A group of rectangular shapes

Description automatically generated with medium confidence

2023

Stefan Cadar

Technical University of Cluj-Napoca

10/15/2023

Communication tools in .NET

A group of rectangular shapes

Description automatically generated with medium confidenceA group of rectangular shapes

Description automatically generated with medium confidenceA group of rectangular shapes

Description automatically generated with medium confidence

# Table of Contents

[Table of Contents 1](#_Toc150931800)

[1 Introduction 2](#_Toc150931801)

[1.1 Context 2](#_Toc150931802)

[1.2 Specification 2](#_Toc150931803)

[1.3 Objectives 2](#_Toc150931804)

[2 Bibliographic study 3](#_Toc150931805)

[2.1 IP/TCP Protocol 3](#_Toc150931806)

[2.2 The Open System Interconnection Model (OSI Model) 3](#_Toc150931807)

[3 Analysis 4](#_Toc150931808)

[3.1 Network sockets 4](#_Toc150931809)

[3.2 WebSockets 5](#_Toc150931810)

[3.2.1 The WebSocket Protocol 5](#_Toc150931811)

[3.2.2 The WebSocket API 5](#_Toc150931812)

[3.2.3 How do WebSockets work? 5](#_Toc150931813)

[3.3 SignalR 6](#_Toc150931814)

[4 Design 6](#_Toc150931815)

[4.1 HTTP 7](#_Toc150931816)

[4.2 WebSockets 7](#_Toc150931817)

[4.3 Server-Sent Events 7](#_Toc150931818)

[4.4 Long Polling 8](#_Toc150931819)

[5 Implementation 9](#_Toc150931820)

[5.1 Basic TCP Client-Server Application in .NET 9](#_Toc150931821)

[5.2 Basic WebSockets Client-Server Application in .NET 12](#_Toc150931822)

[5.3 Basic SignalR Client-Server Application in ASP.NET Core MVC 17](#_Toc150931823)

[5.3.1 Create and run the starter Project. 17](#_Toc150931824)

[5.3.2 Create the first hub 18](#_Toc150931825)

[5.3.3 Implementing the client 21](#_Toc150931826)

[5.3.4 How to figure out which transport SignalR uses? 23](#_Toc150931827)

[5.3.5 Bonus: Integrating authentication with SignalR 23](#_Toc150931828)

[6 Testing and Validation 25](#_Toc150931829)

[7 Conclusion 25](#_Toc150931830)

[8 Bibliography 26](#_Toc150931831)

# Introduction

## 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.

## 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

## 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.

# Bibliographic study

## IP/TCP Protocol

TCP/IP stands for Transmission Control Protocol/Internet Protocol and is a suite of communication protocols used to interconnect network devices on the internet, the main ones being IP and TCP. These protocols express how data should be broken into packets, addressed, transmitted, routed, and received at the destination. Each of these protocols has his own functions.

* IP defines how to packet and route each packet to make sure that it reaches the correct destination.
* TCP defines how the application creates channels of communication. It also manages how a message is assembled into smaller packages.

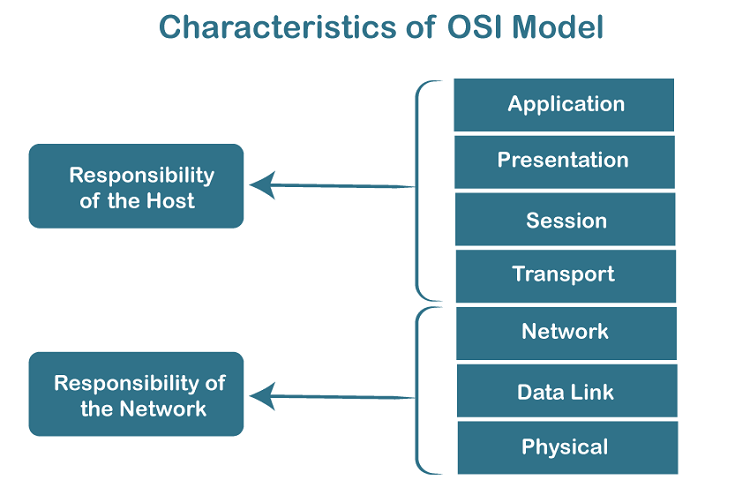
Common TCP/IP protocols include the following:

* HTTP
* HTTP Secure
* File Transfer Protocol

TCP/IP follows the client-server model and it is classified as stateless.

## The Open System Interconnection Model (OSI Model)

The Open Systems Interconnection (OSI) model describes seven layers that computer systems use to communicate over a network.



1. Physical Layer: Transmits raw bit stream over the physical medium.
2. Data Link Layer: Defines the format of data on the network
3. Network Layer: Defines which physical path the data will take
4. Transport Layer: Transmits data using transportation protocols including TCP and UDP.
5. Session Layer: Maintains connections and is responsible for controlling ports and sessions
6. Presentation Layer: Ensures that data is in a usable format and is where data encryption occurs
7. Application Layer: Human-Computer interaction layer, where applications can access the network services

# Analysis

## Network sockets

A **network socket** is a software structure within a network node of a computer network that serves as an endpoint for sending and receiving data across the network. The structure and properties of a socket are defined by an API for the networking architecture. Sockets are created only during the lifetime of a process of an application running in the node.

Because of the standardization of the TCP/IP protocols, the term *network socket* is most commonly used in the context of the Internet protocol suite, and is therefore often also referred to as **Internet socket**. In this context, a socket is externally identified to other hosts by its **socket address**, which is the transport protocol, IP address, and port number.

## 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.

### 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.

### 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

### 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

## 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.

# Design

The implementation of this project will be presented in the form of a laboratory work which will consist of four parts.

1. Study of the cross-platform socket implementation from the System.Net.Sockets namespace, where we will implement the most bare metal form of a Client-Server Application in .Net.
2. Study of the .Net implementation of WebSockets, which will also consist of a Client-Server application
3. Study of the SignalR Library which is an abstraction over Websockets and other transport methods of real time data. For this particular part further specification for the design of this library is required.

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

* WebSockets
* Server-Sent Events
* Long Polling

## 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 cannot notify the client for events.

A black background with white text

Description automatically generated

## WebSockets

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

A black and white screen with white text

Description automatically generated

## Server-Sent Events

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

A screen shot of a black screen

Description automatically generated

## 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 figure out which transport SignalR uses.
3. Which are the different methods of the Hub class.

Bonus: Finishing touches. Connection to the User Identity Service.

# Implementation

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/>

## Basic TCP Client-Server Application in .NET

First we need to create two projects: one for the client one for the server. Open Visual Studio and create a project for the server by doing the following: **Open Visual Studio -> Create a new Project -> Console Application** (make sure you have C# selected as the language) **-> Name the project SocketServer -> Create**

Do the same for the client and name it “SocketClient”

For the Server put the following code in the .cs file that you have in the solution.



In this class a new Socket with with the protocol tcp is created and listens for connections from any ip address on port 1234, once it is connected to the client it wait for data, once it gets the data it will be formatted then written as a string to the console and finally the server will wait for an input from keyboard to quit.

Here is the source code for the server:



The client sends the first line written in the console to the server then it quits.

To test these applications, you need to open two Powershell terminals. On both, navigate to the folder where the projects are located with:

PS> cd C:\path\to\folder\with\projects

Then run the project in each, make sure that you first run the server.

PS> dotnet run --project NameOfProject

Write something on the client and see how it gets sent to the server.

Exercises:

1. For the moment both our server and our client sent only one message at a time, modify the two classes such that both of them keep receiving/sending messages until the command “#quit” is written in the console. Hint: You will need to use loops and another thread for the server.

## Basic WebSockets Client-Server Application in .NET

As for the WebSockets implementation we must not forget that this protocol is also from the TCP/IP suite, therefore we can use the same classes. In order to diversify our knowledge of .NET we will use a simpler implementation of Sockets, mainly the TcpListener and the TcpClient classes which are abstractions over Sockets which have specific responsibilities eigther as Server or as Client.

Firstly, we will initiate the connection to the Client, similar with what we did in the previous part. Notice that this time we use port 80 because WebSockets uses the same port as HTTP.



Now it is time to add after the configuration a loop which takes every message from the client and sends it back. We add a while loop which constantly gets data from the client.

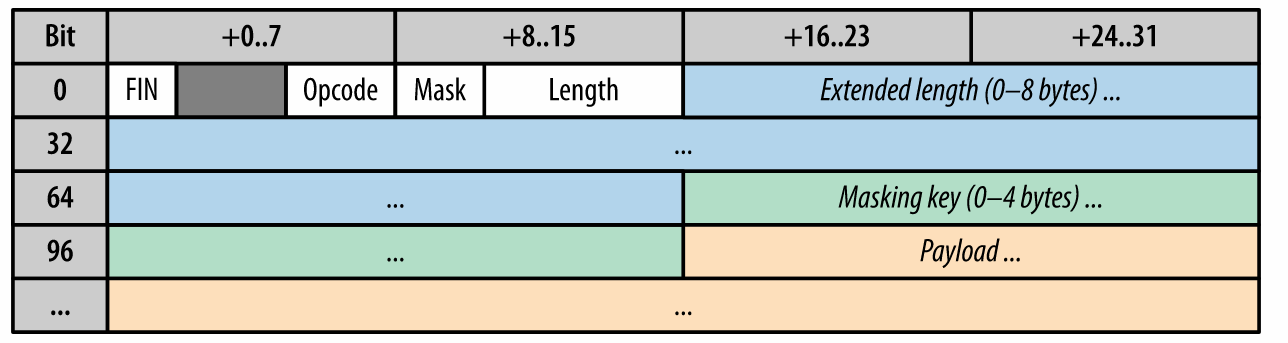


On every iteration of the loop we wait for the stream data to be available and for the data to reach three bytes, which should represent a “GET”, which is the specific HTTP method we are waiting for. If we indeed get that get we initiate the handshake. We compute the Sec-WebSocket-Accept which we sent to the client. Add the following code where we process the handshake.



After we make the handshake, we expect to get messages. Each message consists of bytes and the first bytes are special.

* First byte consists of FIN (byte which indicates if this fragment is the last one or not)
* The second byte contains the mask and the length for short messages. The mask bit represents the if the payload is masked.
* The next bytes up to the tenth one represent extended length in case of a longer message.
* 4 more bytes are the masking key used for decoding masked messages
* Rest are payload



Add this code for processing the messages from the client.



This would be the server-side implementation. As for the client the best way to simulate a client would be using the browser since WebSockets is a http based protocol which is mostly used for browsers. Create a html file with the following code:



Try to run the project and open the html page to a browser you can see that the message is being sent from client to server.

Exercise 2: Using your knowledge about how to use .NET make the server send the same message back to the client.

Exercise 3: Make server able to connect to multiple clients.

## Basic SignalR Client-Server Application in ASP.NET Core MVC

### Create and run the starter 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.

A screenshot of a computer

Description automatically generated

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

A screenshot of a computer

Description automatically generated

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

### 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:



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.



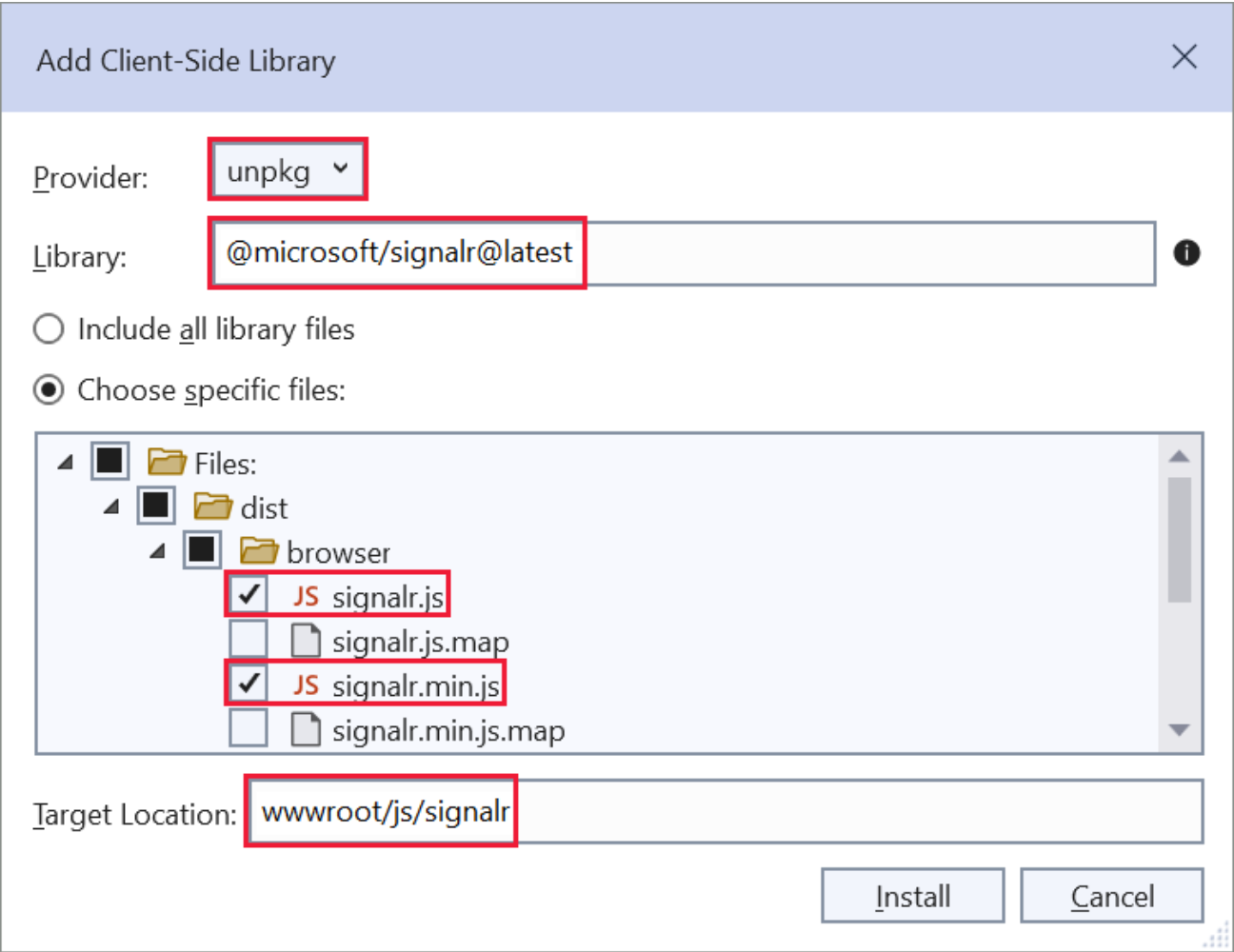
### Implementing the client

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**.



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.



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.

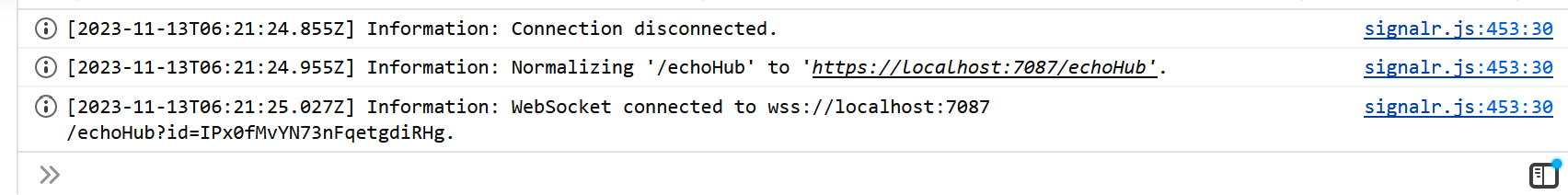
The following code should be added in a new javascript file in called echoChat.js in the wwwroot/js directory.



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

### How to figure out which transport SignalR uses?

Go to your browser and inspect the page open the console and allow infromations. You should see something similar to the following picture.



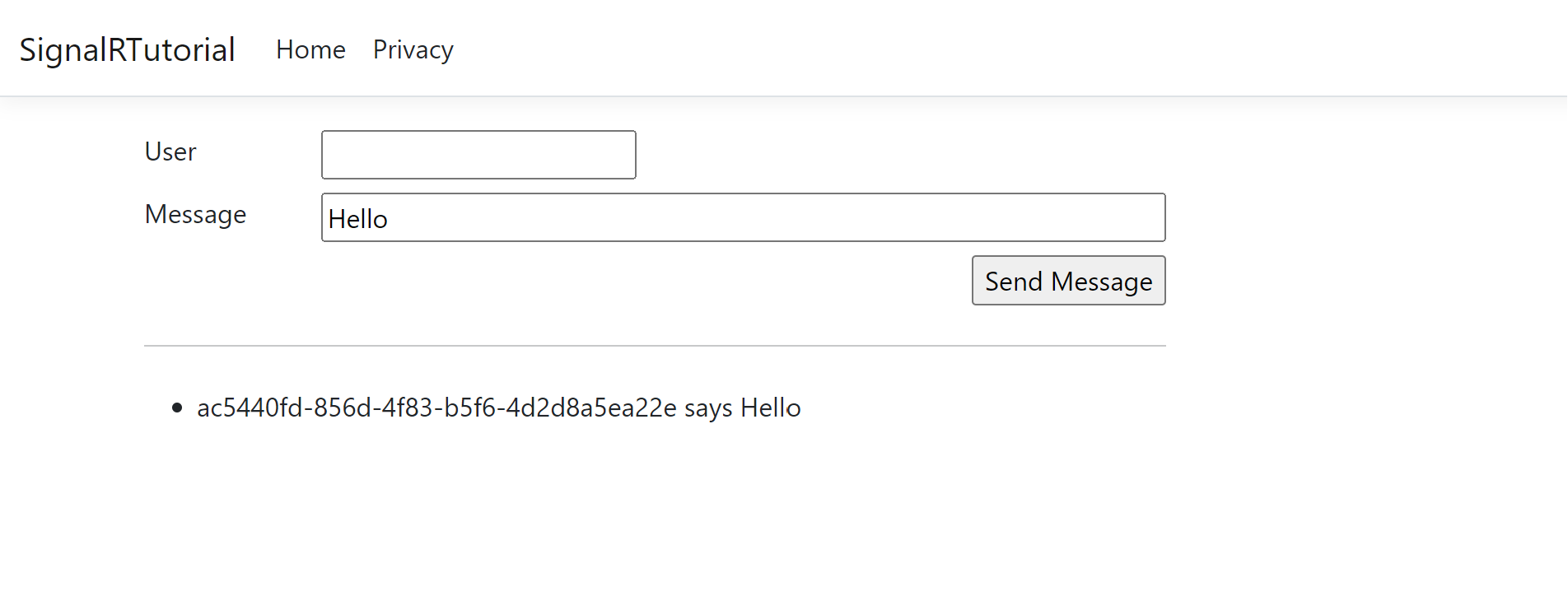
In this particular example WebSockets is the transport that is used.

### Bonus: Integrating authentication with SignalR

At the beginning of this part, we added User Identity as authentication. To use it in .Net is very simple. You just need to add in the EchoHub class the one more line. Which will try to get the User Identifier of the project.



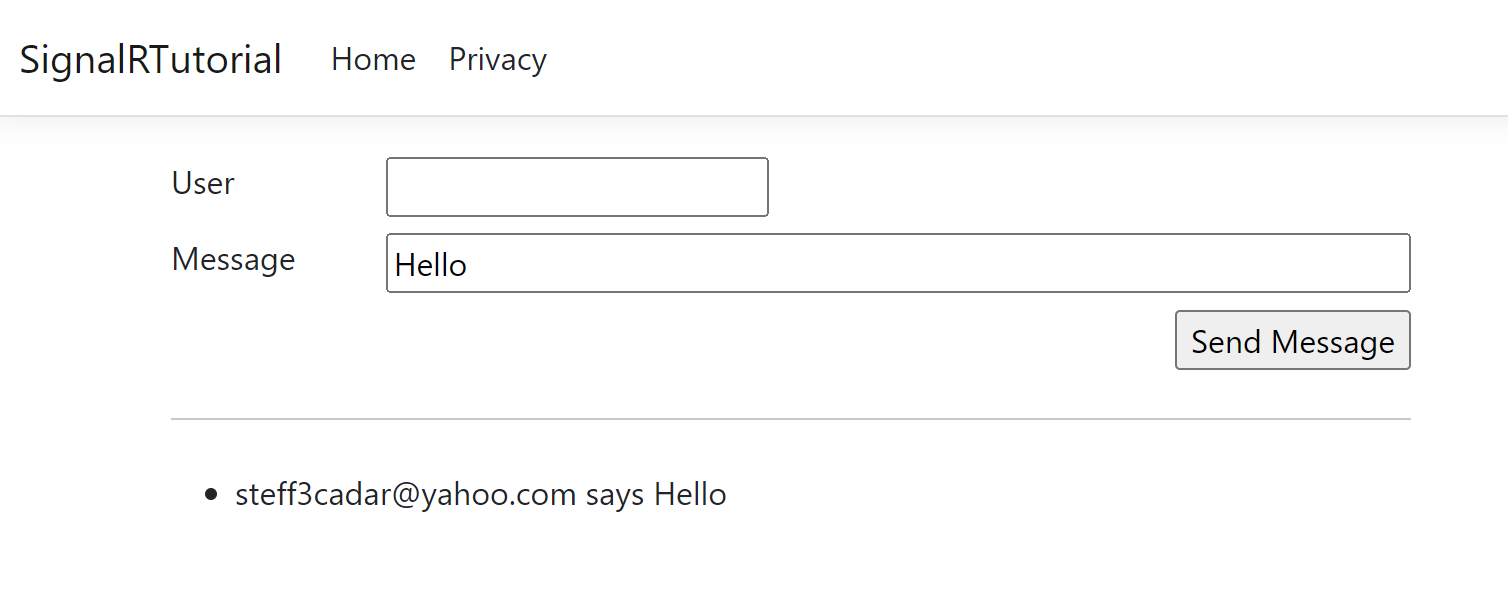
We use Context.UserIdentifier to get the id of the currently logged in user. If the user is not logged in, we use the name that we send as an input. Now the messages we send look like in the following picture.



It does not look good to have the identifier in the chat, so let’s try to fetch the email from the database to have a prettier way to show the sender of that message.



After the modification the result should be something like this.



# Testing and Validation

To test the solutions in all the parts of the laboratory try to have any message sent from Client to Server. For verification try the solutions found in this repository.

# Conclusion

.Net is a feature filled framework which make it one of the most used tools when it comes to networking and web based applications. It supports all kinds of real time communication procedures ranging from basic ones to automatic highly abstracted libraries.

My favorite part about this project is the fact that I got familiar with a framework and a language I never used before. It was a challenging task to create a paper that others are supposed to use for learning. Finally from all the examples I have seen I managed to clarify the basic parts when it comes to real time communication in .NET.

# Bibliography

* Writing a WebSocket server in C#, MDN Docs (<https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_server>)
* Tutorial: Get started with ASP.NET Core SignalR (<https://learn.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-7.0&tabs=visual-studio>)
* Use TcpClient and TcpListener (<https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/sockets/tcp-classes>)
* Internet Protocol Specification (<https://datatracker.ietf.org/doc/html/rfc791>)