# Porozmawiajmy o Blazorze

## Wstęp

Cześć! Nazywam się Łukasz Ziemblicki i w firmie POSSIBE pracuję na stanowisku Technical Architect. Moim głównym zadaniem jest merytoryczne wsparcie klienta, przygotowanie wizji zadań oraz pilnowanie, aby dostarczany przez zespół produkt spełniał wszystkie oczekiwania.

Jaki jest główny cel prezentacji oraz tego dokumentu? Przede wszystkim przedstawienie czegoś niecodziennego, co cieszy się z dnia na dzień coraz większą popularnością. Wszyscy pracujemy w branży, w której pomimo swoich specjalizacji musimy posiadać podstawową wiedzę o trendach dookoła nas. Starałem się utrzymać dokument w jak najlżejszej formie, opierając się na wiedzy czerpanej z artykułów (linki poniżej) i kilku konferencji w których miałem okazję uczestniczyć. Jeżeli znajdziesz jakiś błąd lub masz ochotę dowiedzieć się czegoś więcej to skontaktuj się ze mną drogą mailową (lukasz.ziemblicki@possible.com).

## Czym jest Blazor?

Jest to rozwinięcie istniejącego projektu „Razor” (nie mylić z „językiem tworzenia widoku” w MVC). Stał się on idealnym przykładem dla tworzenia aplikacji internetowych w oparciu o architekturę MVVM (Model-View-ViewModel). Głównym zamysłem Razora było uproszczenie architektury aplikacji internetowej, pozbycie się niepotrzebnych plików i usprawnienie uruchamiania strony. Microsoft pozbył się kontrolerów (znanych z MVC), a widok wykorzystuje metody znajdujące się w ViewModelu bezpośrednio podczas generowania struktury html. Ograniczono w ten sposób ilość komunikacji wymaganej do utworzenia pojedynczego elementu na stronie. Jednak cały czas pozostaje problem miejsca oraz sposobu tworzenia interfejsu użytkownika. Niestety całość opiera się na nieustannej komunikacji urządzenia z serwerem przy jednoczesnym przekazywaniu informacji „gdzie jestem, co robię i co chcę osiągnąć”.

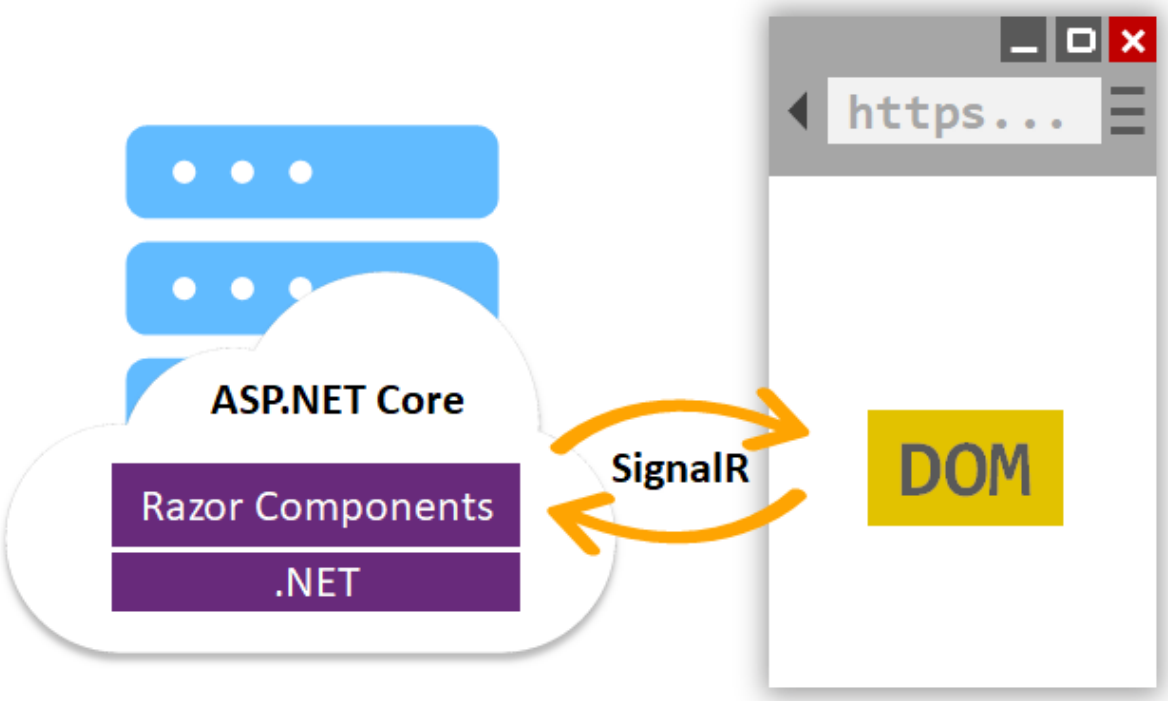
Tutaj z pomocą przybywa Blazor. Opiera się on na zupełnie innym podejściu tworzenia warstwy użytkownika niż do tej pory. Mówiąc w dużym skrócie tworzenie interfejsu nie obywa się na zasadzie pytanie-odpowiedź, tylko reaktywności. Silnik (Blazor) posiada dokładne informacje jak wygląda aktualnie strona u danego użytkownika i przekazuje przeglądarce informacje o aktualizacji wyłącznie konkretnych fragmentów (komponentów).

## Modele Hostingowe

Blazor posiada dwa modele hostingowe aplikacji – każdy z nich posiada swoje zalety i wady, jednak dają one możliwość wyboru pomiędzy tradycyjną a nowatorskim podejściem:

### Blazor Server

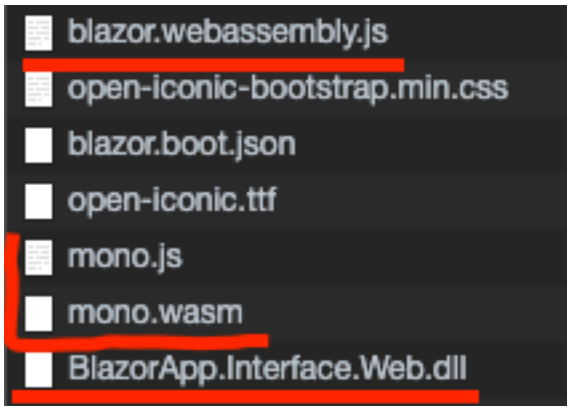
W tym modelu użytkownik po wejściu na stronę tworzy bezpośrednie połączenie SignalR z serwerem, który jest odpowiedzialny za przygotowanie oraz aktualizacje komponentów klienta. W odróżnieniu od tradycyjnego sposobu serwowania aplikacji, silnik Blazor (działający na serwerze) komunikuje się błyskawicznie z przeglądarką, a wbudowane skrypty automatycznie odświeżają wybrane fragmenty strony. Model jest dostępny w .NET Core 3.0.




*Rys 1 – Architektura Blazor Server*

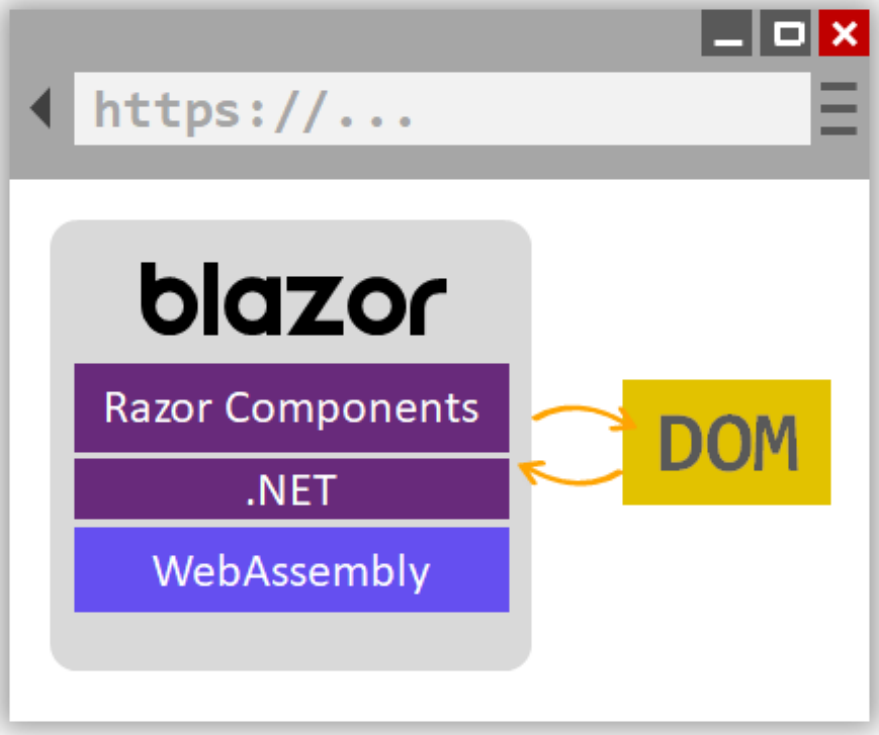
## Blazor WebAssembly (wasm)

W tym modelu silnik Blazor zostaje uruchomiony bezpośrednio w przeglądarce klienta – nie jest wymagany dodatkowy serwer wykonujący operacje oraz dodatkowe narzuty czasowe związane z połączeniem. Jest to możliwe dzięki WebAssembly pozwalającemu realizować operacje bezpośrednio z kodu wykonywalnego. W konsoli przeglądarki warto prześledzić proces uruchamiania aplikacji. W pierwszym kroku zostaje wywołane środowisko uruchomieniowe .NET Core (mono), a następnie załadowane niezbędne pliki dll




*Rys 2 – Uruchomienie Blazor WebAssembly*

Dobrą (a zarazem złą) informacją jest to, że Blazor został zaprojektowany tak, aby wspierać istniejące paczki NuGet przygotowane pod .NET Standard. Musimy pamiętać o tym, że Blazor nie powstał w celu zastąpieniu API, a wyłącznie do zmiany sposobu tworzenia interfejsu użytkownika. Tak samo jak w przypadku istniejących bibliotek opartych o JavaScript (Angular, React) zalecane jest utworzenie podstawowego API w którym będzie znajdować się logika biznesowa, hasła dostępu, połączenie z bazą danych itp.




*Rys 3 – Architektura Blazor WebAssembly*

Niewątpliwą zaletą Blazora w modelu WebAssembly jest ograniczenie narzutów związanych z tworzeniem i dostarczaniem warstwy prezentacji – wystarczy jedynie umieścić skompilowane wcześniej pliki na serwerze statycznej strony www którego jedynym zadaniem będzie dostarczenie ich użytkownikowi. Model jest dostępny w .NET Core 3.1 (oficjalna wersja została opublikowana 03.12.2019r).

## Zalety i wady poszczególnych modeli hostingowych

| Blazor Server | Blazor WebAssembly |
|---|---|
| **Zalety** | |
| • Wsparcie dla większości przeglądarek.<br>• Aplikacja może być napisana w „tradycyjnym" stylu, w którym jasno można określić granicę pomiędzy warstwą prezentacji, a logiką biznesową.<br>• Mniejsza ilość/wielkość paczek do pobrania przez użytkownika.<br>• Szybsze uruchomienie aplikacji.<br>• Tradycyjny sposób debuggowania.<br>• Gwarancja uruchomienia wszystkich istniejących paczek NuGet oraz projektów (o ile zostały przygotowane pod .NET Standard).<br>• Klient nie otrzymuje kodu .NET w którym może potencjalnie znajdować się logika/wrażliwe informacje. | • Serwer nie musi wspierać .NET Core – wystarczy prosty hosting plików (Apache, CDN, Azure Storage).<br>• Operacje tworzenia interfejsu obciążają wyłącznie urządzenie klienta.<br>• Jest możliwe w pełni efektywne wykorzystanie zasobów klienta (rozwiązanie wydajniejsze niż JavaScript). |
| **Wady** | |
| • Dodatkowe narzuty sieciowe związane ze stałym utrzymywaniem połączenia serwer-klient.<br>• Aplikacja nie działa bez połączenia z serwerem.<br>• Utrudnione skalowanie aplikacji – użytkownik posiada otwarte połączenie do konkretnego serwera.<br>• Serwer jest wymagany i musi wspierać .NET Core. | • Brak wsparcia starszych przeglądarek (np. Internet Explorer 11).<br>• Nie ma gwarancji, że aplikacja zawsze będzie działać spójnie.<br>• Ilość danych do pobrania przez klienta.<br>• Debuggowanie aplikacji to aktualnie droga przez mękę.<br>• Sposób pisania aplikacji ma bezpośredni wpływ na urządzenie klienta (np. zbyt zaawansowana logika biznesowa uruchomiona na starszym urządzeniu). |

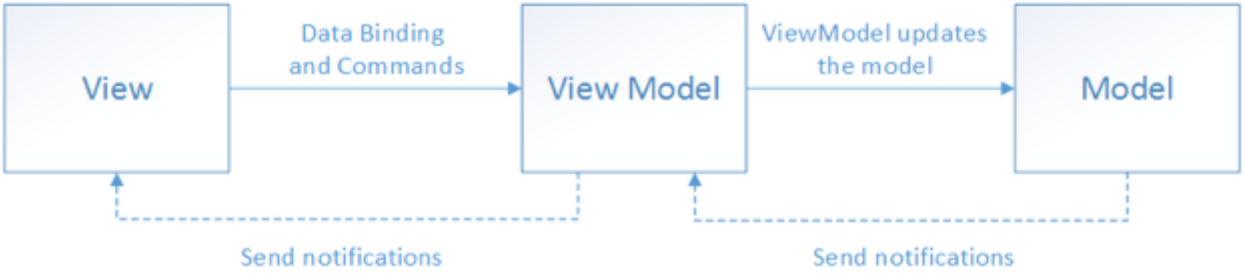
# Architektura i dobre praktyki projektowania strony Blazor

## Blazor i MVVM

Naturalną architekturą dla aplikacji Blazor jest MVVM (Model-View-ViewModel). Została ona przygotowana z myślą o aplikacjach okienkowych i dobrze zastosowana (a przede wszystkim przestrzegana) przynosi wiele benefitów. Blazor z perspektywy programisty powinien być postrzegany jak tradycyjna aplikacja (zarówno przy zastosowaniu modelu Serwer jak i WebAssembly).

Składa się on z trzech podstawowych „elementów”

1. Model – jest to podstawowy „kontener” opisujący i zawierające dane. Nie powinno się w nim stosować logiki biznesowej (wyjątkiem może być podstawowa walidacja – np. implementując interfejs IValidatableObject).
2. View – właściwy interfejs użytkownika. W Blazorze istnieje wbudowany data-binding, przez co zmiany w modelu automatycznie wpływają na to co widzi użytkownik.
3. View Model – warstwa pośrednia pomiędzy modelem a widokiem. Ma na celu dostarczenie/utworzenie/rozszerzenie modelu oraz wypełnienie go danymi (np. pobranymi z logiki biznesowej). Dodatkowo zawiera wszelki kod niezbędny do wykonania operacji. To na poziomie tej warstwy należy określić co jest np. widoczne dla użytkownika.




*Rys 4 – Architektura MVVM*

## Logika biznesowa

Projektując aplikację przy użyciu Blazor musimy pamiętać o tym, aby utrzymywać ilość logiki w projekcie „UI” na możliwie jak najniższym poziomie. Jest to niezwykle istotne przy zastosowaniu modelu WebAssembly w którym całość aplikacji uruchamiana na urządzeniu klienta. Jeżeli w projekcie został użyty Blazor Serwer nie ma to większego znaczenia (wykonaniem logiki zajmuje się zaufany serwer). Pamiętaj jednak, że należy sobie odpowiedzieć na podstawowe pytanie - „Czy aplikacja może być kiedyś przeniesiona na WebAssembly”. Jeżeli nie otrzyma się jednoznacznej odpowiedzi warto utrzymywać jasną granicę pomiędzy logiką biznesową, a interfejsem użytkownika co ułatwi przyszłą migrację.

## Dependency Injection

Podobnie jak w przypadku aplikacji MVC .NET Core posiada wbudowane Dependency Injection dla projektów Blazor. Jedyną różnicą jest to, że posiada trzy, dodatkowe serwisy które dodają się automatycznie do kolekcji:

- **HttpClient** (singleton) – serwis do komunikacji za pomocą http. Z powodów optymalizacji został on domyślnie utworzony jako singleton. Czy ma to duże znaczenie? Jest to bardzo obszerny temat, na podstawie którego można utworzyć osobną prezentację. Jeżeli interesują Cię szczegóły, dlaczego HttpClient nie powinien być umieszczany „tradycyjnie" w bloku using, polecam artykuły:
  - Simon Timms – *You're using HttpClient wrong and it is destabilizing your software* (https://aspnetmonsters.com/2016/08/2016-08-27-httpclientwrong/)
  - Steve Gordon – *HttpClientFactory in ASP.NET Core 2.1* (https://www.stevejgordon.co.uk/introduction-to-httpclientfactory-aspnetcore)
- **IJSRuntime** (singleton) – serwis do "współpracy" kodu .net z Javascriptem. Więcej informacji na ten temat znajduje się w sekcji Javascript Iterop.
- **NavigationManager** (singleton) – serwis zawierający metody pomocnicze używane przy pracy z linkami.

Drobna różnica występuje w czasie życia serwisów przy zastosowaniu poszczególnych modeli hostingowych. W .NET Core występują trzy główne typy dostarczania serwisów przez DI:

- **Transient** – instancja serwisu jest tworzona na nowo przy każdym wywołaniu.
- **Singleton** – instancja tworzona jest tylko raz i jest współdzielona przy każdym wywołaniu we wszystkich aktywnych sesjach.
- **Scoped** – podobnie jak w przypadku Singleton instancja jest tworzona tylko raz, ale jest utrzymywana wyłącznie na czas działania sesji i nie jest współdzielona pomiędzy różnymi użytkownikami.

Jeżeli aplikacja oparta jest na modelu hostingowym Blazor Server to czas życia serwisów w kolekcji Dependency Injection działa tak samo jak w MVC. W przypadku zastosowaniu Blazor WebAssembly nie występuje możliwość utrzymywania kilku różnych sesji w tym samym czasie, więc Singleton oraz Scoped zachowują się identycznie.

## Komponenty

Czym jest komponent w Blazorze? Jest to wszystko co widzi użytkownik. Zgodnie z dobrymi praktykami aplikacje powinny składać się z małych, możliwych do ponownego użycia fragmentów, które razem tworzą spójny interfejs użytkownika. Przykładowo każdy element typu „input" może być oddzielnym komponentem, który wchodzi w skład innego kompotentu (np. formularza), który wraz z menu nawigacyjnym w nagłówku, bannerem i stopką tworzą stronę.

Komponenty mogą być niezależnym bytem względem miejsca, w którym zostały umieszczone (np. reklamy firm trzecich na stronie), przyjmować wartości (parametry) początkowe (np. tytuł), przekazywać informację o interakcji użytkownika (np. wybranie wartości w elemencie typu drop-down) oraz posiadać skomplikowaną, wewnętrzną logikę biznesową.

## Javascript Iterop

Nie wszystkie operacje jesteśmy (aktualnie) wykonać za pomocą samego Blazora. Część podstawowych czynności (takich jak ustawienie „focusa” na konkretnym elemencie) cały czas są poza zasięgiem silnika renderującego. Innym przypadkiem może być chęć użycia istniejącej, sprawdzonej biblioteki javascriptowej zamiast pisać wszystko od podstaw.

Twórcy Blazora przewidzieli taką sytuację i wprowadzili funkcjonalność o nazwie „Javascrip Iterop”. Pozwala ona na wywołanie dowolnej funkcji znajdującej się w obiekcie „window” – jedynym wymogiem jest to, że wartość zwracana musi być w formacie JSon (lub możliwa do zserializowania). Kolejną, ważną dla programistów informacją jest możliwość wywoływania kodu Blazor z poziomu JavaScript. Warto dodać, że aby komponent mógł współpracować z kodem JS należy wstrzyknąć do niego opisany wcześniej serwis implementujący IJSRuntime (jest on domyślnie wbudowany w Dependency Injection). Poniżej przedstawiam proste przykłady użycia Iterop:

1. Wywołanie funkcji JS zwracającej tekst

```
var wynik = await JSRuntime.InvokeAsync<string>("nazwaFunkcji", parametrWejsciowy);
```

2. Wywołanie funkcji JS niezwracającej wyniku (void)

```
JSRuntime.InvokeVoidAsync("nazwaFunkcji", parametrWejsciowy);
```

3. Wywołanie metody komponentu Blazor z JS – tutaj chciałbym dodać, że metoda .NET musi być „opakowana” atrybutem [JSInvokable]

   - Kod .NET

```
[JSInvokable]
public static Task<int[]> ReturnArrayAsync()
{
    return Task.FromResult(new int[] { 1, 2, 3 });
}
```

   - Kod JS

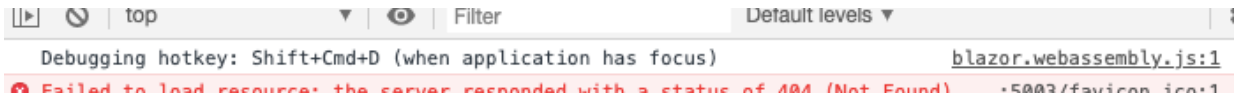
```
function Test() {
    DotNet.invokeMethodAsync('nazwaProjektu', 'ReturnArrayAsync')
      .then(data => {
        data.push(4);
          console.log(data);
    });
```

Bardzo ważna uwaga – silnik Blazor nie posiada informacji o zmianach elementów DOM które nie zostały wykonane przez niego! Jest to niezmiernie ważne, ponieważ każdorazowa zmiana strony z poziomu JavaScript może skutkować losowymi problemami. Z tego powodu należy pamiętać, żeby przywrócić oryginalną strukturę, kiedy przestaje być ona używana.

## Debuggowanie

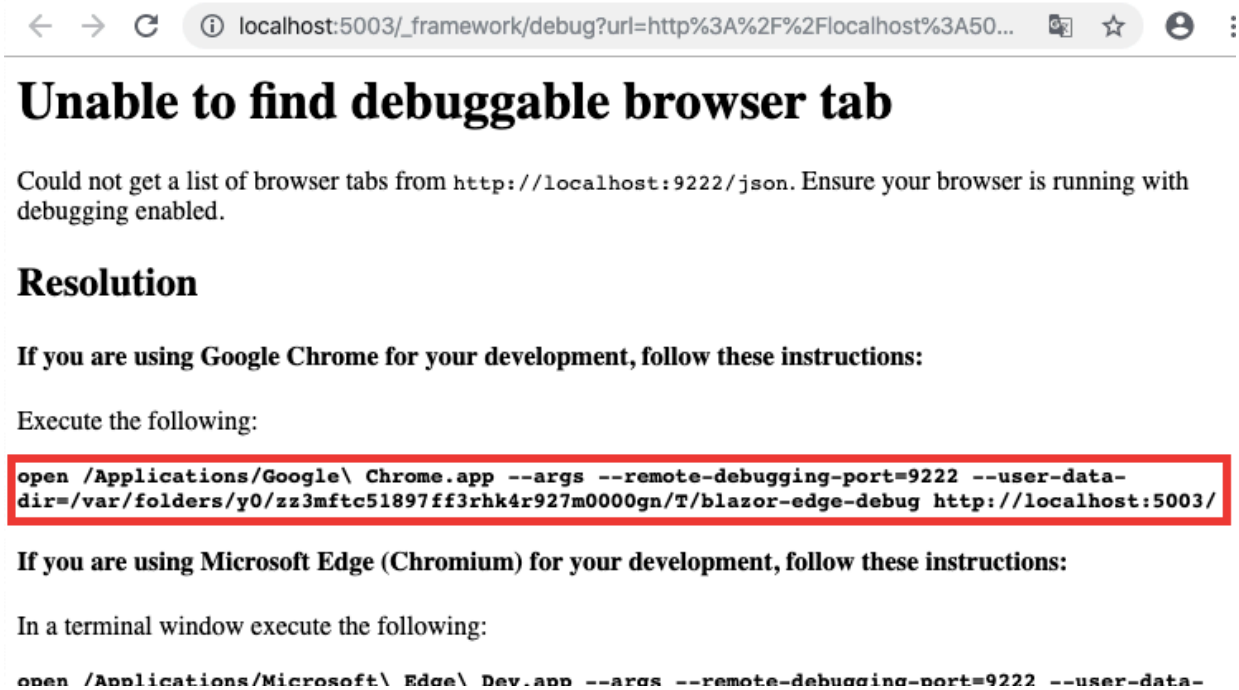
Warto zwrócić uwagę na sposób debuggowania aplikacji Blazor. W przypadku zastosowania modelu Serwer, niczym nie różni się ono od tradycyjnego projektu MVC. Problem pojawia się dopiero przy użyciu WebAssembly. Ponieważ aplikacja jest wykonywana po stronie przeglądarki klienta, programista nie ma możliwości zastosowania Visual Studio. Obecnie jedynie Chrome (Chromium, Edge, Opera itp.) przychodzą nam z pomocą oferując wbudowane narzędzie do debuggowania aplikacji WebAssembly. Z powodów bezpieczeństwa projekt musi być skompilowany w konfiguracji Debug.

Po załadowaniu strony przeglądarka podpowiada następne kroki w zakładce konsoli:

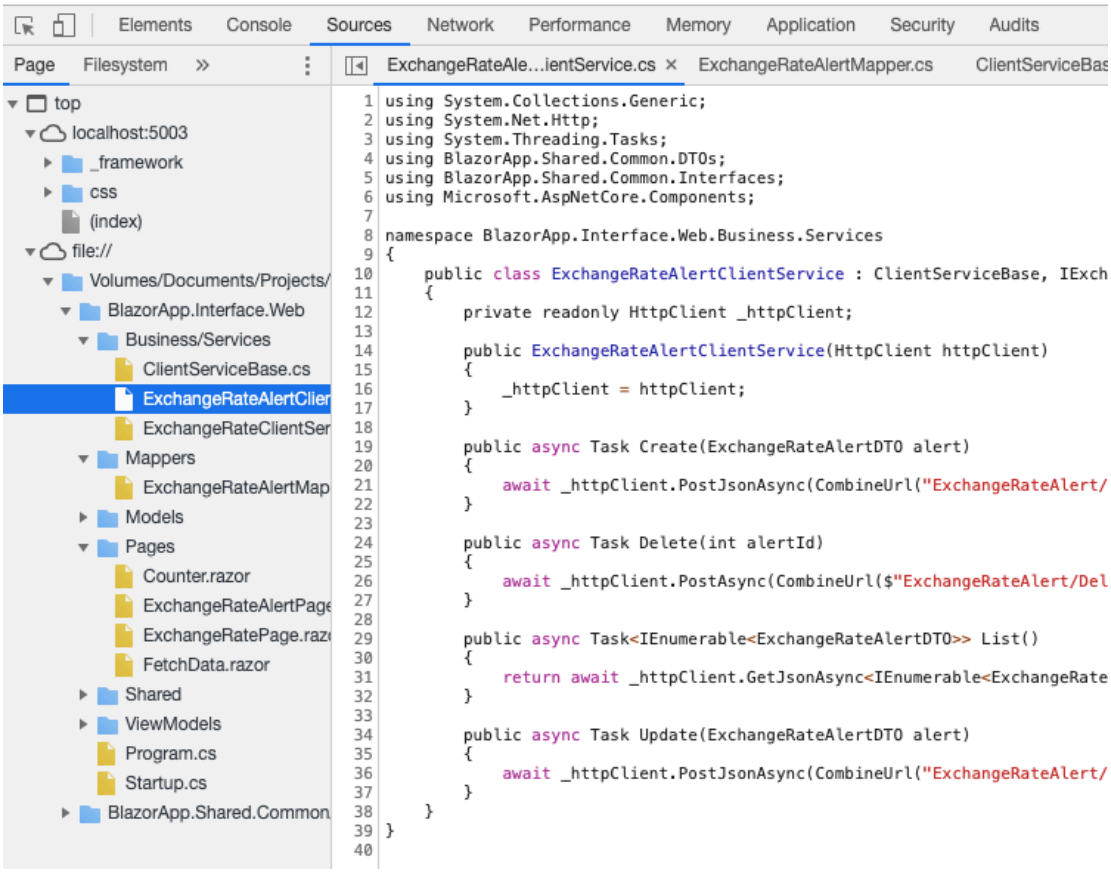



*Rys 5 – Podpowiedź konsoli Chrome*

Po użyciu następującej kombinacji klawiszy przeglądarka uruchomi tryb programisty albo stronę zawierającą dodatkowe kroki do wykonania (najczęściej należy skopiować podaną komendę i uruchomić przeglądarkę z konkretnymi parametrami)




*Rys 6 – Informacja od Chrome z komendą uruchamiającą przeglądarkę z wymaganymi parametrami*




Rys 7 – Widok kodu dostępny w konsoli programisty

Mała uwaga - jeżeli tryb programisty wyświetla komunikat o błędzie połączenia WebSocket warto spróbować połączyć się z aplikacją hostowaną za pomocą nieszyfrowanego protokołu http (w większości przypadków to pomaga).

## Demo

Przykładową aplikację znajdziesz pod adresem:

- Web: https://possibleblazorappweb.azurewebsites.net
- API: https://possibleblazorappapi.azurewebsites.net

Kod źródłowy: https://github.com/LukaszZiemblicki/BlazorApp.git

## Podsumowanie

Blazor stanowi niewątpliwie ciekawe podejście do tworzenia aplikacji. Jest to jedno z pierwszych (wprowadzonych oficjalnie) rozwiązań, zaprezentowanych przez Microsoft, pozwalające na tworzenie prawdziwych, dynamicznych stron internetowych. Najbardziej obiecująco zapowiada się model hostingowy WebAssembly który, pomimo że cały czas znajduje się w fazie testowej, pozwala na przygotowanie pierwszych projektów. Finalnie może on pomóc w zmniejszeniu kosztów utrzymania środowiska produkcyjnego oraz ujednolicenia rozwiązań technologicznych. Jednak największym problemem jest brak wsparcia dla starszych przeglądarek jak Internet Explorer 11 który dla sporej ilości korporacji cały czas jest podstawowym narzędziem pracy.

Mam nadzieję, że zarówno prezentacja jak powyższy dokument był dla Ciebie interesujący i zachęcił do wypróbowania tego rozwiązania. Pobierz proszę przygotowany przeze mnie testowy projekt i dopisz coś swojego – po kilku próbach zauważysz, że Blazor jest spójny i przemyślany, a pisanie aplikacji to czysta przyjemność. Jeżeli masz jakiekolwiek pytania albo uwagi – śmiało napisz do mnie wiadomość email. Postaram się odpisać tak szybko jak tylko możliwe. I pamiętaj, że to my – eksperci w danych dziedzinach – wyznaczamy trendy które będą obowiązywać być może przez kilka następnych lat. Nasza branża cały czas idzie do przodu, a wraz z nią musimy zmieniać nasze podejście i upodobania.

I najważniejsze – mam nadzieję, że pierwsze uruchomienie działającej aplikacji opartej o Blazor da ci tyle samo „dziecięcej radości” to mi ;)

# Bibliografia


- https://docs.microsoft.com/en-gb/aspnet/core/blazor/?view=aspnetcore-3.0
- https://github.com/cradle77
- https://docs.microsoft.com/en-gb/xamarin/xamarin-forms/enterprise-application-patterns/mvvm
- https://www.wintellect.com/model-view-viewmodel-mvvm-explained/
- https://aspnetmonsters.com/2016/08/2016-08-27-httpclientwrong/
- https://www.stevejgordon.co.uk/introduction-to-httpclientfactory-aspnetcore