**Демонстрация возможностей**

**технологий SignalR и JSON Web Token**

**на примере создания интернет-приложения**

**«Чат» в среде Microsoft Visual Studio 2022**

**с использованием платформы .NET framework 6.0**

**Часть 1. Технология SignalR**

Технология SignalR позволяет простым и удобным способом организовать в web-приложении механизм обмена сообщениями в реальном времени между сервером и клиентами.

Создание обработчиков, асинхронная отправка и прием сообщений требуют добавления нескольких простых строк кода.

Для добавления функциональности SignalR в приложение ASP.net сделайте следующее:

1. В браузерном приложении (Client) при помощи менеджера пакетов NuGet добавьте пакет «Microsoft.AspNetCore.SignalR.Client»:



1. В файле «Index.razor» (и других необходимых) добавьте ссылку на SignalR:

@using Microsoft.AspNetCore.SignalR.Client

1. В серверном приложении (Server) создайте новую папку «Hubs». Здесь будут храниться концентраторы сообщений.
2. Создайте в папке «Hubs» новый файл с кодом концентратора:

using Microsoft.AspNetCore.SignalR;

using BlazorWasmChat.Shared;

namespace BlazorWasmChat.Server.Hubs

{

public class ChatHub : Hub

{

public async Task SendMessage(ChatMessage message)

{

// Отправить сообщение всем подключенным клиентам

await Clients.All.SendAsync("ReceiveMessage", message);

}

}

}

Здесь мы создаем обработчик сообщения «SendMessage», который принимает объект типа ChatMessage рассылает всем клиентам полученное сообщение.

Если необходимо отправить сообщение конкретному клиенту или группе, то удалите «All» и укажите нужных получателей через методы

* AllExcept(string[] excludedConnectionIds) – всем, кроме
* Client(string connectionId) – конкретному клиенту
* Clients(string[] excludedConnectionIds) – перечисленным клиентам

или через методы Group, которые позволяют отправлять сообщения группам пользователей.

1. В файле «program.cs» серверного приложения подключите SignalR:

* Добавьте ссылку: using BlazorWasmChat.Server.Hubs;
* Добавьте сервис: builder.Services.AddSignalR();
* Добавьте регистрацию маршрута: app.MapHub<ChatHub>("/chathub");

Создание подключения на стороне клиента удобно разместить в методе OnInitializedAsync:

protected override async Task OnInitializedAsync()

{

hubConnection = new HubConnectionBuilder()

.WithUrl(NavigationManager.ToAbsoluteUri("/chathub"))

.WithAutomaticReconnect()

.Build();

Обработчик входящих сообщений на стороне клиента выглядит следующим образом:

hubConnection.On<ChatMessage>("ReceiveMessage", (message) =>

{

// Принято сообщение типа ChatMessage

});

Отправка сообщения из клиентского приложения:

await hubConnection.SendAsync("SendMessage", message);

Большим достоинством технологии является также то, что возможно создать несколько концентраторов сообщений и обозначить для них разные пути на сервере. Это позволяет оформить разные сервисы в виде отдельных изолированных компонентов, и подключать их к приложению по мере необходимости.

**Часть 2. Технология Json Web Tokens (JWT)**

Технология Json Web Tokens позволяет решить ряд задач по управлению доступом к данным и сервисам приложения:

* Авторизация пользователя. Проверив имя пользователя и пароль, сервер формирует и возвращает токен авторизации. Пользователь (или приложение) сохраняет его на своем компьютере и использует для получения доступа к сервисам. Токен также можно передать любым способом – через электронную почту, на диске или даже на бумажном носителе.
* Ограничение времени доступа к сервису. Токен может иметь ограниченный срок службы, после чего перестает приниматься сервером, и пользователю будет необходимо пройти авторизацию снова (оформить подписку) и получить новый токен. Также, токен может иметь время начала срока действия, и не будет приниматься до указанного времени.
* Разные токены для разных сервисов. Один пользователь может получить несколько токенов для доступа к разным сервисам, а также для изменения своего уровня доступа. Для выполнения ответственных операций можно использовать специальные токены. Точно так же, один токен может одновременно использоваться группой пользователей или приложений.
* После прохождения авторизации не нужны сторонние сервисы для проверки токена. В высоконагруженных приложениях авторизация при обращении к сервису должна выполняться максимально быстро. Web Token содержит цифровую подпись, которая легко проверяется без необходимости обращения к серверу авторизации.
* Независимость от языка программирования и среды. Получив токен, мы можем использовать его для доступа к сервисам из программ, написанных на языке JavaScript, php или даже на встроенном языке программы 1С.

Токен состоит из трех частей:

1. Заголовок – содержит информацию об алгоритме подписи/шифрования и типе токена:

{

"alg": "HS512",

"typ": "JWT"

}

1. Полезная нагрузка – объект Json, в котором можно разместить набор необходимых для идентификации пользователя значений:

{

"name": "Bender",

"role": "Bend"

}

Кроме пользовательских значений, могут быть добавлены служебные значения:

* iss – идентификатор издателя токена
* aud – список получателей токена
* exp – время конца срока действия токена в формате Unix
* nbf – время начала срока действия токена в формате Unix
* jti – уникальный идентификатор токена

1. Цифровая подпись токена по указанному алгоритму.

Сформировав 3 вышеуказанные части, необходимо закодировать их по-отдельности алгоритмом Base64-URL и склеить вместе в одну строку, поместив символ «.» (точка) между частями. Конечно же, создавать токен вручную нет необходимости – можно использовать готовые библиотеки, как будет показано далее.

Пример обращения к сервису из консольного приложения с использованием JWT:

var client = new HttpClient();

client.DefaultRequestHeaders.Authorization =

new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer",

"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy5" +

"4bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjo" +

"idGVzdCIsImV4cCI6MTY3OTUwMTMwMCwiaXNzIjoiQ2hhdFNlcnZlciJ9.Gb" +

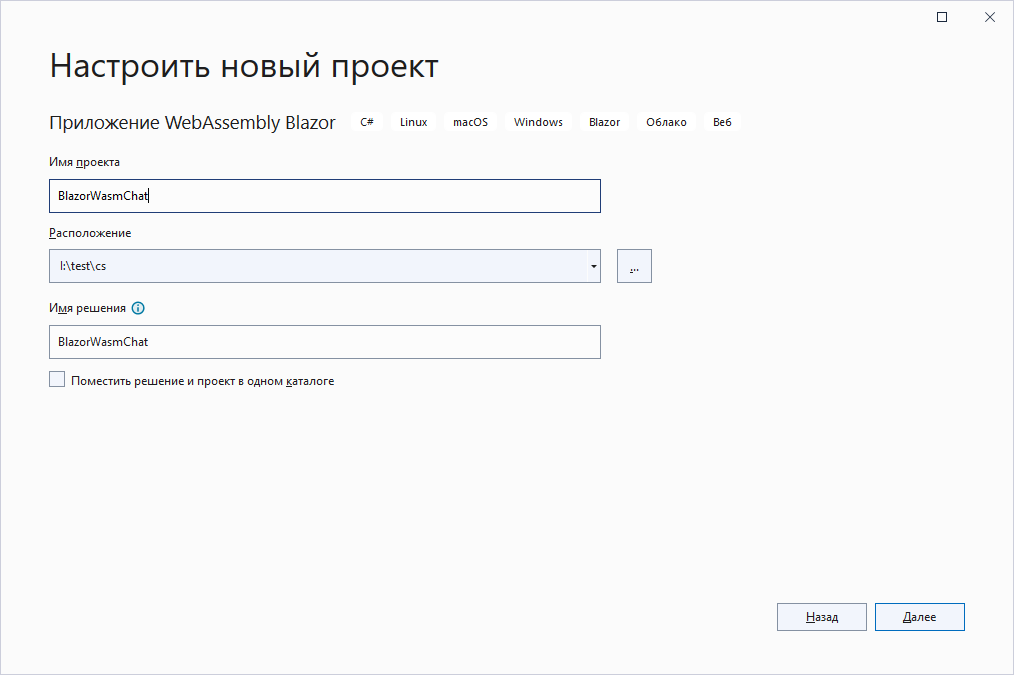
"3DziykIZykXcHTfIPxG8i62ROJPrAl5jQyeema8HU");

var s = await client.GetStringAsync("https://localhost:7121/private");

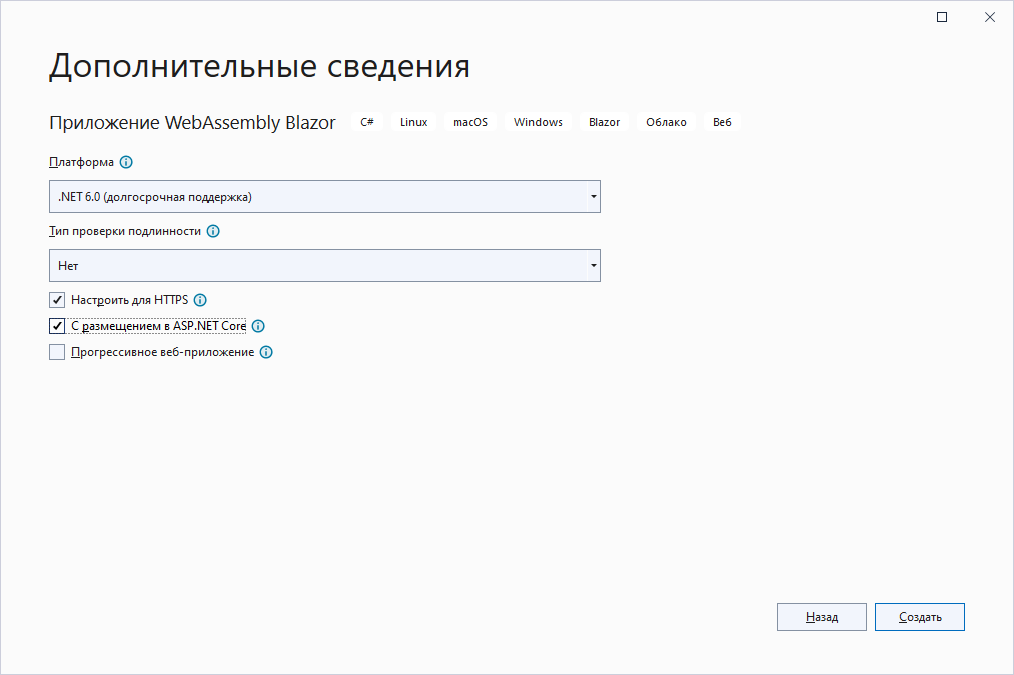
Console.WriteLine(s);

**Часть 3. Создание и настройка проекта**

1. В среде Visual Studio 2022 выберите Файл → Создать → Проект.
2. Выберите тип проекта «Приложение WebAssembly Blazor».
3. Введите название проекта и имя решения «BlazorWasmChat»:



1. Выберите расположение папки проекта и нажмите «Далее».
2. Выберите версию платформы «.NET 6.0 (долгосрочная поддержка) (LTS)»:



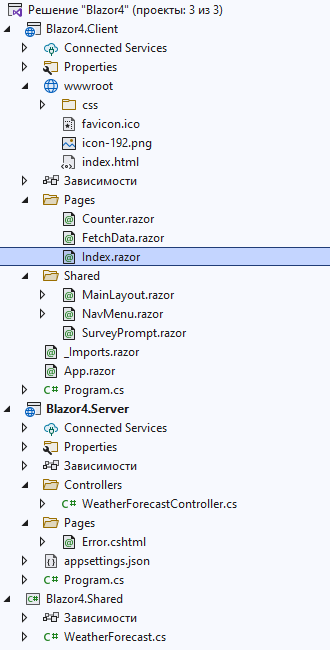
1. Тип проверки подлинности выставьте «Нет». Мы будем настраивать это вручную.
2. Поставьте флажок «С размещением в ASP.NET Core» и нажмите «Создать».
3. Будет создано приложение с несколькими страницами, содержащими тестовые данные для демонстрации базовой функциональности ASP.NET.

Также можно создать приложение из командной строки. Для этого откройте Командную строку или PowerShell, перейдите в папку, в которую нужно поместить проект:

D:\> md BlazorWasmChat

D:\> cd BlazorWasmChat

D:\BlazorWasmChat> dotnet new blazorwasm –ho



Среда Visual Studio создала для нас 3 проекта:

* Client – клиентское приложение, выполняемое в браузере клиента при помощи технологии WebAssembly.
* Server – основное приложение для выполнения на стороне сервера.
* Shared – библиотека общих типов, разделяемых между проектами.

1. Запустите приложение и убедитесь, что оно работает корректно.

**Часть 4. Очистка проекта**

1. В проекте «Client»:
   * Откройте папку «Pages» и удалите файлы «Counter.razor» и «FetchData.razor».
   * В файле «Index.razor» удалите строку с элементом «SurveyPrompt».
   * В папке «Shared» удалите «NavMenu.razor» и «SurveyPrompt.razor».
   * В файле «MainLayout.razor» оставьте только такой код:

@inherits LayoutComponentBase

<div class="page">

<main>

@Body

</main>

</div>

1. В проекте «Server»:
   * В папке «Controllers» удалите файл «WeatherForecastController.cs».
2. В проекте «Shared»:
   * Удалите файл «WeatherForecast.cs».
3. Соберите и запустите приложение. Если все сделано правильно, то в браузере должна появиться надпись «Hello, world! Welcome to your new app».

**Часть 5. Приложение «Чат»**

Для демонстрации применения технологий SignalR, WebTokens (JWT) и авторизации пользователя, создадим приложение «Чат».

Основа чата – обмен сообщениями между пользователями. Для удобства общения чат можно разделить на виртуальные комнаты. Давайте создадим типы, описывающие сообщения и комнаты:

1. Добавьте в проект «Shared» новый файл с названием «ChatMessage.cs». У сообщения могут быть следующие свойства:
   * Идентификатор (Id)
   * Время отправки (Issued)
   * Имя пользователя, от которого отправлено сообщение (FromUser)
   * Название комнаты (RoomId)
   * Текст сообщения (Text)

Код может выглядеть так:

namespace BlazorWasmChat.Shared

{

public class ChatMessage

{

public DateTime Issued { get; set; } = DateTime.Now;

public string? FromUser { get; set; }

public string? RoomId { get; set; }

public string? Text { get; set; }

public override string ToString()

{

return $"{FromUser}: {Text}";

}

}

}

Переопределение метода ToString() нужно для однообразного форматирования сообщений, если мы решим отображать их в нескольких местах программы.

1. Добавьте в проект «Shared» новый файл с названием «ChatRoom.cs». У комнаты могут быть следующие свойства:

* Идентификатор (Id) для сохранения в базе данных
* Название комнаты (Name)

Код может выглядеть так:

namespace BlazorWasmChat.Shared

{

public class ChatRoom

{

public int Id { get; set; } = 0;

public string Name { get; set; } = string.Empty;

}

}

Давайте сделаем html-разметку главной web-страницы приложения (Pages/Index.razor).

1. Заголовок файла:

@page "/"

@using Microsoft.AspNetCore.SignalR.Client

@using BlazorWasmChat.Shared;

@inject HttpClient Http

@inject NavigationManager NavigationManager

@implements IAsyncDisposable

<**PageTitle**>Чат</**PageTitle**>

Здесь мы обозначаем, что это корневая страница «/», добавляем ссылки и настраиваем внедрение зависимостей.

1. Боковая панель:

<div class="container">

<div class="sidebar">

<div style="background-color:whitesmoke;">

<label>Ваше имя:

<input @bind="userName" style="width: 280px;" />

</label>

@if (rooms is not null)

{

<p></p><label>Комнаты:

<select @bind="@roomId" size="10" style="width: 280px;">

@foreach (var room in rooms)

{

<option>@room.Name</option>

}

</select>

</label>

}

else

{

<p>Загрузка списка комнат, подождите...</p>

}

</div>

</div>

Здесь создается поле ввода имени, от которого отправляется сообщение (позже мы это поменяем), и отображается список комнат чата. Если список еще не загружен, выводится сообщение «Загрузка».

1. Область сообщений:

<div class="main">

<h2>@(roomId ?? "Выберите комнату")</h2>

<div>

<form action="javascript:void()">

<label>Сообщение:

<br><input @bind="messageText" style="width: 480px;" />

</label>

<input type="submit" @onclick="Send" hidden />

</form>

</div>

<div>

@foreach (var message in messages

.Where(x => x.RoomId == roomId)

.TakeLast(maxMessages))

{

<div class="message-time">@message.Issued</div>

<div class="message-text">@message</div>

}

</div>

</div>

</div>

Здесь мы определяем поле, где можно ввести текст сообщения, с невидимой кнопкой submit, реагирующей на нажатие клавиши Enter. Метод Send будет при помощи SignalR удаленно вызывать процедуру «SendMessage» на сервере. Далее следует поле вывода сообщений с фильтрацией по номеру комнаты.

1. Определим стили в файле «wwwroot/css/app.css»:

@import url('open-iconic/font/css/open-iconic-bootstrap.min.css');

html, body {

font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;

}

html {

height: 100%;

}

body {

min-height: 100%;

}

input, select, button {

margin: 4px;

padding: 4px;

}

select {

min-width: 200px;

}

div {

padding: 4px;

}

.message-time {

font-size: 9pt;

font-weight: 800;

color: cadetblue;

margin: 0px;

padding: 0px;

}

.message-text {

margin: 0px;

padding: 0px 0px 8px 8px;

}

.container {

display: table;

}

.main {

display: table-cell;

vertical-align: top;

}

.sidebar {

display: table-cell;

vertical-align: top;

width: 300px;

}

p {

padding-bottom: 0px;

}

Так мы создали простую форму со списком комнат и полями ввода и отображения сообщений.

Теперь давайте добавим клиентские компоненты технологии SignalR в проект:

1. В проект «Client» при помощи менеджера пакетов NuGet добавьте пакет «Microsoft.AspNetCore.SignalR.Client»:



1. В файле «Index.razor» добавьте ссылку на SignalR:

@using Microsoft.AspNetCore.SignalR.Client

1. В проекте «Server» создайте новую папку «Hubs».
2. Создайте в папке «Hubs» новый файл «ChatHub.cs»:

using Microsoft.AspNetCore.SignalR;

using BlazorWasmChat.Shared;

namespace BlazorWasmChat.Server.Hubs

{

public class ChatHub : Hub

{

public async Task SendMessage(ChatMessage message)

{

// Отправить сообщение всем клиентам, чтобы они добавили

// текст в свои хранилища сообщений

await Clients.All.SendAsync("ReceiveMessage", message);

}

}

}

1. В файле «program.cs» проекта «Server» подключите SignalR:

* Добавьте ссылку: using BlazorWasmChat.Server.Hubs;
* Добавьте сервис: builder.Services.AddSignalR();
* Добавьте регистрацию маршрута: app.MapHub<ChatHub>("/chathub");

Теперь создадим основной код браузерного приложения. Нам нужно определить:

* Константу «maxMessages», в которой будет храниться максимальное количество сообщений, отображаемых на экране
* Объект подключения к концентратору SignalR «hubConnection»
* Списки комнат и сообщений
* Поля формы: имя пользователя «userName», текст сообщения «messageText», идентификатор выбранной комнаты «roomId»
* Таймер для периодического обновления списка комнат (для демонстрации получения содержимого в формате Json)

В файле «Pages/Index.razor» добавьте код:

@code {

private readonly int maxMessages = 10;

private HubConnection? hubConnection;

private System.Threading.Timer? timer;

private List<ChatMessage> messages = new List<ChatMessage>();

private ChatRoom[]? rooms;

private string? userName;

private string? messageText;

private string? roomId;

Сразу после загрузки страницы мы можем подключиться к концентратору SignalR сервера. Добавьте код:

protected override async Task OnInitializedAsync()

{

hubConnection = new HubConnectionBuilder()

.WithUrl(NavigationManager.ToAbsoluteUri("/chathub"))

.WithAutomaticReconnect()

.Build();

hubConnection.On<ChatMessage>("ReceiveMessage", (message) =>

{

messages.Add(message);

if (message.FromUser == userName &&

message.RoomId == roomId &&

message.Text == messageText)

{

messageText = "";

}

StateHasChanged();

});

await hubConnection.StartAsync();

Здесь мы подключаемся к концентратору по адресу «/chathub», мы его ранее определили в файле «program.cs» проекта «Server». Далее добавляем обработчик сообщения «ReceiveMessage» и запускаем в работу подключение. Теперь SignalR сделает остальную работу.

Также давайте сделаем загрузку списка комнат по таймеру:

var timerLocked = false;

timer = new System.Threading.Timer(async (object? stateInfo) =>

{

// Простейший механизм блокировки повторного входа в процедуру

if (timerLocked)

{

return;

}

timerLocked = true;

try

{

rooms = await Http.GetFromJsonAsync<ChatRoom[]>("/api/chatrooms");

StateHasChanged();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

timerLocked = false;

}, null, 2000, 5000);

}

Здесь через 2 секунды запускается таймер, и каждые 5 секунд он отправляет запрос серверу для получения списка комнат.

Далее сделаем отправку сообщений (Send), привязанную к невидимому элементу submit формы:

private async Task Send()

{

// Проверка заполнения полей формы перед отправкой

if (userName is null || userName.Length == 0)

{

return;

}

if (messageText is null || messageText.Length == 0)

{

return;

}

if (roomId is null)

{

return;

}

// Проверка подключения к hub

if (hubConnection is null ||

hubConnection.ConnectionId is null)

{

return;

}

// Подготовить и отправить новое сообщение

var message = new ChatMessage

{

FromUser = userName,

RoomId = roomId,

Text = messageText

};

await hubConnection.SendAsync("SendMessage", message);

}

Здесь нам нужно проверить корректность заполнения полей формы, убедиться в наличии соединения с сервером, а затем одной простой строкой кода отправить сообщение путем вызова удаленной процедуры «SendMessage». Объект типа ChatMessage будет автоматически сериализован – десериализован.

Для закрытия соединения при завершении работы приложения дополним код методом «DisposeAsync»:

public async ValueTask DisposeAsync()

{

if (hubConnection is not null)

{

await hubConnection.DisposeAsync();

}

}

}

Соединение SignalR теперь будет корректно завершаться.

Теперь нам нужно добавить контроллер, отдающий список комнат. В папке «Controllers» проекта «Server»добавьте файл «ChatRoomsController.cs»:

using Microsoft.AspNetCore.Mvc;

using BlazorWasmChat.Shared;

using BlazorWasmChat.Server.Hubs;

using Microsoft.AspNetCore.SignalR;

using Microsoft.AspNetCore.Authorization;

namespace BlazorWasmChat.Server.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class ChatRoomsController : ControllerBase

{

[HttpGet]

public IEnumerable<ChatRoom> Get()

{

return Chat.Rooms.ToArray();

}

}

}

Этот простейший контроллер отдает список комнат в формате Json по адресу «/api/chatrooms».

Нам нужно создать еще хранилище данных. Пусть это будет простое хранилище в оперативной памяти компьютера. Создайте новый файл «chat.cs»:

using BlazorWasmChat.Shared;

public class Chat

{

public static List<ChatRoom> Rooms { get; } = new List<ChatRoom>();

public static List<ChatMessage> Messages { get; } = new List<ChatMessage>();

static Chat()

{

Rooms.Add(new ChatRoom { Id = 1, Name = "Общий чат" });

Rooms.Add(new ChatRoom { Id = 2, Name = "Новости" });

Rooms.Add(new ChatRoom { Id = 3, Name = "Политика" });

Rooms.Add(new ChatRoom { Id = 4, Name = "Торренты" });

}

}

Здесь мы вручную создаем список комнат, но так же можно загрузить его из базы данных или из другого источника.

Приложение готово к запуску. Соберите проект и запустите приложение. Через 2 секунды после загрузки должен появиться список комнат.

В поле «Ваше имя» введите свое имя, в поле «Комнаты» выберите нужную комнату. После этого можно будет ввести текст сообщения для отправки. Клавиша Enter отправляет сообщение.

Для тестирования приложения откройте его во втором окне браузера. Набрав сообщение в одном окне, можно увидеть появление его в другом.

**Часть 6. Установка и настройка JWT**

1. При помощи менеджера пакетов NuGet добавьте в клиентское приложение пакеты «Microsoft.AspNetCore.Components.Authorization» и «Blazored.LocalStorage». Второй пакет не является обязательным, но он значительно упрощает работу с локальным хранилищем браузера. Без этого пакета нам пришлось бы сохранять значения при помощи функций JavaScript.
2. Добавьте в клиентское приложение файл «CustomAuthStateProvider.cs»:

using Blazored.LocalStorage;

using Microsoft.AspNetCore.Components.Authorization;

using System.Net.Http.Headers;

using System.Security.Claims;

using System.Text.Json;

namespace BlazorWasmChat.Client

{

public class CustomAuthStateProvider : AuthenticationStateProvider

{

Нам понадобятся LocalStorage и HttpClient. Их получим через DI:

ILocalStorageService \_localStorage;

HttpClient \_httpClient;

public CustomAuthStateProvider(ILocalStorageService localStorage, HttpClient httpClient)

{

\_localStorage = localStorage;

\_httpClient = httpClient;

}

Эта функция будет декодировать токен, обновлять и возвращать состояние аутентификации по нему:

public override async Task<AuthenticationState> GetAuthenticationStateAsync()

{

var token = await \_localStorage.GetItemAsStringAsync("token");

\_httpClient.DefaultRequestHeaders.Authorization = null;

var identity = new ClaimsIdentity();

if (!string.IsNullOrEmpty(token))

{

identity = new ClaimsIdentity(ParseClaimsFromJwt(token), "jwt");

\_httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token.Replace("\"", ""));

}

var user = new ClaimsPrincipal(identity);

var state = new AuthenticationState(user);

NotifyAuthenticationStateChanged(Task.FromResult(state));

return state;

}

Здесь особо стоит отметить вызов «NotifyAuthenticationStateChanged». Вызов этого метода позволяет обновить все элементы «Authorized».

Для быстрого извлечения данных из токена удобно использовать такую функцию:

public static IEnumerable<Claim> ParseClaimsFromJwt(string jwt)

{

var elements = jwt.Split('.');

if (elements.Length < 3)

{

return Array.Empty<Claim>();

}

var payload = elements[1];

switch (payload.Length % 4)

{

case 2: payload += "=="; break;

case 3: payload += "="; break;

}

var data = Convert.FromBase64String(payload);

var kv = JsonSerializer.Deserialize<Dictionary<string, object>>(data);

return kv!.Select(p => new Claim(p.Key, p.Value.ToString()!));

}

}

}

1. Обычный файл «App.razor» нам больше не подходит – в него нужно внести изменения для поддержки авторизации:

<**CascadingAuthenticationState**>

<**Router** **AppAssembly**="@typeof(App).Assembly">

<**Found** **Context**="routeData">

<**AuthorizeRouteView** **RouteData**="@routeData" **DefaultLayout**="@typeof(MainLayout)">

</**AuthorizeRouteView**>

<**FocusOnNavigate** **RouteData**="@routeData" **Selector**="h1" />

</**Found**>

<**NotFound**>

<**PageTitle**>Not found</**PageTitle**>

<**LayoutView** **Layout**="@typeof(MainLayout)">

<p role="alert">Страница не найдена!</p>

</**LayoutView**>

</**NotFound**>

</**Router**>

</**CascadingAuthenticationState**>

1. Давайте добавим кнопку авторизации пользователя на главную страницу приложения. Для этого в файле «Index.razor» внесите изменения:

<div style="background-color:whitesmoke;">

<**AuthorizeView**>

<**Authorized**>

<label>Ваше имя:

<input value="@context?.User?.Identity?.Name" style="width: 280px;" />

</label>

<p/><button @onclick="Logout" >Выйти</button>

</**Authorized**>

<**NotAuthorized**>

Пожалуйста, <a href="/login">авторизуйтесь</a>

</**NotAuthorized**>

</**AuthorizeView**>

Теперь мы не используем поле «Ваше имя», и пользователь не сможет кем-то другим. Если пользователь не вошел в систему (NotAuthorized), то предложим ему авторизоваться.

1. В файле «Index.razor» давайте вынесем запуск подключения SignalR в отдельный метод, ему мы будем передавать токен авторизации:

protected override async Task OnInitializedAsync()

{

var token = await LocalStorage.GetItemAsStringAsync("token");

await startHubConnection(token);

Метод будет выглядеть так:

private async Task startHubConnection(string? token)

{

// Создать подключение SignalR

hubConnection = new HubConnectionBuilder()

.WithUrl(NavigationManager.ToAbsoluteUri("/chathub"), options =>

{

options.AccessTokenProvider = () => Task.FromResult(token);

})

.WithAutomaticReconnect()

.Build();

// Добавить обработчик входящего сообщения

hubConnection.On<ChatMessage>("ReceiveMessage", (message) =>

{

messages.Add(message);

if (message.FromUser == userName &&

message.RoomId == roomId &&

message.Text == messageText)

{

messageText = "";

}

StateHasChanged();

});

await hubConnection.StartAsync();

}

Здесь добавляется токен авторизации, остальное без изменений.

Там также нужно внести изменения в метод Send:

private async Task Send()

{

var auth = await AuthStateProvider.GetAuthenticationStateAsync();

var userName = auth?.User?.Identity?.Name;

Пока для демонстрации мы будем сами получать из токена имя пользователя. Позже подстановку имени нужно перенести на сторону сервера.

1. Следующая важная функция – выход пользователя из системы:

async Task Logout()

{

// 1. Удалить токен авторизации из хранилища

await LocalStorage.RemoveItemAsync("token");

// 2. Обновить состояние авторизации

await AuthStateProvider.GetAuthenticationStateAsync();

// 3. Перезапустить соединение SignalR с пустым токеном

await hubConnection!.DisposeAsync();

await startHubConnection("");

// 4. Перейти на главную страницу

NavigationManager.NavigateTo("/");

}

Когда пользователь нажимает кнопку «Выход», необходимо удалить токен авторизации из локального хранилища, после чего завершить и снова открыть соединение SignalR. В рабочем приложении, возможно, нам также стоит сделать токен недействительным на сервере, иначе пользователь через API сможет отправлять сообщения после выхода из системы. Если это поведение не является ожидаемым, то необходимо отозвать токен.

1. Теперь нам необходима страница авторизации. Добавьте в папку «Pages» новый компонент Razor «Login.razor»:

@page "/login"

@inject HttpClient Http

@inject AuthenticationStateProvider AuthStateProvider

@inject ILocalStorageService LocalStorage

@inject NavigationManager NavigationManager

<div class="d-flex justify-content-center">

<div>

<**AuthorizeView**>

<**NotAuthorized** **Context**="Auth">

<h3>Авторизация</h3>

<**EditForm** **Model**="user" **OnSubmit**="HandleLogin">

<p/>Имя пользователя:

<br/><**InputText** @bind-Value="user.UserName" />

<p/>Пароль:

<br/><**InputText** @bind-Value="user.Password" type="password" />

<p/><button type="submit">Вход</button>

</**EditForm**>

</**NotAuthorized**>

<**Authorized**>

<div>

<p/>Вот ваш токен авторизации.

<p/>Вы можете использовать его для доступа к сервисам без ввода пароля:

<p/><textarea cols="40" rows="10">@tok</textarea>

<p/><a href="/">Перейти в чат</a>

</div>

</**Authorized**>

</**AuthorizeView**>

</div>

</div>

Если пользователь не авторизован, то предложим ему ввести имя и пароль. После авторизации предоставим пользователю его токен и предложим перейти на главную страницу приложения.

@code {

UserLogin user = new UserLogin();

string tok = string.Empty;

async Task HandleLogin()

{

var result = await Http.PostAsJsonAsync("api/auth", user);

var token = await result.Content.ReadAsStringAsync();

await LocalStorage.SetItemAsync("token", token);

await AuthStateProvider.GetAuthenticationStateAsync();

tok = token;

// Раскомментируйте эту строку, чтобы сразу переходить в чат

// NavigationManager.NavigateTo("/");

}

}

Сначала обращаемся к контроллеру авторизации, он возвращает токен. По токену обновляем состояние и в случае успеха, NotifyAuthenticationStateChanged внутри GetAuthenticationStateAsync обновит поля Authorized, пользователь получит токен и сможет отправлять сообщения.

1. При помощи менеджера пакетов NuGet добавьте в **серверное** приложение пакет «Microsoft.AspNetCore.Authentication.JwtBearer».
2. В файл «program.cs» добавьте директивы using:

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.AspNetCore.Authorization;

using BlazorWasmChat.Server.Hubs;

using BlazorWasmChat.Shared;

using BlazorWasmChat.Server.Authorization;

1. Добавьте сервисы авторизации:

builder.Services.AddAuthorization();

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = Auth.MakeTokenValidationParameters();

options.Events = new JwtBearerEvents

{

// Для корректной работы авторизации в SignalR нам необходимо

// вручную извлечь из строки запроса токен авторизации

// и преобразовать его к исходному виду

OnMessageReceived = context =>

{

var path = context.HttpContext.Request.Path;

// Нам нужны только запросы к концентраторам сообщений

// Добавьте здесь другие пути при необходимости

if (path.StartsWithSegments("/chathub"))

{

// Это запрос SignalR

// Получить токен из параметров строки запроса

var accessToken = context.Request.Query["access\_token"].ToString();

// Удалить лишние символы

if (!string.IsNullOrEmpty(accessToken) && accessToken.Length > 1 && accessToken.StartsWith("\""))

{

accessToken = accessToken.Substring(1, accessToken.Length - 2);

}

// Передать токен

context.Token = accessToken;

}

return Task.CompletedTask;

}

};

});

builder.Services.AddAuthenticationCore();

Кроме подключения сервисов, здесь необходимо настроить параметры проверки токенов авторизации и предварительную обработку сообщений. В связи с тем, что SignalR не использует значение заголовка Authorization, а передает токен через адресную строку, нам нужно извлечь токен и отправить его в обычном виде.

1. Настройте после строки «app.UseRouting()» использование JWT:

app.UseAuthentication();

app.UseAuthorization();

В примерах использования авторизации .net 6 написано, что порядок объявления служб имеет значение, поэтому не перемещайте app.Use\* и app.Map\* без необходимости.

1. Добавьте тестовый контроллер авторизации в приложение. Для этого в папку «Controllers» добавьте новый файл «AuthController.cs»:

using Microsoft.AspNetCore.Mvc;

using BlazorWasmChat.Shared;

using BlazorWasmChat.Server.Authorization;

namespace BlazorWasmChat.Server.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class AuthController : ControllerBase

{

[HttpPost]

public async Task<ActionResult<string>> Login(UserLogin request)

{

var token = Auth.AuthorizeByNameAndPassword(request.UserName, request.Password);

return token;

}

}

}

Здесь мы настроили POST-метод «Login», который будет возвращать токен авторизации по параметрам request (имя пользователя и пароль).

1. Создайте новую папку «Auth» и поместите в нее файл «Auth.cs». В этом файле будет размещены данные для создания токенов и процедура авторизации:

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.IdentityModel.Tokens;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Text;

namespace BlazorWasmChat.Server.Authorization

{

public class Auth

{

Добавим приватный ключ, название издателя и срок действия токена по умолчанию:

private static readonly string \_jwtPrivateKey = "2EF2D6BCB91D44E5877FF39200409E97";

private static readonly string \_jwtIssuer = "ChatServer";

private static readonly int \_jwtDays = 366; // Будем выдавать токен на 1 год

Функция создания симметричного ключа:

private static SymmetricSecurityKey CreateSecurityKey()

{

return new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_jwtPrivateKey));

}

Параметры проверки токенов. Здесь вы можете добавить дополнительные значения:

public static TokenValidationParameters MakeTokenValidationParameters()

{

// Создать параметры проверки токенов

var parameters = new TokenValidationParameters

{

// Разрешить проверку ключа безопасности

ValidateIssuerSigningKey = true,

IssuerSigningKey = CreateSecurityKey(),

// Включить проверку издателя токена (это наш сервер)

ValidateIssuer = true,

ValidIssuer = \_jwtIssuer,

// Не нужно проверять получателя токена

ValidateAudience = false,

// Включить проверку времени дайствия

ValidateLifetime = true

};

return parameters;

}

Далее будет следовать функция авторизации, возвращающая токен. Здесь в случае успеха мы создаем токен с параметрами Claims. Если имя пользователя или пароль не найдены, то можно вернуть пустой токен.

public static string AuthorizeByNameAndPassword(string username, string password)

{

// Здесь необходимо проверить имя пользователя и пароль

// После чего можно сформировать и вернуть токен

// В этом месте добавляем наши произвольные значения

var claims = new List<Claim>

{

new Claim(ClaimTypes.Name, username)

};

var signingCredentials = new SigningCredentials(CreateSecurityKey(), SecurityAlgorithms.HmacSha256);

var result = new JwtSecurityToken(

issuer: \_jwtIssuer,

claims: claims,

expires: DateTime.UtcNow.Add(TimeSpan.FromDays(\_jwtDays)),

signingCredentials: signingCredentials);

return new JwtSecurityTokenHandler().WriteToken(result);

}

}

}

Код в блоке проверки будет создан позже. Это могут быть несколько проверок по различным спискам.

1. Добавьте тестовую защищенную страницу для проверки ограничения доступа:

app.Map("/private", [Authorize] () =>

{

return "Защищенное содержимое";

});

1. Теперь необходимо добавить атрибут «Authorize» в обработчик сообщений ChatHub (файл Hubs/ChatHub.cs):

[Authorize]

public async Task SendMessage(ChatMessage message)

1. И последнее, в проект «Shared» нам необходимо добавить модель «UserLogin». Создайте новый файл «UserLogin.cs»:

namespace BlazorWasmChat.Shared

{

public class UserLogin

{

public string UserName { get; set; } = string.Empty;

public string Password { get; set; } = string.Empty;

}

}

Проверьте работу приложения. Авторизация и выход из системы должны проходить успешно. Отправка сообщений должна выполняться только в том случае, если пользователь авторизован.

Т.к. мы не добавили проверку имени пользователя и пароля, можно вводить любые данные при авторизации. Мы сделаем проверку позже.