Dominik Słapa, Wojciech Kopacki

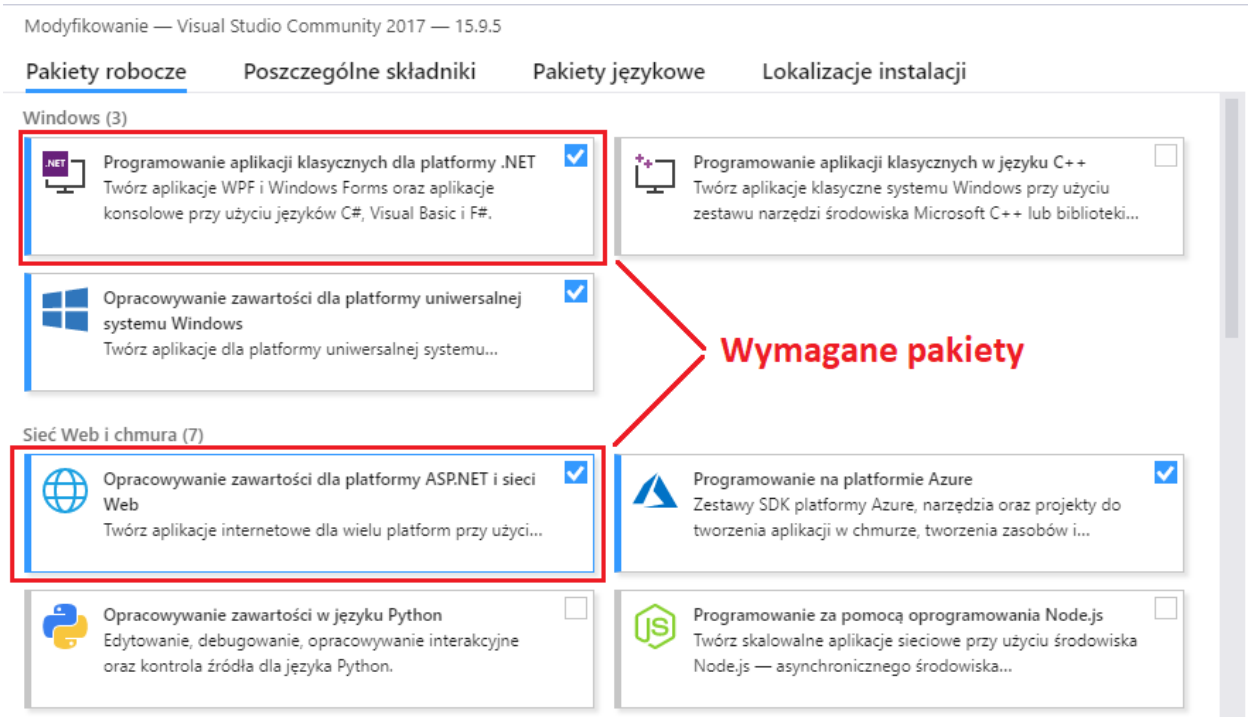
Wprowadzenie do .NET

Instrukcja ćwiczenia labolatoryjnego

Temat: Wstęp do technologii ASP.NET Core oraz SignalR

1. Opis ćwiczenia
   Celem ćwiczenia jest dokończenie implementacji części serwerowej aplikacji – czatu, pozwalającego na tekstową komunikację w czasie rzeczywistym z innymi użytkownikami. Komunikator ma działać w architekturze klient-serwer. Klient czatu został już stworzony za pomocą biblioteki ReactJS i nie jest przedmiotem ćwiczenia. Do stworzenia części serwerowej należy wykorzystać: język C#, platformę ASP.NET Core oraz bibliotekę SignalR.

2. Wymagania do zrealizowania ćwiczenia
   - Zainstalowane .NET Core SDK wersja 2.2 (https://dotnet.microsoft.com/download)
   - Zalecane IDE - Visual Studio lub Visual Studio Code.
   - Postman (lub inna aplikacja do przetestowania API)
   - Zainstalowane środowisko Node.JS (https://nodejs.org/en/)
   - Zainstalowane pakiety ze zdjęcia:



3. Przygotowanie do realizacji ćwiczenia

   Należy pobrać paczkę znajdującą się pod adresem https://github.com/slapadominik/ChatNET. Zawiera ona katalogi:

   Chat.client - kod źródłowy klienta czatu

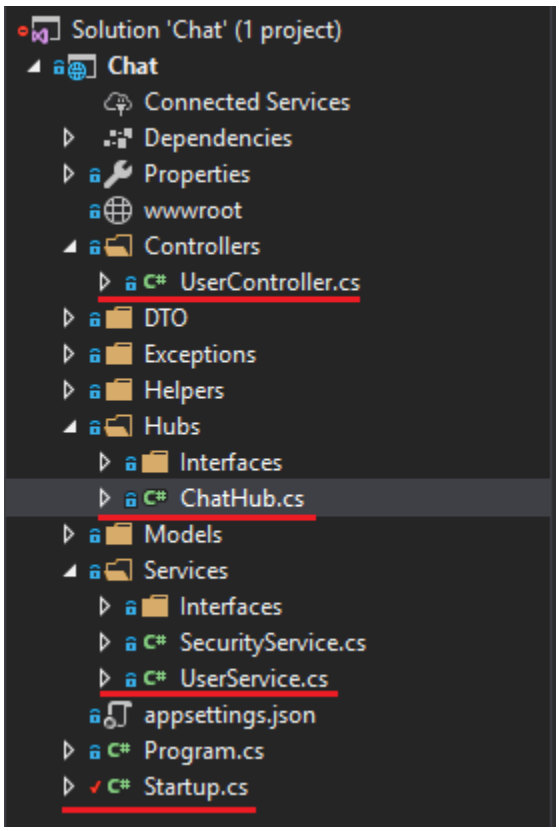
   Chat - przygotowany projekt z częścią kodu serwera, który trzeba będzie uzupełnić.

   Aby rozpocząć zadania należy przejść do folderu *Chat/Chat* i uruchomić plik projektu (*Chat.csproj*) za pomocą programu Visual Studio.

4. Zadania

Zad. 0

Zapoznać się z istniejącą strukturą kodu. Klasy, które będą wymagały dodatkowej implementacji, są podkreślone czerwoną linią na poniższym zrzucie ekranu:



Opis katalogów ze względu na odpowiedzialności:

Controllers – zawiera klasy, których zadaniem jest obsługa żądań HTTP. Metody tych klas są konkretnymi akcjami HTTP, które użytkownik może wywołać. Metody są oznaczane atrybutami `[HttpPost]`, `[HttpGet(„{id}”)]` w celu określenia typu metody. Kontrolery posiadają zależności do klas z logiką biznesową (*Services*) oraz do klas *DTO*.

DTO (ang. Data Transfer Object) – są to anemiczne klasy (zawierają same property, bez metod) służące tylko jako środek komunikacyjny ze światem zewnętrznym. Służą jako struktura danych wejściowych/wyjściowych w metodach klas *Controllers* oraz *Hubs*.

Hubs – klasy z tego katalogu służą do obsługi funkcjonalności w czasie rzeczywistym. Dziedziczą po klasie `Hub` pochodzącej z biblioteki `Microsoft.AspNetCore.SignalR`.

Services – klasy zawierające logikę biznesową aplikacji (np. tworzenie użytkowników, usuwanie użytkowników, reguły uwierzytelniania/autoryzacji).

Klasa *Startup* – klasę tę dostajemy od razu podczas tworzenia projektu w ASP.NET Core (info - https://docs.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-2.2). Zawiera ona dwie, główne metody publiczne:

```
public void ConfigureServices(IServiceCollection services)
```

metoda ta służy do ustalenia konfiguracji w naszej aplikacji (np. konfiguracja sposobu uwierzytelniania, konfiguracja połączenia z bazą danych) oraz do dodawania klas do wbudowanego w ASP.NET Core tzw. **kontenera zależności** (realizującego wzorzec **Dependency Injection** – więcej na ten temat https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2) Tutaj kolejność dodawania klas konfiguracyjnych nie ma znaczenia.

oraz

```
public void Configure(IApplicationBuilder app,
IHostingEnvironment env)
```

metoda ta służy do określenia jak aplikacja powinna reagować na przychodzące żądania (np. żądania HTTP lub WebSocket) tworząc tzw. **middleware. Tutaj kolejność wywoływania metod ma znaczenie!**

Zad. 1

Obsługa tworzenia użytkownika w aplikacji. Tworzenie użytkownika będzie realizowane za pomocą zapytania HTTP typu POST. Zostaną wykorzystane dwie klasy: *UserController* oraz *UserService.*

```
public class UserService : IUserService
{
    private static List<User> _users = new List<User>();
    private readonly IDictionary<string, string> _connectedChatUsers;
    private readonly ISecurityService _securityService;

    public UserService(ISecurityService securityService)
    {
        _connectedChatUsers = new Dictionary<string, string>();
        _securityService = securityService;
    }
```

Klasa *UserService* posiada pola:

_users – statyczna lista, w której będą przetrzymywani wszyscy użytkownicy aplikacji.

_connectedChatUsers – słownik, gdzie kluczem jest identyfkator połączenia WebSocket, a wartością nazwa użytkownika. Reprezentuje on użytkowników aktualnie połączych z czatem.

_secuirtyService – klasa zawierająca obsługę uwierzytelnienia, a konkretnie tworzenie JWT Tokenu oraz serializacja Tokenu do łańcucha znakowego (string).

Należy zaimplementować metodę *CreateUser* w następujący sposób:

```
public User CreateUser(string username)
{
    if (_users.SingleOrDefault(x => x.Username == username) != null)
    {
        throw new UsernameIsTakenException($"User {username} already exists.");
    }

    var user = new User { Id = Guid.NewGuid(), Username = username };
    _users.Add(user);

    var token = _securityService.CreateToken(username);
    user.Token = _securityService.WriteToken(token);

    return user;
}
```

Po zaimplementowaniu dodawania użytkownika w *UserService* należy przejść do klasy *UserController*, w której będziemy obsługiwać żądnie HTTP typu POST. Przyjmujemy, że klient chcąc zalogować się do chatu, musi utworzyć użytkownika z unikalną nazwą użytkownika w systemie. Tworząc takiego użytkownika przesyła jedynie nazwę, którą będzie identyfikował się podczas rozmowy przez komunikator. Przyjmujemy więc z ciała zapytania HTTP dopasowany, zserializowany obiekt typu `UserCredentials`, zawierający nazwę użytkownika.

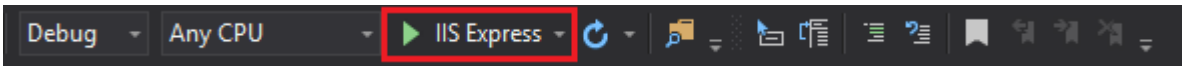
Należy dodać następujący kod do klasy *UserController*:

```
[Route("api/[controller]")]
[ApiController]
public class UserController : ControllerBase
{
    private readonly IUserService _userService;

    public UserController(IUserService userService)
    {
        _userService = userService;
    }

    [HttpPost]
    public ActionResult<User> CreateUser([FromBody] UserCredentials userCredentials)
    {
        try
        {
            var user = _userService.CreateUser(userCredentials.Username);
            return StatusCode(201, user);
        }
        catch (UsernameIsTakenException ex)
        {
            return StatusCode(409, ex.Message);
        }
    }

}
```

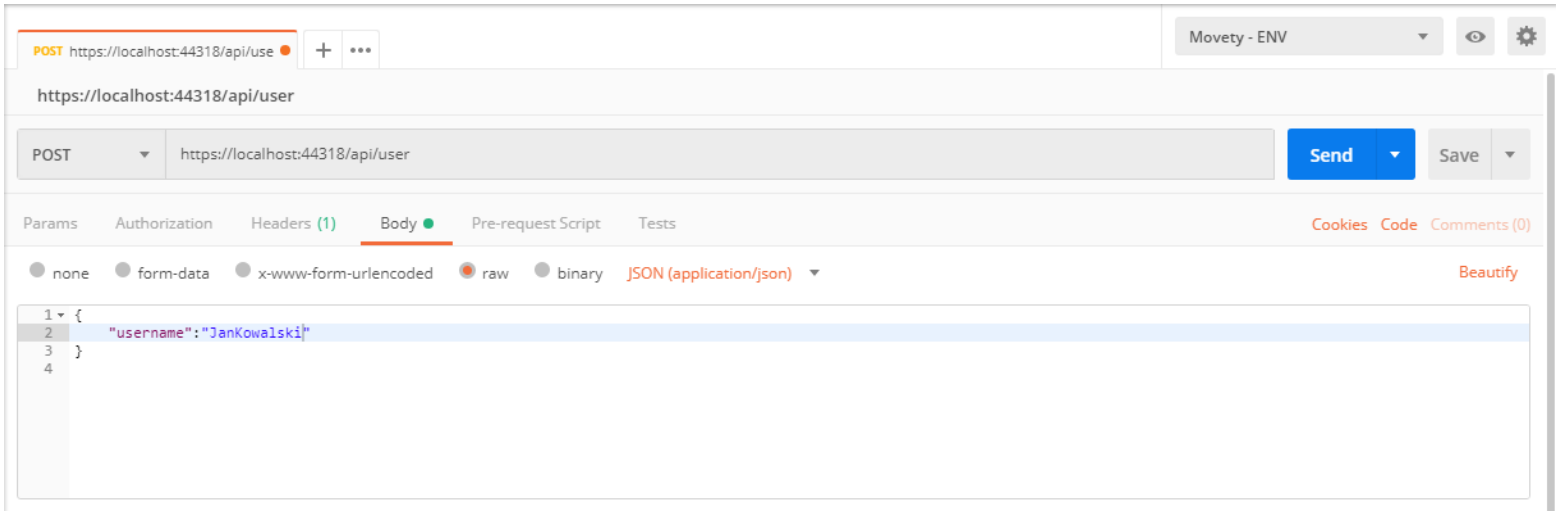
Aby funkcjonalność tworzenia użytkownika działała, w klasie Startup należy skonfigurować routing akcji kontrolera (odwzorowanie adresu URL na konkretną metodę kontrolera) oraz dodać klasę UserService do kontenera zależności:

```
public void ConfigureServices(IServiceCollection services)
        {
            var appSettingsSection =
Configuration.GetSection("AppSettings");
            services.Configure<AppSettings>(appSettingsSection);
            ConfigureAuthentication(services, appSettingsSection);
            services.AddTransient<ISecurityService,
SecurityService>();
            services.AddCors();


services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Versi
on_2_1);
            services.AddSingleton<IUserService, UserService>();
        }


public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseHsts();
            }

            app.UseCors(x => x
                .AllowCredentials()
                .AllowAnyOrigin()
                .AllowAnyMethod()
                .AllowAnyHeader());
            app.UseAuthentication();

            app.UseMvc();
        }
```

Teraz możemy uruchomić aplikację na lokalnym serwerze webowym IIS Express:



I przetestować funkcjonalność tworzenia użytkownika za pomocą Postmana:



Należy wybrać typ zapytania POST oraz wpisać adres URL: https://localhost:**PORT**/api/user

Zmienić PORT na taki, na jakim została uruchomiana aplikacja.

W sekcji body należy zaznaczyć opcję raw JSON i wpisać strukturę:

```
{
    "username":"JanKowalski"
}
```

Jest to zserializowany w postaci JSON odpowiednik klasy `UserCredentials`.

Przykładowa odpowiedź na zrzucie ekranu.

Zad. 2

Kolejna część to dodanie funkcjnonalności czatu, działających w czasie rzeczywistym. Zajmiemy się implementacją klasy *ChatHub* oraz metod z klasy *UserService*.

Zacznijmy od klasy UserService. Mamy do zaimplementowania 4 metody:

`public void DeleteUser(string username) – usuwa użytkownika z użytkowników aplikacji`

`public void JoinChat(string connectionId, string username) – dodaje użytkownika do aktualnie połączonych z czatem użytkowników`

`public void LeaveChat(string connectionId) – usuwa użytkownika z aktualnie połączonych z czatem użytkowników`

`public IEnumerable<string> GetConnectedUsers() – pobiera aktualnie połączonych użytkowników`

Przykładowa implementacja:

```
public void DeleteUser(string username)
   {
      var user = Users.SingleOrDefault(x => x.Username ==
      username);
       if ( user == null)
       {
           throw new UserNotFoundException($"User {username}
      not    found.");
       }
       Users.Remove(user);
   }

   public void JoinChat(string connectionId, string username)
   {
       if (_connectedChatUsers.ContainsKey(connectionId))
       {
      throw new InvalidOperationException($"User {username}
      connected with id {connectionId} is connected to chat.");
       }

       _connectedChatUsers.Add(connectionId, username);
   }
```

```
public void LeaveChat(string connectionId)
{
    if (!_connectedChatUsers.ContainsKey(connectionId))
    {
   throw new InvalidOperationException($"Connection with id
   {connectionId} isn't connected to chat.");
    }

    _connectedChatUsers.Remove(connectionId);
}

public IEnumerable<string> GetConnectedUsers()
{
    return _connectedChatUsers.Values;
}
```

Logika zawarta w tym kodzie będzie potrzebna do powiadamiania użytkowników, kto dołączył do czatu, kto z niego wyszedł oraz ilu aktualnie użytkowników online na czacie.

Należy wykorzystać metody z UserService w klasie ChatHub. W tej klasie są do zaimplementowania 3 metody:

`public Task SendMessage(GeneralMessage msg) – rozsyłanie wiadomości, która przyszła od jednego użytkownika, do reszty użytkowników`

`public override async Task OnConnectedAsync() – metoda uruchamiana raz dla danego klienta, podczas nawiązania połączenia z nowym klientem`

`public override async Task OnDisconnectedAsync(Exception exception) - metoda uruchamiana raz dla danego klienta, podczas zakończenia połączenia z klientem`

Przykładowa implementacja:

```
public override async Task OnConnectedAsync()
 {
    var username = Context.User?.Identity?.Name;

    try
    {
        var connectedUsers = _userService.GetConnectedUsers();
        await Clients.Caller.SetConnectedUsers(connectedUsers);

        _userService.JoinChat(Context.ConnectionId, username);
```

```
            await Clients.All.UserJoined(username);
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }


public override async Task OnDisconnectedAsync(Exception exception)
{
      var username = Context.User?.Identity?.Name;

      try
      {
          _userService.LeaveChat(Context.ConnectionId);
          _userService.DeleteUser(username);
          await Clients.All.UserLeft(username);
      }
      catch (UserNotFoundException ex)
      {
          Console.WriteLine($"Error: {ex.Message}");
      }
  }

  public async Task SendMessage(GeneralMessage msg)
  {
      await Clients.All.MessageAdded(msg);
  }
```

Aby funkcjonalności czatu działały, w klasie Startup należy skonfigurować routing akcji Hub'u (odwzorowanie adresu URL na konkretną metodę Hub'u).

W klasie *Startup* dodać:

```
public void ConfigureServices(IServiceCollection services)
{
    var appSettingsSection =
    Configuration.GetSection("AppSettings");
    services.Configure<AppSettings>(appSettingsSection);
    ConfigureAuthentication(services, appSettingsSection);
    services.AddTransient<ISecurityService, SecurityService>();
    services.AddCors();


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.
    Version_2_1);
    services.AddSingleton<IUserService, UserService>();
    services.AddSignalR();
```

```
        }



public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseHsts();
            }

            app.UseCors(x => x
                .AllowCredentials()
                .AllowAnyOrigin()
                .AllowAnyMethod()
                .AllowAnyHeader());

            app.UseAuthentication();

            app.UseSignalR(route =>
            {
                route.MapHub<ChatHub>("/hubs/chat");
            });

            app.UseMvc();
        }
```

Serwer jest gotowy do działania!

Zad. 3

Mamy gotowy kod serwera, należy go uruchomić. Przetestujemy serwer czatu na kliencie z materiałów. Do odpowiedniego działania klienta musimy podmienić adres serwera, który przed chwilą uruchomiliśmy. Należy skopiować adres serwera w postaci:

http://localhost:PORT

i wkleić pomiędzy apostrofy do pliku o ścieżce Chat.client/src/constants.js.

Aby uruchomić klienta należy przejść do folderu *Chat.client*, otworzyć konsolę w tym katalogu i wpisać polecenie do konsoli:

```
npm install
```

Polecenie to instaluje niezbędne paczki do uruchomienia klienta. Po instalacji należy wpisać do konsoli polecenie:

```
npm start
```

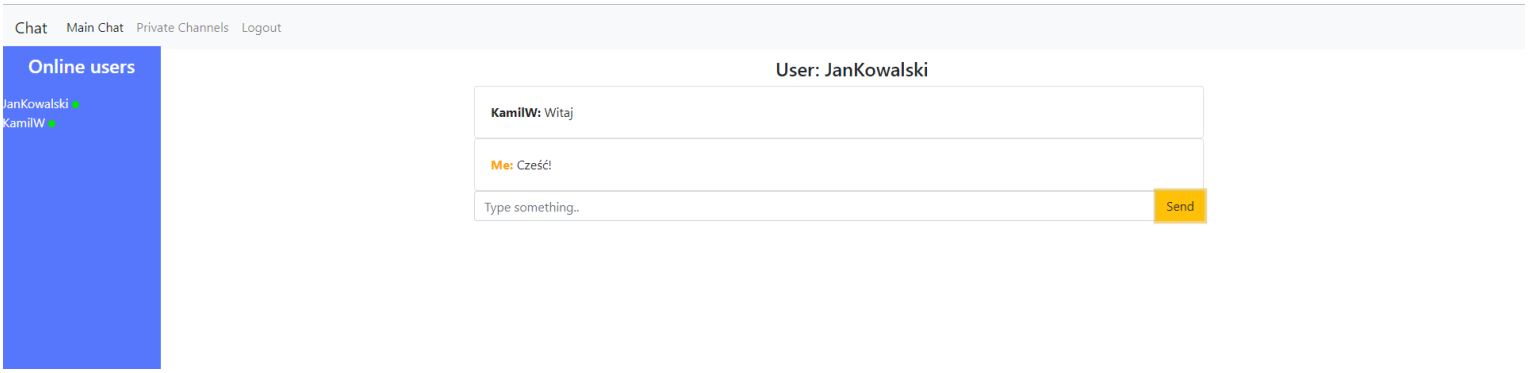
Po uruchomieniu środowiska powinna automatycznie otworzyć się przeglądarka pod adresem:

http://localhost:3000/login

Należy wpisać login użytkownika i kliknąć przycisk „Send”, jeśli wykonaliśmy wszystkie kroki poprawnie to użytkownik powinen zostać stworzony, a w oknie przeglądarki powinen pojawić się interfejs czatu z możliwością pisania na nim. Możemy otworzyć nową kartę w przeglądarce, wejść pod adres

http://localhost:3000/login

utworzyć użytkownika z inną nazwą i ze sobą pisać.



Prawidłowy sposób funkcjonowania aplikacji można zobaczyć pod adresem: https://chatnet.azurewebsites.net/