a fast, global content delivery network for everything on **npm**.

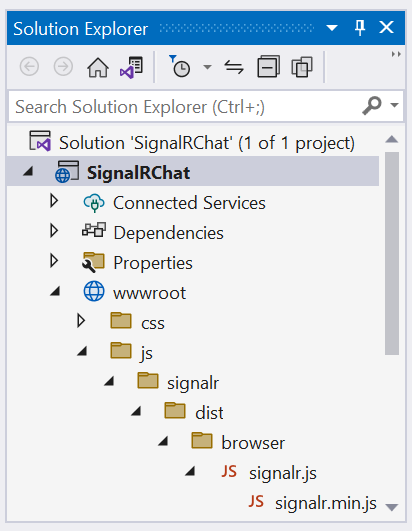
1. In **Solution Explorer**, right-click the project, and select **Add > Client-Side Library**.



1. In the **Add Client-Side Library** dialog:
   1. Select **unpkg** for Provider
   2. Enter **@microsoft/signalr@latest** for Library
   3. Select **Choose specific files**, expand the dist/browser folder, and select **signalr.js** and **signalr.min.js**.
   4. Set **Target Location** to wwwroot/js/signalr/
   5. Select **Install**

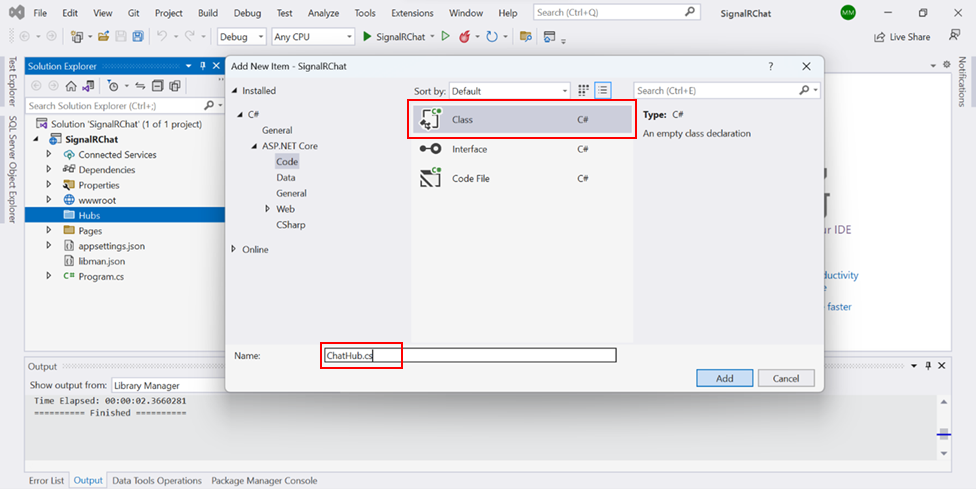


1. LibMan creates a wwwroot/js/signalr folder and copies the selected files to it.



Step 3 - Create a SignalR hub

1. In the SignalRChat project folder, create a **Hubs** **folder**.



1. In the Hubs folder, create the **ChatHub class** 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. SignalR code is asynchronous to provide maximum scalability.

Step 4 - 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;

namespace SignalRChat

{

public class Program

{

public static void Main(string[] args)

{

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");

}

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.MapHub<ChatHub>("/chatHub");

app.Run();

}

}

}

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

Step 5 - Add SignalR client code

1. Replace the content in **Pages/Index.cshtml** with the following code.

@page

<div class="container">

<div class="row">&nbsp;</div>

<div class="row">

<div class="col-2">User</div>

<div class="col-4"><input type="text" id="userInput" /></div>

</div>

<div class="row">

<div class="col-2">Message</div>

<div class="col-4"><input type="text" id="messageInput" /></div>

</div>

<div class="row">&nbsp;</div>

<div class="row">

<div class="col-6">

<input type="button" id="sendButton" value="Send Message" />

</div>

</div>

</div>

<div class="row">

<div class="col-12">

<hr />

</div>

</div>

<div class="row">

<div class="col-6">

<ul id="messagesList"></ul>

</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.

1. In the wwwroot/js folder, create a **chat.js** file with the following code.

Graphical user interface, application

Description automatically generated

"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);

li.textContent = `${user} - ${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.

Step 6 - Run the app

1. Press **CTRL+F5** to run the app without debugging.
2. Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.
3. Choose either browser, enter a name and message, and select the Send Message button. The name and message are displayed on both pages instantly.

Graphical user interface, text, application

Description automatically generated

# ANGULAR IMPLEMENTATION - CLIENT

The following are the steps to implement a real-time chat application using SignalR and Angular. This is the client-side implementation of SignalR application using Angular. In this chat application, messages from any client can be sent to all connected clients.

PREREQUISITES

The prerequisites for this application are

* **Node.js** with **npm**
* **Visual Studio Code** or other editor for modifying project files
* **Angular CLI**

To install the Angular CLI, open a terminal window and run the following command

npm install -g @angular/cli

Step 1 – Create an angular project

1. Open **command prompt** in your project directory and run the following command. This command creates an angular project in the name of **‘SignalRChat’**
2. Type **yes** to add angular routing.
3. Select **CSS** as the stylesheet format.

Text

Description automatically generated

1. After successful creation of the angular project, open the project using Visual Studio Code. The project structure will look like the one shown below.

Graphical user interface, application

Description automatically generated

Step 2 – Install the required packages

1. Go to **Terminal > New Terminal**
2. In the command prompt, run the following command to install **Bootstrap** and **SignalR** packages using NPM.

Graphical user interface, text, application, email

Description automatically generated

After successful installation, the packages will be included under **dependencies** section in the **package.json** file.

Text, application

Description automatically generated

1. In **angular.json** file, include the following lines in **projects->SignalRChat->architect->build->styles** and **projects->SignalRChat->architect->build->scripts** arrays respectively.

            "styles": [

              "node\_modules/bootstrap/dist/css/bootstrap.min.css",

              "src/styles.css"

            ],

            "scripts": [

              "node\_modules/bootstrap/dist/js/bootstrap.min.js"

            ]

Step 3 – Import the required modules

In **app.module.ts**, add **FormsModule** in the imports section.

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    FormsModule,

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

Step 4 – Create the components needed

In the command prompt, run the following **command prompt** commands to create **header**, **footer,** and **chat** components.

Graphical user interface, text, application

Description automatically generated

For each component, the corresponding **html**, **css** and **typescript** files get generated as shown below.

Graphical user interface, text, application

Description automatically generated

Step 5 – Add the code in components

Add the following codes in the corresponding files for each component.

**header.component.html**

<nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white

    border-bottom box-shadow mb-3">

    <div class="container">

        <a class="navbar-brand" [routerLink]="['/chat']">SignalRChat</a>

        <button class="navbar-toggler" type="button" data-bs-toggle="collapse"

                data-bs-target=".navbar-collapse"

                aria-controls="navbarSupportedContent"

                aria-expanded="false" aria-label="Toggle navigation">

            <span class="navbar-toggler-icon"></span>

        </button>

        <div class="navbar-collapse collapse d-sm-inline-flex

            justify-content-between">

            <ul class="navbar-nav flex-grow-1">

                <li class="nav-item">

                    <a class="nav-link text-dark"

                        [routerLink]="['/chat']">Chat</a>

                </li>

            </ul>

        </div>

    </div>

</nav>

**footer.component.html**

<div class="container">

    &copy; 2022 - SignalRChat

</div>

**chat.component.html**

<div class="container">

    <div class="row">&nbsp;</div>

    <div class="row">

        <div class="col-2">User</div>

        <div class="col-4"><input type="text" [(ngModel)]="user"

            class="form-control"/></div>

    </div>

    <div class="row pt-3">

        <div class="col-2">Message</div>

        <div class="col-4"><input type="text" [(ngModel)]="message"

            class="form-control"/></div>

    </div>

    <div class="row">&nbsp;</div>

    <div class="row">

        <div class="col-6">

            <input type="button" (click)="sendMessage()" id="sendButton"

                value="Send Message" />

        </div>

    </div>

</div>

<div class="row">

    <div class="col-12">

        <hr />

    </div>

</div>

<div class="row">

    <div class="col-6">

        <ul \*ngFor="let info of messages" id="messagesList">

            <li>{{info.user}} - {{info.message}}</li>

        </ul>

    </div>

</div>

**chat.component.ts**

import { Component, OnInit } from '@angular/core';

import { HubConnectionBuilder } from '@microsoft/signalr';

const url='http://localhost:5294/chatHub';

@Component({

  selector: 'chat',

  templateUrl: './chat.component.html',

  styleUrls: ['./chat.component.css']

})

export class ChatComponent implements OnInit {

  constructor() { }

  user:string='';

  message:string='';

  messages:any[]=[];

  connection:any;

  ngOnInit(): void {

    this.connection = new HubConnectionBuilder()

                            .withUrl(url)

                            .build();

    this.connection.on('ReceiveMessage',

                        (user:string,message:string)=>

                            this.messages.push({user:user,message:message})

                      );

    this.connect();

  }

  async connect(){

    try{

      await this.connection.start();

    }

    catch(error:any){

      console.log('error on conection', error)

    }

  }

  sendMessage(){

    if(this.user!=''&&this.message!='')

      this.connection.invoke('SendMessage', this.user,this.message);

  }

}

**app.component.html**

<app-header></app-header>

<div class="main-content">

  <router-outlet></router-outlet>

</div>

<app-footer class="footer fixed-bottom border-top text-muted"></app-footer>

Step 6 – Add the routing module code

Add the following code in the **app-routing.module.ts** to configure the **route** paths for the application.

**app-routing.module.ts**

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { ChatComponent } from './modules/core/components/chat/chat.component';

const routes: Routes = [

  {path:"chat", component:ChatComponent}

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

Step 7 – Configure CORS in the server

1. Open the **server-side** project of the application implemented using **ASP.NET Core**.
2. Add the highlighted lines of code in the **program.cs** file to configure **CORS** (Cross Origin Request Security).

using SignalRChat.Hubs;

namespace SignalRChat

{

public class Program

{

public static void Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddSignalR();

builder.Services.AddCors();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Error");

}

app.UseStaticFiles();

app.UseRouting();

app.UseCors(options=>

options.AllowAnyMethod()

.AllowAnyHeader()

.SetIsOriginAllowed(origin => true)

.AllowCredentials());

app.UseAuthorization();

app.MapRazorPages();

app.MapHub<ChatHub>("/chatHub");

app.Run();

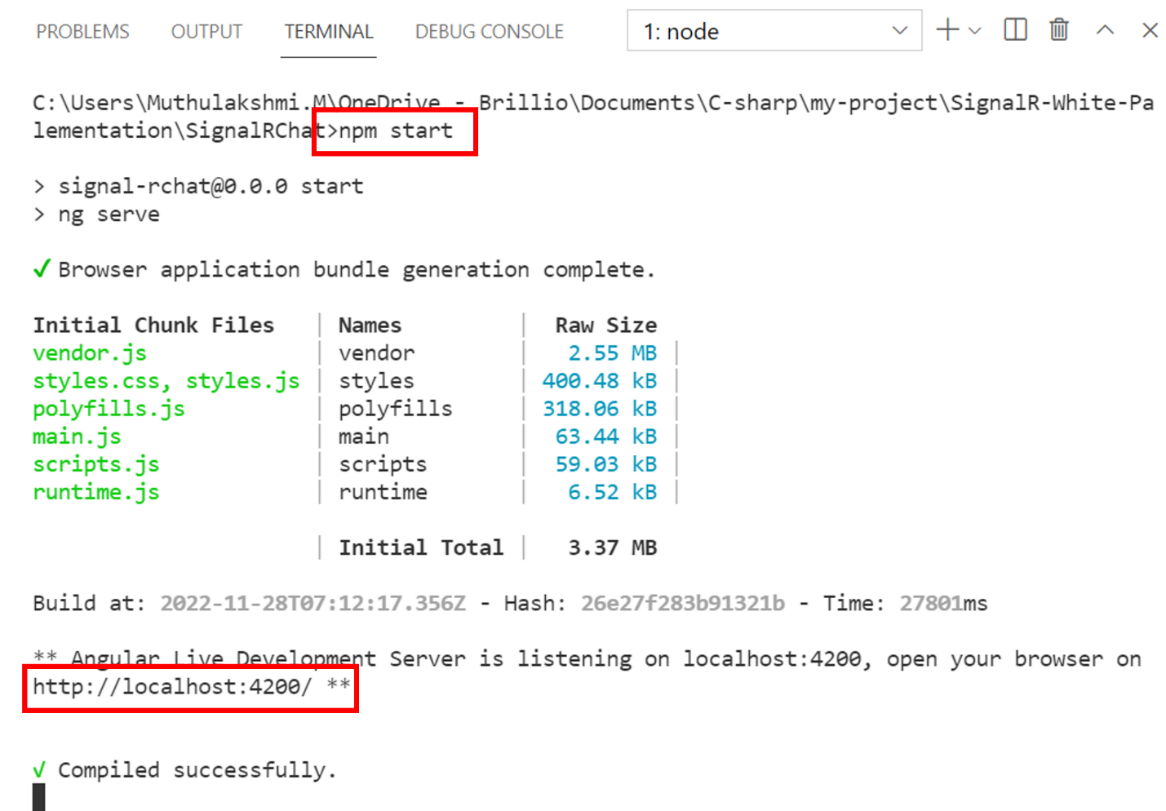
}

}

}

Step 8 – Run the app

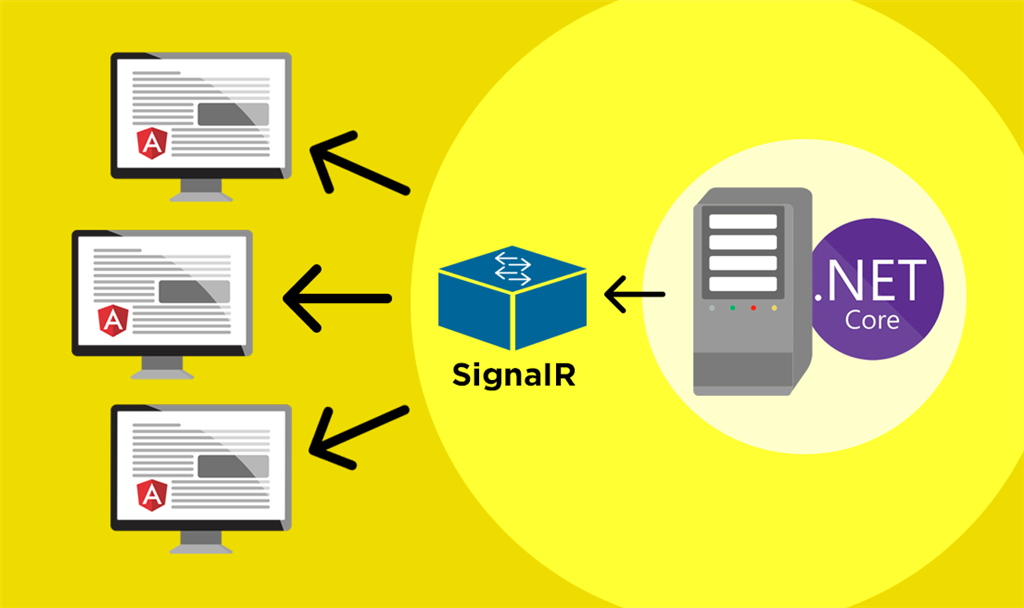
1. In the **server**, Press **CTRL+F5** to run the app without debugging.
2. In the **client**, run the command “**npm start**” in the command prompt to run the application.



1. Copy the URL “**http://localhost:4200**” from the command prompt.
2. Open two different browsers and paste the URL in the address bar.
3. Navigate to ‘**chat’** in both browsers.
4. Choose either browser, enter a name and message, and select the Send Message button. The name and message are displayed on both pages instantly.

Graphical user interface, text, application

Description automatically generated



# Conclusion

SignalR provides an API for creating server-to-client remote procedure calls (RPC). The RPCs invoke functions on clients from server-side .NET Core code. SignalR is open source on **GitHub**, just like the rest of .NET. In addition to the source code, the protocol specification for communication between hubs and clients is open too. With client SDKs for **JavaScript**, **.NET** (C#, F#, and Visual Basic), and **Java**, we can connect to our SignalR hub and start receiving real-time messages on almost any platform including web, mobile, desktop, and games. SignalR will use WebSockets when it's available, and gracefully falls back on other technologies when it isn't, while our application code stays the same.