Projektowanie aplikacji ASP.NET

Wykład 11/15
Architektura/Testowanie/SignalR

Wiktor Zychla 2024/2025

Spis treści

2 SignalR

SignalR jest biblioteką, uławiającą na platformie ASP.NET programowanie aplikacji wykorzystujących protokół WebSocket. Protokołu tego używa się do uzyskania efektu wysyłania powiadomień z serwera do klienta, co w normalnym trybie pracy protokołu HTTP jest oczywiście niemożliwe (to klient nawiązuje połączenie). WebSocket realizuje to przez otwarcie długotrwałego połączenia z klienta (czyli to nadal klient nawiązuje połączenie), ale po otwarciu połączenia klient przechodzi w tryb nasłuchu.

O ile programowanie WebSockets na poziomie samego protokołu jest technicznie możliwe, biblioteki takie jak SignalR rozwiązują dodatkowo następujące kwestie:

- Obsługa starszych przeglądarek, które nie wspierają protokołu WebSockets (obsługiwane jest to przez zarządzanie długotrwałymi połączeniami HTTP)
- Organizacja API po stronie serwera – np. możliwość przypisywania użytkowników do grup i komunikacja tylko w obrębie grupy

SignalR ma dwie wersje, dla ASP.NET Core i dla ASP.NET Framework. Obie mają zbliżony, choć nie identyczny interfejs programistyczny.

Interfejs ten obejmuje:

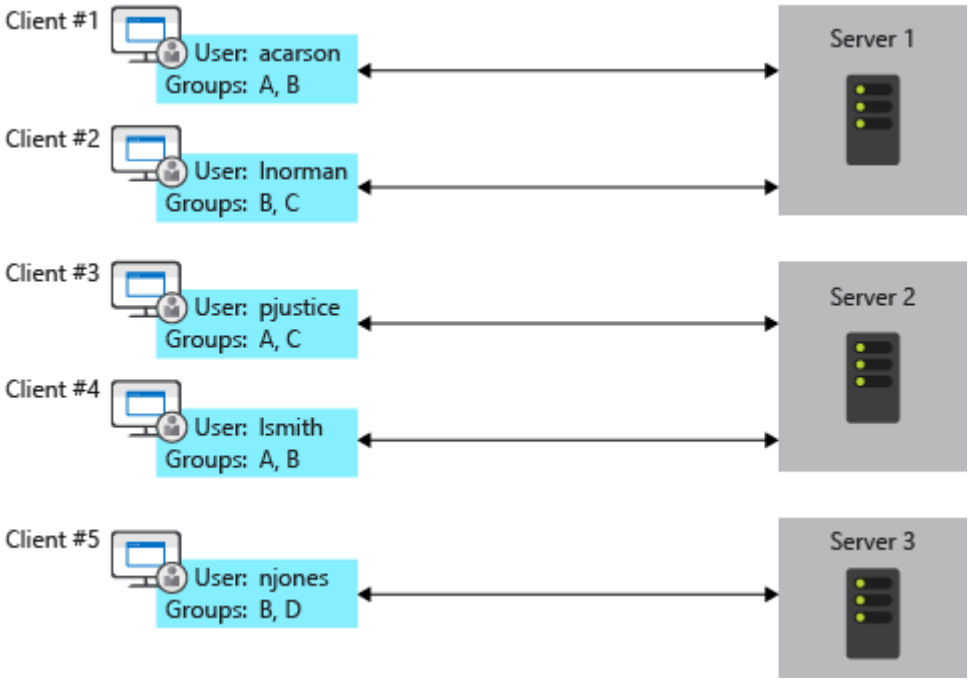
- Obiekt serwerowy typu **Hub** – jest to klasa, która z jednej strony implementuje zdarzenia dołączania i odłączania klientów, z drugiej strony – tworzy wszystkie dodatkowe opcjonalne metody, te które będą wywoływane przez klientów (jeśli w Hub pojawi się metoda **Foo**, to klient będzie mógł ją wywołać z określonymi argumentami)
- Część kliencką interfejsu, wykonującą się w przeglądarce. W części klienckiej trzeba umieć nawiązać połączenie do serwera, umieć wywołać metody serwera (na przykład jeśli w **Hub** jest metoda **Foo** …) ale w części klienckiej definiuje się w Javascript metody, które może wywołać serwer – czyli jeśli na kliencie jest metoda **Bar** to serwer może wywołać metodę **Bar** – u wszystkich połączonych klientów, u klientów tylko z określonej grupy lub u tylko jednego klienta

Serwerowa część SignalR jest częścią biblioteki standardowej ASP.NET Core, co znaczy że nie trzeba instalować żadnych dodatkowych pakietów. Część kliencka (biblioteki Javascript) musi być natomiast zainstalowana, jest to opisane w dokumentacji. W .NET.Core middleware SignalR integruje się z potokiem uwierzytelnienia

Praca z grupami jest opisana w dokumentacji.

Skalowanie wymaga dodatkowej konfiguracji.

Jest to związane z tym że jeśli aplikacja działa na wielu serwerach, każdy serwer obsługuje pewną część użytkowników, to użytkownicy z grup w ramach konkretnego serwera otrzymywaliby komunikaty, ale użytkownicy z tej samej grupy połączeni do innego serwera – nie.



Rysunek 1 https://docs.microsoft.com/en-us/aspnet/core/signalr/scale?view=aspnetcore-6.0 Ilustracja problemu skalowania

Przykładowy Hub:

```
    public class ChatHub : Hub
    {
        public override Task OnConnectedAsync()
        {
            // identyfikator łączącego się klienta
            var clientId = this.Context.ConnectionId;

            // dodanie klienta do grupy i wysłanie komunikatu tylko do grupy
            //await this.Groups.AddToGroupAsync(clientId, "nazwa grupy");
            //await this.Clients.Group("nazwa grupy").SendAsync(...);

            return base.OnConnectedAsync();
        }

        public async Task SendMessage(string message)
        {
            // metoda wywołana po stronie klienta
            await Clients.All.SendAsync( "DisplayMessage",
                this.Context.User.Identity.Name,
                message);
        }
    }
```

Przykładowa część kliencka:

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

connection.on("DisplayMessage", function (name, message) {
    // Html encode display name and message.
    var encodedName = $('<div />').text(name).html();
    var encodedMsg = $('<div />').text(message).html();
```

```
    // Add the message to the page.
    $('#discussion').append('<li><strong>' + encodedName
        + '</strong>:  ' + encodedMsg + '</li>');
});

connection.start().then(function () {

    $('#sendmessage').click(function () {

        var message = $('#message').val();

        // Call the Send method on the hub.
        connection.invoke("SendMessage", message);

        // Clear text box and reset focus for next comment.
        $('#message').val('').focus();
    });

}).catch(function (err) {
    return console.error(err.toString());
});
```

3 Archiektura heksagonalna / MediatR

3.1 Architektura heksagonalna

W trakcie rozwijania aplikacji w technologiach webowych (nie tylko ASP.NET), zawsze pojawiają się pytania o właściwą architekturę.

Na przykład:

- Czy logika dostępu do danych powinna znajdować się bezpośrednio w kontrolerach?

Odpowiedź jest:

- Nie powinna, ponieważ trudno jest w takim podejściu pisać testy jednostkowe.

Zacznijmy więc od pokazania jak pisać testy jednostkowe:

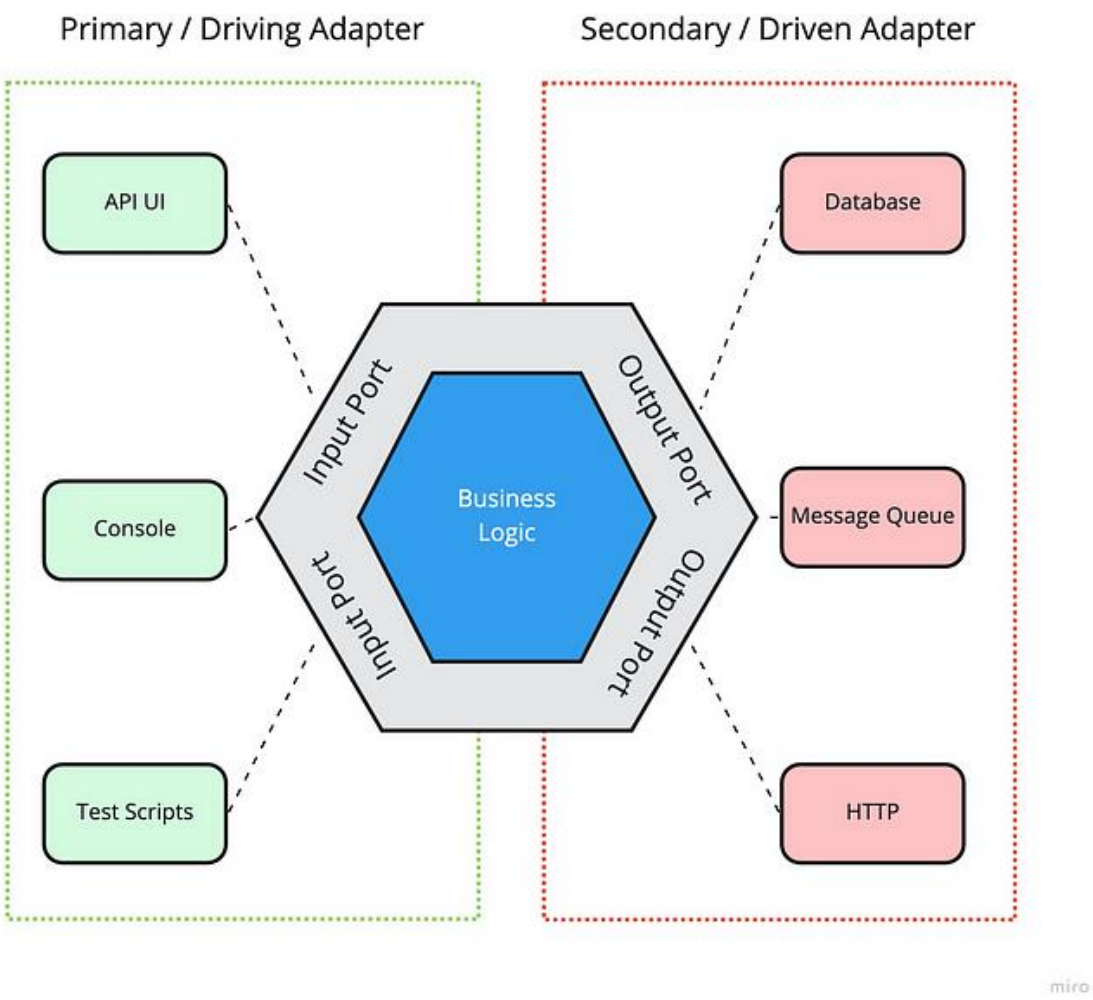
- W dodatkowym projekcie należy zainstalować **MSTest.TestFramework** i **MSTestTestAdapter**. Na .NET.Core również **Microsoft.NET.Test.SDK**
- Pisać testy używając atrybutów m.in. [TestClass] i [TestMethod]
- Uruchamiać testy z widoku Test/Test Explorer

Próba napisania testu jednostkowego napotkamy problem – metoda w kontrolerze odwołuje się do kontekstu http (Request, Response itd.)

Architektura heksagonalna (aka **architektura portów i adapterów** aka **architektura cebulowa**) (*hexagonal architecture*, *ports and adapters*, *onion architecture*) to ogólniejsze podejście do projektowania aplikacji, w którym jednym z celów jest ułatwienie testowania.

Ten rodzaj architektury nadaje się zarówno do aplikacji typu rich-client jak i do aplikacji webowych, stąd duża popularność tego podejścia w ostatnich latach.

Zasadniczy pomysł polega na odizolowaniu wejścia i wyjścia od rdzenia aplikacji za pomocą tzw. **portów**.



Rysunek 2 https://medium.com/idealo-tech-blog/hexagonal-ports-adapters-architecture-e3617bcf00a0

Wyróżnia się tu dwa rodzaje portów:

- **Porty pierwotne** (wejściowe) (primary/driving) – służą do sterowania wejściem do logiki biznesowej aplikacji i odizolowania warstwy aplikacyjnej od logiki biznesowej
- **Porty wtórne** (wyjściowe) (secondary/driven) – służą do realizacji komunikacji z zapleczem (backend), na przykład bazą danych, usługami poczty itd. Chętnie sięga się tu po wzorzec **Repository**

Jako że oba rodzaje portów pełnią inną funkcję – są też technicznie inaczej implementowane:

- **Porty pierwotne** są konkretnymi klasami, zwykle odwzorowującymi przypadki użycia poszczególnych rodzajów wejścia. W warstwie aplikacyjnej zachodzi mapowanie informacji z infrastruktury (komponentów na formularzach, parametrów żądań HTTP itp.) na argumenty wywołania metod odpowiedzialnych za wykonanie przypadku użycia. Dobrze sprawdza się tu wzorzec **Command**, bo z punktu widzenia architektonicznego – każdy przypadek użycia trzeba „wykonać”, co najwyżej przekazując mu jakieś parametry

- **Porty wtórne** są opisane interfejsami i logika biznesowa aplikacji nie używa konkretnych implementacji. Zamiast tego, za pomocą wstrzykiwania zależności (na przykład wzorca **Local Factory** który już poznaliśmy, opcjalnie wspartego kontenerem **Inversion of Control**), na etapie uruchomienia, dostarczane są konkretne implementacje usług, z punktu widzenia architektury jest to nieistotne jakie to będą konkretne implementacje.

Ta różnica w podejściu do portów – konkretne klasy na portach pierwotnych a interfejsy na portach wtórnych – jest w terminologii związanej z architekturą heksagonalną nazwana **left-right assymetry** (por. oryginalny artykuł Alistaira Cockburna)

Uwaga! Ten rodzaj architektury skupia się na otoczeniu (odizolowaniu) warstwy nazwanej tu szeroko Logiką Biznesową ale nie wnika w to jak ta warstwa jest zorganizowana.

Testowalność kodu jest tu zapewniona ponieważ cała logika biznesowa aplikacji jest sterowana przez porty pierwotne, a te są oddzielone od warstwy aplikacyjnej. W testach jednostkowych można więc łatwo powoływać do życia obiekty reprezentujące porty pierwotne i wywoływać ich metody. Z kolei porty wtórne mogą być w testach jednostkowych zastąpione takimi implementacjami, które nie mają skutków ubocznych albo wręcz – całowicie zastępują zaplecze.

3.2 MediatR

Biblioteka MediatR to przykład narzędzia wspierającego architekturę heksagonalną.

W praktyce zaczyna się od modelowania wejścia/wyjścia portu pierwotnego:

```
public class LogonUseCaseParameters : IRequest<LogonUseCaseResults>
{
    public string Username { get; set; }
    public string Password { get; set; }
}

public class LogonUseCaseResults
{
    public bool LogonStatus { get; set; }
}
```

I implementuje się port:

```
public class LogonUseCaseHandler : IRequestHandler<LogonUseCaseParameters,
LogonUseCaseResults>
{
    public async Task<LogonUseCaseResults> Handle(
        LogonUseCaseParameters request,
        CancellationToken cancellationToken )
```

```
    {
        return new LogonUseCaseResults()
        {
            LogonStatus = request.Username == request.Password
        };
    }
}
```

Całość trzeba jeszcze zarejestrować w kontenerze usług:

```
builder.Services.AddMediatR( cfg =>
{
    cfg.RegisterServicesFromAssembly( typeof( Program ).Assembly );
} );
```

I już można używać portu wewnątrz kontrolera:

```
public class AccountController : Controller
 {
     private IMediator _mediator;

     public AccountController( IMediator mediator )
     {
         this._mediator = mediator;
     }

     . . .

     [HttpPost]
     public async Task<IActionResult> Logon(AccountLogonModel model)
     {
         var logonRequest = new LogonUseCaseParameters()
         {
             Username = "foo",
             Password = "foo"
         };

         var logonResult = this._mediator.Send( logonRequest );
```

Jeżeli port pierwotny wymaga użycia portu wtórnego, kontener załatwia sprawę. Przykładowy kontrakt usługi portu wtórnego:

```
public interface IEmailSender
{
    bool Send( string to );
}

public class EmailSender : IEmailSender
{
    public bool Send( string to )
    {
        return true;
```

```
    }
}
```

I zwyczajowe zbudowanie zależności wstrzykiwanej przez konstruktor:

```
public class LogonUseCaseHandler : IRequestHandler<LogonUseCaseParameters,
LogonUseCaseResults>
{
    private IEmailSender _emailSender;
    public LogonUseCaseHandler( IEmailSender emailSender )
    {
        this._emailSender = emailSender;
    }
```