

# Možnosti implementace frameworku Blazor do webové aplikace založené na frameworku React

Bc. Miloš Petrenčák

---

Diplomová práce  
2022



Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky

---

Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky  
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

# ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: **Bc. Miloš Petrenčák**  
Osobní číslo: **A20824**  
Studijní program: **N0613A140022 Informační technologie**  
Specializace: **Softwarové inženýrství**  
Forma studia: **Prezenční**  
Téma práce: **Možnosti implementace frameworku Blazor do webové aplikace založené na frameworku React**  
Téma práce anglicky: **Options for Implementing the Blazor Framework in a Web Application Based on the React Framework**

---

## Zásady pro vypracování

1. Popište současný stav technologií pro vývoj a zabezpečení webových aplikací.
2. Zaměřte se především na frameworky ASP.NET Core, Blazor a React.
3. Navrhněte aplikaci, definujte funkční a nefunkční požadavky, případy použití.
4. Navrhněte způsob zabezpečení komunikace mezi klientem a serverem.
5. Realizujte vývoj navržené aplikace ve frameworku React s implementací částí aplikace ve frameworku Blazor a popište klíčové části řešení.
6. Demonstrujte výsledky a formulujte závěr.

Forma zpracování diplomové práce: **tištěná/elektronická**

**Seznam doporučené literatury:**

1. PERES, Ricardo. Mastering ASP.NET Core 2.0: MVC patterns, configuration, routing, deployment, and more. Birmingham: Packt, 2017, xi, 471 s. ISBN 9781787283688.
2. PECINOVSKÝ, Rudolf. Návrhové vzory: [33 vzorových postupů pro objektové programování]. Brno: Computer Press, 2007, 527 s. ISBN 9788025115824.
3. FREEMAN, Adam. Pro ASP.NET Core 3: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages. 8th ed. 2020. Berkeley, CA: APress, 2020. ISBN 9781484254400.
4. Blazor [online]. Redmond, Washington, USA: Microsoft, 2021 [cit. 2021-10-12]. Dostupné z: <https://blazor.net/>
5. React [online]. Palo Alto, Kalifornie: Facebook, 2021 [cit. 2021-10-12]. Dostupné z: <https://reactjs.org/>
6. ASP.NET [online]. Redmond, Washington, USA: Microsoft, 2021 [cit. 2021-10-12]. Dostupné z: <https://docs.microsoft.com/cs-cz/aspnet/core/?view=aspnetcore-5.0>

Vedoucí diplomové práce: **Ing. Erik Král, Ph.D.**  
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **3. prosince 2021**

Termín odevzdání diplomové práce: **23. května 2022**

**doc. Mgr. Milan Adámek, Ph.D. v.r.**  
děkan



**prof. Mgr. Roman Jašek, Ph.D., DBA v.r.**  
ředitel ústavu

Ve Zlíně dne 24. ledna 2022

### **Prohlašuji, že**

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

### **Prohlašuji,**

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 23. 05. 2022

Miloš Petrenčák

## **ABSTRAKT**

Předmětem této práce je vývoj webové aplikace pro demonstraci implementace frameworku Blazor WebAssembly do webové aplikace založené na technologii React. V teoretické části jsou popsány technologie, které se dnes využívají na vývoj webových aplikací, důraz je kladen na frameworky WebAssembly, React a ASP .NET Core. V praktické části je kladen důraz na návrh a vývoj takové webové aplikace. Důraz je také kladen na zabezpečení. Nakonec je vytvořená aplikace demonstrována.

Klíčová slova: Blazor, Blazor WebAssembly, React, C#, Web

## **ABSTRACT**

The subject of this work is the development of a web application to demonstrate the implementation of the Blazor WebAssembly framework into a web application based on React technology. The theoretical part describes the technologies that are used today for web application development, the emphasis is on the frameworks WebAssembly, React and ASP .NET Core. The practical part emphasizes the design and development of such a web application. The emphasis is also on security. Finally, the created application is demonstrated.

Keywords: Blazor, Blazor WebAssembly, React, C#, Web

Především bych chtěl poděkovat vedoucímu své diplomové práce, panu Ing. Eriku Královi Ph.D, za trpělivost, ochotu a především za cenné rady při vypracovávání práce. Dále bych chtěl celé své rodině za jejich podporu po celou dobu mého studia.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

# OBSAH

<b>ÚVOD</b> .....	<b>9</b>
<b>I TEORETICKÁ ČÁST</b> .....	<b>10</b>
<b>1 SOUČASNÝ STAV TECHNOLOGIÍ PRO VÝVOJ WEBOVÝCH APLIKACÍ</b> .....	<b>11</b>
1.1 FRONT END TECHNOLOGIE.....	11
1.1.1 HTML .....	12
1.1.1.1 Struktura HTML stránky .....	13
1.1.1.2 Tagy atributy a elementy .....	14
1.1.1.3 Element head.....	15
1.1.1.4 Element Body .....	18
1.1.2 CSS.....	20
1.1.3 Tailwind CSS .....	20
1.1.3.1 Instalace .....	20
1.1.4 JavaScript .....	22
1.1.5 Angular.....	24
1.1.5.1 Instalace .....	24
1.1.6 React.....	25
1.1.6.1 Instalace .....	25
1.1.6.2 JSX.....	26
1.1.6.3 DOM .....	26
1.1.6.4 Komponenty.....	27
1.1.6.5 Stav a životní cyklus komponent .....	29
1.1.7 Blazor .....	34
1.1.7.1 Blazor WebAssembly .....	34
1.1.7.2 Standalone a Hosted Blazor WebAssembly .....	34
1.1.7.3 Razor Komponenty .....	36
1.2 BACKEND TECHNOLOGIE.....	41
1.2.1 ASP. NET Core.....	41
1.2.1.1 Dependency Injection .....	41
1.2.1.2 ASP.NET Core Middleware .....	44
1.2.1.3 Třída Program.cs.....	45
1.2.1.4 Kontroléry .....	45
1.2.1.5 Clean architecture .....	47
<b>2 ZABEZPEČENÍ WEBOVÝCH APLIKACÍ</b> .....	<b>49</b>
2.1 DUENDE IDENTITYSERVER.....	49
2.1.1 JWT .....	49
2.1.2 OAuth 2.0.....	50
2.1.3 OpenID Connect.....	50
2.1.4 Duende Identity server .....	50
<b>II PRAKTICKÁ ČÁST</b> .....	<b>52</b>
<b>3 NÁVRH APLIKACE</b> .....	<b>53</b>
3.1 FUNKČNÍ POŽADAVKY .....	53
<b>4 KLÍČOVÉ ČÁSTI APLIKACE</b> .....	<b>59</b>
4.1 APLIKACE REACT.....	59
4.1.1 Implementace frameworku Blazor .....	59

4.2	APLIKACE IDENTITYSERVER .....	62
4.3	APLIKACE MINIMALAPI .....	64
4.3.1	Zabezpečení endpointů.....	65
4.4	APLIKACE BLAZOR WEBASSEMBLY.....	66
4.4.1	Atomic design .....	66
4.4.2	Struktura projektu.....	67
4.4.3	Zabezpečení stránek .....	68
<b>5</b>	<b>ZABEZPEČENÍ KOMUNIKACE MEZI KLIENTEM A SERVEREM .....</b>	<b>71</b>
5.1	KONFIGURACE KLIENTA .....	71
5.2	KONFIGURACE IDENTITYSERVERU .....	72
5.3	KONFIGURACE API.....	74
<b>6</b>	<b>DEMONSTRACE APLIKACE .....</b>	<b>75</b>
6.1	SPUŠTĚNÍ APLIKACE .....	75
6.2	ÚVODNÍ OBRAZOVKA .....	75
6.3	PŘIHLÁŠENÍ .....	76
6.4	REGISTRACE.....	76
6.5	ADMINISTRÁTORSKÁ A MODERÁTORSKÁ STRÁNKA.....	77
6.6	STRÁNKA TOKEN .....	78
	<b>ZÁVĚR .....</b>	<b>79</b>
	<b>SEZNAM POUŽITÉ LITERATURY.....</b>	<b>80</b>
	<b>SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK .....</b>	<b>85</b>
	<b>SEZNAM OBRÁZKŮ .....</b>	<b>87</b>
	<b>SEZNAM TABULEK.....</b>	<b>89</b>
	<b>SEZNAM PŘÍLOH.....</b>	<b>90</b>



## ÚVOD

Nejpoužívanější technologie pro vývoj uživatelských rozhraní webových aplikací jsou JavaScriptové technologie a mezi nimi knihovna React. Framework Blazor WebAssembly je relativně nový framework, který má pro některé tu výhodu, že se nepíše v JavaScriptu, nýbrž v C#. Toto přináší bezesporu výhodu v tom, že vývojáři, kteří umí programovat v C# se mohou velmi rychle naučit v tomto frameworku. Framework Blazor WebAssembly je založený na principu WebAssembly, což znamená že uživatelské rozhraní běží v prohlížeči klienta a využíváme tedy výpočetní výkon klienta. Díky tomu nemusíme mít tak silný server, což ve výsledku může ušetřit nějaké náklady. Jelikož je React nejpoužívanější knihovnou pro vývoj uživatelských rozhraní a např. firma se rozhodne, že vyzkouší implementovat nové funkcionality pomocí frameworku Blazor WebAssembly, tak častým případem užitím může být právě implementace nového frameworku do již stávajících zaběhlých technologií.

Teoretická část se zabývá popisem aktuálně používaných technologií pro vývoj webových aplikací, přičemž je zaměřena především na knihovnu React, framework Blazor WebAssembly a ASP .NET Core. Dále je v teoretické části věnovaná část pro technologie HTML a CSS, jelikož bez nich se při vývoji uživatelských rozhraní neobejdeme, ať už použijeme jakýkoliv framework či knihovnu. V úvodu teoretické části je vyhodnocení dotazníku o oblibě JavaScriptových frameworků z webu stateofjs. Z těchto JavaScriptových frameworků je stručně popsán Angular a podrobněji je popsán React. Dále je pozornost zaměřena na framework ASP .NET Core, ve kterém je vyvíjena serverová část aplikace. V poslední řadě je také zmíněna, v dnešní době obzvlášť důležitá, technologie pro zabezpečení aplikací.

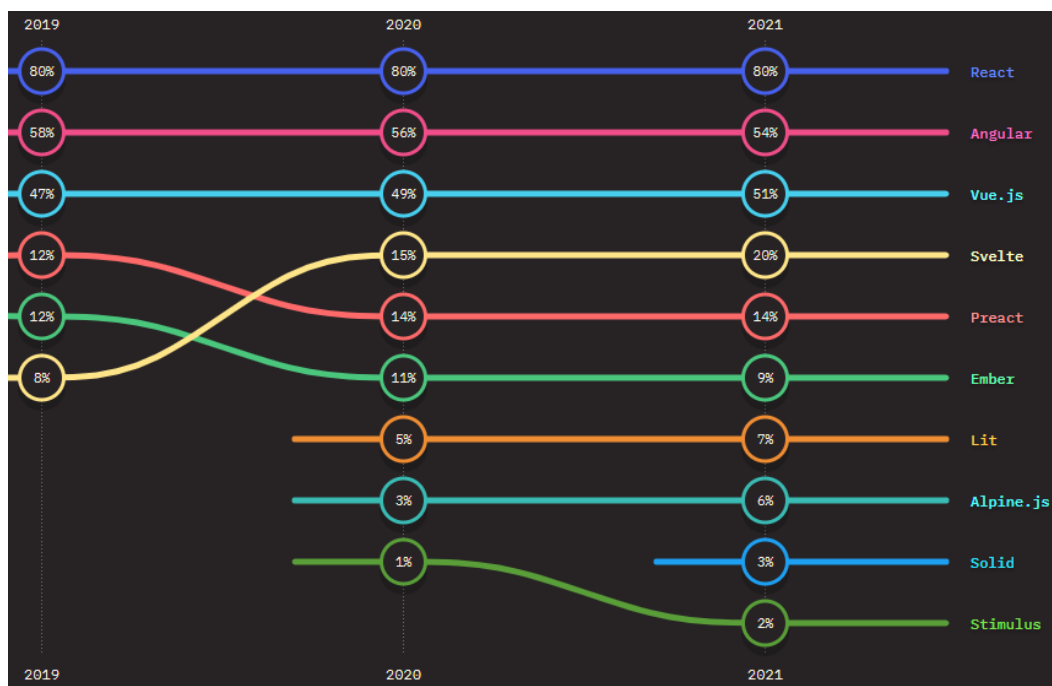
Praktická část práce se primárně zabývá vývojem dané webové aplikace. Jako první jsou při návrhu aplikace popsány funkční požadavky, nefunkční požadavky a případné scénáře použití aplikace. V další kapitole praktické části jsou popsány všechny důležité součásti webové aplikace. Jsou zde popsány všechny dílčí projekty, ze kterých se webová aplikace skládá. Dále je zde popsán způsob implementace aplikace vytvořené ve frameworku Blazor WebAssembly do aplikace vytvořené v Reactu. U jednotlivých projektů je popis použitých technologií. Dále jsou popsány použité návrhové vzory a způsob jakým se vytváří uživatelské rozhraní. Dále se v praktické části nachází podrobný popis konfigurace zabezpečení dané webové aplikace. Na závěr jsou demonstrovány výsledky vytvořené webové aplikace.

## **I. TEORETICKÁ ČÁST**

# 1 SOUČASNÝ STAV TECHNOLOGIÍ PRO VÝVOJ WEBOVÝCH APLIKACÍ

V této kapitole si jako první představíme, jak to vypadá aktuálně na trhu webových technologií. Jako první se zaměříme na frontendové technologie a poté si něco řekneme o backendových technologiích.

Na následujícím grafu máme zobrazeny některé JavaScriptové frameworky či knihovny od roku 2019 až po rok 2021. Procenta v jednotlivých rocích znamenají, kolik % dotázaných by tuto technologii použilo znovu. Z grafu vyplývá, že největší procento dotázaných, kteří by znovu použili danou technologii získal React. Na druhé pozici se umístil framework Angular. Za připomenutí by stála určitě i technologie Svelte. Tato technologie je zatím na trhu chvíli, ale v komunitě vývojářů začíná být v oblibě.



Obrázek 1: Graf procentuální znovu použitelnosti jednotlivých technologií v čase přejato z [1].

## 1.1 Front end technologie

Frontendový vývoj se zaměřuje na to, co vidí uživatel při návštěvě webové stránky, jako tlačítka, odkazy, animace a další. V dnešní době je pro vývoj takových aplikací celá řada různých frameworků, knihoven a nástrojů. Není si některé popíšeme.

### 1.1.1 HTML

HTML je zkratka pro Hyper Text Markup Language a je jedním ze základních stavebních kamenů ve vývoji webových aplikací. Html oficiálně vzniklo v roce 1993 [2] a od té doby se vyvíjelo až doposud do verze HTML5.

HTML je značkovací jazyk, který používáme ke strukturování obsahu webových stránek. HTML je do prohlížeče doručováno různými způsoby. HTML může být generováno aplikací na straně serveru, která jej vytváří v závislosti na požadavku nebo datech relace. HTML může být dále generováno pomocí aplikace na straně klienta, a to pomocí JavaScriptu, který tento kód generuje za běhu. V nejjednodušším případě může být HTML uloženo do souboru a doručen do prohlížeče pomocí webového serveru.

Podle konvence je HTML soubor uložen s příponou .html nebo .htm. Uvnitř tohoto souboru organizujeme obsah pomocí tzv. tagů. Tyto tagy obalují svůj obsah a každý tag dává speciální význam svému obsahu. Každý tag se zapisuje do ostrých závorek.

Uveďme nyní pár příkladů.

```
<p>Obsah tagu</p>
```

Tento fragment HTML kódu vytvoří pomocí tagu p odstavec, v němž bude "Obsah tagu".

V dalším příkladu vytvoříme seznam položek pomocí tagu ul. Tag ul znamená neuspořádaný seznam. Tento seznam obsahuje položky li tzv. položka seznamu. Takto vytvořený seznam položek pomocí tagů ul a li je naznačen na následujícím úryvku kódu.

```
<ul>
  <li>Neuspořádaná položka seznamu1</li>
  <li>Neuspořádaná položka seznamu2</li>
  <li>Neuspořádaná položka seznamu3</li>
</ul>
```

Jakmile prohlížeč zobrazí stránku HTML, všechny značky jsou interpretovány a prohlížeč vykreslí jejich vizuální podobu. Některá z těchto pravidel jsou vestavěná, například jak se vykresluje seznam nebo že odkaz je modře podtržený. Další taková pravidla lze nastavit pomocí kaskádových stylů CSS. HTML není prezentační, nezáleží na to, jak položky vypadají, nýbrž co jednotlivé položky znamenají. Je na prohlížeči, aby pomocí direktiv určil, jak položky vypadají. Tento vzhled je definován tvůrcem stránky pomocí jazyka CSS. [3]

### 1.1.1.1 Struktura HTML stránky

Každá správná HTML stránka by měla začít deklarací typu dokumentu. Tato deklarace sděluje prohlížeči, že se jedná o dokument typu HTML, případně o jakou verzi HTML se jedná. Taková deklarace HTML dokumentu je naznačena na dalším obrázku.

```
<!DOCTYPE html>
```

Po deklaraci typu dokumentu následuje element HTML. Tento element má otevírací a uzavírací tag. Element HTML ohraničuje veškerý HTML obsah stránky. Většina HTML tagů je párových, tedy s uzavíracím a otevíracím tagem. Uzavírací tag se píše stejně jako otevírací tag, ale před svým názvem má lomítko /. Takový element HTML je vyobrazen na následujícím úryvku zdrojového kódu.

```
<!DOCTYPE html>  
<html>  
</html>
```

Uvnitř elementu HTML se nachází další 2 důležité elementy – head a body.

Uvnitř elementu head se nachází tagy, kterou jsou nezbytné pro vytvoření stránky, jako je název, metadata, interní nebo externí styly CSS a JavaScript. Většina těchto věcí se na stránce vůbec nezobrazuje, ale pomáhá prohlížečům nebo vyhledávačům zobrazit stránku správně.

Uvnitř elementu body se nachází veškerý obsah stránky, jako jsou odstavce, tlačítka, nadpisy, odkazy a mnoho dalšího. [3]

Taková základní struktura HTML dokumentu se nazývá boilerplate a je naznačena na následujícím úryvku zdrojového kódu. Takovýto boilerplate by měl obsahovat každý HTML dokument. Většina vývojových prostředí nebo pokročilejších textových editorů umí takový boilerplate vygenerovat a není nutno jej psát ručně.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE-edge">
    <title>Titulek stránky</title>
    <meta name="viewport" content="width=device-width initial-scale=1">
    <link rel="stylesheet" type="text/css" media="screen" href="main.css">
    <script src="main.js"></script>
  </head>
</html>
```

### 1.1.1.2 Tagy atributy a elementy

Již několikrát byly zmíněny tagy a elementy. Jaký je tedy mezi nimi rozdíl?

Element se skládá ze sady:

- Otevírací tag
- Textový obsah (případně další elementy)
- Uzavírací tag

Pokud element nemá párové tagy můžeme psát buď otevírací nebo uzavírací tag. Takový element neobsahuje žádný textový obsah a můžeme jej nazvat i jako tag. Jako příklad si můžeme uvést element `br`, který zalomí řádek textu. Takovýto element je naznačen v následujícím výstřižku zdrojového kódu.

```
<br>
<br />
```

Atributy jsou vlastnosti HTML elementů. Připojují se k otevíracímu tagu elementu a nesou v sobě informace o daném elementu. Atributy se zapisují pomocí syntaxe klíč = "hodnota". Element může obsahovat i více než 1 atribut. Na následujícím obrázku je naznačen element `button` s atributy `type` a `id`. Atribut `type` říká prohlížeči, o jaký typ tlačítka se jedná a atribut `id` tento element jednoznačně identifikuje.

```
<button type="submit" id="submitButton">Submit tlačítko</button>
```

Dalším velmi používaným atributem u elementů je atribut `class`. Tento atribut se používá většinou při stylování HTML elementů pomocí CSS. Hlavní rozdíl v `class` atributu je ten, že může v sobě obsahovat i více hodnot, které se navzájem od sebe oddělují mezerou. Na

dalším řádku zdrojového kódu je zobrazen element p s atributem class, který obsahuje více hodnot v tomto atributu. [3]

```
<p class="a-class other-class">Text Odstavce</p>
```

### 1.1.1.3 Element head

Element head je tzv. záhlaví dokumentu. Jak již bylo zmíněno záhlaví obsahuje tagy, které jsou nezbytné pro vytvoření stránky. U tohoto elementu nikdy nepoužíváme atributy a nepíšeme do něj žádný textový obsah. Element head se používá jako kontejner, který obsahuje další elementy, resp. tagy.

Tento element zpravidla obsahuje tagy:

- Title
- Script
- Noscript
- Link
- Style
- Base
- Meta

#### Title tag

Tag title určuje název stránky. Tento název stránky se zobrazuje v prohlížeči, zpravidla v horní části prohlížeče, kde se nachází záložky se stránkami. Název stránky je velmi důležitý, protože je to jeden z klíčových faktorů pro optimalizaci vyhledávacích enginů (SEO). [3]

#### Script tag

Tento tag se používá pro přidání JavaScriptu na HTML stránku. JavaScript se může přidat tzv. inline, tzn., že se JavaScript píše přímo do HTML kódu. Takovýto inline skript je naznačen na následujícím obrázku.

```
<script>  
  console.log("JavaScript kód");  
</script>
```

Obvyklejší způsob přidání JavaScriptu do HTML souboru je takový, že se načte externí soubor pomocí atributu `src`. Takové přidání JavaScript souboru je zobrazeno na následujícím zdrojovém kódu.

```
<script src="main.js"></script>
```

Je důležité také podotknout, že často se načítání JavaScriptu do HTML souboru přidává až na konec stránky, před uzavírací tag `body`, nikoliv do hlavičky HTML souboru. Je to z důvodu toho, že pokud umístíme načítání JavaScriptu do hlavičky tak se JavaScriptové soubory načítají ještě před tím, než prohlížeč vykreslí HTML elementy v `body`. Takové načítání blokuje vykreslování elementů v `body`, dokud není skript analyzován a načten. Toto načítání může zpomalit celkové načtení stránky.

Pokud umístíme načítání JavaScriptu na konec stránky, tak se JavaScript načte a spustí až poté, co je celá stránka vykreslena. Takové načítání JavaScriptu na konci stránky vede k lepšímu uživatelskému zážitku při užívání stránky.

Dnes se již metoda načítání JavaScriptu na konci těla stránky označuje za zastaralou. Nejlepší způsob, jak načíst JavaScript je v hlavičce, ale tag `script` musí obsahovat atribut `defer`. Tento atribut zaručí to, že načítání skriptu se provádí asynchronně, tzn., že načítání skriptu není blokující operace. Načítání JavaScriptu s atributem `defer` je zahájeno paralelně s vykreslováním stránky a jeho spuštění nastane, jakmile je stránka vykreslena. Takovéto načítání JavaScriptu je nastíněno na následujícím úryvku zdrojového kódu. [3]

```
<script src="main.js" defer></script>
```

### Noscript tag

Tento tag se používá ke zjištění, zda jsou skripty v prohlížeči zakázány. Skripty mohou být vypnuty buď samotným uživatelem nebo je starší verze prohlížečů nemusí podporovat. Tag `noscript` lze použít buďto v hlavičce nebo v těle dokumentu. V hlavičce může tento tag obsahovat:

- Link tag
- Style tag
- Meta tag



Tyto tagy používá ke změně zdrojů stránky, nebo metainformací, pokud jsou skripty zakázány. Použití noscript tagu je zobrazeno na následujícím zdrojovém kódu.

```
<noscript>
  <style>
    .no-script-alert {
      display: block;
    }
  </style>
</noscript>
```

Při použití v těle HTML dokumentu může navíc tag noscript obsahovat elementy, které může prohlížeč vykreslit.

### Link tag

Tento tag se využívá k vytváření vztahů mezi HTML dokumentem a jinými zdroji. Link tag není párový a používá se zejména k připojení externích CSS dokumentů. Příklad použití link tagu je naznačen na následujícím řádku zdrojového kódu.

```
<link rel="stylesheet" type="text/css" media="screen" href="main.css">
```

### Style tag

Tento tag slouží podobně jako link tag k přidání CSS stylů, ale místo načítání stylů ze souboru se využívá přímo k psaní CSS kódu. Takový příklad psaní CSS kódu přímo do HTML souboru je naznačen na dalším útržku zdrojového kódu.

```
<style>
  .some-css {background-color: black;}
</style>
```

### Meta tag

Meta tagy plní na stránce různé úkoly, nejsou párové, a jsou velmi důležité zejména pro search engine optimization (SEO). Meta tag představuje metadata, která nemohou být reprezentována jinými tagy v hlavičce, jako jsou například link, script, base. Meta tagů existuje celá řada, uveďme pár příkladů.

- Description

Meta tag s názvem description slouží jako popisek stránky. Tento popis využívá např. Google při vyhledávání stránek. Příklad je nastíněn na následujícím řádku zdrojového kódu.

```
<meta name="description" content="Eshop s elektronikou">
```

- Viewport

Viewport je uživatelsky viditelná oblast webové stránky, která se liší podle typu zařízení uživatele. Na mobilním zařízení bude tato oblast menší než např. na širokoúhlém monitoru. Pokud je tento meta tag umístěn na webové stránce, při otevření na mobilním zařízení se stránka zobrazí na šířku zařízení. Dále lze nastavit i tzv. škálování tzn., že stránka se při otevření na mobilním zařízení zvětší. Příklad takového meta tagu je naznačen na následujícím řádku zdrojového kódu.

```
<meta name="viewport" content="width=device-width initial-scale=1">
```

- Refresh

Meta tag refresh lze použít k přesměrování na jinou stránku. Toto přesměrování lze nastavit i se zpožděním. Příklad přesměrování na jinou stránku je naznačeno dalším řádku zdrojového kódu. [3]

```
<meta http-equiv="refresh" content="0;url=http://google.com/">
```

#### **1.1.1.4 Element Body**

Element body je tzv. Tělo HTML dokumentu. V tomto těle se nachází veškerý obsah stránky. Elementy, které vytváří obsah rozdělujeme na 2 základní skupiny, blokové a inline elementy.

##### **Blokové a inline elementy**

Blokové elementy jsou elementy, které zabírají vždy celou dostupnou šířku, každý blokový element začíná vždy na samostatném řádku. U blokových elementů můžeme navíc měnit vlastnosti jako šířku, výšku, margin, padding, což u inline elementů nemůžeme. Některé příklady blokových elementů:

- p – odstavec
- ul – neuspořádaný seznam

- h – nadpis
- section – sekce
- div – blokový kontejner

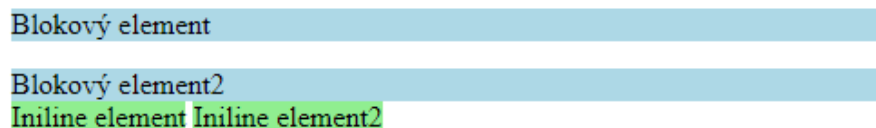
Inline elementy jsou elementy, které zabírají pouze svou vlastní šířku, lze na 1 řádek dát více elementu vedle sebe. Některé příklady inline elementů:

- button – tlačítko
- input – uživatelský vstup
- span – inline kontejner

Praktická ukázka rozdílu mezi inline a blokovým elementem je naznačena na následujících následujícím úryvku zdrojového kódu. V prvním z nich si definujeme jednoduché CSS třídy, které změní pouze barvu pozadí a přidají margin-top. Následně na to vykreslíme 2 blokové a inline elementy.

```
<style>
  .light-blue-background {
    background-color: lightblue;
    margin-top: 15px;
  }
  .light-green-background {
    background-color: lightgreen;
    margin-top: 15px;
  }
</style>

<div class="light-blue-background">Blokový element</div>
<div class="light-blue-background">Blokový element2</div>
<span class="light-green-background">Inline element</span>
<span class="light-green-background">Inline element2</span>
```



Obrázek 2: Blokový a inline element

Na obrázku 2 je vidět, jaký je rozdíl mezi inline a blokovými elementy. Důležité je podotknout, že pomocí CSS jsme nastavili `margin-top` pro oba typy elementů, avšak odsazení se projevuje pouze na blokových elementech. [3]

### 1.1.2 CSS

CSS je označení pro Cascading Style Sheet a spolu s HTML patří k základním stavební kamenům ve vývoji webových aplikací. Historie CSS sahá někdy do 90. let a vyvíjí se až dodnes. [4]

CSS je jazyk, který používáme pro stylování HTML souborů. Jazyk CSS je založený na pravidlech. Každé pravidlo se skládá ze 2 částí:

- Selektor
- Deklační blok

Selektor je řetězec, který definuje 1 nebo více elementů na webové stránce. Za selektorem následuje deklarační blok, ve kterém se nachází různá pravidla. Tato pravidla jsou zapsána ve tvaru pravidlo – hodnota. Na následujících obrázcích je naznačeno vytvoření CSS selektoru a jeho použití na HTML elementu. [4]

```
.selektor {  
  /*blok deklarací*/  
  background-color: aqua;  
}
```

```
<div class="selektor">Použití selektoru</div>
```

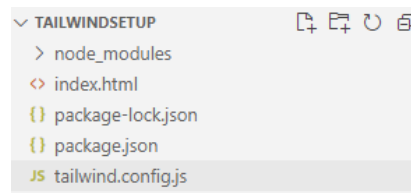
### 1.1.3 Tailwind CSS

Tailwind CSS je framework založený na CSS, který slouží pro rychlé vytváření vlastních uživatelských rozhraní. Tailwind framework funguje tak, že prohledá všechny soubory HTML, komponenty JavaScriptu a jakékoliv další šablony a vyhledá názvy tříd, vygeneruje odpovídající styly a poté je zapíše do statického souboru CSS.

#### 1.1.3.1 Instalace

Nejllepší způsob, jak nainstalovat Tailwind CSS do projektu je přes příkazy npm. Tyto příkazy jsou součástí node.js, které se musí taktéž nainstalovat. Příklad instalace do projektu bude naznačeno ve vývojovém prostředí Visual Studio Code.

Jako první si vytvoříme složku, do které umístíme index.html soubor. Tuto složku otevřeme ve VS Code a otevřeme si nový terminál. Do terminálu napíšeme příkaz “npm init -y”. Tento příkaz inicializuje projekt, tím že vytvoří nový soubor package.json. Poté napíšeme příkaz “npm i -D tailwindcss“ a “npx tailwindcss init”. První příkaz stáhne všechny potřebné soubory a druhý příkaz vytvoří konfigurační soubor. Složka s projektem by měla vypadat jako na následujícím obrázku 3.



Obrázek 3: Tailwind projekt

Nyní musíme otevřít soubor “tailwind.config.cs”, do kterého musíme napsat cesty na všechny html dokumenty a JavaScript komponenty. Tyto cesty budeme psát do vlastnosti “content”. Jelikož máme pouze index.html zapíšeme jen jej. Obsah souboru “tailwind.config.cs” je zobrazen na následujícím zdrojovém kódu.

```
module.exports = {
  content: ["../index.html"],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Nyní nám chybí už jen přidat CSS soubor, kde přidáme direktivy pro tailwind. Tento soubor můžeme pojmenovat libovolně, např “main.css” a obsah tohoto souboru je naznačen na následujícím útržku zdrojového kódu.

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Všechny soubory jsou připraveny, nyní je potřeba nastavit cesty pro sestavení výstupního CSS souboru. Na to použijeme příkaz “npx tailwindcss -i ./styles.css -o ./tailwind/output.css --watch”. Po vykonání tohoto příkazu se ve složce projektu vytvoří nová složka s názvem tailwind a bude obsahovat vygenerovaný CSS soubor output.css. Tento soubor se generuje

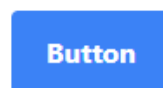
vždy při uložení souborů, které jsme v konfiguračním souboru uvedli. Nyní zbývá poslední krok přilinkovat vygenerovaný CSS soubor `output.css` do hlavičky HTML souboru `index.html`. Toto propojení je nastíněno na dalším řádku zdrojového kódu. [5]

```
<link rel="stylesheet" href="./tailwind/output.css">
```

Na následujícím výstřižku zdrojového kódu je naznačeno použití `tailwind` CSS přímo v HTML kódu.

```
<button class="bg-blue-500 rounded text-white font-bold px-5 py-3">  
  Button  
</button>
```

Výsledné tlačítko může, vytvořeno pomocí `tailwind` CSS je ukázáno na dalším obrázku.



Obrázek 4: Tlačítko

### 1.1.4 JavaScript

JavaScript byl vytvořen v roce 1995 [6] a od té doby prošel rozsáhlým vývojem. Byl to první skriptovací jazyk který, byl nativně podporován webovými prohlížeči a díky tomu získal značnou výhodu oproti konkurenci. Ve své podstatě je to stále jediný jazyk, který můžeme použít pro vytváření webových aplikací. Existují sice jiné jazyky, ale všechny se kompilují do JavaScriptu. V dnešní době se začíná používat i `WebAssembly`, ale o tom později.

Vlastnosti JavaScriptu:

- Vysokoúrovňový programovací jazyk

JavaScript patří do tzv. Vysokoúrovňových programovacích jazyků. Tyto jazyky poskytují vyšší úroveň abstrakce, která umožňuje lepší čitelnost kódu pro člověka. Javascript spravuje paměť pomocí `garbage collectoru`, takže se programátor může více soustředit na vývoj samotné aplikace než na správu paměti.

- Dynamický

Na rozdíl od statických programovacích jazyků, dynamický programovací jazyk provádí za běhu mnoho věcí, které statický programovací jazyk dělá při kompilaci. Tato funkcionality má své klady i zápory a poskytuje nám to výhodné funkce jako dynamické typování, reflexi a mnoho dalšího.

- Dynamicky typovaný

Programovací jazyk, který je dynamicky typovaný nevyžaduje datový typ při deklaraci proměnných. Můžeme také například přiřazovat čísla do proměnných, ve kterých se nachází řetězec. Takové přiřazení je zobrazeno na dalším obrázku.

```
let variable1 = 'abc';
let variable2 = 10;
console.log(variable1);
//Vypíše abc
console.log(variable2);
//Vypíše 10
```

- Volně typovaný

Opakem volně typovaných jazyků jsou jazyky, které jsou silně typované. Volně typový jazyk nevynucuje typ objektu, což umožňuje větší flexibilitu, ale odepírá nám možnost kontroly tohoto typu.

- Interpretovaný

U interpretovaných programovacích jazyků je pro spuštění programu nezbytný jeho zdrojový kód a program zvaný interpret, který zdrojový kód provádí. Opakem interpretovaných jazyků jsou jazyky kompilované. Kompilované jazyky je nutné před spuštěním přeložit překladačem do strojového kódu.

- Multi paradigmatický

Multi paradigmatický jazyk nevynucuje žádné zvláštní programovací vzory jako například Java, která nás nutí objektově orientovanému programování.

S čistým JavaScript se dnes již velmi často nesetkáme. U vývojářů je v oblibě používat spíše JavaScriptové frameworky. JavaScriptových frameworků existuje celá řada např. React, Angular, Vue.js, Svelte, Preact, Ember a mnoho dalších. [7]

### 1.1.5 Angular

Framework Angular je framework založený na komponentách, které slouží k vytváření re-sponzivních webových aplikací. Obsahuje celou řadu integrovaných knihoven, které poskytují celou řadu funkcí, včetně routingu, formulářů, komunikaci klienta se serverem a spoustu dalších. Ekosystém angularu se skládá z různorodé skupiny více než 1,7 milionu vývojářů, autorů knihoven a tvůrců jiného obsahu.

#### 1.1.5.1 Instalace

Podobně jako u Tailwind CSS budeme vytvářet nový projekt v příkazovém řádku. Příklad instalace bude stejně jako minule v terminálu VS Code pomocí příkazů npm.

Jako první musíme nainstalovat CLI – Command Line Interface pro Angular. K tomu slouží příkaz:

```
npm install -g @angular/cli
```

Jakmile máme nainstalováno CLI můžeme vytvořit nový projekt, který vytvoříme příkazem:

```
ng new my-first-project
```

Tento příkaz vytvoří nový adresář s názvem „my-first-project“ a stáhne do něj veškerý obsah. Nyní již stačí jen otevřít vytvořenou složku příkazem:

```
cd .\my-first-project\
```

A jako poslední spustíme projekt. K tomu slouží příkaz:

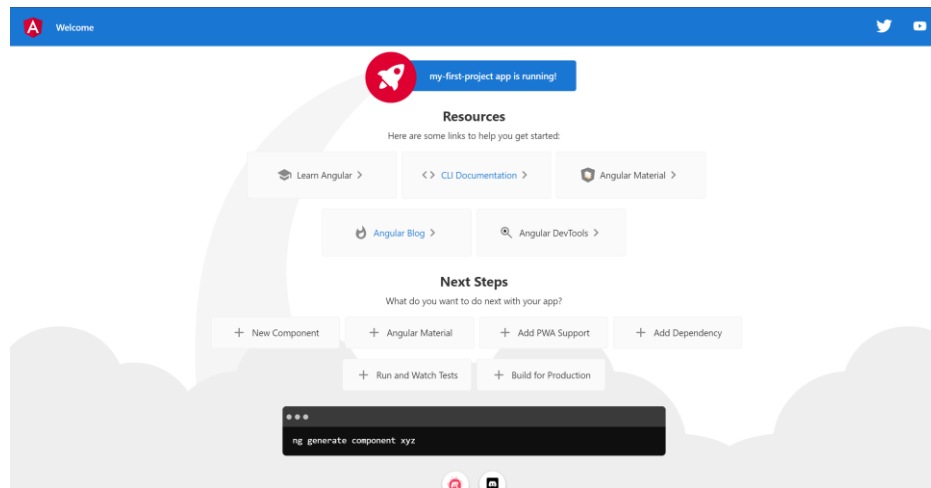
```
ng serve
```

Projekt je nasazen lokálně na portu 4200. Pro otevření stačí do prohlížeče zadat adresu:

```
http://localhost:4200/
```

Takto vytvořený projekt v Angularu je naznačen na následujícím obrázku 5. [8]





Obrázek 5: Spuštění základního projektu

## 1.1.6 React

React je jednou z nejpopulárnějších technologií pro vývoj frontendových částí aplikace. Vývoj Reactu zprostředkovává společnost Meta, starším názvem Facebook. Narozdíl například od Angularu, React není framework, ale pouze JavaScriptovou knihovnou. [9]

Obdobně jako u ostatních frontendových technologií, tak i React klade důraz na komponenty. Tyto komponenty jsou různé znovupoužitelné elementy HTML, které mají své vzhledové vlastnosti a zapouzdřenou funkcionalitu. Skládáním těchto komponentů vzniká kompletní uživatelské rozhraní aplikace.

### 1.1.6.1 Instalace

Ukázka instalace Reactu bude naznačena v již známém vývojovém prostředí VS Code. Začneme tím, že si ve VS Code otevřeme prázdnou složku a v této složce si otevřeme nový terminál.

Jako první nainstalujeme správce balíčků yarn příkazem:

```
npm install -g yarn
```

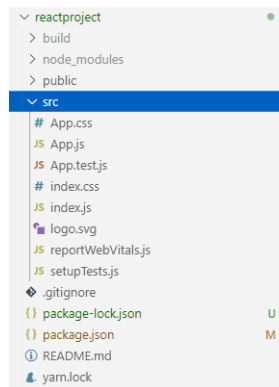
Jakmile máme nainstalováno můžeme přejít na vytvoření React projektu příkazem:

```
yarn create react-app <jmenoaplikace>
```

Jakmile proběhne vytvoření React aplikace, tak se vytvoří nový adresář, který se bude jmenovat stejně jako název aplikace v předchozím příkaze. Následně přejdeme do tohoto vytvořeného adresáře a v tomto adresáři spustíme projekt pomocí příkazu: [9]

```
yarn start
```

Základní struktura takto vytvořeného projektu je naznačena na obrázku 6.



Obrázek 6: Struktura základního React projektu

### 1.1.6.2 JSX

Jednou z největších předností této knihovny je JSX a jedná se o syntaktické rozšíření JavaScriptu. JSX umožňuje přiřazovat proměnným v JavaScriptu elementy z HTML. Takové přiřazení je naznačeno v následující výstřižku kódu. [10]

```
const JSXelement = <h1>HTML element</h1>;
```

### 1.1.6.3 DOM

Zkrátka DOM znamená Document Object Model. DOM je standardní logická reprezentace jakékoliv webové stránky. Zjednodušeně řečeno, DOM je stromová struktura, která obsahuje všechny prvky s jejími vlastnostmi na dané webové stránce. DOM poskytuje jazykově neutrální rozhraní, které umožňuje přístup a aktualizaci obsahu libovolného prvku webové stránky.

Předtím než přišel React vývojáři přímo manipulovali s DOM prvky, což vedlo k časté manipulaci s DOM, a pokaždé, když byla provedena aktualizace musel prohlížeč přepočítat a překreslit celou stránku, což způsobovalo, že celý proces potřeboval větší množství času. React přinesl jako vylepšení virtuální DOM. Virtuální DOM lze označit jako kopii skutečné

DOM, která se používá k uchování aktualizací provedených uživatelem, a nakonec je přenesena do původního DOM prohlížeče najednou, což vyžaduje mnohem méně času.

K manipulaci s DOM elementy je k dispozici v Reactu balíček s názvem React-dom. Tento balíček poskytuje metody sloužící k manipulaci s DOM. Tento balíček importujeme pomocí následujícího kódu.

```
import * as ReactDOM from 'react-dom';
```

V tomto balíčku se nachází metody jako:

- createPortal
- flushSync

Metoda createPortal

Tato metoda vytváří tzv. portál. Portály poskytují způsob, jak vykreslit child komponentu do DOM uzlu, který existuje mimo hierarchii komponenty DOM. Užití této metody je nastíněno na následujícím úryvku kódu.

Metoda createPortal nahradila metodu render od verze Reactu 18.

```
ReactDOM.createPortal(child, container);
```

Metoda flushSync

Tato metoda se využívá k synchronní aktualizaci uvnitř poskytnutého callbacku. Tímto je zajištěna okamžitá aktualizace DOM. Na následujícím výstřižku kódu je příklad použití takové metody. [11]

```
ReactDOM.flushSync(callback);
```

#### ***1.1.6.4 Komponenty***

Komponenty jsou základní prvky při tvorbě uživatelských rozhraní. Komponenty umožňují rozdělit uživatelské rozhraní na nezávislé a opakovaně použitelné části.

Konceptuálně jsou komponenty jako funkce JavaScriptu. Přijímají libovolné vstupy, které nazýváme tzv. props a vracejí React elementy, které popisují, co by se mělo uživateli zobrazit na obrazovce.

Nejjednodušším způsobem, jak definovat React komponentu je napsat ji jako JavaScriptovou funkci. Taková funkce je naznačena na následujícím útržku kódu.

```
function WelcomeComponent(props) {  
  return <h1>Ahoj, {props.name}</h1>;  
}
```

Taková JavaScriptová funkce je validní React komponenta, jelikož přijímá jediný argument, objekt s daty props, a vrací React element. Tyto komponenty nazýváme funkční komponenty, protože jsou to doslova funkce JavaScriptu.

Na následující kódu je naznačena komponenta, která již není ve tvaru funkce, nýbrž ve tvaru třídy, která implementuje rozhraní.

```
class WelcomeComponent extends React.Component {  
  render() {  
    return <h1>Ahoj, {this.props.name}</h1>;  
  }  
}
```

Podívejme se nyní na následující kód a popišme si jeho funkci.

```
function WelcomeComponent(props) {  
  return <h1>Ahoj, {props.name}</h1>;  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
const element = <Welcome name="Sara" />;  
root.render(element);
```

Jako první si definujeme vlastní komponentu s názvem WelcomeComponent. Tato komponenta přijímá ve svém parametru proměnnou typu objekt s názvem props. Komponenta vrací HTML element h1. V těle tohoto HTML elementu se nachází text „Ahoj,“, za kterým se nachází vlastnost objektu props s názvem name. K těmto vlastnostem daného objektu přistupujeme pomocí tečky a umísťujeme je do složených závorek. Tyto složené závorky slouží k výpisu hodnoty dané vlastnosti.

V další části kódu si definujeme 2 proměnné s názvem root a element. Tyto proměnné se definují pomocí klíčového slova const. Jako první definujeme root element pomocí metody

`createRoot`, která se nachází v balíčku `ReactDOM`. K této metodě přistupujeme obdobně jako k vlastnostem, a to pomocí tečky. Jako parametry předáváme HTML element, který vrací JavaScript metoda `getElementById`. Uvnitř této metody se nachází řetězec, který jednoznačně identifikuje HTML element na stránce v našem případě `root`. HTML element `root` je `div` element a umísťuje se především na hlavní stránku `index.html` a je v něm zobrazován React obsah.

Nyní nám stačí jen definovat proměnnou pro náš vlastní komponentu `WelcomeComponent` a tento element zobrazit v DOM `root`. [12]

### 1.1.6.5 Stav a životní cyklus komponent

Když jsme si řekli, jakým způsobem vytváříme své vlastní komponenty a zobrazujeme je v DOM, představme si nyní stav a životní cyklus těchto komponent.

Uvažme nyní příklad tikajících hodin a popišme si jej v úryvcích zdrojového kódu.

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function tik() {
  const element = (
    <div>
      <h1>Tik!</h1>
      <h2>Čas: {new Date().toLocaleTimeString()}</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

Na začátku kódu je již známé definování `root` proměnné.

Následně si definujeme funkci `tik`. Uvnitř této funkce se nachází JSX definice elementu, který obsahuje HTML tagy `h1` a `h2`, které jsou zároveň obě uzavřeny v 1 `div` elementu. Tyto elementy pouze vypisují na obrazovku text „Tik!“, „Čas: “ a aktuální hodnotu času podle časové zóny ve které máme jazyk prohlížeče. Následuje volání metody `render` pro zobrazení daného elementu.

Nakonec, až za tělem funkce, voláme `setInterval`, kde jako první parametr předáváme naši funkční komponentu a jako druhý parametr interval v ms. Metoda `setInterval` ve svém zpětném volání volá metodu `tick` s periodou 1000.

Takové definování vlastní komponenty není moc praktické a neumožňuje znovu používání dané komponenty. Aby komponenta byla lépe znovupoužitelná je potřeba implementovat state do této komponenty. Definujme nyní tuhle komponentu jako třídu. Implementace takové komponenty jako třída je naznačena na dalším úryvku kódu.

```
const root = ReactDOM.createRoot(document.getElementById('root'));

class Hodiny extends React.Component {
  render() {
    return (
      <div>
        <h1>Tik!</h1>
        <h2>Čas: {this.props.datum.toLocaleTimeString()}</h2>
      </div>
    );
  }
}

function tick() {
  root.render(<Hodiny datum={new Date()} />);
}

setInterval(tick, 1000);
```

Naši vlastní komponentu jsme deklarovali jako třídu, která dědí od třídy `React.Component`. V těle této třídy se nachází metoda `render` a uvnitř této metody se nachází klíčové slovo `return`, které obsahuje nám již známe HTML elementy. Rozdílem však je, že hodnota aktuálního času je předávána komponentě `Hodiny` v parametru `props`. Jelikož se odkazujeme na objekt `props`, který náleží třídě, ve které pracujeme, použijeme klíčové slovo `this`.

Komponenta `Hodiny` je nyní definována jako třída. Metoda `render` bude volána pokaždé, když je vyžádána aktualizace. Pokaždé když vykreslíme komponentu `Hodiny` do stejného DOM uzlu, bude použita stejná instance této třídy. Tato vlastnost nám umožňuje další funkce jako je stav a životní cyklus.

Implementujeme nyní vnitřní stav naší vlastní komponenty. Abychom mohli tuto funkcionálnost implementovat musíme k třídě `hodiny` přidat konstruktor. Konstruktor je metoda, která se volá při vytvoření nové instance dané třídy. Jako další úpravu uděláme to, že se aktuální čas nebude předávat pomocí parametru komponenty. Tyhle úpravy jsou naznačeny na následujícím zdrojovém kódu.

```
class Hodiny extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Tik!</h1>
        <h2>Čas: {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hodiny />);
```

Tímto jsme úspěšně nastavili stav komponenty. Jelikož jsme nikde nenastavili časovač, který bude komponentu aktualizovat, aktuální čas se nám ukáže akorát při načtení stránky. Abychom docílili aktualizování komponenty každou vteřinu musíme u komponenty hodiny nastavit časovač, když je tato komponenta poprvé zobrazena v DOM.

Činnost, při které něco nastavujeme při prvním vykreslení komponenty nazýváme v Reactu `mountign`. Opačná činnost, tedy činnost, při které komponentu odstraníme z DOM se nazývá `unmounting`. Obě tyto metody lze naimplementovat do naší třídy, kde máme vlastní komponentu. Takové metody jsou nastíněny na následujícím úryvku zdrojového kódu. Tyhle metody nazýváme jako `lifecycle` metody.

```
componentDidMount() {
}

componentWillUnmount() {
}
```

Jako první implementujeme metodu `componentDidMount`. V těle této metody nastavíme časovač. Implementaci provedeme způsobem, že vlastnost `timerID` nastavíme interval pomocí metody `setInterval`. Této metodě předáme jako parametr třídní funkci `tik`, kterou vytvoříme za okamžik. Implementace metody tedy může vypadat následovně.

```
componentDidMount() {  
  this.timerID = setInterval(  
    () => this.tik(),  
    1000  
  );  
}
```

V implementaci druhé lifecycle metody pouze zavoláme metodu, která odstraní časovač. Taková implementace může vypadat následovně.

```
componentWillUnmount() {  
  clearInterval(this.timerID);  
}
```

Nyní nám chybí implementovat pouze metoda `tik`. Tuto metodu bude volat komponenta `Hodiny` každou vteřinu. Tato metoda nastaví do stavu komponenty vždy aktuální čas. Tento stav by se neměl nastavovat přímo, ale měla by se pro to využít metoda `setState`. Implementace metody `tik` může vypadat následovně.

```
tik() {  
  this.setState({  
    date: new Date()  
  });  
}
```

Tímto jsme úspěšně implementovali všechny potřebné metody. Pokud kód spustíme, uvidíme, že se čas aktualizuje každou vteřinu. Nyní je naše vlastní komponenta připravena na použití na více místech v projektu. Podívejme se tedy na celý zdrojový kód naší vlastní komponenty a zrekapitulujme si její.

Jakmile je komponenta `Hodiny` předána metodě `root.render`, spustí se konstruktor komponenty `Hodiny`. V konstruktoru se inicializuje objekt `state`, kterému se přiřadí nový objekt typu `Date`.

React poté zavolá metodu `render` uvnitř komponenty `Hodiny`. Tohle je způsob, jak se React dozví, jaký obsah má být zobrazen na obrazovce. React poté aktualizuje DOM tak, aby odpovídal obsahu uvnitř metody `render`.

Jakmile je tento obsah metody `render` vložen do DOM, React zavolá lifecycle metodu `componentDidMount`. Uvnitř této metody se komponenta `Hodiny` požádá prohlížeč, aby nastavil časovač, který zavolá metodu `tik` každou vteřinu.



Každou vteřinu zavolá prohlížeč metodu `tik`. Komponenta `Hodiny` naplňuje aktualizaci uživatelského rozhraní voláním metody `setState` s parametrem typu objekt, který obsahuje aktuální čas. Díky tomu, že zavoláme metodu, React ví že se změnil stav komponenty, a zavolá metodu `render`, která překreslí uživatelské rozhraní. [13]

```
class Hodiny extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tik(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tik() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Tik!</h1>
        <h2>Je: {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hodiny />);
```

### 1.1.7 Blazor

Blazor je framework, který slouží pro vytváření interaktivního webového uživatelského rozhraní na straně klienta. Formuje bohaté interaktivní uživatelské rozhraní pomocí jazyka C# namísto JavaScriptu. Umožňuje sdílet logiku aplikací na straně serveru a na straně klienta napsanou v .NET. Vykresluje uživatelské rozhraní jako HTML a CSS. Tyto technologie podporuje celá řada webových prohlížečů, včetně mobilních.

#### 1.1.7.1 *Blazor WebAssembly*

WebAssembly zkráceně WASM je binární formát instrukcí pro virtuální stroj fungující na principu zásobníku. WebAssembly je navržen jako přenositelný kód, který umožňuje nasazení na webových stránkách. WebAssembly je vyvíjen pod zastřešením W3C. Do této skupiny přísluší společnosti Mozilla, Microsoft, Google a Apple. [14]

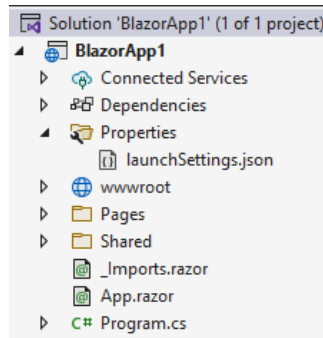
Blazor WebAssembly je zdarma open-source webový framework vyvíjený společností Microsoft. Jako jiné frontendové frameworky, tak i Blazor WebAssembly je framework založený na komponentách.

#### 1.1.7.2 *Standalone a Hosted Blazor WebAssembly*

Máme několik způsobů, jak navrhnout architekturu Blazor WebAssembly projektu. Každá verze má své klady i zápory, a odlišují se i jejich případ použití. Asi nejtypičtější jsou verze standalone a verze hosted.

##### Standalone

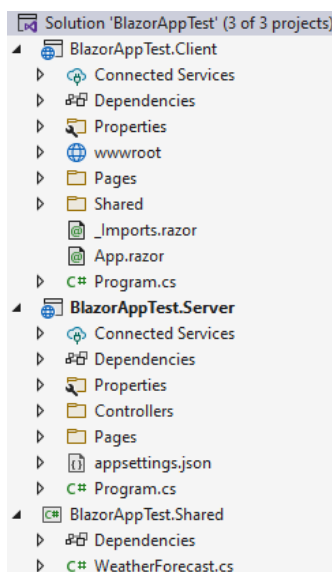
Ve standalone verzi není WebAssembly hostována na ASP.NET Core backendu. Nejčastěji je takto navržená aplikace hostována přímo u uživatele v prohlížeči. Takto navržená aplikace nám umožňuje hostovat WebAssembly bez jakéhokoliv serveru. Samozřejmě, že standalone WebAssembly aplikace většinou komunikuje s nějakým backendovým API, ale tohle API nemusí být vůbec ve stejném řešení jako standalone WebAssembly, a už vůbec tohle API nemusí být ASP.NET Core. V mém případě jsem zvolil verzi WebAssembly jako standalone, protože se Blazor WebAssembly bude spouštět z projektu, který běží na Reactu. Pro vytvoření takového projektu použijeme Visual Studio 2022, který má grafické uživatelské rozhraní, kde vytvoříme podle průvodce nový projekt typu Blazor WebAssembly App a máme hotovo. Takto vytvořený projekt je naznačen na následujícím obrázku 7. [15]



Obrázek 7 Standalone Blazor WebAssembly projekt

## Hosted

Ve verzi hosted je WebAssembly hostována na ASP.NET Core backendu. ASP.NET Core backend slouží jako API pro WebAssembly, se kterým komunikuje např. pomocí HTTP požadavků. Pro tuto verzi většinou umístíme API a Blazor WebAssembly do jednoho řešení. Obvykle máme v řešení ještě alespoň projekt, který obsahuje objekty, resp. třídy, které jsou společné jak pro API, tak pro Blazor WebAssembly. Pro vytvoření takového projektu použijeme Visual Studio 2022, který má grafické uživatelské rozhraní, kde vytvoříme podle průvodce nový projekt typu Blazor WebAssembly App. Následně v průvodci vybereme, že bude aplikace hostovaná na ASP.NET Core. Jak by měl vypadat vytvořený projekt je naznačeno na následujícím obrázku 8. [15]



Obrázek 8 Hosted Blazor WebAssembly projekt

### 1.1.7.3 Razor Komponenty

Princip Blazoru, stejně jako jiných frontendových frameworků či knihoven je založen na komponentách. V Blazoru těmto komponentám říkáme razor komponenty a tyto soubory mají příponu .razor. Konvence Blazoru udává že komponenty jsou pojmenovány tzv. CamelCasem, tzn že názvy komponent píšeme s velkým počátečním písmenem a každé další nové slovo je taky velkým písmenem, ostatně tak jako v celém C#. Přibližme si nyní, jaké možnosti nám Blazor nabízí, co se týče vytváření komponent.

#### Rozložení razor komponenty

Razor komponenta se skládá primárně ze dvou částí, a to z části pro HTML kód, a částí pro C# kód. Tyto části mohou být buď v 1 společném razor souboru nebo lze HTML kód uchovávat v odlišném souboru jako C# kód.

Na následujícím výstřižku kódu je naznačeno jak vypadá razor komponenta s HTML a C# kódem v 1 souboru.

```
@page "/cestaStranky"
<h1>Ahoj, @_jmeno </h1>
@code
{
    private string? _jmeno;

    protected override async Task OnInitializedAsync()
    {
        Console.WriteLine("Kódová sekce stránky");
        _jmeno = "Josef";

        await ValueTask.CompletedTask;
    }
}
```

Popišme si nyní tento kód, kde se nachází C# a HTML v 1 souboru.

Jak je již zvykem z C# nebo i z jiných programovacích jazyků, tak se na začátku souboru nachází direktivy či importy. Direktivy v souborech typu .razor začínají vždy znakem @.

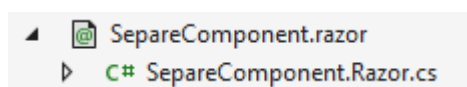
Jako první se nachází direktiva page. Ta to direktiva určuje, jakou adresou přistupujeme na tuto stránku pomocí adresového řádku v prohlížeči.

Za direktivy následuje HTML kód. V této sekci se nachází většinou razor komponenty a je zde definované celé uživatelské rozhraní. Pro jednoduchost se v HTML kódu nachází pouze

1 element, nadpis h1, ve svém těle vypíše řetězec „Ahoj, “ a doplní hodnoty proměnné `_jmeno`, v našem případě Josef.

Jako poslední sekce se v souboru nachází sekce s C# kódem. Tato sekce je vždy jednoznačně definována pomocí klíčového slova `code`, kde veškerý C# kód je psán do složených závorek. V této sekci se zpravidla nachází logika uživatelského rozhraní.

Pokud bychom chtěli rozdělit C# a HTML část do dvou souborů, vytvoříme novou třídu, která se jmenuje stejně jako razor komponenta. Příklad takové komponenty je nastíněn na následujícím obrázku 9.



Obrázek 9 Rozdělení komponenty razor do 2 souborů

Jediným rozdílem v implementaci je ten, že tuto třídu musíme definovat jako `partial`. Výhodou takového rozdělení je, že pokud máme komponentu, která obsahuje velké množství kódu je implementace přehlednější, jinak principy zůstávají stejné jako když máme kód v 1 souboru. Na následujícím úryvku kódu je naznačena definice `partial` třídy.

```
public partial class SepareComponent
{
}
```

### Parametry komponenty

Základní věcí, kterou budeme potřebovat při psaní vlastních komponent je, aby komponenty přijímaly parametry zvenčí.

Parametry komponent předávají komponentám data a jsou definovány pomocí veřejných vlastností. Tyto vlastnosti mají navíc atribut `[Parameter]`.

Jako příklad si uveďme komponentu `CustomPage`, která bude přijímat jako textový řetězec parametr s názvem stránky. [16]

```
<div class="h1">
  @Title
</div>

@code
{
  [Parameter]
  public string? Title { get; set; }
}
```

Volání takto vytvořené vlastní komponenty vypadá následovně.

```
<CustomPage Title="Parametr"></CustomPage>
```

Abychom mohli dovnitř tagu CustomPage vkládat další komponenty, potřebujeme tuto funkcionalitu implementovat. Implementace spočívá v přidání vlastnosti RenderFragment do třídy komponenty, a je naznačena na následujícím kódu. [17]

```
<div class="h1">
  @Title
</div>

@ChildContent

@code
{
  [Parameter]
  public string? Title { get; set; }

  [Parameter]
  public RenderFragment? ChildContent { get; set; }
}
```

Pomocí nového atributu ChildContent typu RenderFragment můžeme v této komponentě vykreslit veškerý obsah uvnitř této komponenty. Volání takové komponenty může vypadat následovně.

```
<CustomPage Title="Parametr">
  <FetchData />
</CustomPage>
```

Uvnitř naší vlastní komponenty se nachází komponenta FetchData, která je v projektu při jeho vytvoření. Místo razor komponent můžeme vkládat i obyčejné HTML elementy a jiné.

Kaskádové parametry komponent

Představme si nyní příklad, že máme v sobě zanořených 2 a více komponent. Chceme, aby se při předání parametru první rodičovské komponentě předal parametr až poslední komponentě v celé hierarchii.

Jeden způsob, jak bychom mohli tento druh problému vyřešit, je předávat si ve všech komponentách parametry postupně. Ovšem pokud budeme mít např. 10 úrovní komponent, a ne všechny úrovně budou tento parametr využívat je toto řešení ne úplně praktické.

K dalšímu možnému řešení toho problému lze použít kaskádové parametry. Kaskádové parametry poskytují pohodlný způsob toku dat z rodičovské komponenty do libovolného počtu podřízených komponent. Výhodou tohoto řešení je si nemusíme předávat parametry postupně všemi úrovněmi hierarchie.

K Implementaci této funkcionality využijeme komponentu s názvem `CascadingValue` a použijeme ji v rodičovské komponentě. V rodičovské komponentě navíc přebíráme parametr, který dále svoji hodnotu posílá v celé hierarchii. Taková implementace je naznačena na následujícím výstřižku zdrojového kódu.

```
<h3 style="@Style">rodičovská komponenta</h3>
<CascadingValue Value="@Style">
  <ChildComponent />
</CascadingValue>

@code
{
  [Parameter]
  public string Style { get; set; }
}
```

Vykreslení takto vytvořené vlastní komponenty je naznačeno v následujícím kódu. V tomto kódu předáme komponentně řetězec ve vlastnosti `Style`, která se předá všem následujícím komponentám.

```
<ParentComponent Style="color: lightcoral"></ParentComponent>
```

Implementace dalších dvou navazujících komponent může být například taková. Na následujícím kódu je následující komponenta za rodičovskou komponentou. Tato komponenta žádným způsobem nepracuje a ani nepředává dál vlastnost `Style`.

```
<h3>potomek rodiče</h3>
```

```
<GrandChildComponent />
```

```
@code  
{  
}
```

Komponenta, která je v hierarchii až poslední může mít následující implementaci.

```
<h3 style="@Style">potomek potomka</h3>
```

```
@code  
{  
    [CascadingParameter]  
    public string? Style { get; set; }  
}
```

Kdybychom kód zpustili, dostali bych výstup, který by vypadal následovně. [18]

rodičovská komponenta  
potomek rodiče  
potomek potomka

Obrázek 10 Demonstrace kaskádových parametrů

## Vykreslování komponent

Nyní si povíme něco o vykreslování komponent v ASP.NET Core Blazor WebAssembly aplikacích, včetně toho, kdy používat metodu `StateHasChanged`. Tato metoda totiž při jejím zavolání překreslí danou komponentu.

Jako první se komponenta vždy musí vykreslit, jakmile je přidána do komponentové hierarchie v těle své rodičovské komponenty. Toto je jediný případ kdy se komponenta musí vykreslit. Uvedme si nyní další případy, kdy se komponenty renderují.

- Při aktualizování parametrů převzatých z rodičovské komponenty.
- Při aktualizování kaskádových parametrů.
- Při vyvolání jedné z vlastních obslužných rutin událostí.
- Při volání metody `StateHasChanged`.

Jelikož se komponenty renderují při výše zmíněných událostech, není tedy potřeba při nich volat znovu metodu `StateHasChanged`.

Uvedme si nyní pár příkladu, kdy takové volání naopak vhodné je.



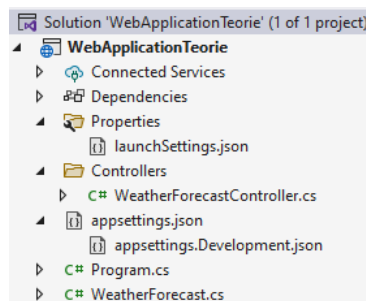
- Při asynchronní obslužné rutině událostí, která zahrnuje více asynchronních fází.
- Když je renderování a zpracování událostí voláno externě.
- Při renderování komponenty mimo podstrom, který je znovu vykreslen konkrétní událostí. [19]

## 1.2 Backend technologie

### 1.2.1 ASP.NET Core

ASP.NET Core je multiplatformní open source framework pro vytváření moderních webových aplikací, který je vyvíjen společností Microsoft. Aktuální verze tohoto frameworku verze 6. [20]

Začneme rovnou vytvořením nového projektu. Nový projekt se stejně jako u Blazor WebAssembly vytváří pomocí vývojového prostředí Visual Studio 2022, které má grafické uživatelské rozhraní. V něm vytvoříme nový projekt typu ASP.NET Core Web API, ve kterém můžeme rovnou nechat nakonfigurovat kontroléry. Takto vytvořené základní webové API může vypadat jako na obrázku 11.



Obrázek 11 Výchozí ASP .NET Core projekt

Vysvětleme si první pár základních pojmů a pak si popíšeme, jak takový projekt vypadá. V nové verzi ASP.NET Core 6 vidíme, že základní projekt je poněkud jednodušší, není zde již třída startup.cs.

#### 1.2.1.1 Dependency Injection

Dependency injection je softwarový návrhový vzor, který nám umožňuje implementovat Inversion of Control (IoC) mezi třídami a jejich závislostmi. [21]

Každá aplikace se skládá z mnoha tříd, které spolu vzájemně interagují a navzájem na sobě závisí. Obvyklá implementace je taková, že každá třída je zodpovědná za reference ke svým závislostem. Taková implementace vede k tomu, že třídy mají příliš mnoho vzájemných odkazů a kód se špatně spravuje.

Existují 3 základní typy dependency injection. [22]

- Injection v konstruktoru
- Injection setrem
- Injection rozhraním

Pravděpodobně nejčastější metodou dependency injection v ASP.NET Core je za pomoci konstrukturu třídy.

Uvedme si nyní příklad služby, kterou injektujeme do třídy. Na následujícím úryvku kódu se nachází třída kontroléru, rozhraní služby a třída této služby.

```
public class EventController : ControllerBase
{
    private readonly ILogger<EventController> _logger;
    private readonly IEventService _eventService;

    public EventController(
        IEventService eventService,
        ILogger<EventController> logger
    )
    {
        _eventService = eventService;
        _logger = logger;
    }
}

public interface IEventService
{
    public Task AddEventAsync(CreateEventRequest createEventRequest);
    public Task<IReadOnlyCollection<EventResponse>> GetAllAsync();
}

class EventService : IEventService
{
    public async Task AddEventAsync(CreateEventRequest createEventRequest)
    {
        await DoSomethingAsync();
    }

    public async Task<IReadOnlyCollection<EventResponse>> GetAllAsync()
    {
        await DoSomethingAsync();
        return new List<EventResponse>();
    }
}
```

Popišme si nyní tento kód. Máme zde třídu `EventController`, která dědí od třídy `ControllerBase`.

Jako první v těle této třídy se nachází deklarace dvou vlastností. Obě vlastnosti jsou nastaveny jako privátní a jen pro čtení, jak je zvykem. První z nich je rozhraní pro logger a druhá vlastnost je rozhraní pro službu.

Po deklaraci vlastností se nachází konstruktor dané třídy. V konstruktoru probíhá injektování závislostí do této třídy. V parametru konstruktoru předáváme této třídě rozhraní zvenčí a v těle tohoto konstruktoru inicializujeme dříve deklarované vlastnosti dané třídy.

Následně za třídou pro kontrolér se nachází rozhraní `IEventService`. Toto rozhraní definuje metody, které musí všechny jeho třídy implementovat.

Jako poslední se v tomto úryvku kódu nachází třída `EventService`. Tato třída implementuje rozhraní `IEventService`, a s tím i všechny metody tohoto rozhraní.

Mohlo by se zdát, že máme závislosti úspěšně injektovány, ale pokud bychom se snažili aplikaci spustit, bez toho, aniž bychom služby registrovali, aplikace by nemusela fungovat správně, či by si nemusela spustit vůbec.

### Konfigurace služeb

Pro konfiguraci služeb ve frameworku ASP.NET Core využíváme rozhraní `IServiceCollection` a máme 3 základní typy, jak můžeme služby registrovat. [23]

- `Transient`
- `Scope`
- `Singleton`

Registrace služeb určuje životní cyklus daných služeb a určuje, jak často dependency injection kontejner vytváří nové instance daných služeb. Představme si nyní tyto způsoby, můžeme služby registrovat.

**Singleton** registrace služeb vytváří instanci pouze jednou pro dependency injection kontejner. Tato instance je vytvořena na požádání a její životní cyklus je po celou dobu běhu aplikace. Vraťme se není k příkladu, kdy jsme do kontroleru injektovali službu `IEventService`. Na následujícím řádku je naznačena registrace služby jako singleton. [23]

```
services.AddSingleton<IEventService, EventService>();
```

Všechny 3 zmíněné metody pro registraci služeb se nejčastěji volají se 2 generickými parametry, přičemž jako první z nich je rozhraní pro danou službu, a druhým z generických parametrů je třída této služby. Podmínkou je také že daná třída ve své definici implementuje toto rozhraní.

**Transient** registrace metod vytváří novou instanci pokaždé, když je služba využita. Jinými slovy, všechny třídy, které tuto závislost injektují vždy v konstruktoru dostanou jinou instanci. Pokud si při vývoji zatím nejsme jistí, jakým způsobem chceme životní cyklus dané služby udržovat, je transient nejbezpečnější varianta. Na následujícím řádku kódu je naznačena registrace služby `IService` jako transient pomocí metody `AddTransient`. [23]

```
services.AddTransient<IService, Service>();
```

**Scoped** registrace služeb vytváří vždy novou instanci pro 1 požadavek (HTTP kontext). Takový způsob implementace zaručuje sdílení instance dané služby po celou dobu trvání požadavku. Na následujícím řádku kódu je naznačen příklad registrace scoped služby. [23]

```
services.AddScoped<IService, Service>();
```

### 1.2.1.2 ASP.NET Core Middleware

Middleware je software, který se nachází v pipeline (kanálu) pro zpracování požadavků a odpovědí. Tato pipeline se skládá z posloupnosti delegátů jednotlivých požadavků, který se volají závisle jeden na druhém. Příklad takové pipeline se nachází ve třídě `Program.cs` a je naznačena na následujícím úryvku kódu. [24]

```
// Konfiguruje pipeline HTTP požadavků .
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.MapControllers();

app.Run();
```

Popišme si nyní princip této základní pipeline. Jako první se do pipeline přidá swagger. Swagger je sada open-source nástrojů postavena na OpenAPI specifikaci, která vývojářům umožní navrhovat, sestavovat, testovat, dokumentovat a používat REST API. [25]

Další v pipeline se nachází `UseHttpsRedirection()`. Tahle metoda slouží přesměrování obyčejných HTTP požadavků na zabezpečený HTTPS protokol. [26]

Jako další se v pipeline nachází `MapControllers()`. Pomocí této metody se namapují koncové body kontrolérů a my k nim tak můžeme přistupovat. [27]

A jako poslední se nachází metoda `Run()` na spuštění webové aplikace.

### 1.2.1.3 Třída *Program.cs*

Třída `Program.cs` obsahuje spouštěcí kód aplikace. V této třídě se nachází 2 základní sekce.

- Sekce pro konfiguraci služeb
- Sekce pro definování kanálu na zpracování požadavků (pipeline)

Již jsme si zmínili, jakým způsobem se registrují služby v ASP.NET Core. Tyto služby se obvykle registrují ve třídě `Program.cs`. Na následujícím úryvku kódu je naznačena malá část sekce pro konfiguraci služeb.

```
var builder = WebApplication.CreateBuilder(args);  
var configuration = builder.Configuration;  
var services = builder.Services;
```

Popišme si nyní těchto pár řádků kódu. Na prvním řádku voláme metodu `CreateBuilder(args)` nad třídou `WebApplication`, která do proměnné `builder` přiřadí novou instanci `WebApplicationBuilder` s předdefinovanými vlastnostmi. [28]

Poté si deklaruujeme dvě další nové proměnné. Proměnná `configuration` slouží především pro přístup k nastavení aplikace, které se nachází v souboru `appsettings.json`.

V proměnné `services` se nachází rozhraní `IServiceCollection`, které spravuje služby dané aplikace. [29]

### 1.2.1.4 Kontroléry

Nyní si představme další důležitou vlastnost tohoto frameworku, což jsou kontroléry. Abychom lépe pochopili, k čemu kontroléry slouží vysvětlíme si nejprve co je to MVC.

MVC architektonický vzor

Princip architektonického vzoru Model View Controller je takový, že rozděluje aplikaci na 3 hlavní části. [30]

- Model
- View
- Controller

Tento architektonický vzor nám umožňuje lepší oddělení kódu uživatelské části a backend kódu. Hlavním důvodem, proč tento architektonický vzor používáme, je lepší udržitelnost, testovatelnost, či další rozšiřování kódu.

Princip tohoto architektonického vzoru je takový, že uživatelský požadavek je přeměřován na kontrolér. Tento kontrolér zajišťuje spolupráci mezi modelem a požadavky z uživatelského rozhraní. Implementace takového kontroléru v ASP.NET Core je naznačena na dalším úryvku zdrojového kódu.

```
[Route("api/[controller]")]
[ApiController]
public class EventController : ControllerBase
{
    private readonly ILogger<EventController> _logger;
    private readonly IEventService _eventService;

    public EventController(
        IEventService eventService,
        ILogger<EventController> logger)
    {
        _eventService = eventService;
        _logger = logger;
    }

    [HttpPost]
    [Route(nameof(Create))]
    public async Task Create(CreateEventRequest createEventRequest)
    {
        _logger.LogInformation("POST request to 'api/event/Create' ");
        await _eventService.AddEventAsync(createEventRequest);
    }

    [HttpGet]
    [Route(nameof(GetAll))]
    public async Task<IReadOnlyCollection<EventResponse>> GetAll()
    {
        _logger.LogInformation("GET request to 'api/event/GetAll' ");
        return await _eventService.GetAllAsync();
    }
}
```

Na tomto úryvku kódu se nachází třída `EventController`, který dědí od třídy `ControllerBase`. Dále má tato třída 2 atributy, které jsou v hranatých závorkách a jsou umístěny nad deklarácí názvu třídy.

První z atributů je atribut `ApiController`. Tento atribut používáme, když u tohoto kontroléru chceme povolit následující typy chování. [31]

- Je požadovaný atribut `Route`
- Automatická HTTP 400 odpověď
- Odvození binding parametrů
- Zobrazení detailu HTTP chyby

Jelikož tento kontrolér má atribut `ApiController` další atribut musí být atribut `Route`. Tomuto atributu předáváme v kulatých závorkách řetězec, který slouží k přístupu ke kontroléru. Cesta k tomuto kontroléru bude vypadat „`api/event/`“.

V těle třídy pro kontrolér se jako první nachází deklaráce služeb, které se zde injektují v konstruktoru tohoto kontroléru za pomoci `dependency injection`.

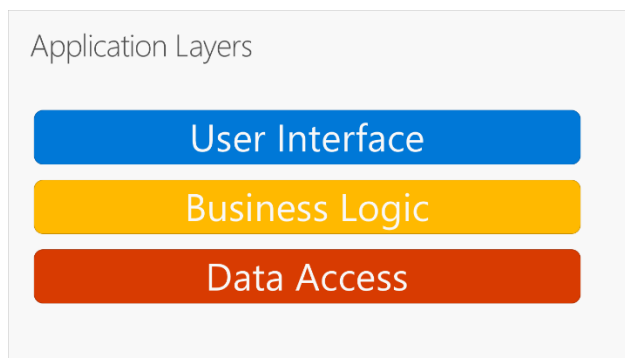
Za konstruktorem následují definice metod kontroléru. Jako první metoda je metoda s názvem `Create`. Tato metoda, stejně jako celý kontrolér má nad svou deklarácí atributy, atribut s názvem `HttpPost`. Tento parametr definuje, jaký typ HTTP požadavku metoda obsluhuje, v toto případě je to metoda `POST`. Dále je tato metoda definována jako `asynchronní` a ve svém parametru přebírá data z těla metody `POST`, které pak poskytuje službě, a ta data zpracuje. Navíc, pokud je zaznamenán přístup na tento koncový bod, měl by se zaznamenat do logu. Takové logování může velmi pomoci při ladění aplikace na produkčním prostředí.

Další metoda je `GetAll`. Stejně jako předchozí metoda, tak i tato metoda má atribut, ve kterém se nachází, jaký typ HTTP požadavku daná metoda obsluhuje, v tomto případě je to HTTP metoda `GET`. Tato metoda nepřebírá žádný parametr, ale pro změnu nějaká data vrací. V těle této metody je opět volání služby a následně zapsání informace do logu.

### 1.2.1.5 *Clean architecture*

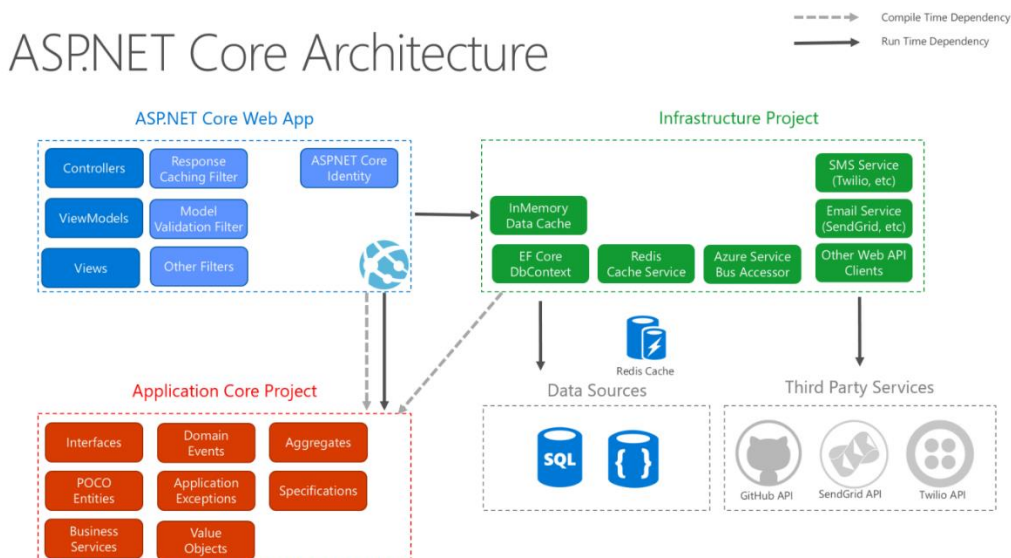
Klasická vrstvená architektura využívá principu, kdy požadavky z uživatelského rozhraní jsou zpracovány v jádru aplikace, které posílá požadavky na databázi. Hlavní nevýhodou tohoto tradičního vrstvení je, že závislosti při kompilaci jdou od uživatelského rozhraní až

po databázi. Znamená to, že uživatelské rozhraní je závislé na jádře aplikace, a to je závislé na databázi. Jádro aplikace, které většinou obsahuje hlavní logiku aplikace je závislé na přístupu do databáze, což není nejlepší stav. Na následujícím obrázku 12 je nastíněna taková architektura.



Obrázek 12 Klasické vrstvení aplikace přejato z [39]

Oproti tomu Clean architecture má hlavní logiku své aplikace ve svém středu. Hlavní výhodou je, že zde nemáme závislost jádra na závislosti přístupu k datům. V Clean architecture je tato závislost obrácená, tedy přístup k datům závisí na jádře. Implementace této funkcionality je pomocí rozhraní v jádru aplikace. Na následujícím obrázku 13 je naznačeno, jak by mohla vypadat implementace Clean architecture v ASP .NET Core projektu. [39]



Obrázek 13 Clean architecture v ASP .NET Core přejato z [39]



## 2 ZABEZPEČENÍ WEBOVÝCH APLIKACÍ

Jelikož jsou webové aplikace nejčastěji nasazovány veřejně na internet, měl by se na zabezpečení klást obzvlášť důraz. V této kapitole si přiblížíme, jakým způsobem bude v aplikaci řešeno celkové zabezpečení ASP .NET Core aplikace. Pro zabezpečení jsem zvolil Duende IdentityServer.

### 2.1 Duende IdentityServer

Identity server je autentizační server, který implementuje standardy jako OpenID Connect a OAuth 2.0. Identity server je navržen tak aby, poskytoval autentizaci pro všechny druhy aplikací, ať už jsou to webové, nativní, mobilní nebo prosté API. Identity server lze použít pro implementaci jednotného přihlášení Single Sign-On. [32]

#### 2.1.1 JWT

JSON Web Token (JWT) je otevřený standard (RFC 7519), který definuje samostatný a kompaktní způsob pro bezpečný přenos informací mezi stranami jako JSON objekt. Tyto informace lze ověřit a důvěřovat jim, jelikož jsou digitálně podepsány. JWT může být podepsán pomocí páru veřejného a soukromého klíče (RSA, ECDSA), nebo pomocí soukromého klíče algoritmem HMAC.

JWT se skládá ze 3 částí. [33]

- Header
- Payload
- Verify Signature

Na následujícím obrázku 14 je naznačen příklad JWT.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEyMzQ1Njc4OTAiLCJ1c2VybmFtZSI6IjE1VzZXJ0YW11IiwiaWF0IjoiJ0cnVlIiwiaWF0IjoiIjoxNTE2MjM5MDIyfQ.kG_Ph3pXW10ixtZnm07vWcI2QLzMJCzUgrqcX1e3wo
```

Obrázek 14 Příklad JWT

### 2.1.2 OAuth 2.0

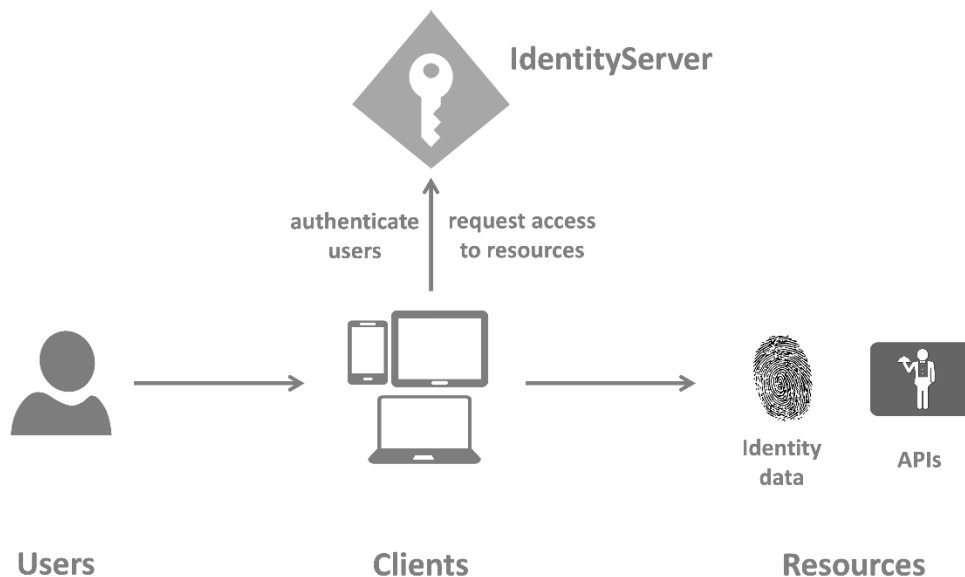
OAuth 2.0, což je zkratka pro Open Authorization, je standard navržený tak, aby umožnil webu nebo aplikaci přístup ke zdrojům hostovanými jinými webovými aplikacemi jménem uživatele. V roce 2012 nahradil OAuth 1.0 a je nyní de facto průmyslovým standardem pro online autorizaci. OAuth 2.0 poskytuje přístup nebo omezuje akce klientské aplikace, aniž by uživatel musel s danou aplikací sdílet své přihlašovací údaje. [34]

### 2.1.3 OpenID Connect

OpenID Connect 1.0 je jednoduchá vrstva identity nad protokolem OAuth 2.0. Umožňuje klientům ověřit identitu koncového uživatele na základě autentizace provedené autorizačním serverem a rovněž získat základní profilové informace o koncovém uživateli interoperabilním způsobem podobným REST. [35]

### 2.1.4 Duende Identity server

Princip zabezpečení aplikace je naznačen na následujícím obrázku.



Obrázek 15 Komunikace mezi klientem, IdentityServerem a API přejato z [36]

Popišme si nyní tento obrázek. User je člověk, který používá danou aplikaci.

Klient je nějaká aplikace, která žádá IdentityServer o přístupový token, který používá dále k autentizaci, či ke přístupu pomocí tohoto tokenu. Takovýto klient musí být první registrován na straně IdentityServeru, aby mohl klientovi přidělovat přístupové tokeny.

Zdroje jsou nějaká data, která se snažíme pomocí IdentityServeru zabezpečit.

Identity token je token, který je výstupem autentizačního procesu. Tento token obsahuje minimální potřebné informace, které nám umožní jednoznačně identifikovat uživatele.

Přístupový token nám umožňuje přistupovat k zabezpečeným API nebo k zabezpečeným zdrojům. Klient si tento přístupový token, např. JWT vyžádá u IdentityServeru, který zároveň i ověřuje platnost tohoto tokenu. Klient poté tento přístupový token přikládá ke každému požadavku při přístupu k API. Přístupový token obsahuje informace jak o uživateli, tak i o klientovi a určuje k jakým zdrojům má uživatel oprávnění. [36]

## **II. PRAKTICKÁ ČÁST**

### 3 NÁVRH APLIKACE

Nyní se dostáváme k samotnému vývoji aplikace. Před začátkem návrhu softwarové aplikace je vždy vhodné se napřed zamyslet nad tím, co chceme, aby aplikace vlastně uměla. Aplikace by měla umožňovat jednoduchou správu událostí. V aplikaci je také několik úrovní přístupu administrátor, moderátor, uživatel.

#### 3.1 Funkční požadavky

Přibližme si nyní jak by mohly vypadat funkční požadavky. Na následujícím obrázku jsou tyto funkční požadavky naznačeny.

Funkční požadavky uživatele	Funkční požadavky moderátora
R01: Aplikace musí umožnit registraci uživatele	R05: Aplikace musí umožnit přidání události
R02: Aplikace musí umožnit přihlášení uživatele	R06: Aplikace musí umožnit smazání události
R03: Aplikace musí umožnit odhlášení uživatele	
R04: Aplikace musí umožnit autorizaci uživatelů	
	Funkční požadavky administrátora
	R07: Aplikace musí umožnit přidělení práv moderátora
	R08: Aplikace musí umožnit odebrání moderátora

Obrázek 16 Funkční požadavky aplikace

R01: Všichni uživatelé, kteří nemají účet se mohou na webové stránce registrovat.

R02: Všichni uživatelé, kteří nejsou přihlášení se mohou na webové stránce přihlásit.

R03: Všichni uživatelé, kteří jsou přihlášení se mohou na webové stránce odhlásit.

R04: Webová aplikace umožňuje uživatelům přístup podle uživatelských rolí. Role v aplikaci jsou: uživatel, administrátor, moderátor.

R05: Webová aplikace umožňuje uživatelům s právy moderátora přidávat nové události.

R06: Webová aplikace umožňuje uživatelům s právy moderátora mazat již vytvořené události.

R07: Webová aplikace umožňuje uživatelům s právy administrátora přidělovat jiným uživatelům práva moderátora.

R08: Webová aplikace umožňuje uživatelům s právy administrátora odebrat moderátorům jejich práva.

### 3.2 Nefunkční požadavky

V této kapitole je nastíněno, jak by mohly vypadat nefunkční požadavky vytvořené aplikace. Tyto požadavky jsou nastíněny na následujícím obrázku.

Nefunkční požadavky
R09: Aplikace musí umožnit ukládání hesel v nečitelné podobě
R10: Aplikace musí autorizovat jednotlivé uživatele
R11: Aplikace musí udržovat vytvořené události

Obrázek 17 Nefunkční požadavky aplikace

R09: Webová aplikace musí ukládat hesla v nečitelné formě do databáze.

R10: Webová aplikace musí obsahovat autorizaci. Sekce pro přidání moderátorů je viditelná a přístupná pouze pro administrátora. Sekce pro správu eventů je pro dostupná administrátorovi i moderátorovi. Vytvořené události může uživatel vidět i bez přihlášení.

R11: Webová aplikace musí ukládat vytvořené události do databáze.

### 3.3 Scénáře použití

V této části práce si představíme pár základních scénářů použití aplikace.

#### Scénář : Vytvoření události

Tento scénář popisuje případ, kdy moderátor vytváří novou událost.

Tabulka 1 Scénář vytvoření nové události

<b>Jméno scénáře</b>	Vytvoření nové události
<b>Zúčastnění aktéři</b>	Nepřihlášený uživatel
<b>Hlavní scénář</b>	<ol style="list-style-type: none"> <li>1. Use case začíná, když uživatel přejde do záložky manage</li> <li>2. Uživatel se přihlásí</li> <li>3. Uživatel vyplní formulář pro vytvoření události</li> <li>4. Aplikace ověří zadaná data</li> <li>5. Aplikace vytvoří událost v databázi</li> </ol>
<b>Vedlejší scénář</b>	<ol style="list-style-type: none"> <li>2a. Uživatel zadal špatné přihlašovací údaje</li> <li>4a. Uživatel špatně vyplnil údaj ve formuláři</li> </ol>

**Alternativní scénář: Vytvoření události – uživatel zadal špatné přihlašovací údaje**

Při tomto alternativní scénáři je popsán případ, kdy uživatel zadá nesprávné přihlašovací údaje.

Tabulka 2 Alternativní scénář – Uživatel zadal špatné přihlašovací údaje

Jméno scénáře	Vytvoření nové události – uživatel zadal špatné přihlašovací údaje
Alternativní scénář	1. Systém zobrazí uživateli chybovou hlášku o neúspěšném přihlášení

**Alternativní scénář: Vytvoření události – uživatel špatně vyplnil údaj ve formuláři**

Při tomto alternativní scénáři je popsán případ, kdy uživatel zadá nesprávné přihlašovací údaje.

Tabulka 3 Alternativní scénář – Uživatel špatně vyplnil údaj ve formuláři

Jméno scénáře	Vytvoření nové události – špatně vyplnil údaj ve formuláři
Alternativní scénář	1. Systém zobrazí uživateli chybovou hlášku o nesprávném vyplnění formuláře.

**Scénář : Zobrazení administrátorské sekce**

Tento scénář popisuje situaci, kdy se uživatel pokusí přistoupit do administrátorské sekce.

Tabulka 4 Scénář zobrazení administrátorské sekce

Jméno scénáře	Zobrazení administrátorské sekce
Zúčastnění aktéři	Nepřihlášený uživatel
Hlavní scénář	1. Use case začíná, když uživatel přejde do sekce admin 2. Uživatel se přihlásí 3. Aplikace zobrazí stránku administrátorské sekce
Vedlejší scénář	2a. Uživatel zadal špatné přihlašovací údaje 2b. Uživatel nemá administrátorská práva

**Alternativní scénář: Zobrazení administrátorské sekce – uživatel zadal špatné přihlašovací údaje**

Při tomto alternativní scénáři je popsán případ, kdy uživatel zadá nesprávné přihlašovací údaje.

Tabulka 5 Alternativní scénář – Uživatel zadal špatné přihlašovací údaje

Jméno scénáře	Zobrazení administrátorské sekce – uživatel zadal špatné přihlašovací údaje
Alternativní scénář	1. Systém zobrazí uživateli chybovou hlášku o neúspěšném přihlášení

**Alternativní scénář: Zobrazení administrátorské sekce – uživatel nemá administrátorská práva**

Tento alternativní scénář popisuje situaci, kdy se uživatel úspěšně přihlásil, ale nemá administrátorská práva.



Tabulka 6 Alternativní scénář – Uživatel nemá administrátorská práva

Jméno scénáře	Zobrazení administrátorské sekce – nemá administrátorská práva
Alternativní scénář	1. Systém zobrazí uživateli výchozí stránku pro neautorizované uživatele

**Scénář : Registrace uživatele**

Tento scénář popisuje situaci při registraci nového uživatele do systému. Uživatel při registraci musí vyplnit všechny potřebné údaje a heslo musí mít odpovídající složitost.

Tabulka 7 Scénář registrace uživatele

Jméno scénáře	Registrace uživatele
Zúčastnění aktéři	Neregistrovaný uživatel
Hlavní scénář	1. Use case začíná, když uživatel klikne na tlačítko Registration 2. Uživatel vyplní formulář pro registraci 3. Systém zkontroluje všechny vyplněné údaje 4. Systém vytvoří nového uživatele
Vedlejší scénář	2a. Uživatel zadal špatné registrační údaje 2b. Registrační údaje jsou již využívány

**Alternativní scénář: Registrace uživatele – uživatel zadal špatné registrační údaje**

Tento scénář popisuje situaci, kdy uživatel zadá do registračního formuláře špatné přihlašovací údaje. Uživatel může například špatně zadat e-mail nebo může mít slabé heslo.

Tabulka 8 Alternativní scénář – Uživatel zadal špatné registrační údaje

Jméno scénáře	Registrace uživatele – uživatel zadal špatné registrační údaje
Alternativní scénář	1. Systém zobrazí uživateli chybovou hlášku o neúspěšném registraci

**Alternativní scénář: Registrace uživatele – registrační údaje jsou již využívány**

Tento scénář popisuje situaci, kdy uživatel zadá do registračního formuláře špatné přihlašovací údaje. Uživatel může například špatně zadat e-mail nebo může mít slabé heslo.

Tabulka 9 Alternativní scénář – Registrační údaje jsou již využívány

Jméno scénáře	Registrace uživatele – registrační údaje jsou již využívány
Alternativní scénář	1. Systém zobrazí uživateli chybovou hlášku o neúspěšném registraci

**Scénář : Přihlášení uživatele**

Tento scénář popisuje situaci při přihlášení uživatele do systému.

Tabulka 10 Scénář přihlášení uživatele

Jméno scénáře	Přihlášení uživatele
Zúčastnění aktéři	Nepřihlášený uživatel
Hlavní scénář	<ol style="list-style-type: none"> <li>1. Use case začíná, když uživatel klikne na tlačítko login</li> <li>2. Uživatel vyplní formulář pro přihlášení</li> <li>3. Systém zkontroluje všechny vyplněné údaje</li> <li>4. Systém přihlásí uživatele do aplikace</li> </ol>
Vedlejší scénář	2a. Uživatel zadal špatné přihlašovací údaje

**Alternativní scénář: Přihlášení uživatele – uživatel zadal špatné přihlašovací údaje**

Tento scénář popisuje situaci, kdy uživatel zadá špatné přihlašovací údaje.

Tabulka 11 Alternativní scénář – Uživatel zadal špatné přihlašovací údaje

Jméno scénáře	Přihlášení uživatele – uživatel zadal špatné přihlašovací údaje
Alternativní scénář	1. Systém zobrazí uživateli chybovou hlášku o neúspěšném přihlášení

## 4 KLÍČOVÉ ČÁSTI APLIKACE

V následující části práce jsou popsány klíčové části webové aplikace. Aplikace je rozdělena do 4 samostatných projektů, spouští se každý zvlášť, ale všechny spolu komunikují. První z projektů je projekt kde se nachází React aplikace. Dalším projektem je projekt, který se stará o autentizaci a autorizaci uživatelů. Předposlední projekt je backend webové aplikace neboli API. Oba tyto projekty jsou vytvořeny v ASP .NET Core. Posledním z projektů je frontendová část aplikace, která je ve frameworku Blazor WebAssembly.

Klíčové prvky této webové aplikace jsou tedy autentizace, autorizace, ukládání dat do databáze, uživatelské rozhraní a v neposlední řadě validace uživatelských vstupů. V následujících kapitolách si tyto klíčové části blíže popíšeme.

### 4.1 Aplikace React

V této části práce se budeme věnovat React projektu. Tento projekt slouží primárně k tomu abychom v něm spouštěli Blazor. Jakým způsobem se takový projekt vytváří je popsáno v teoretické části.

#### 4.1.1 Implementace frameworku Blazor

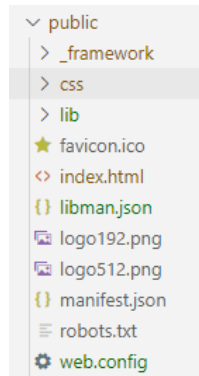
K tomu, abychom mohli spustit Blazor WebAssembly v Reactu, bude potřeba publikovat Blazor projekt. Jakmile publikujeme projekt Blazoru, v adresáři projektu se vytvoří potřebně souboru pro spuštění Blazoru. Při publikaci vznikne jeden z mnoha adresářů, adresář `_framework`. Relativní cesta k tomuto adresáři, která začíná ze složky Blazor projektu, je naznačena na následujícím obrázku 18.

```
bin > Release > net6.0 > browser-wasm > publish > wwwroot > _framework
```

Obrázek 18 Cesta ke složce `_framework`

Tento adresář je nutný, aby se nacházel v React projektu. Samozřejmě, že se tahle složka může nacházet nasazená i někde jinde na serveru, ale je poté třeba přesměrovávat požadavky. Já jsem zvolil umístění do stejného projektu.

Obsah složky `_framework` zkopírujeme do složky `public` v React projektu. Na následujícím obrázku 19 je vyobrazena. Jak by mohla vypadat složka `public`.



Obrázek 19 Složka public v projektu React

Nyní máme v projektu všechny potřebné binární soubory z Blazor WebAssembly. Přesuneme se nyní do souboru `index.html`. Do tohoto souboru přidáme následující řádky kódu.

```
<script defer src="https://localhost:44001/_framework/blazor.webassembly.js">
</script>
```

```
<script defer src="https://localhost:44001/_content/Microsoft.AspNetCore.Components.WebAssembly.Authentication/AuthenticationService.js">
</script>
```

První načítání JavaScriptového souboru je na adresu `https://localhost:44001`. Na této adrese se nachází nasazená aplikace Blazor WebAssembly. Soubor `blazor.webassembly.js` slouží ke stažení všech .NET runtime, stažení zavilostí a samotné aplikace. [37]

Následující načítání je načítání souboru `AuthenticationService.js`. Tento JavaScriptový soubor definuje autentizační službu. Tato autentizační služba zpracovává nízko úroňové podrobnosti protokolu OIDC. Aplikace poté využívá metody této služby k poskytování autentizace. [38]

Nyní je aplikace připravena na spuštění pomocí příkazu „`yarn start`“, který napíšeme v terminálu VS Code. Vývojové prostředí by nám mělo vypsát na jakém portu běží aplikace. V mém případě projekt Reactu běží na `http://localhost:3000`.

Pokud jsou všechny ostatní projekty správně nakonfigurovány a spuštěny, aplikace by měla bez problému fungovat. Otevřeme si vývojářské nástroje v prohlížeči a v záložce network si popíšeme, jakým způsobem se Blazor WebAssembly načítá. Na následujícím obrázku 20 je část prohlížečem načítaných souborů.

Name	Status	Type
ws		websocket
localhost	200	document
bundle.js	200	script
blazor.webassembly.js	200	script
AuthenticationService.js	200	script
bootstrap.min.css	200	stylesheet
bootstrap-dark.min.css	200	stylesheet
toggle-bootstrap.min.css	200	stylesheet
toggle-bootstrap-dark.min.css	200	stylesheet
app.css	200	stylesheet
all.css	200	stylesheet
blazor.boot.json	304	fetch
openid-configuration	200	xhr

Obrázek 20 Načítané soubory po spuštění aplikace

Prvním zajímavým souborem je soubor `Blazor.webassembly.js`. Po bližším prozkoumání můžeme zjistit zdroj tohoto souboru. Na následujícím obrázku 21 je naznačeno odkud se tento soubor stáhnul.

```
Request URL: https://localhost:44001/_framework/blazor.webassembly.js
Request Method: GET
Status Code: 200
```

Obrázek 21 Detail souboru `blazor.webassembly.js`

Není překvapení, že zdroj odkud se soubor stáhnul je `https://localhost:44001`, protože adresu jsme zadali v souboru `index.html`. Jak jsem již zmínil tento soubor slouží ke stažení dat aplikace.

Dalším ze zajímavých souborů je `blazor.boot.json` a i na něj se podíváme zblízka. Na následujícím obrázku 22 můžeme vidět část jeho podrobností.

```
Request URL: http://localhost:3000/_framework/blazor.boot.json
Request Method: GET
Status Code: 304 OK
```

Obrázek 22 Detail souboru `blazor.boot.json`

Zde vidíme, že zdroj odkud se tento soubor stahuje je jiný než u `blazor.webassembly.js`. Adresa `http://localhost:3000` je adresa kde je nasazený React projekt a složka `_framework` se zde nachází. Tímto způsobem prohlížeč stáhne všechny ostatní závislosti Blazor aplikace.

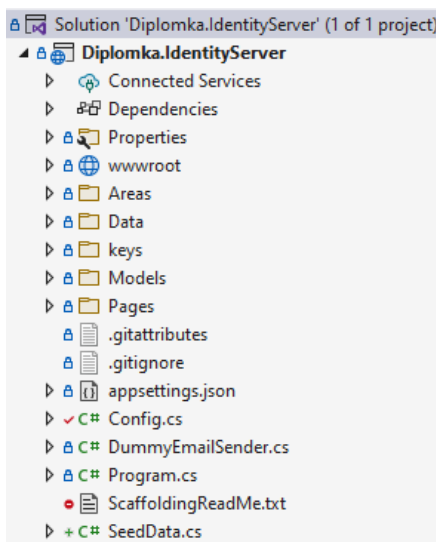
Toto je jedna z možností, jakým můžeme Blazor WebAssembly implementovat do projektu, který je vytvořený pomocí Reactu.

## 4.2 Aplikace IdentityServer

V této části práce se budeme věnovat IdentityServer projektu, který je vytvořený podle šablony Duende IdentityServer.

Tento projekt slouží pro celkovou správu uživatelů jako tzv Identity Provider. Identity Provider slouží k vytváření, uchovávání a spravování informací o identitě uživatele.

Projekt je naznačen na následujícím obrázku 23.



Obrázek 23 Struktura projektu IdentityServer

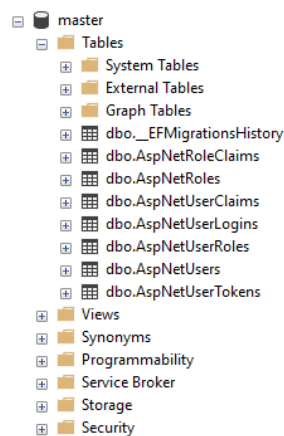
Na obrázku vidíme ASP .NET Core projekt, který se skládá z několika adresářů a tříd.

Aby mohla aplikace ukládat své registrované uživatele, musí mít přístup do databáze. Projekt pro tuto funkcionalitu využívá Microsoft Entity Framework. Microsoft Entity Framework využívá pro práci s databází DbContext, který se společně s migracemi nachází ve složce data. Pro připojení do databáze je nutné v souboru appsettings.json nastavit ConnectionString. Na následujícím obrázku 24 je zobrazen příklad ConnectionStringu.

```
{  
  "ConnectionStrings": {  
    "DefaultConnection": "Server=(localdb)\\PETRENAKIF;Initial_Catalog=master;Trusted_Connection=True;MultipleActiveResultSets=true"  
  },  
}
```

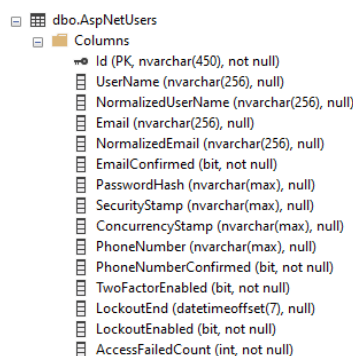
Obrázek 24 Connection string

Jakmile je správně nastaven ConnectionString je potřeba vytvořit inicializační migraci. Tuto migraci vytvoříme pomocí příkazu „Add-Migration ‘NazevMigrace’“, který zadáme do konzole správce balíčků. Tento příkaz vytvoří schéma databáze podle modelu. Modelem pro databázi je třída ApplicationUser, která nemá žádné své vlastnosti, ale dědí je od třídy IdentityUser, který je součástí balíčku Microsoft.AspNetCore.Identity. Nyní už stačí jen příkaz do konzole „update-database“. Výsledkem tohoto příkazu může být databáze, která je naznačena na následujícím obrázku 25.



Obrázek 25 Struktura databáze pro uživatele

Microsoft Entity Framework, za pomoci modelu IdentityUser vytvořil do databáze několik tabulek. Základní tabulka je tabulka pro uchovávání uživatelů AspNetUsers. Tato tabulka se skládá z velkého množství sloupců. Na následujícím obrázku 26 jsou takové sloupce zobrazeny.



Obrázek 26 Struktura tabulky AspNetUsers

Pro testovací účely se v projektu nachází třída SeedData.cs, která po spuštění přidá do databáze 3 uživatele s různými právy, jinak je zde standardní způsob přidání uživatelů pomocí registrace z uživatelského rozhraní.

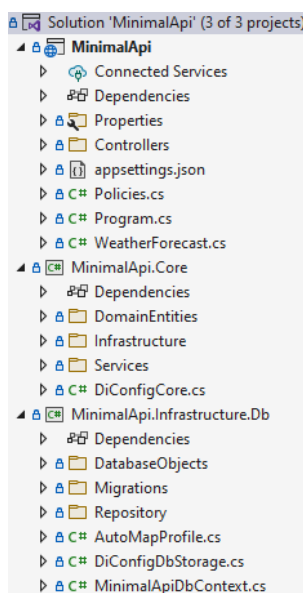
Pro vlastní upravení stránek pro registraci, přihlášení, změnu hesla a mnoho dalšího zde existuje funkce Scaffolding.

Další důležitá třída v projektu je třída Config.cs. V této třídě, jak už název napovídá, se nachází konfigurace. V této třídě se nastavují potřebné parametry pro komunikaci mezi klientem a IdentityServerem.

### 4.3 Aplikace MinimalApi

Tento projekt s názvem MinimalApi slouží jako backend celé aplikace.

Struktura celého projektu je založena na architektonickém vzoru Clean architecture. Na následujícím obrázku 27 je tato struktura vyobrazena.



Obrázek 27 Struktura projektu MinimalApi

Na obrázku 27 můžeme vidět, že se projekt skládá z několika různých menších projektů. Toto rozdělení pomáhá lépe se v projektu orientovat, lepší udržitelnost kódu a lepší testovatelnost.

Jako první se nachází projekt této webové aplikace. Nachází se zde např. kontroléry, nastavení pro aplikaci nebo třída Program.cs.

Jako druhý projekt je projekt pro jádro aplikace. Zde se nachází 3 adresáře. V adresáři DomainEntities se nachází modely, se kterými aplikace pracuje, např. model pro událost. Dalším adresářem je adresář Infrastructure, kde se nachází rozhraní pro vrstvu infrastructure.



V implementaci tohoto rozhraní se nachází metody, které slouží pro práci s databázovými repositáři. Jako poslední složka se zde nachází složka *Service*, kde se nachází implementaci tzv. bussines logiky pro danou aplikaci. Jako příklad si můžeme uvést službu pro události, která má za úkol zpracovávat požadavky z uživatelského rozhraní, které přijdou na kontrolér.

Posledním projektem je projekt pro práci s databází. Tento projekt opět využívá pro práci s databází Microsoft Entity Framework. V tomto projektu se taky nachází 3 adresáře. Prvním z nich je adresář *DatabaseObjects*, který obsahuje modely pro práci s databází. Další složkou je složka pro migrace, kterou Microsoft Entity Framework využívá. Jako poslední je složka s repositáři. V této složce se nachází třídy, kdy většinou 1 třída znamená 1 repositář a tedy 1 tabulku v databázi, např tabulka pro události. Dále tento projekt obsahuje AutoMapper, který implicitně přetypovává databázové objekty na objekty používané jádrem.

### 4.3.1 Zabezpečení endpointů

K autorizaci endpointů na backendu se využívá atribut *Authorize*. Tento atribut můžeme přiřadit buď celé třídě kontroléru nebo jen metodám tohoto kontroléru. V parametru konstruktoru atributu *Authorize* můžeme předávat pravidla, která určují, kdo má k daným zdrojům přístup.

Pro stanovení těchto pravidel je v projektu třída *Policies.cs*. Na následujícím úryvku kódu je tato třída zobrazena.

```
public static class Policies
{
    public const string CanHaveAdmin = "CanHaveAdmin";
    public static AuthorizationPolicy CanHaveAdminPolicy()
    {
        return new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .RequireClaim("role", "admin")
            .RequireClaim("role", "mod")
            .Build();
    }
}
```

Tato statická třída má ve svém těle vlastnost *CanHaveAdmin*, do které je přiřazena hodnota stejného názvu. Dále zde následuje metoda *CanHaveAdminPolicy*, která podle claimů v přichozím JWT zařazuje uživatele do různých skupin, které jsou definovány jako vlastnosti této třídy. Aby tato metoda přiřadila správná práva správným skupinám musíme navíc spárovat tuto metodu s danou vlastností. Toto spárování se provádí ve třídě *Program.cs* a je naznačeno na následujícím kódu.

```
builder.Services.AddAuthorizationCore(authorizationOptions =>
{
    authorizationOptions.AddPolicy(
        Policies.CanHaveAdmin,
        Policies.CanHaveAdminPolicy()
    );
});
```

Nyní jsou pro tuto aplikaci definována pravidla pro přístup ke zdrojům a můžeme je použít. Takové použití je naznačeno na metodě kontroleru a je naznačeno na následujícím zdrojovém kódu.

```
[HttpPost]
[Route(nameof(Create))]
[Authorize(Policy = Policies.CanHaveAdmin)]
public async Task Create(CreateEventRequest createEventRequest)
{
    _logger.LogInformation("POST request to 'api/event/create' ");
    await _eventService.AddEventAsync(createEventRequest);
}
```

Výhodou takovéto implementace je, že máme správu skupin pro autorizaci v jedné třídě, a navíc můžeme do 1 skupiny zařadit více uživatelů s různými claimy.

## 4.4 Aplikace Blazor WebAssembly

Tento projekt s názvem DPFrontend slouží pro správu uživatelského rozhraní. Jelikož hlavní součástí uživatelských rozhraní jsou komponenty budeme se jim nyní věnovat.

### 4.4.1 Atomic design

Atomic design je způsob vytváření uživatelských rozhraní. Při vytváření stránek pro aplikaci rozdělujeme komponenty do menších znovu použitelných komponent. Základní struktura Atomic designu je následovná.

- Atomy
- Molekuly
- Organismy
- Šablony
- Stránky

Atomy jsou základní HTML elementy jako `label`, `input`, nebo `button`.

Molekuly se skládají z více atomů spojených dohromady. Jako příklad si můžeme uvést komponentu pro vyhledávání. Takováto komponenta se bude pravděpodobně skládat

z nějakého vstupního pole, toto pole bude mít nějaký svůj nadpis, a nakonec tlačítko na odeslání požadavku.

Organismy jsou už komplexnější komponenty, které se skládají z více molekul či atomů. Jako příklad si můžeme uvést prázdnou šablonu stránky. Tato šablona bude mít svůj nadpis, menu, název atd.

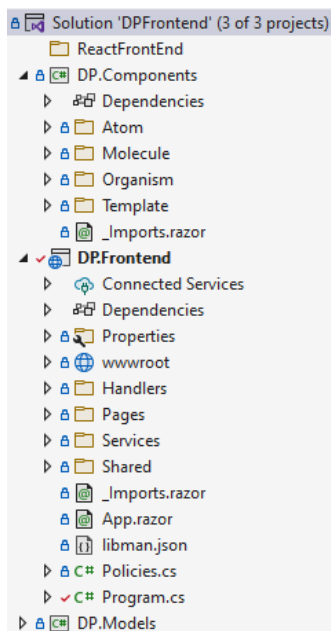
Šablony jsou již více či méně celé stránky poskládané z organismů molekul nebo atomů. Tyto šablony můžeme pak volat v kódu s různými vstupními parametry a vykreslovat je na různých stránkách.

Stránky jsou již specifické instance šablon. [40]

U Atomic designu je občas těžké rozhodnout, jestli daná komponenta patří do molekuly, či je to ještě atom, nebo zda patří už do organismu. Každý vývojář to bude nejspíš vidět trochu jinak, takže víceméně je to o stanovení konvencí mezi sebou, které by se dále měly dodržovat.

#### 4.4.2 Struktura projektu

Celý projekt je členěn do několika menších projektů, které jsou naznačeny na následujícím obrázku.



Obrázek 28 Struktura projektu Blazor WebAssembly

Jako první dílčí projekt je projekt s komponenty. Tento projekt je rozčleněn pomocí několika složek do Atomic designu.

Dalším projekt je projekt pro celkovou Blazor WebAssembly aplikaci. Tento projekt je rozdělen také do několika složek. Důležitou složkou je například složka Services, kde se nacházejí služby pro komunikaci s backendem aplikace. Další složkou je složka Pages. V této složce se nachází už konkrétní stránky aplikace. Na následujícím zdrojovém kódu je naznačena konkrétní stránka s názvem Admin.razor.

```
@page "/admin"
@using DP.Frontend.Services
@using DP.Models.Event

[Authorize(Policy = Policies.Admin)]

<AdminPage CreateEvent="(createEventModel) => CreateEvent(createEventModel)">
</AdminPage>

@code
{
    [Inject]
    IEventService EventService { get; set; }

    private async Task CreateEvent(EventModel createEventModel)
    {
        Console.WriteLine("/admin CreateEvent(EventModel createEventModel)");
        await EventService.CreateNewEventAsync(createEventModel);
    }
}
```

Jako první řádek na tomto úryvku kódu se nachází cesta ke stránce. Poté následují direktivy a atribut stránky Authorize. Tento atribut zajišťuje autorizaci stránky a funguje stejně jako na backendu. Po atributu následuje HTML sekce této razor stránky. Jelikož je v projektu použitý Atomic design, stránka má pouze 1 komponentu, a to je šablona dané stránky.

Za sekci pro HTML se nachází sekce C# kódu stránky. Jako první zde injektujeme službu, která posílá požadavek na backend aplikace. Tato metoda je definována jako asynchronní Task s názvem CreateEvent, který je do komponenty posílán jako parametr.

#### 4.4.3 Zabezpečení stránek

V uživatelské rozhraní nejen že zabezpečujeme jednotlivé endpointy, ale dokonce ani nezobrazujeme stránky kam nemá uživatel přístup. Typický příklad je navigační menu nebo přihlášení a registrace. V navigačním menu obvykle administrátor vidí nějakou položku navíc, např. položka administrace. Uveďme si nyní příklad. Pokud na stránce nejsme přihlášení obvykle v pravém horním rohu máme odkazy na přihlášení či registraci. Pokud jsme

přihlášení chceme vidět tlačítko na odhlášení. Abychom dosáhli takové funkcionality Blazor WebAssembly používá element `AuthorizedView`. Dovnitř tohoto elementu umístíme další 2 elementy. První z elementů je element `Authorized`, a jak už z názvu napovídá, v těle tohoto elementu se nachází obsah, když je uživatel přihlášen. Druhým elementem je element `NotAuthorized`, který má opačnou funkcionality.

Na následujícím zdrojovém kódu je příklad použití těchto elementů.

```
<AuthorizeView>
  <Authorized>
    <div class="d-flex justify-content-between">
      <button type="button" class="btn btn-dark btn-lg btn-link mx-1
d-inline-flex" @onclick="NavigateToProfile">
        <div class="d-inline-flex align-items-center">
          @context.User.Identity.Name
          <i class="fas fa-gear pl-2"></i>
        </div>
      </button>

      <button type="button" class="btn btn-dark btn-lg btn-link mx-1
d-inline-flex" @onclick="NavigateToLogout">
        <div class="d-inline-flex align-items-center">
          Logout
          <i class="fas fa-right-from-bracket pl-2"></i>
        </div>
      </button>
    </div>
  </Authorized>

  <NotAuthorized>
    <div class="d-flex justify-content-between">
      <button type="button" class="btn btn-dark btn-lg btn-link mx-1
d-inline-flex" @onclick="NavigateToLogin">Login</button>

      <button type="button" class="btn btn-dark btn-lg btn-link mx-1
d-inline-flex" @onclick="NavigateToRegister">Registration</button>
    </div>
  </NotAuthorized>
</AuthorizeView>
```

Tyto razor elementy nám pouze zobrazí nějaké jiné elementy, na základě toho, jestli jsme přihlášení nebo ne, což nestačí. Pokud bychom znali adresu, kam tyto tlačítka či odkazy směřují mohli bychom je zadat do adresního řádku a na endpoint bychom se bez problému dostali.

Pro autorizace těchto endpointů používáme stejně jako v ASP .NET Core atribut `Authorize`. Stejně jako na backendu je zde třída `Policies.cs`, která má stejnou funkcionalitu. Příklad použití atributu `Authorize` v razor komponentě je naznačen na následujícím řádku zdrojového kódu.

```
@attribute [Authorize(Policy = Policies.Admin)]
```

## 5 ZABEZPEČENÍ KOMUNIKACE MEZI KLIENTEM A SERVEREM

Návrh celkového zabezpečení byla jednou z klíčových částí aplikace. Jelikož se Blazor WebAssembly spouští z Reactu musí být Blazor standalone. Při tomto způsobu implementace nemůžeme požitím zabezpečení pomocí same site cookies.

Nejlepším a v podstatě jediným způsobem, jak navrhnu takové zabezpečení je pomocí JWT. ASP .NET Core a Blazor WebAssembly již mají implementovanou funkcionalitu, jak s takovým tokenem zacházet, stačí jen dodržet potřebné kroky, aby autentizace a autorizace fungovala.

Celý tento proces je popsán v teoretické části v kapitole 2 a je naznačen na obrázku 13. Nyní se prakticky podíváme, co konkrétně pro tuhle funkcionalitu musíme implementovat.

### 5.1 Konfigurace klienta

Jako první se budeme věnovat nastavení na straně klienta, což je v tomto případě Blazor WebAssembly aplikace. Začneme tím, že si nainstalujeme balíček Microsoft.AspNetCore.Components.WebAssembly.Authentication. Jako další je zapotřebí přidat JavaScript autentizační službu na stránku index.html. Tento skript je již zmíněn v popisu React projektu, kde se přidává také. Aplikace bude poté volat autentizační metody definované v tomto skriptu.

V projektu Blazoru musíme dále přidat novou komponentu s názvem Authentication.razor. Tato komponenta definuje cesty, které obsluhují různá stádia autentizace, např login nebo logout. Obsah této komponenty je nastíněn na následujícím úryvku zdrojového kódu.

```
@page "/authentication/{action}"

@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action">
  <LoggingIn>
    <span class="h1">We're logging you in, please wait...</span>
  </LoggingIn>
  <CompletingLoggingIn>
    <span class="h1">Logged in, redirecting back...</span>
  </CompletingLoggingIn>
</RemoteAuthenticatorView>

@code
{
  [Parameter] public string Action { get; set; }
}
```

Po přidání skriptu a komponenty, zbývá pouze nakonfigurovat Blazor WebAssembly pro použití OIDC Autentizace. Přesuneme se do třídy Program.cs a do kolekce služeb přidáme novou službu s názvem AddOidcAuthentication. Přidání a nastavení takové služby je nastíněno na následujícím zdrojovém kódu.

```
builder.Services.AddOidcAuthentication(options =>
{
    options.ProviderOptions.Authority = "https://localhost:44100/";
    options.ProviderOptions.ClientId = "BlazorWasm";
    options.ProviderOptions.RedirectUri =
        "http://localhost:3000/authentication/login-callback";
    options.ProviderOptions.PostLogoutRedirectUri =
        "https://localhost:3000/authentication/login-callback";
    options.ProviderOptions.DefaultScopes.Add("openid");
    options.ProviderOptions.DefaultScopes.Add("profile");
    options.ProviderOptions.DefaultScopes.Add("email");
    options.ProviderOptions.DefaultScopes.Add("dpapi");
    options.ProviderOptions.DefaultScopes.Add("role");
    options.ProviderOptions.ResponseType = "code";
});
```

V tomto úryvku kódu definujeme různé možnosti jako např, adresu IdentityServeru, jméno klienta, adresu uri pro přesměrování, scopy a typ odpovědi.

Těmito kroky by měl být klient připraven na OIDC autentizace za pomoci IdentityServeru. Podívejme se nyní jakým způsobem musíme nakonfigurovat IdentityServer.

## 5.2 Konfigurace IdentityServeru

Podívejme se nyní do souboru Config.cs. V této třídě se nachází konfigurace klientů, API a samotného IdentityServeru.

Na následujícím zdrojovém kódu je jako první nastíněna konfigurace pro klienta.

```
public static IEnumerable<Client> Clients =>
    new Client[]
    {
        new Client
        {
            ClientId = "BlazorWasm",
            ClientName = "Blazor standalone app",
            AllowedGrantTypes = GrantTypes.Code,
            RequireClientSecret = false,
            RequirePkce = true,
            RedirectUris =
                { "http://localhost:3000/authentication/login-callback" },
            PostLogoutRedirectUris =
                { "http://localhost:3000/authentication/logout-callback" },
            AllowedScopes = { "openid", "profile", "email", "dpapi", "role" },
            AllowedCorsOrigins = { "http://localhost:3000" }
        }
    };
```



Tato metoda s názvem `Clients` vrací kolekci všech klientů, kteří jsou pro tento `IdentityServer` nakonfigurováni. Zde je pouze 1 klient, a to `Blazor WebAssembly`. Všimněme si, že vlastnost `ClientID` má přiřazenou stejnou hodnotu jak na straně klienta, tak na straně `IdentityServeru`. Jednou z dalších vlastností je `AllowedScopes`, kde si můžeme opět všimnout, že všechny tyto scopy opět používá klient. Jako poslední vlastnost musíme správně nastavit `CORS`, kde nastavíme url nasazeného projektu.

Dále se v konfiguračním souboru nachází definice podporovaných zdrojů. Vysvětleme si to na následujícím zdrojovém kódu.

```
public static IEnumerable<IdentityResource> IdentityResources =>
    new IdentityResource[]
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile(),
        new IdentityResources.Email(),
        new IdentityResource("role", new [] { "role" })
    };
```

Podobně jako u nastavení klienta, máme zde metodu, která vrací kolekci `IdentityResource`. Takhle konfigurace znamená, že `IdentityServer` může poskytnout přístup k těmto informacím o uživateli.

Dále musíme nakonfigurovat API, na které bude klient posílat své požadavky. Konfigurace API je velmi podobná jako konfigurace klienta či `IdentityServeru`, opět zde máme metodu, která vrací kolekci `ApiResources`. Na následující ukázce zdrojového kódu je naznačena konfigurace pro API.

```
new ApiResource[]
{
    new ApiResource("dpapi", "Diplomka Api", new [] {"role"})
    {
        Scopes = { "dpapi" }
    }
};
```

V konstruktoru `ApiResources` předáme jméno pro API, zobrazované jméno a kolekci uživatelských claimů, které musí JWT obsahovat při dotazu na toto API. Dále pro toto `ApiResource` nastavíme jméno API scopu a poté jej i nakonfigurujeme obdobně jako vše ostatní. Konfigurace API scopu je naznačena na následujícím zdrojovém kódu.

```
public static IEnumerable<ApiScope> ApiScopes =>
    new ApiScope[]
    {
        new ApiScope("dpapi"),
    };
```

Všechny metody této statické třídy nyní využijeme ve třídě Program.cs, při přidávání služby s názvem AddIdentityServer. Na následujícím zdrojovém kódu je přidání naznačena konfigurace a přidání takové služby.

```
builder.Services.AddIdentityServer(option =>
    {
        option.Events.RaiseErrorEvents = true;
        option.Events.RaiseInformationEvents = true;
        option.Events.RaiseFailureEvents = true;
        option.Events.RaiseSuccessEvents = true;

        }).AddAspNetIdentity<ApplicationUser>()
        .AddInMemoryIdentityResources(Config.IdentityResources)
        .AddInMemoryApiResources(Config.ApiResources)
        .AddInMemoryApiScopes(Config.ApiScopes)
        .AddInMemoryClients(Config.Clients);
```

### 5.3 Konfigurace API

Jako poslední zbývá nakonfigurovat API. Konfigurace API je poněkud jednodušší než ostatních projektů. Jediné, co je potřeba, tak přidat do třídy Program.cs následující zdrojový kód.

```
builder.Services.AddAuthentication(IdentityServerAuthenticationDefaults.AuthenticationScheme)
    .AddIdentityServerAuthentication(
        options =>
        {
            options.Authority = "https://localhost:44100";
            options.ApiName = "dpapi";
        });
```

V tomto zdrojovém kódu přidáváme do kolekce služeb službu pro autentizaci. V parametru předáváme název schématu určený pro autorizaci, Bearer. Poté zavoláme metodu, kterou určíme autentizaci pomocí IdentityServeru a v možnost zadáme adresu serveru a název API.

## 6 DEMONSTRACE APLIKACE

V této kapitole se zaměříme na prezentace vytvořené aplikace ve frameworku Blazor WebAssembly, kdy je tato aplikace poté spuštěna v jiné aplikaci založené v Reactu.

### 6.1 Spuštění aplikace

Jak již bylo zmíněno celková aplikace se skládá ze 4 projektů. IdentityServer, API a Frontend jsou projekty ve vývojovém prostředí Visual Studio. Všechny projekty lze jednoduše spustit pomocí tohoto vývojového prostředí, přičemž pořadí nehraje žádnou roli. Aplikace je navržena tak, že při výpadku jedné z těchto částí to do jisté míry neovlivní ostatní. Samozřejmě pokud bude fungovat pouze IdentityServer a frontend, tak frontend spustíme, přihlásíme se, ale jelikož nefunguje API, tak nezískáme z něj žádná data.

Jakmile IdentityServer, API, a frontend aplikace běží, můžeme spustit React aplikaci.

### 6.2 Úvodní obrazovka

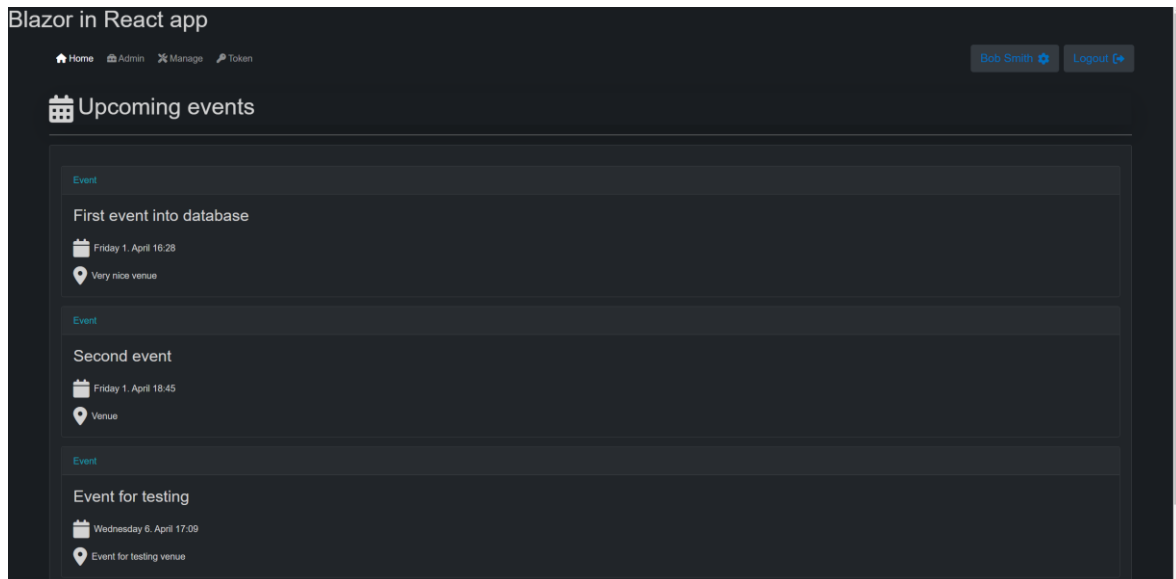
Po úspěšném spuštění a přihlášení se uživateli zobrazí domovská stránka aplikace. Informace o uživateli a tlačítko na odhlášení se klasicky nachází v pravém horním rohu. Pokud by uživatel nebyl přihlášen byly by zde tlačítka pro registraci či přihlášení.

V levé horní části obrazovky se nachází nadpis „Blazor in React App“. Tento nadpis je součástí React projektu a naznačuje že tahle aplikace běží v jiné aplikaci.

Pod tímto nadpisem se nachází menu aplikace, které slouží k navigaci mezi jednotlivými stránkami. Tyto položky menu jsou odlišné pro různé úrovně přístupu, které má daný uživatel.

V obsahu domovské stránky se zobrazují všechny již vytvořené události. Jednotlivá událost se skládá z názvu, místa události a času, kdy se událost koná.

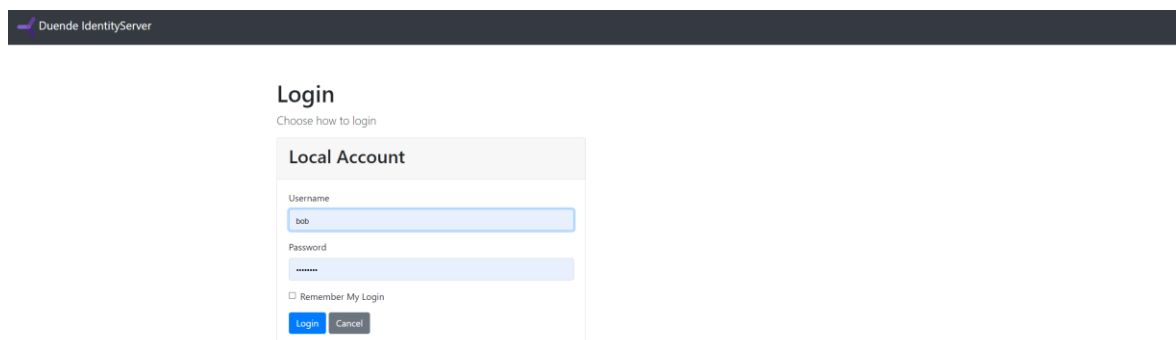
Na následujícím obrázku 29 je naznačeno, jak domovská stránka vypadá.



Obrázek 29 Domovská stránka aplikace

### 6.3 Přihlášení

Další stránkou je stránka pro přihlašování uživatelů. Uživatelé se zde přihlašují pomocí uživatelského jména a hesla. Dále se na stránce nachází checkbox pro zapamatování si přihlášeného uživatele. Na následujícím obrázku 30 je naznačena stránka pro přihlašování uživatelů.

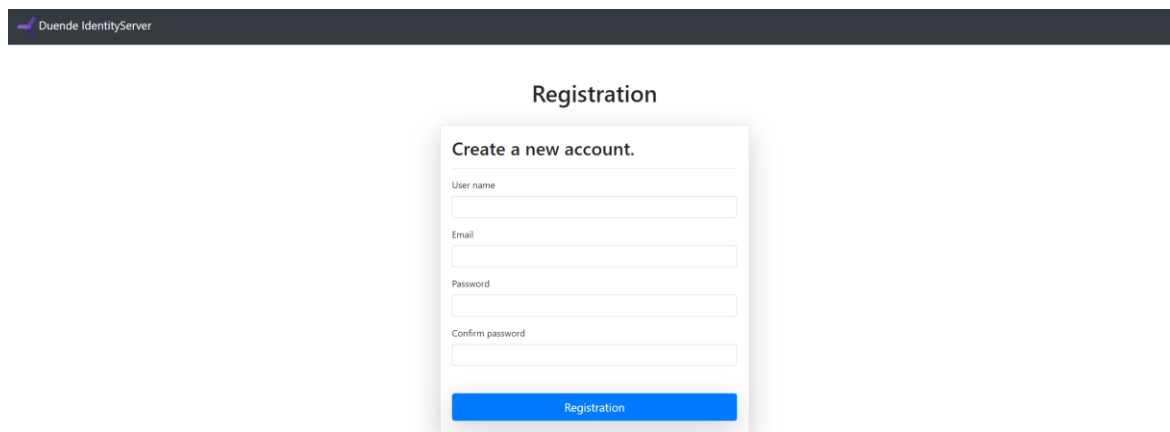


Obrázek 30 Stránka pro přihlášení do aplikace

### 6.4 Registrace

Následující stránka je stránka pro registraci uživatelů. Aby se mohl uživatel registrovat musí vyplnit všechny údaje v registračním formuláři. Uživatel zde vyplní své uživatelské jméno,

email a heslo. Heslo musí obsahovat velká a malá písmena, číslice, a navíc i speciální znak. Nakonec je po uživateli vyžadováno heslo ještě jednou. Takováto stránka pro registraci je naznačena na následujícím obrázku 31.

The image shows a registration form titled "Registration" with the subtitle "Create a new account." It contains four input fields: "User name", "Email", "Password", and "Confirm password". Below the fields is a blue button labeled "Registration". The form is displayed on a dark background with the text "Duende IdentityServer" in the top left corner.

Obrázek 31 Stránka pro registraci do aplikace

## 6.5 Administrátorská a moderátorská stránka

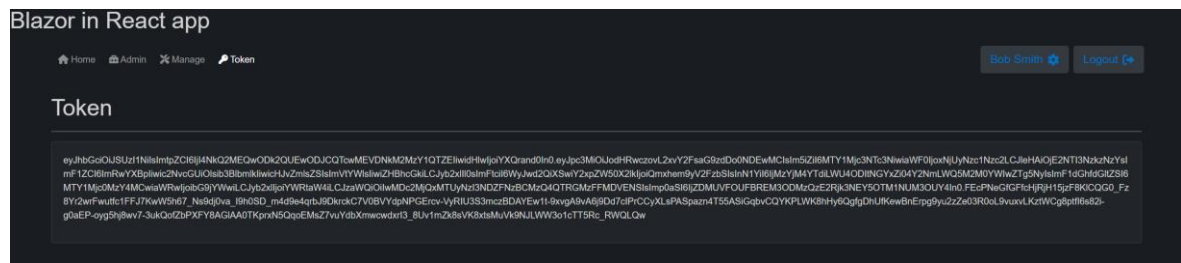
Administrátorská i moderátorská stránka slouží k vytváření událostí. Obě tyto stránky jsou stejné, ale moderátor nemá přístup do administrátorské stránky. Pro vytvoření události je potřeba vyplnit název události, místo konání události, datum startu a datum konce události. Tato pole jsou zároveň i validována, pokud by se uživatel snažil vytvořit událost bez názvu, nepodaří se mu to. Na následujícím obrázku 32 je nastíněna administrátorská stránka.

The image shows a screenshot of an "Admin page" in a "Blazor in React app". The page has a dark theme and a navigation bar with links for "Home", "Admin", "Manage", and "Token". In the top right corner, there are buttons for "Bob Smith" and "Logout". The main content area is titled "Admin page" and contains an "Add event" section. This section has four input fields: "Event name", "Event venue", "Start time" (with a date value of 05/17/2022), and "End time" (with a date value of 05/17/2022). A "create event" button is located at the bottom right of the form.

Obrázek 32 Administrátorská stránka aplikace

## 6.6 Stránka token

Na této stránce se nachází JWT, který je přidělen uživateli. Na Následujícím obrázku 33 je naznačena stránka pro takový token.



Obrázek 33 Stránka pro zobrazení JWT

## ZÁVĚR

Výsledkem praktické části diplomové práce je webová aplikace. Tato celková aplikace se skládá z několika menších aplikací, aplikace vytvořena v technologii React, aplikace vytvořená ve frameworku Blazor WebAssembly, API a IdentityServer vytvořené v ASP .NET Core. Hlavním cílem této práce bylo implementovat Blazor WebAssembly do projektu vytvořeného pomocí Reactu. Tato implementace byla pomocí načítání JavaScriptového souboru Blazor.webassembly.js do souboru index.html v aplikaci Reactu. V nové verzi ASP .NET Core 6 Microsoft přidal experimentální funkcionalitu pro renderování Blazor WebAssembly jako komponentu Reactu. Jelikož je tahle funkcionalita na trhu malou chvíli a není k ní dostatečná dokumentace, nepodařil se mi tento způsob implementace.

V teoretické části diplomové práce byl popsán aktuální stav technologií pro vývoj webových aplikací a technologií pro zabezpečení webových aplikací. Práce klade především důraz na praktičnost, je tedy v práci použito velké množství příkladů zdrojových kódů. V teoretické části se popisovali technologie jako HTML, CSS, JavaScript, Angular, React, Blazor WebAssembly a ASP .NET Core.

V praktické části diplomové práce byl vytvořen návrh takové webové aplikace, která umožňuje vytváření jednoduchých událostí. V praktické části byly dále navrhnuty funkční a nefunkční požadavky a byly definovány případy užití takové aplikace. Následně byly popsány všechny dílčí projekty dané aplikace a jak danou aplikaci spouštět. Dále byly popsány klíčové součásti aplikace, kde je například popsáno, jakým způsobem je rozvržena architektura jednotlivých projektů. Následně je v diplomové práci popsáno, jakým způsobem komunikuje klient s API a je popsána podrobně i konfigurace tohoto zabezpečení. Na závěru praktické části je demonstrace samotné aplikace.

Při vypracování práce byl největší problém s autentizací a autorizací. Jelikož tato webová aplikace je dost specifická nešla použít předem nakonfigurovaná autorizace a autentizace uživatele od Microsoftu.

Na závěr bych chtěl říci, že implementace frameworku Blazor WebAssembly do projektu založeném na technologii React je možná ale zatím není úplně ideální. Jelikož je Blazor WebAssembly poměrně nová technologie můžeme očekávat, že tahle funkcionalitu bude nadále vyvíjena.

## SEZNAM POUŽITÉ LITERATURY

- [1] *Stateofjs* [online]. [cit. 2022-05-03]. Dostupné z: <https://2021.stateofjs.com/en-US/libraries/front-end-frameworks>
- [2] RAGGETT, Dave. A Review of the HTML+ Document Format. *A Review of the HTML+ Document Format* [online]. [cit. 2022-05-03]. Dostupné z: [https://www.w3.org/MarkUp/htmlplus\\_paper/htmlplus.html](https://www.w3.org/MarkUp/htmlplus_paper/htmlplus.html)
- [3] COPEs, Flavio. *The HTML Handbook* [online]. [cit. 2022-05-03]. Dostupné z: <https://flaviocopesbooks.fra1.digitaloceanspaces.com/html-handbook.pdf>
- [4] COPEs, Flavio. *The CSS Handbook* [online]. [cit. 2022-05-03]. Dostupné z: <https://flaviocopesbooks.fra1.digitaloceanspaces.com/css-handbook.pdf>
- [5] *Tailwind CSS* [online]. [cit. 2022-05-03]. Dostupné z: <https://tailwindcss.com/docs/installation>
- [6] *JavaScript history* [online]. [cit. 2022-05-03]. Dostupné z: [https://www.w3schools.com/js/js\\_history.asp](https://www.w3schools.com/js/js_history.asp)
- [7] *JavaScript beginner handbook* [online]. [cit. 2022-05-03]. Dostupné z: <https://flaviocopesbooks.fra1.digitaloceanspaces.com/javascript-beginner-handbook.pdf>
- [8] *What is Angular?* [online]. [cit. 2022-05-03]. Dostupné z: <https://angular.io/guide/what-is-angular>
- [9] *Create a New React App* [online]. [cit. 2022-05-03]. Dostupné z: <https://reactjs.org/docs/create-a-new-react-app.html>
- [10] *Introducing JSX* [online]. [cit. 2022-05-03]. Dostupné z: <https://reactjs.org/docs/introducing-jsx.html>
- [11] *ReactDOM* [online]. [cit. 2022-05-03]. Dostupné z: <https://reactjs.org/docs/react-dom.html>



- [12] *Components and Props* [online]. [cit. 2022-05-03]. Dostupné z: <https://reactjs.org/docs/components-and-props.html>
- [13] *State and Lifecycle* [online]. [cit. 2022-05-03]. Dostupné z: <https://reactjs.org/docs/state-and-lifecycle.html>
- [14] *Webassembly* [online]. [cit. 2022-05-03]. Dostupné z: <https://webassembly.org/>
- [15] *Securing Blazor Client-side Applications* [online]. [cit. 2022-05-03]. Dostupné z: <https://app.pluralsight.com/library/courses/securing-blazor-client-side-applications/transcript>
- [16] *ASP.NET Core Blazor data binding* [online]. [cit. 2022-05-03]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/blazor/components/data-binding?view=aspnetcore-6.0#binding-with-component-parameters>
- [17] *ASP.NET Core Razor components* [online]. [cit. 2022-05-03]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/blazor/components/?view=aspnetcore-6.0>
- [18] *ASP.NET Core Blazor cascading values and parameters* [online]. [cit. 2022-05-03]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/blazor/components/cascading-values-and-parameters?view=aspnetcore-6.0>
- [19] *ASP.NET Core Razor component rendering* [online]. [cit. 2022-05-03]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/blazor/components/rendering?view=aspnetcore-6.0>
- [20] *ASP.NET Core Razor component rendering* [online]. [cit. 2022-05-03]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0>

- [21] *Dependency injection in ASP.NET Core* [online]. [cit. 2022-05-03]. Dostupné z: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0>
- [22] *Dependency Injection* [online]. [cit. 2022-05-03]. Dostupné z: <https://www.tutorialsteacher.com/ioc/dependency-injection>
- [23] GORDON, Steve. *ASP.NET CORE DEPENDENCY INJECTION: WHAT IS THE ISERVICECOLLECTION?* [online]. [cit. 2022-05-03]. Dostupné z: <https://www.stevejgordon.co.uk/aspnet-core-dependency-injection-what-is-the-IServiceCollection>
- [24] *ASP.NET Core Middleware* [online]. [cit. 2022-05-06]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/fundamentals/middleware/?view=aspnetcore-6.0>
- [25] *Overview to ASP.NET Core* [online]. [cit. 2022-05-03]. Dostupné z: <https://swagger.io/docs/specification/about/>
- [26] *HttpsPolicyBuilderExtensions.UseHttpsRedirection(IApplicationBuilder) Method* [online]. [cit. 2022-05-06]. Dostupné z: <https://docs.microsoft.com/en-gb/dotnet/api/microsoft.aspnetcore.builder.httpspolicybuilderextensions.usehttpsredirection?view=aspnetcore-6.0>
- [27] *ControllerEndpointRouteBuilderExtensions.MapControllers(IEndpointRouteBuilder) Method* [online]. [cit. 2022-05-06]. Dostupné z: <https://docs.microsoft.com/en-gb/dotnet/api/microsoft.aspnetcore.builder.controllerendpointroutebuilderextensions.mapcontrollers?view=aspnetcore-6.0>
- [28] *ASP.NET Core fundamentals overview* [online]. [cit. 2022-05-06]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/fundamentals/?view=aspnetcore-6.0&tabs=windows>

- [29] *WebApplication.CreateBuilder Method* [online]. [cit. 2022-05-06]. Dostupné z: <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplication.createbuilder?view=aspnetcore-6.0#microsoft-aspnetcore-builder-webapplication-createbuilder>
- [30] *Overview of ASP.NET Core MVC* [online]. [cit. 2022-05-09]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/mvc/overview?view=aspnetcore-6.0>
- [31] *Create web APIs with ASP.NET Core* [online]. [cit. 2022-05-09]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/web-api/?view=aspnetcore-6.0#problem-details-for-error-status-codes>
- [32] *IdentityServer for cloud-native applications* [online]. [cit. 2022-05-09]. Dostupné z: <https://docs.microsoft.com/en-gb/dotnet/architecture/cloud-native/identity-server>
- [33] *What is JSON Web Token?* [online]. [cit. 2022-05-09]. Dostupné z: <https://jwt.io/introduction>
- [34] *What is OAuth 2.0?* [online]. [cit. 2022-05-09]. Dostupné z: <https://auth0.com/intro-to-iam/what-is-oauth-2/>
- [35] *Welcome to OpenID Connect* [online]. [cit. 2022-05-09]. Dostupné z: <https://openid.net/connect/>
- [36] *Terminology* [online]. [cit. 2022-05-09]. Dostupné z: <https://docs.duendesoftware.com/identityserver/v6/overview/terminology/>
- [37] *ASP.NET Core Blazor hosting models* [online]. [cit. 2022-05-09]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/blazor/hosting-models?view=aspnetcore-6.0>

- [38] *Secure an ASP.NET Core Blazor WebAssembly standalone app with the Authentication library* [online]. [cit. 2022-05-09]. Dostupné z: <https://docs.microsoft.com/en-gb/aspnet/core/blazor/security/webassembly/standalone-with-authentication-library?view=aspnetcore-6.0&tabs=visual-studio>
- [39] *Common web application architectures* [online]. [cit. 2022-05-09]. Dostupné z: <https://docs.microsoft.com/en-gb/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>
- [40] *Atomic design* [online]. [cit. 2022-05-09]. Dostupné z: <https://bradfrost.com/blog/post/atomic-web-design/>

**SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK**

HTML	Hypertext markup language.
CSS	Cascading style sheets.
SEO	Search engine optimization.
VS	Visual studio
CLI	Command line interface
JSX	JavaScript xml
DOM	Document object model
WASM	WebAssembly
API	Application programming interface
Xml	Extensible markup language
ASP	Microsoft active server pages
IoC	Inversion of control
HTTP	Hypertext transfer protocol
REST	Representational state transfer
HTTPS	Hypertext transfer protocol secured
MVC	Model view controller
JWT	JSON web token
JSON	JavaScript object notation
OIDC	OpenID Connect
RFC	Request for comments
RSA	Reverts-Shamir-Adleman
ECDSA	The elliptic curve digital signature
HMAC	Hash based message authentication codes
OAuth	Open Authorization

CORS Cross origin resource sharing

URI Uniform resource identifier

URL Uniform resource locator

## SEZNAM OBRÁZKŮ

Obrázek 1: Graf procentuální znovu použitelnosti jednotlivých technologií v čase přejato z [1]. .....	11
Obrázek 2: Blokový a inline element .....	19
Obrázek 3: Tailwind projekt .....	21
Obrázek 4: Tlačítko .....	22
Obrázek 5: Spuštění základního projektu .....	25
Obrázek 6: Struktura základního React projektu .....	26
Obrázek 7 Standalone Blazor WebAssembly projekt.....	35
Obrázek 8 Hosted Blazor WebAssembly projekt .....	35
Obrázek 9 Rozdělení komponenty razor do 2 souborů.....	37
Obrázek 10 Demonstrace kaskádových parametrů.....	40
Obrázek 11 Výchozí ASP .NET Core projekt .....	41
Obrázek 25 Klasické vrstvení aplikace přejato z [39] .....	48
Obrázek 26 Clean architecture v ASP .NET Core přejato z [39] .....	48
Obrázek 12 Příklad JWT.....	49
Obrázek 13 Komunikace mezi klientem, IdentityServerem a API přejato z [36] .....	50
Obrázek 14 Funkční požadavky aplikace .....	53
Obrázek 15 Nefunkční požadavky aplikace .....	54
Obrázek 16 Cesta ke složce _framework.....	59
Obrázek 17 Složka public v projektu React.....	60
Obrázek 18 Načítané soubory po spuštění aplikace .....	61
Obrázek 19 Detail souboru blazor.webassembly.js .....	61
Obrázek 20 Detail souboru blazor.boot.json .....	61
Obrázek 21 Struktura projektu IdentityServer.....	62
Obrázek 22 Connection string .....	62
Obrázek 23 Struktura databáze pro uživatele .....	63
Obrázek 24 Struktura tabulky AspNetUsers.....	63
Obrázek 27 Struktura projektu MinimalApi.....	64
Obrázek 28 Struktura projektu Blazor WebAssembly .....	67
Obrázek 29 Domovská stránka aplikace.....	76
Obrázek 30 Stránka pro přihlášení do aplikace .....	76
Obrázek 31 Stránka pro registraci do aplikace .....	77

---

Obrázek 32 Administrátorská stránka aplikace .....	77
Obrázek 33 Stránka pro zobrazení JWT .....	78



**SEZNAM TABULEK**

Tabulka 1 Scénář vytvoření nové události.....	55
Tabulka 2 Alternativní scénář – Uživatel zadal špatné přihlašovací údaje .....	55
Tabulka 3 Alternativní scénář – Uživatel špatně vyplnil údaj ve formuláři.....	55
Tabulka 4 Scénář zobrazení administrátorské sekce .....	56
Tabulka 5 Alternativní scénář – Uživatel zadal špatné přihlašovací údaje .....	56
Tabulka 6 Alternativní scénář – Uživatel nemá administrátorská práva .....	57
Tabulka 7 Scénář registrace uživatele .....	57
Tabulka 8 Alternativní scénář – Uživatel zadal špatné registrační údaje .....	57
Tabulka 9 Alternativní scénář – Registrační údaje jsou již využívány.....	58
Tabulka 10 Scénář přihlášení uživatele .....	58
Tabulka 11 Alternativní scénář – Uživatel zadal špatné přihlašovací údaje .....	58

## SEZNAM PŘÍLOH

P1 CD

## **PŘÍLOHA P I: CD**

Přiložené CD obsahuje:

- Diplomovou práci ve formátu docx: fulltext.docx
- Diplomovou práci ve formátu pdf: fulltext.pdf
- Zdrojové kódy: příloha.zip