Projektowanie aplikacji ASP.NET

Wykład 13/15
React vs Blazor

Wiktor Zychla 2024/2025

Spis treści

1 Wprowadzenie

W naszych opowieściach do tej pory pomijaliśmy temat bibliotek do tworzenia aplikacji frontendowych. Pokazaliśmy jak budować interfejs użytkownika sterowany „z serwera” (WebForms, MVC, ClickOnce) ale oczywiście istnieją biblioteki do budowania aplikacji, w których przeglądarka w całości odpowiada za interfejs, a serwer wyłącznie za dane. Od strony architektury wskazywaliśmy że taka hipotetyczna aplikacja pracuje najczęściej z usługami typu REST (WebAPI).

To nie miejsce na skomentowanie wszystkich możliwych bibliotek (Vue, Angular, jQuery, itd.) ale zdecydowanie, w kontekście ASP.NET, należy wspomnieć o bibliotece Blazor, ponieważ jest ona proponowana jako integralna część ASP.NET 5+. Dostawca technologii nie traktuje więc tej biblioteki już jako eksperymentu, tylko stawia ją na równi z innymi podsystemami ASP.NET.

Naszą prezentację obu technologii oprzemy o pokazanie dwóch przykładów – ta sama aplikacja powstanie w React i w Blazor. Pokażemy jak poszczególne elementy architektury aplikacji realizuje się w jednym i drugim przypadku.

Następnie pokusimy się o subiektywną opinię – o podsumowanie co z tego zestawienia wynika.

2 React + TypeScript

React to biblioteka typu open-source udostępniona na licencji MIT, której rozwój nadzoruje Facebook. Pierwsza wersja React została udostępniona w roku 2013. Za przełomową uznaje się wersję 16.8 z 2019 roku, która dodała wsparcie dla tzw. hooks czyli możliwości tworzenia kodu wyłącznie na modłę funkcyjną (komponenty są funkcjami).

Kluczowa dla React jest komunikacja z przeglądarką – biblioteka utrzymuje wirtualny obraz dokumentu (tzw. Virtual DOM) a automat (na który programista nie musi mieć wpływu) synchronizuje zmiany w wirtualnym dokumencie z rzeczywistym dokumentem renderowanym przez przeglądarkę.

O ile React jest biblioteką dla silnika Javascript w przeglądarce, o tyle programowanie w React zyskuje na kulturze wytwarzania jeśli zamiast Javascript użyje się Typescript.

Typescript to nadzbiór Javascript (poprawny Javascipt jest poprawnym Typescriptem, a taki Typescript który nie jest Javascriptem kompiluje się do Javascriptu), zaprezentowany po raz pierwszy w 2012 roku. Typescript jest utrzymywany przez Microsoft. Kamieniem milowym była wersja 2.0 z 2016 roku, w której oprócz dodanych wcześniej typów generycznych, pojawiło się znacznie rozbudowane śledzenie typów w ścieżkach przepływu kontroli. Wśród unikalnych cech Typescript należy wymienić m.in. unie i przecięcia typów. Programiści z doświadczeniem w językach obiektowych (C++/C#/Java), odnajdą w Typescript wiele znajomych elementów i jeden nowy – to tzw. typowanie strukturalne (w przeciwieństwie do typowania nominalnego w w/w językach).

Skondensowany przykład mechanizmów języka:

```
type Filter = <T>(t: T[], f: (elem:T) => boolean) => T[];

const filter: Filter = (t, f) => {
    const result = [];

    for ( let e of t ) {
        if ( f(e) ) {
            result.push(e);
        }
    }

    return result;
}

console.log(filter( [1,2,3], a => a >= 2 ));
console.log(filter( ['a', 'aa', 'aaa'], a => a.length >= 2 ));
```

Jako ciekawostkę można przywołać informację, że jednym z architektów języka Typescript jest Anders Hejlsberg, który wcześniej brał udział w projektowaniu języków Turbo Pascal, Delphi i C#.

Opcjonalnym, ale bardzo poprawiającym jakość pracy elementem, jest Webpack. Webpack pozwala zautomatyzować proces kompilacji i łączenia modułów Typescript do pojedynczego pliku Javascript, zawierającego całą aplikację.

Przepis na uzyskanie działającej aplikacji Typescript:

Zainstalować node.js. Zainstalować Visual Studio Code.

Utworzyć pusty folder. Z foldera uruchomić VSC poleceniem powłoki

```
code .
```

W VSC, otworzyć terminal i utworzyć aplikację oraz jej zależności

```
npm init -y
npm install webpack webpack-cli
npm install typescript ts-loader
npm install react react-dom
npm install @types/react @types/react-dom
```

W folderze aplikacji dodać plik **tsconfig.json**

```
{
    "compilerOptions": {
        "allowJs": false,
        "baseUrl": "./",
        "jsx": "react",
        "declaration": false,
        "esModuleInterop": true,
        "lib": ["ES6", "DOM"],
        "module": "commonjs",
        "moduleResolution": "node",
        "noImplicitAny": true,
        "outDir": "./dist/",
        "paths": {
        "@/*": ["src/*"]
        },
        "sourceMap": true,
        "target": "ES6"
    }
}
```

i **webpack.config.js**

```
    const path = require('path');
    const TerserPlugin = require("terser-webpack-plugin");

    module.exports = {
    //mode: 'production',
    mode: 'development',
    target: ['web','es6'],
    entry: {
        'index': './src/index.tsx'
    },
    output: {
        path: path.resolve(__dirname, 'dist'),
        filename: '[name].js'
```

```
    },
    devtool: 'source-map',
    module: {
        rules: [{
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
        }, ],
    },
    externals: {
        // Use external version of React
        //'react': 'React',
        //'react-dom': 'ReactDOM'
    },
    resolve: {
        // https://getfrontend.tips/shorten-import-paths-in-webpack/
        alias: {
        // Assume that the `src` folder is located at the root folder
        '@': path.join(__dirname, 'src'),
        },
        extensions: ['.tsx', '.ts', '.js'],
    },
    optimization: {
        minimize: false, // true/false
        minimizer: [
        new TerserPlugin({
            extractComments: false,
            terserOptions: {
            format: {
                comments: false,
            },
            },
        })
        ],
    },
    };
```

Utworzyć dwa puste foldery, **src** i **dist**, do pierwszego trafi kod źródłowy, do drugiego – wynik kompilacji aplikacji.

Utworzyć dwa pliki kodu TypeScript. Punkt wejścia aplikacji, **index.tsx**

```
import React from 'react';
import ReactDOM from 'react-dom';

import App from '@/app';

/**
 * Entry point
 */
class Program {

    Main() {

        var app = (
                <App />
        );

        ReactDOM.render(app, document.getElementById('root'));
    }
}

new Program().Main();
```

Drugi to komponent React, **app.tsx**

```
import React from 'react';

const App = () => {

  return <>
    Hello world from React & Typescript
  </>
};

export default App;
```

Zasada nazewnicza: moduły kodu mogą mieć rozszerzenie ***.ts**. Moduły zawierające kod JSX powinny mieć rozszerzenie ***.tsx**.

Ostatni plik to strona w której hostowana jest aplikacja, **app.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script defer src="/dist/index.js"></script>
    <style>
    * , *:before, *:after {
         box-sizing:border-box;
    }
    html {
        background-color: rgb(234, 238, 243);
    }

    </style>
</head>
<body>
    <div id="root"></div>
</body>
</html>
```

Teraz należy wydać polecenie

```
webpack
```

które zbuduje aplikację (jeśli webpack nie jest dostępny globalnie należy go dodatkowo zainstalować globalnie

```
npm install -g webpack webpack-cli
```

)

Aplikacja zostanie zbudowana, a kompilat trafi do **dist/index.js**

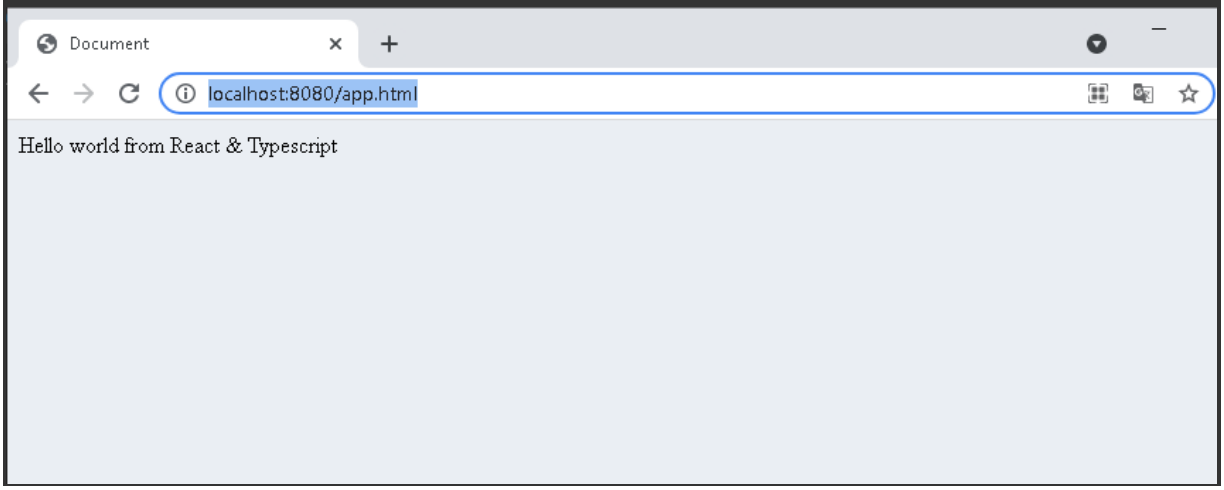
Do przetestowania aplikacji wymagany jest serwer zasobów statycznych. Aplikację można połączyć na przykład z widokiem MVC. Do prostego testu wystarczy prosty serwer, np. **live-server**

```
npm install -g live-server
```

```
live-server
```

Plik konfiguracyjny (launch.json) w VSC

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "pwa-chrome",
            "request": "launch",
            "name": "Launch Chrome against localhost",
            "url": "http://localhost:8080/app.html",
            "webRoot": "${workspaceFolder}"
        }
    ]
}
```



3 Blazor

Blazor to biblioteka open-source, udostępniona na licencji Apache, której rozwój nadzoruje Microsoft. Pierwsza wersja udostępniona w roku 2018. Językiem w którym tworzy się logikę aplikacji jest C#, stąd kontrowersyjny wybór języka docelowego w przeglądarce – nie jest nim Javascript tylko WebAssembly.

WebAssembly to format binarnego języka niezależnego od architektury systemu, dostępnego w przeglądarkach od mniej więcej 2018 roku. To język bardzo różny od Javascript – dzięki m.in. instrukcji skoku bezwarunkowego umożliwia łatwiejszą i w teorii wydajniejszą kompilację dowolnego języka wysokiego poziomu.

```
(module
  (func $add (param $lhs i32) (param $rhs i32) (result i32)
     local.get $lhs
     local.get $rhs
     i32.add)
  (export "add" (func $add))
)

WebAssembly.instantiateStreaming(fetch('add.wasm'))
  .then(obj => {
     console.log(obj.instance.exports.add(1, 2));
   });
```

Najważniejsze ograniczenie WebAssembly to brak bezpośredniego dostępu do dokumentu przeglądarki. Środowisko uruchomienione oparte o WebAssembly musi więc posiłkować się dodatkowym komponentem Javascript, który służy jako pomost między kodem WebAssembly a dokumentem. Ta konieczność użycia dodatkowego pośrednika obniża wydajność aplikacji wymagających intensywnych operacji na strukturze widoku.

Blazor posiada dwa modele hostowania:

3.1 Blazor Server

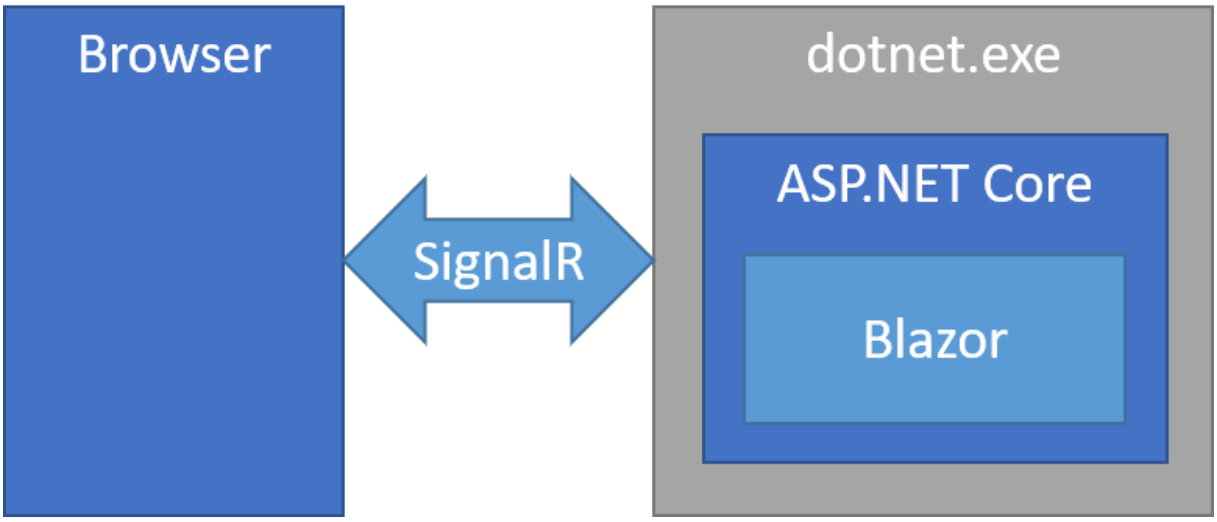
Blazor Server to model w którym aplikacja jest w całości wykonywana po stronie serwera.

Zalety:

- Rozmiar pobierania jest mniejszy niż aplikacja Blazor WebAssembly – brak narzutu na komponenty biblioteki standardowej
- Aplikacja wykonuje się po stronie serwera, ma więc łatwy dostęp do infrastruktury (np. do połączenia z bazą danych)
- Debugowanie jest naturalne (proces wykonuje się na serwerze)

Wady:

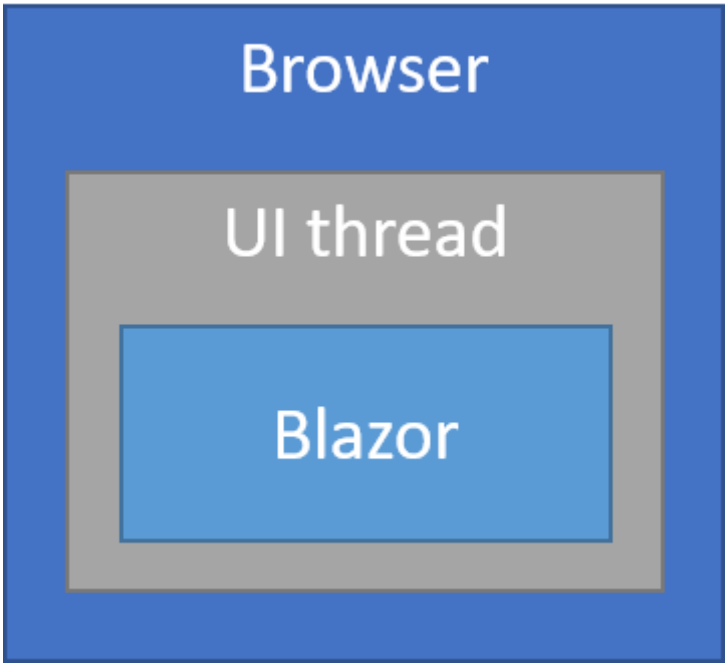
- Każda zmiana stanu aplikacji oznacza że trzeba użyć SignalR do poinformowania części klienckiej o tym że coś się zmieniło
- Nie ma obsługi trybu offline
- Skalowanie aplikacji z wieloma użytkownikami wymaga zasobów serwera do obsługi wielu połączeń klienta i stanu klienta – **jest to poważna wada**! Każdy klient Blazor Server to klient usługi SignalR. W praktyce jeden serwer obsługuje do kilku tysięcy równoległych użytkowników. Jest to model skalowania dopuszczalny w aplikacjach intranetowych, w aplikacji w sieci rozległej może być dużym ograniczeniem



Rysunek 1 Blazor Server, źródło: dokumentacja

3.2 Blazor WebAssembly

Blazor WebAssembly to model przypominający React – aplikacja wykonuje się w przeglądarce, kod aplikacji wykonuje silnik WebAssembly, a każdy kontakt z serwerem wymaga użycia obiekty **HttpClient**.



Rysunek 2 Blazor WebAssembly, źrodło: dokumentacja

Zalety:

- Łatwa skalowalność – ograniczona przepustowością usług REST/WebApi (setki lub tysiące razy większa niż Blazor Server)
- Nie wymaga określonego serwera WWW, można hostować nawet w serwerach zasobów statycznych

Wady:

- Duży rozmiar aplikacji i środowiska uruchomieniowego – **jest to poważna wada**! W sytuacji gdy typowe biblioteki klienckie mają rozmiar ~100kb, Blazor WebAssembly wymaga > 10MB runtime po stronie klienta

4 Porównanie

Podobieństwa:

- Model komponentów
- VirtualDOM / Incremental DOM

Różnice:

	React	**Blazor**
Rok udostępnienia	2013	2018
Pochodzenie	Facebook	Microsoft
Język kodu	Javascript/Typescript	C#
Język widoków	JSX	Razor
Kompilacja do	Javascript	WebAssembly
Zewnętrzne komponenty	Przeogromny wybór m.in. MUI, AntD, itp.	Zauważalnie mniejszy wybór
Obsługa formularzy	Zewnętrzne pakiety, np. react-hook-form	Wbudowana
Routing	Zewnętrzne pakiety, np. react-router-dom	Wbudowany
Klient HTTP	Zewnętrzny np. axios lub wbudowany w przeglądarkę np. fetch	Wbudowany
Rozmiar runtime	~150kb	> 10MB