**Демонстрация возможностей**

**технологий SignalR и JSON Web Token**

**на примере создания интернет-приложения**

**«Чат»**

**в среде Microsoft Visual Studio 2022**

**с использованием платформы .NET 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.

Обработка сообщений на стороне сервера осуществляется при помощи концентраторов SignalR:

public class ChatHub : Hub

{

[Authorize]

public async Task SendMessage(ChatMessage message)

{

// Отправить сообщение всем подключенным клиентам

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

}

}

Клиент удаленно вызывает метод «SendMessage» на сервере, который может быть асинхронным. Обработав данные, сервер, в свою очередь, может вызвать удаленный метод одного или нескольких клиентов. Сериализация-десериализация данных происходит автоматически.

Создание обработчика 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);

SignalR поддерживает ограничение доступа на основе WebToken, достаточно лишь указать атрибут «Authorize», чтобы выполнялась проверка уровня доступа. В случае разрыва связи, подключение может быть восстановлено автоматически.

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 различных языках программирования. WebToken может быть использован даже IoT-устройствами, имеющими всего несколько килобайт ОЗУ.

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" +

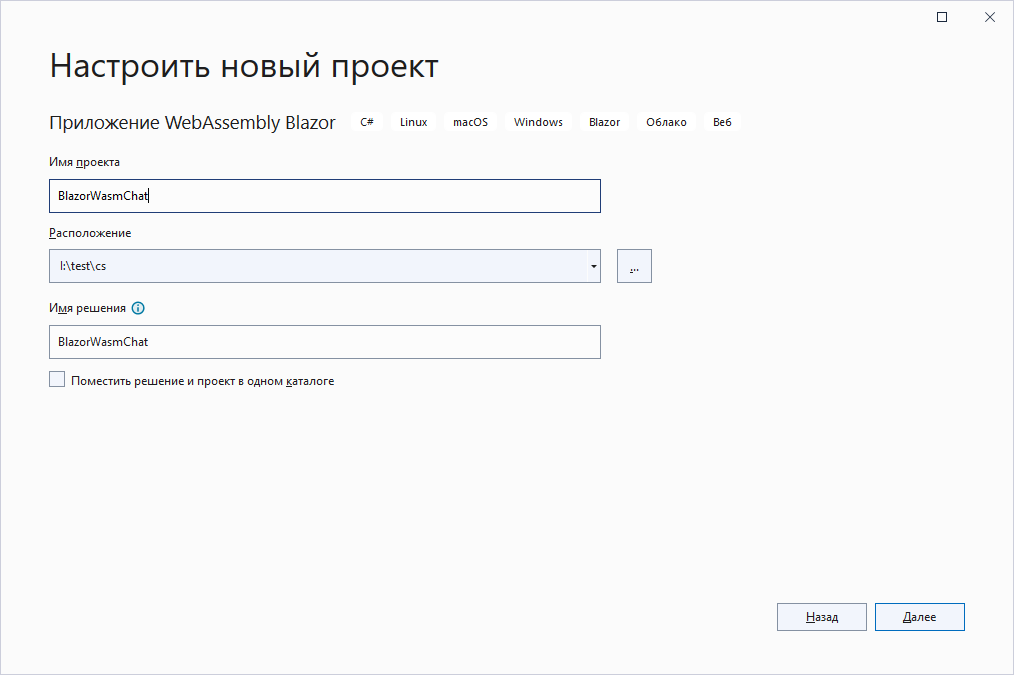
"3DziykIZykXcHTfIPxG8i62ROJPrAl5jQyeema8HU");

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

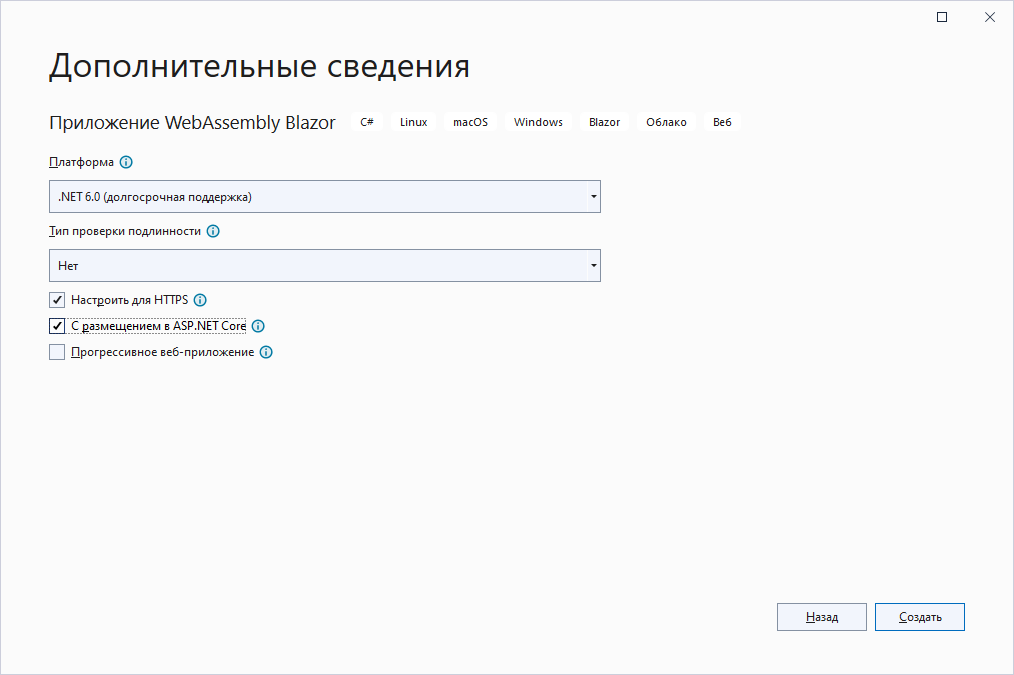
Console.WriteLine(s);

**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)»:



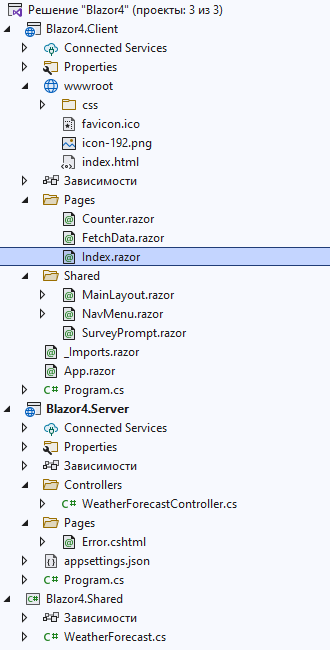
1. Тип проверки подлинности выставьте «Нет». Мы будем настраивать это вручную.
2. Поставьте флажок «С размещением в ASP.NET Core» и нажмите «Создать».
3. Будет создано приложение с несколькими страницами, содержащими тестовые данные для демонстрации базовой функциональности ASP.NET.

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».

Сначала необходимо добавить SignalR в проект Client. Откройте менеджер пакетов NuGet и установите пакет «Microsoft.AspNetCore.SignalR.Client»:



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;

}

}

1. Нам также понадобится информация о пользователе: имя и пароль. Добавьте в проект «Shared» файл «UserLogin.cs»:

namespace BlazorWasmChat.Shared

{

public class UserLogin

{

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

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

}

}

Давайте сделаем html-разметку главной web-страницы приложения «Pages/Index.razor»:

1. Заголовок файла:

@page "/"

@using BlazorWasmChat.Shared

@using Microsoft.AspNetCore.SignalR.Client

@inject HttpClient Http

@inject NavigationManager NavigationManager

@implements IAsyncDisposable

<**PageTitle**>Чат</**PageTitle**>

<div class="container">

1. Боковая панель:

<div class="sidebar">

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

Поле для ввода имени пользователя:

<label>Ваше имя:

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

</label>

Список комнат:

@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 будем получать имя из учетной записи при авторизации), 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>

Форма ввода текста сообщения с невидимой кнопкой submit:

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

<label>Сообщение:

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

</label>

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

</form>

</div>

При нажатии на клавишу Enter будет вызван метод «Send», который отправит сообщение.

Добавим поле для вывода сообщений чата. Т.к. имеется несколько комнат и общий список сообщений, при помощи Linq сделаем фильтр:

<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>

1. Теперь создадим код страницы:

@code {

Определим максимальное количество сообщений, видимых на экране:

private readonly int maxMessages = 10;

Добавим подключение SignalR:

private HubConnection? hubConnection;

Загрузку списка комнат сделаем по таймеру для демонстрации обращения к контроллеру при помощи HttpClient:

private System.Threading.Timer? timer;

Создадим хранилища сообщений и комнат:

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

private ChatRoom[]? rooms;

Поля формы:

private string? userName;

private string? messageText;

private string? roomId;

Теперь нам нужно при открытии страницы создать соединение с SignalR и запустить таймер:

protected override async Task OnInitializedAsync()

{

Подключимся к концентратору SignalR:

await startHubConnection("");

Запустим таймер. Первое срабатывание – через 2 секунды, далее каждые 5 секунд. Чтобы таймер не мог быть запущен повторно во время выполнения его функции, добавим простейший механизм блокировки:

var timerLocked = false;

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

{

// Простейший механизм блокировки повторного входа в процедуру

if (timerLocked)

{

return;

}

timerLocked = true;

// Загрузить список комнат

try

{

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

StateHasChanged();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

timerLocked = false;

}, null, 2000, 5000);

}

Создадим функцию подключения к концентратору SignalR:

private async Task startHubConnection(string? token)

{

// Создать подключение SignalR

hubConnection = new HubConnectionBuilder()

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

.WithAutomaticReconnect()

.Build();

Для приема входящих сообщений добавим функцию ReceiveMessage, которая будет добавлять сообщение в список и перерисовывать страницу:

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

{

messages.Add(message);

StateHasChanged();

});

Запустим подключение:

await hubConnection.StartAsync();

}

Теперь создадим функцию Send, отправляющую сообщение:

private async Task Send()

{

Проверим правильность заполнения полей:

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

{

return;

}

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

{

return;

}

if (roomId is null)

{

return;

}

Нам нужно активное соединение SignalR:

if (hubConnection is null ||

hubConnection.ConnectionId is null)

{

return;

}

Подготовим и отправим сообщение:

var message = new ChatMessage

{

FromUser = userName,

RoomId = roomId,

Text = messageText

};

// Очистить поле ввода текста сообщения

messageText = string.Empty;

// Вызвать метод SendMessage на сервере

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

}

И, наконец, нам следует корректно завершить подключение при закрытии страницы:

public async ValueTask DisposeAsync()

{

if (hubConnection is not null)

{

await hubConnection.DisposeAsync();

}

}

}

1. Переопределим некоторые стили в файле «wwwroot/css/app.css» для того, чтобы элементы были аккуратно размещены на формах:

@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 серверное приложение:

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:

using BlazorWasmChat.Server.Hubs;

Добавьте в коллекцию сервисов SignalR (перед builder.Build()):

builder.Services.AddSignalR();

В нижней части файла, перед «app.Run()» зарегистрируйте концентратор сообщений SignalR:

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

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

using Microsoft.AspNetCore.Mvc;

using BlazorWasmChat.Shared;

namespace BlazorWasmChat.Server.Controllers

{

[ApiController]

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

public class ChatRoomsController : ControllerBase

{

[HttpGet]

public IEnumerable<ChatRoom> Get()

{

По запросу GET будем отдавать список комнат в формате Json по адресу «/api/chatrooms»:

return Chat.Rooms.ToArray();

}

}

}

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

{

// Данные приложения

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

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

public static List<UserLogin> Users { get; } = new List<UserLogin>();

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" });

Users.Add(new UserLogin { UserName = "admin", Password = "1234" });

}

}

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()

{

Получим токен из локального хранилища:

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

Создадим пустое удостоверение пользователя:

\_httpClient.DefaultRequestHeaders.Authorization = null;

var identity = new ClaimsIdentity();

Проверим токен. Если он правильный, то попробуем декодировать его:

if (IsTokenValid(token))

{

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

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

}

Обновим состояние авторизации:

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)

{

Всего в токене 3 части, разделенные точками. Нам нужна 2-я часть, payload:

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

if (elements.Length < 3)

{

return Array.Empty<Claim>();

}

var payload = elements[1];

// Это Base64 без дополняющих символов "=", добавить их при необходимости

switch (payload.Length % 4)

{

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

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

}

// Раскодировать строку

var data = Convert.FromBase64String(payload);

// Преобразовать в словарь значений

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

// Вернуть значения как список

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

}

Создадим функцию простой проверки соответствия токена шаблону:

public bool IsTokenValid(string token)

{

if (token is null)

{

return false;

}

// Правильно составленный токен состоит из 3 частей, разделенных точками

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

if (elements.Length < 3)

{

return false;

}

return true;

}

}

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**>

Если пользователь авторизован – выведем его имя и кнопку выхода:

<**Authorized**>

<label>Ваше имя:

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

</label>

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

</**Authorized**>

Если пользователь не авторизован – предложим войти в систему:

<**NotAuthorized**>

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

</**NotAuthorized**>

</**AuthorizeView**>

Теперь мы не используем поле «Ваше имя», и пользователь не сможет представиться кем-то другим. Если пользователь не вошел в систему (NotAuthorized), то предложим ему авторизоваться. Удалите поле «string userName», оно больше не будет использоваться.

1. В файле «Index.razor» нам понадобится токен авторизации:

protected override async Task OnInitializedAsync()

{

// Получить токен авторизации из локального хранилища

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

Добавим токен в параметры соединения SignalR:

await startHubConnection(token);

Теперь 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

{

Вначале объявим единый интерфейс для различных поставщиков процедур авторизации:

public interface IChatAuthProvider

{

bool TryAuthorize(UserLogin user);

}

Теперь класс Auth:

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));

}

Это 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 действия

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

Последовательно проверим результат наших процедур авторизации:

var auth = false;

// Проверим всех наших провайдеров

foreach (var provider in Chat.AuthProviders)

{

auth |= provider.TryAuthorize(user);

if (auth)

{

break;

}

}

if (!auth)

{

// Ни один из модулей не смог выполнить авторизацию

return string.Empty;

}

Одна из процедур прошла успешно, создадим токен:

// Реквизиты пользователя

var claims = new List<Claim>();

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);

}

}

}

1. Добавьте тестовую защищенную страницу для проверки ограничения доступа:

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

{

return "Защищенное содержимое";

});

Чтобы проверить доступ при помощи полученного токена, можно использовать пример консольного клиента из части 2. Если токен введен правильно – в консоли появится сообщение «Защищенное содержимое».

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

[Authorize]

public async Task SendMessage(ChatMessage message)

1. Откройте файл «Chat.cs» и добавьте в него список поставщиков авторизации:

public static List<IChatAuthProvider> AuthProviders { get; } = new List<IChatAuthProvider>();

1. Создайте временного тестового поставщика авторизации (можно в файле Chat.cs):

public class TestAuthProvider : IChatAuthProvider

{

public bool TryAuthorize(UserLogin user)

{

return true;

}

}

1. Добавьте поставщика в файле «Program.cs» перед app.Run():

Chat.AuthProviders.Add(new TestAuthProvider());

Когда создадим настоящих поставщиков, эту строку будет необходимо удалить!

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

Можно вводить любое имя пользователя и пароль, т.к. проверка всегда возвращает true.

**Часть 7. Локальная авторизация пользователя**

1. Откройте файл «Chat.cs» и добавьте в него методы проверки и добавления пользователя в списки пользователей сервера:

public static bool UserExists(UserLogin user)

{

return Chat.Users.Exists(x => x.UserName == user.UserName);

}

public static bool AddUser(UserLogin user)

{

Chat.Users.Add(new UserLogin { UserName = user.UserName, Password = user.Password });

return true;

}

public static bool PasswordCorrect(UserLogin user)

{

return Chat.Users.Exists(x => x.UserName == user.UserName && x.Password == user.Password);

}

Код этих методов можно легко поменять на код, обращающийся к базе данных.

1. В папку «Auth» добавьте новый файл «LocalAuthProvider.cs»:

using BlazorWasmChat.Shared;

namespace BlazorWasmChat.Server.Authorization

{

// Поставщик локальной авторизации

public class LocalAuthProvider : IChatAuthProvider

{

public bool TryAuthorize(UserLogin user)

{

// Если имя пользователя и пароль совпадают, то разрешить вход

bool res = Chat.PasswordCorrect(user);

return res;

}

}

}

Здесь мы проверяем наличие пользователя и соответствие пароля данным, имеющимся на сервере, и в случае соответствия, разрешаем доступ.

1. Добавьте поставщика в файле «Program.cs» перед app.Run():

Chat.AuthProviders.Add(new LocalAuthProvider());

Запустите приложение, и попробуйте авторизоваться с тестовым именем «admin»и паролем «1234». Авторизация должна проходить, а в случае использования любого другого имени будет выдаваться сообщение об ошибке.

**Часть 8. Авторизация пользователя через ActiveDirectory**

1. При помощи менеджера пакетов NuGet добавьте в серверный проект пакет «System.DirectoryServices».
2. Для проверки авторизации пользователя в ActiveDirectory мы можем попробовать получить содержимое каталога. Для этого в папке «Auth» добавьте новый файл «ActiveDirectoryAuthProvider.cs»:

using BlazorWasmChat.Shared;

using System.DirectoryServices;

namespace BlazorWasmChat.Server.Authorization

{

// Поставщик авторизации в ActiveDirectory

public class ActiveDirectoryAuthProvider : IChatAuthProvider

{

public bool TryAuthorize(UserLogin user)

{

Сначала проверим входные данные:

if (user.UserName is null || user.Password is null)

{

return false;

}

Имя пользователя в ActiveDirectory должно содержать символ «\»

if (!user.UserName.Contains('\\'))

{

return false;

}

bool res = false;

Попробуем авторизоваться и прочитать каталог:

try

{

// Извлечь из полного имени пользователя домен

var domainName = user.UserName.Substring(0, user.UserName.IndexOf('\\'));

// и имя пользователя

var username = user.UserName.Substring(domainName.Length + 1);

// Попробовать подключиться и прочитать каталог

var direenty = new DirectoryEntry("LDAP://" + domainName, username, user.Password);

var search = new DirectorySearcher(direenty);

var results = search.FindOne();

res = true;

}

catch

{

}

return res;

}

}

}

1. Добавьте поставщика в файле «Program.cs» перед app.Run():

Chat.AuthProviders.Add(new ActiveDirectoryAuthProvider());

После этого авторизация будет проходить успешно в случае присутствия пользователя в ActiveDirectory.

Подобным образом можно добавить авторизацию в других сервисах.

**Часть 9. Регистрация пользователя**

В этом разделе мы создадим собственную форму регистрации пользователя с обратной связью.

1. В клиентском приложении создайте в папке Pages новый файл «Register.razor»:

@page "/register"

@inject HttpClient Http

@inject AuthenticationStateProvider AuthStateProvider

@inject ILocalStorageService LocalStorage

@inject NavigationManager NavigationManager

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

<div>

<**AuthorizeView**>

Если пользователь не авторизован, то отобразим форму с полем ввода имени и двумя полями для ввода пароля. Для организации обратной связи сделаем элемент с сообщением об ошибке:

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

<h3>Регистрация</h3>

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

<p/>Имя пользователя:

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

<p/>Пароль:

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

<p/>Пароль (повторите ввод):

<br/><**InputText** @bind-Value="passwordRepeat" type="password" />

<p/><span class="text-danger">@errorMessage</span>

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

</**EditForm**>

</**NotAuthorized**>

Если пользователь уже авторизован, то значит, он перешел вручную по ссылке. Отобразим сообщение об ошибке:

<**Authorized**>

<div>

<p/>Вы уже авторизованы!

</div>

</**Authorized**>

</**AuthorizeView**>

Далее предложим вернуться на главную страницу:

<p/><a href="/">Перейти на главную страницу</a>

</div>

</div>

1. Добавим поля для хранения данных формы:

@code {

UserLogin user = new UserLogin();

string passwordRepeat = string.Empty;

string errorMessage = string.Empty;

Напишем обработчик, срабатывающий при нажатии кнопки «Зарегистрироваться». Сначала нам нужно проверить, совпадает ли введенный пользователем пароль в основном и проверочном полях ввода:

async Task HandleRegister()

{

if (user.Password != passwordRepeat)

{

errorMessage = "Пароль не совпадает!";

return;

}

Теперь обратимся к контроллеру авторизации (мы его сделаем позже), и отобразим пользователю результат его работы:

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

errorMessage = await result.Content.ReadAsStringAsync();

}

}

1. В файле «Login.razor» добавим ссылку на регистрацию:

<p/>Нет учетной записи? <a href="/register">Зарегистрируйтесь</a>

</**EditForm**>

1. В серверном приложении в папке «Controllers» создайте новый файл «RegisterController.cs» с классом контроллера RegisterController:

using Microsoft.AspNetCore.Mvc;

using BlazorWasmChat.Shared;

namespace BlazorWasmChat.Server.Controllers

{

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

[ApiController]

public class RegisterController : ControllerBase

{

[HttpPost]

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

{

Метод «AddUser» будет принимать имя пользователя и пароль. Сделаем проверку полученных данных:

if (string.IsNullOrEmpty(request.UserName))

{

return "Требуется имя пользователя";

}

if (string.IsNullOrEmpty(request.Password))

{

return "Требуется пароль";

}

if (Chat.Users.Exists(x => x.UserName == request.UserName))

{

return "Пользователь с таким именем уже зарегистрирован";

}

1. Если все данные проходят проверку, добавим пользователя в базу данных:

string res;

try

{

if (Chat.AddUser(request))

{

res = "OK";

}

else

{

res = "Не удалось добавить пользователя";

}

}

catch (Exception ex)

{

res = "Ошибка регистрации: " + ex.Message;

}

return res;

}

}

Соберите и запустите приложение. Теперь будет доступна регистрация нового пользователя.