2023

Communication tools in .NET

Stefan Cadar

Technical University of Cluj-Napoca

10/15/2023

Table of Contents

1 Introduction

1.1 Context

The goal of this project is to study communication tools that are available in the .NET framework. In addition, this work will also include a demonstrative application which exemplifies how to implement the client-server application in .NET.

These tools have a wide range of use cases in industries such as instant messaging, finance, data broadcasting, multiplayer collaboration, GPS location tracking and many more. There are multiple ways clients and servers communicate on the internet such as HTTP, Sockets and Websockets, .NET has implementation for each one of them.

1.2 Specification

The source code of what we are going to build can be seen on “link” and the source code of the application is available on “github-link”.

We will send messages from client to server and backwards in the form of a instant messaging application that will be accessed using the web.

To complete this work, we will need the following tools:

- Microsoft Visual Studio IDE
- .Net SDK

1.3 Objectives

Design and implement a client-server web-based application with the .NET framework which will use different communication tools for communication via the internet. We will have two or more users which will write messages in a room. The messages will be stored in the database along with which the user has access to them.

2 Bibliographic study

3 Analysis

3.1 Websockets

WebSocket is a realtime technology that enables bidirectional, full-duplex communication between client and server over a persistent, single-socket connection. This technology consists of two core building blocks: The Websocket protocol and the Websocket API.

The first real time web applications started to appear in the 2000s, but they were difficult to achieve and slower, they were built by "hacking" established HTTP-based technologies (AJAX and Comet). In 2008 two developers Michael Carter and Ian Hickson created with in collaboration with IRC and W3C mailing lists a new modern standard for real time communication, mainly Websockets.

3.1.1 The WebSocket Protocol

In December 2011, the Internet Engineering Task Force (IETF) standardized the WebSocket protocol through RFC 6455. In coordination with IETF, the Internet Assigned Numbers Authority (IANA) maintains the Websocket Protocol Registries, which define many of the codes and parameter identifiers used by the protocol.

3.1.2 The WebSocket API

Included in the HTML Living Standard, the WebSocket API is a programming interface for creating WebSocket connections and managing the data exchange between a client and a server in a web app. It provides a simple and standardized way for developers to use the WebSocket protocol in their applications.

Nowadays, almost all modern browsers support the WebSocket API. Additionally, there are plenty of frameworks and libraries — both open-source and commercial solutions — that implement WebSocket APIs

3.1.3 How do WebSockets work?

It envolves three main steps:

- Opening a connection consists of a HTTP request/response between client and server.
- Data transmission, client, and server exchange messages between frames.
- Closing a connection

3.2 SignalR

ASP.NET SignalR is an abstraction over WebSockets and as well over many other real time communication API. It can be used to add any sort of "real-time" web functionality to your ASP.NET application. While chat is often used as an example, you can do a whole lot more. Any time a user refreshes a web page to see new data, or the page implements long polling to retrieve new data, it is a candidate for using SignalR. Examples include dashboards and monitoring applications, collaborative applications (such as simultaneous editing of documents), job progress updates, and real-time forms. For this project we will mainly use simple WebSockets and SignalR.

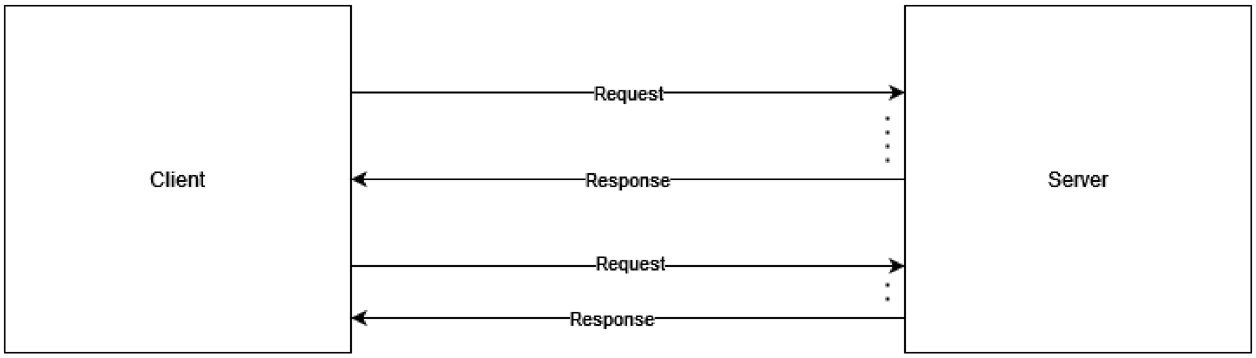
4 Design

For this project SignalR will be used. SignalR supports the following tehniques (transports) for handling real time communication:

- WebSockets
- Server-Sent Events
- Long Polling

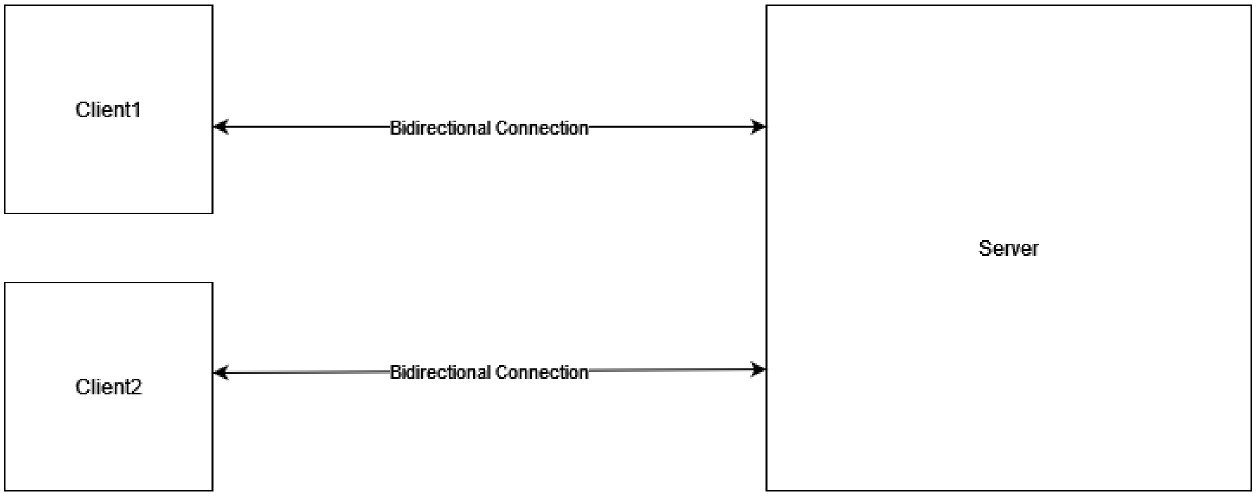
4.1 HTTP

To understand the transports we first need to familiarize ourselves with http. Http is an protocol that does not imply real time communication. The client is the only one that can initiate a request and the server gives back a response. Everything is stateless and the server can not notify the client for events.



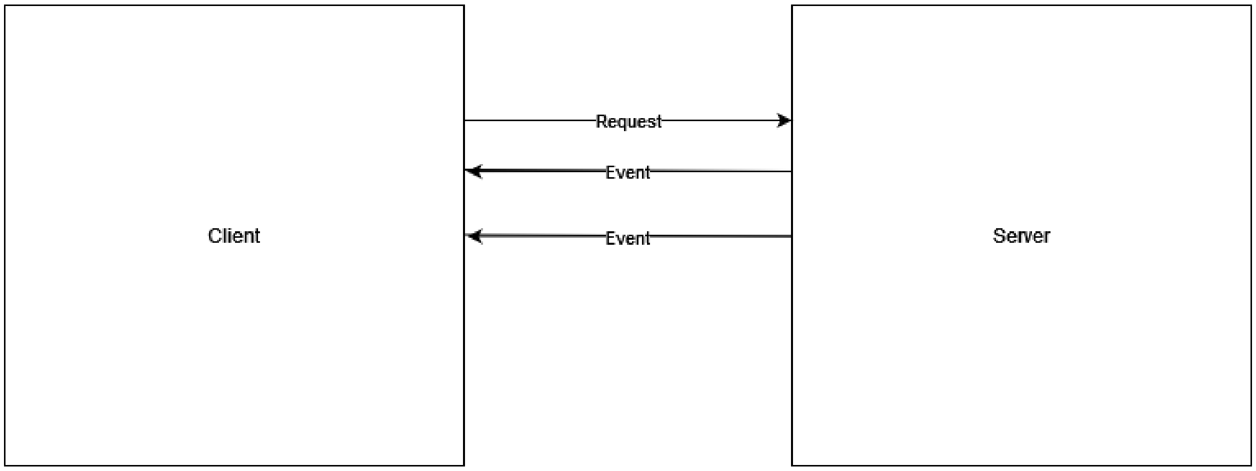
4.2 WebSockets

The first transport that SignalR tries to use is WebSockets. This implies that there is bidirectional communication between client and server.



4.3 Server-Sent Events

Server-Sent Events (SSE) is when the client sends one request and based on this request the sever sends multiple events to the client.



4.4 Long Polling

When using this transport, the client sends requests to the server constantly and keeps waiting for a response mimicking a bidirectional connection. It is very similar to HTTP, but the point is for the client to wait as long as possible and to automatically send the request again when it times out.

The order in which they are written here represents also their fallback order, in other words WebSockets is most preferred then Server-Sent Events then Long Polling.

The Server is responsible for exposing a SignalR endpoint, which is mapped to a Hub class. The Server exposes the methods the clients can call and the events they subscribe to.

The information in SignalR can be transmitted in one of two formats: JSON which is default or MessagePack which contains binary data.

There exist also groups in SignalR which send messages to multiple clients. As for the connection, it is represented by a unique identifier.

There are tree clients:

- JavaScript Client
- .Net Client
- Java Client

For the implementation we will use the JavaScript client because we will make a web application.

We will design a chat application in the following phases. Each phase represents a different aspect of SignalR. They will be the following:

1. How to open a connection between client and server and send messages to all clients.
2. How to manually switch which transport SignalR uses.
3. Which are the different methods of the Hub class.
4. How to implement groups.
5. Finishing touches. Connection to database and User Identity.

5 Implementation

5.1 How to open a connection between client and server and send messages to all clients.

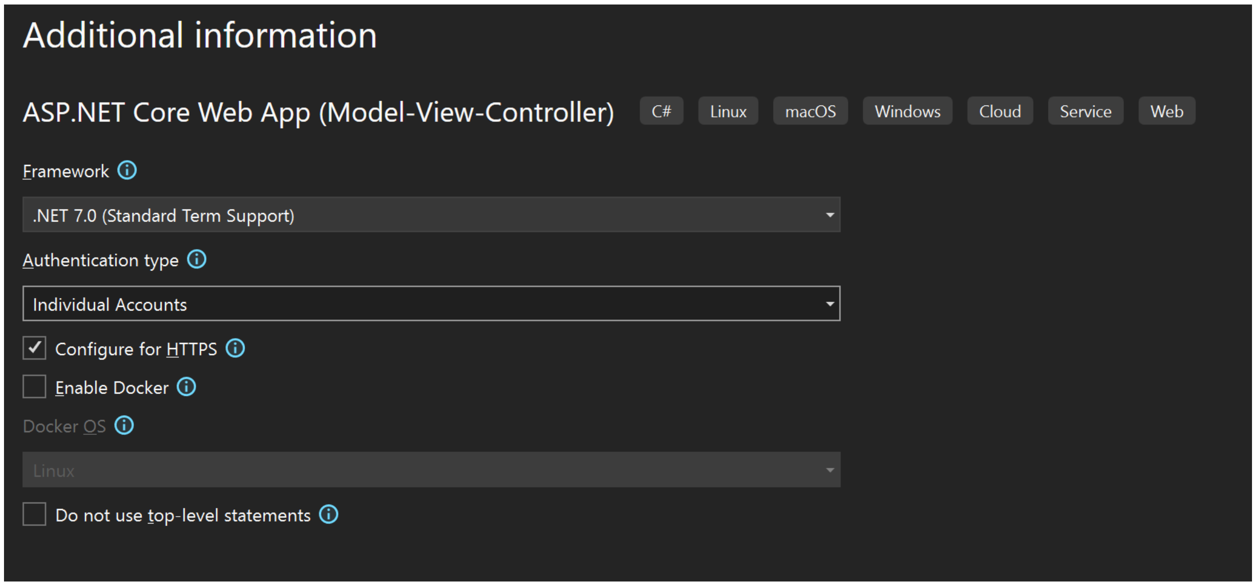
5.1.1 Install requirements.

To work with .Net applications we need to install the .Net 7 SDK and Visual Studio IDE.

- Link for .Net SDK: https://dotnet.microsoft.com/en-us/download
- Link for Visual Studio: https://visualstudio.microsoft.com/downloads/

5.1.2 Create and run the new project.

1. Open Visual Studio.
2. Create a new Project.
3. Search and select ASP.NET Core Web App (Model-View-Controller), make sure you have C# as the selected language.
4. Choose a name and select.
5. For the additional information menu make sure you have the same configuration as in the picture bellow. The Authentication Type will be used in the next steps for connecting the Hubs to the users in the database.



Additional information

ASP.NET Core Web App (Model-View-Controller)   C#   Linux   macOS   Windows   Cloud   Service   Web

Framework: .NET 7.0 (Standard Term Support)

Authentication type: Individual Accounts

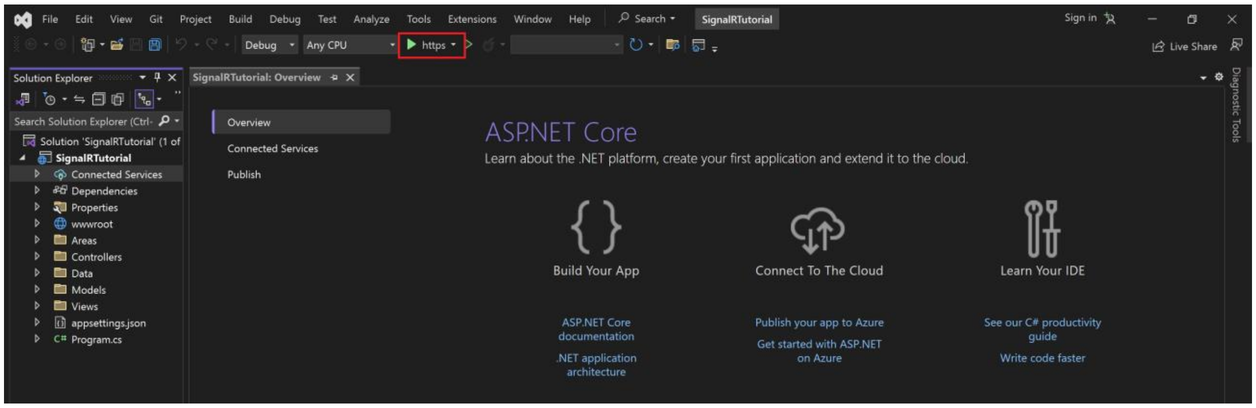
[✓] Configure for HTTPS

[ ] Enable Docker

Docker OS: Linux

[ ] Do not use top-level statements

6. After creating the project press the green filled play button like the one from the picture bellow.


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
SignalRTutorial
Sign in
Live Share
Debug
Any CPU
https
Solution Explorer
Search Solution Explorer (Ctrl-
Solution 'SignalRTutorial' (1 of
SignalRTutorial
Connected Services
Dependencies
Properties
wwwroot
Areas
Controllers
Data
Models
Views
appsettings.json
Program.cs
SignalRTutorial: Overview
Overview
Connected Services
Publish
ASP.NET Core
Learn about the .NET platform, create your first application and extend it to the cloud.
Build Your App
Connect To The Cloud
Learn Your IDE
ASP.NET Core documentation
.NET application architecture
Publish your app to Azure
Get started with ASP.NET on Azure
See our C# productivity guide
Write code faster
Diagnostic Tools

Now you should have an open browser with the starter ASP.NET Core Web MVC project. Now let’s look at the solution structure of our project. You have the following structure:

1. Connected Services: Here are the links to our connected services (authentication and database in our case).
2. Dependencies: Here are all the packages we have installed.
3. Properties: Here are the properties of our application such as environment variables, application url etc.
4. wwwroot: Static files (javascript, css, images, favicon, etc.)
5. Areas: Parts of html
6. Controllers: Our controllers of the MVC
7. Data: Migrations
8. Models: Our models
9. Views: cshtml files which we will send to the client
10. Appsetting.json: setting such as connection url to db
11. Program.cs : the main file

5.1.3 Create the first hub.

Create a new Hubs folder where we will store the hubs we create by right clicking on the solution and clicking on add. Now create a new class in that folder named EchoHub. This will be a hub that simply echoes to all created clients what it receives. The class should look like this:

```
using Microsoft.AspNetCore.SignalR;

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

If we look at the code, we are using the SignalR library. Our class is in the Hubs namespace, a namespace in .NET is the equivalent of a package in java. We define our class which extends Hub, making it the server side of our application. The method SendMessage send to all clients a command to execute the function ReciveMessage, with the parameters user and message.

We also need to connect the SignalR service to our app by modifying the Program.cs file like in the next page. Note that the url which we will use to connect to the hub class is /echoHub.

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using SignalRTutorial.Data;

using SignalRTutorial.Hubs;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection") ?? throw new
InvalidOperationException("Connection string 'DefaultConnection' not found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddControllersWithViews();

builder.Services.AddSignalR();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Home/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.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.MapRazorPages();

app.MapHub<EchoHub>("/echoHub");

app.Run();
```

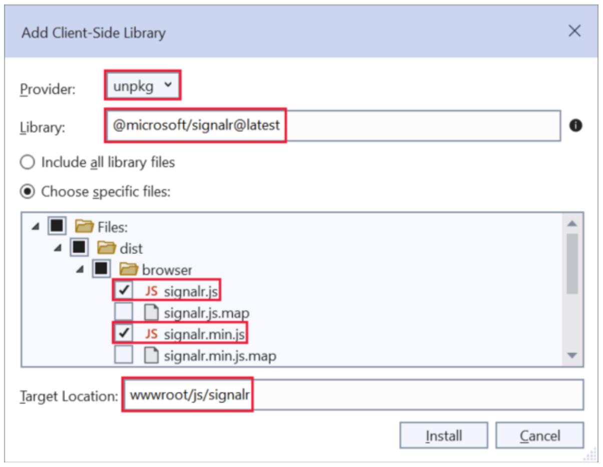
5.1.4 Implementing the client

We will implement a the signalR javascript client. To do this we first need to install the javascript source code of the client and add it to the project for this we will use the unpkg provider.

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

In the **Add Client-Side Library** dialog:

- Select **unpkg** for **Provider**
- Enter @microsoft/signalr@latest for **Library**.
- 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**.


Add Client-Side Library

Provider: unpkg

Library: @microsoft/signalr@latest

Include all library files

Choose specific files:

- Files:
  - dist
    - browser
      - ✓ signalr.js
      - signalr.js.map
      - ✓ signalr.min.js
      - signalr.min.js.map

Target Location: wwwroot/js/signalr

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

Now replace the content of Views/Index.cshtml with the following code bellow.

In the HTML file we an input for user, an input for message which are the text boxes where we will write the two parameters which we send to the hub. We also have a button which will send the data when we press and a unordered list for displaying the messages. There are two script tags one which links to the client side library which we previously added and one which links to the client side code which we will implement.

```
<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/echoChat.js"></script>
```

In order to implement the client side functionality we need to create a connection and based on what happened in the DOM we will send data to the server and also receive.

First, we use the strict mode. This is good practice otherwise we might have errors slipping by.

Then we create a new connection to the hub with the link /echoHub and disable the send button, we only want it enabled after the connection is established.

We define what happens when the connection has the event **“ReceiveMessage”**, it adds a new element to the **“messagesList”** containing the message and the user that sent it.

On connection start we want to enable the button otherwise throws an error.

When the button is clicked the client fetches the input from the DOM and sends it to the server using the **“SendMessage”** event from the library and throws an error if it fails.

The following code should be added in a new javascript file in called echoChat.js in the wwwroot/js directory.

```
"use strict";

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

document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var li = document.createElement("li");
    document.getElementById("messagesList").appendChild(li);
    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();
});
```

Now you should test the application. Note that if you open two different browsers, they communicate with each other.

6 Testing and Validation

7 Conclusion

8 Bibliography