Srovnání implementace aplikace pro pojišťovnictví ve frameworcích Blazor a Angular

Jakub Horák

Bakalářská 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 BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jakub Horák**
Osobní číslo:	**A19034**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Srovnání implementace aplikace pro pojišťovnictví ve frameworcích Blazer a Angular**
Téma práce anglicky:	**Comparison of Insurance Application Implementation in Blazer and Angular Frameworks**

Zásady pro vypracování

1. Popište současný stav technologií pro vývoj a zabezpečení webových aplikací.
2. Zaměřte se na frameworky Blazor a Angular.
3. Navrhněte aplikaci z oblasti pojišťovnictví, definujte funkční a nefunkční požadavky, případy použití.
4. Navrhněte způsob zabezpečení aplikace.
5. Realizujte vývoj navržené aplikace ve frameworku Blazor, popište její klíčové části a porovnejte řešení s možnou implementací v knihovně Angular.
6. Demonstrujte výsledky a formulujte závěr.

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

Seznam doporučené literatury:

1. MURRAY, Nate, Felipe COURY, Eri LERNER a Carlos TABORDA. Ng-book: The Complete Guide to Angular. Revision 77. San Francisco: Fullstack.io, 2020.
2. ALBAHARI, Joseph a Eric JOHANNSEN. C# 8.0 in a Nutshell: definitive reference. Sebastopol, CA: OŘeilly, [2020]. ISBN 9781492051138.
3. FREEMAN, Adam. Pro ASP.NET Core MVC: Develop cloud-ready web applications using Microsoft’s latest framework, ASP.NET Core MVC. 6th ed. New York: Springer Science+Business Media, 2016. ISBN 9781484203972.
4. HIMSCHOOT, Peter. Microsoft Blazor: Building Web Applications in .NET. 2nd ed. New York: Springer Science+Business Media, 2020. ISBN 9781484259283.
5. ASP.NET documentation | Microsoft Docs [online]. Redmond: Microsoft, ©2021 [cit. 2021-10-19]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core
6. Introduction to ASP.NET Core Blazor | Microsoft Docs [online]. Redmond: Microsoft, ©2021 [cit. 2021-10-19]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/blazor
7. Angular – Introduction to the Angular Docs [online]. Mountain View: Google, ©2010-2021 [cit. 2021-10-19]. Dostupné z: https://angular.io/docs

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

Datum zadání bakalářské práce: **3. prosince 2021**
Termín odevzdání bakalářské práce: **23. května 2022**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

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 bakalářské 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 bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou 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 – bakalářskou 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 bakalářské 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 bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské 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 bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 17. 5. 2022

Jakub Horák, v. r.
podpis studenta

ABSTRAKT

Cílem bakalářské práce je porovnat implementaci aplikace z oblasti pojišťovnictví ve frameworku Blazor s využitím webového standardu WebAssembly a ve frameworku Angular v kombinaci s frameworkem ASP.NET Core. Teoretická část je zaměřena na momentálně nejvyužívanější technologie pro vývoj a zabezpečení webových aplikací s hlavním zaměřením na frameworky Blazor a Angular. V praktické části je pak popsán návrh a vytvoření aplikace z oblasti pojišťovnictví ve frameworcích Blazor a Angular a také návrh a implementace zabezpečení dané aplikace. Dále definice funkčních a nefunkčních požadavků a případů užití. Jsou zde také popsány klíčové části obou verzí aplikace a porovnání implementací v již zmíněných dvou frameworcích. Na závěr jsou klíčové části obou verzí aplikace srovnány a je zformulován závěr a popis výhod a nevýhod implementace ve frameworcích Angular a Blazor.

Klíčová slova: Webová aplikace, Blazor, WebAssembly, Angular, ASP.NET, C#

ABSTRACT

The aim of this bachelor thesis is to compare the implementation of an insurance application in the Blazor framework using the WebAssembly web standard and in the Angular framework in combination with the ASP.NET Core framework. The theoretical part focuses on the most currently used technologies for web application development and security with the main focus on Blazor and Angular frameworks. The practical part describes the design and creation of an insurance application in the Blazor and Angular frameworks, as well as the design and implementation of the application security measures. It also includes the definition of functional and non-functional requirements and use cases. The key parts of both versions of the application and a comparison of the implementations in the two frameworks already mentioned are also described. Finally, the key parts of both versions of the application are compared and a conclusion and description of the advantages and disadvantages of implementing in the Angular and Blazor frameworks is formulated.

Keywords: Web application, Blazor, WebAssembly, Angular, ASP.NET, C#

Tímto bych chtěl poděkovat vedoucímu mé bakalářské práce Ing. et Ing. Eriku Králi, Ph.D. za ochotu, odborné rady, trpělivost a také za možnost velmi častých konzultací. Také chci poděkovat své rodině a přátelům, kteří mě při studiu podporovali.

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

OBSAH

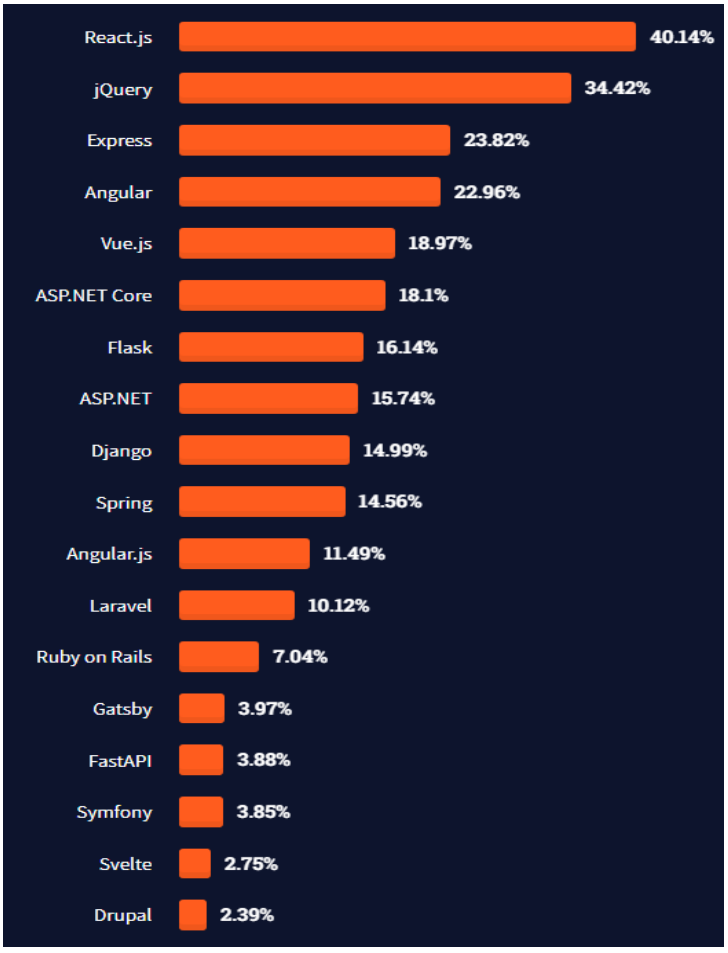
ÚVOD

Bakalářská práce se zabývá srovnáním implementace aplikace z oblasti pojišťovnictví mezi poměrně novým frameworkem Blazor, který využívá standard WebAssembly, a již starším a vyspělejším frameworkem Angular. Teoretická část práce je zaměřena na současný stav technologií pro vývoj a zabezpečení webových aplikací. Dále se v teoretické části nachází popis frameworků Blazor a Angular. V první kapitole teoretické části je využito dotazníku webu StackOverflow z roku 2021. Z tohoto dotazníku jsou získány tři nejpoužívanější frontendové frameworky, tři nejpoužívanější backendové frameworky a tři nejpoužívanější databázové technologie. Každý framework a databázová technologie je popsána ve vlastní podkapitole. V podkapitole zabezpečení byl vytvořen seznam osmi nejčastějších kybernetických útoků na webové aplikace. Každý z útoků je popsán, jakým způsobem funguje a také jak se proti němu dá bránit. Další kapitolou teoretické části je popis frameworku Blazor. Zde je popsána historie frameworku, jeho hostovací modely a zajímavé funkcionality. Stejně tak je popsán framework Angular, kde je popsána jeho historie, architektura a zajímavé funkcionality. Praktická část bakalářské práce je zaměřena na návrh a implementaci webové aplikace. V rámci návrhu aplikace jsou vytvořeny funkční požadavky, nefunkční požadavky, scénáře případů užití a popis zabezpečení aplikace. Následuje popis klíčových částí aplikace, ve kterém jsou popsány endpointy, služby a propojení s databází. Dále také implementace komponent, modelů, validací a možnosti komunikace se serverem, a to vše ve frameworku Blazor i Angular. Popis klíčových částí aplikace je doprovázen ukázkami ze zdrojového kódu. Následuje podkapitola popisující rozdíly a podobnosti v implementaci aplikace v již zmíněných frameworcích. Nakonec je vytvořená aplikace demonstrována z pohledu uživatele a popsán postup skrz aplikaci s obrázky, které ukazují, jak aplikace vypadá.

I. TEORETICKÁ ČÁST

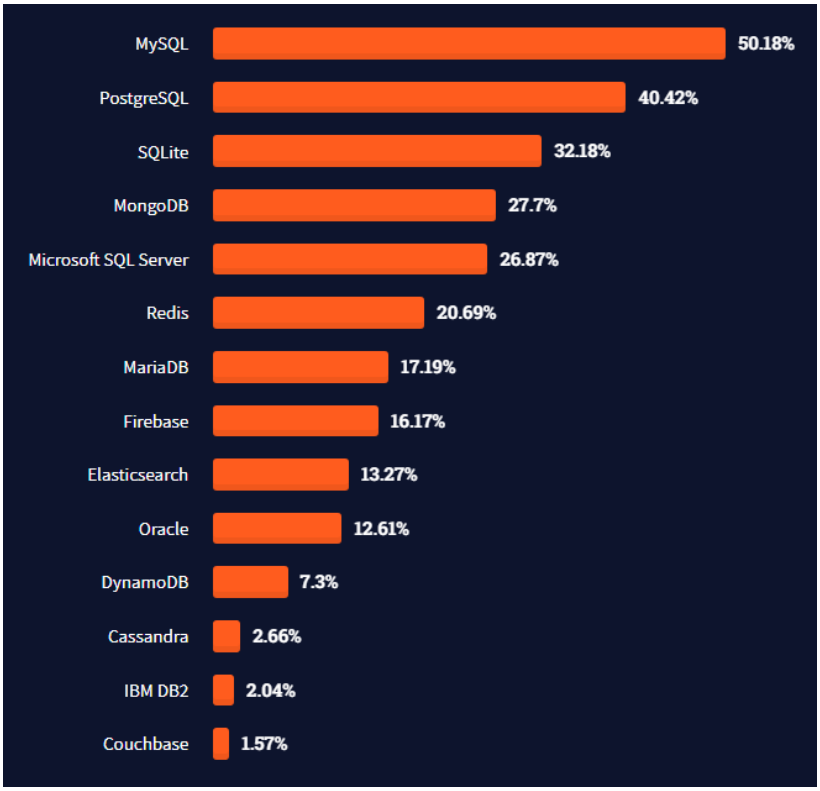
1 SOUČASNÝ STAV TECHNOLOGIÍ PRO VÝVOJ A ZABEZPEČENÍ WEBOVÝCH APLIKACÍ.

Výkonné mnohostránkové webové aplikace jsou velmi důležitou součástí online podnikání. Vyvíjet takovouto webovou aplikaci je velmi časově náročné a je vyžadována velká technická dovednost a schopnost. Využití frameworků tento vývojový proces značně usnadní a zrychlí. V dnešní době již existuje celá řada frameworků pro vývoj webových aplikací. Na obrázku číslo 1 můžeme vidět výsledky dotazníku webu StackOverflow [1], kde 67 593 dotazovaných odpovědělo na otázku „Se kterými webovými frameworky a knihovnami jste v uplynulém roce intenzivně pracovali, a se kterými chcete pracovat v roce příštím?“. Nejpoužívanějším webovým frameworkem roku 2021 je dle dotazníku React.js, který v průběhu let postupně nabýval na popularitě (zhruba o 5 % populárnější každým rokem) a sesadil z prvního místa po dlouhé době jQuery, který naopak na popularitě každoročně upadá (pokles o necelých 9 % oproti minulému roku). Dalšími velmi populárními a využívanými frameworky jsou Express, Angular, Vue.js nebo ASP.NET Core.



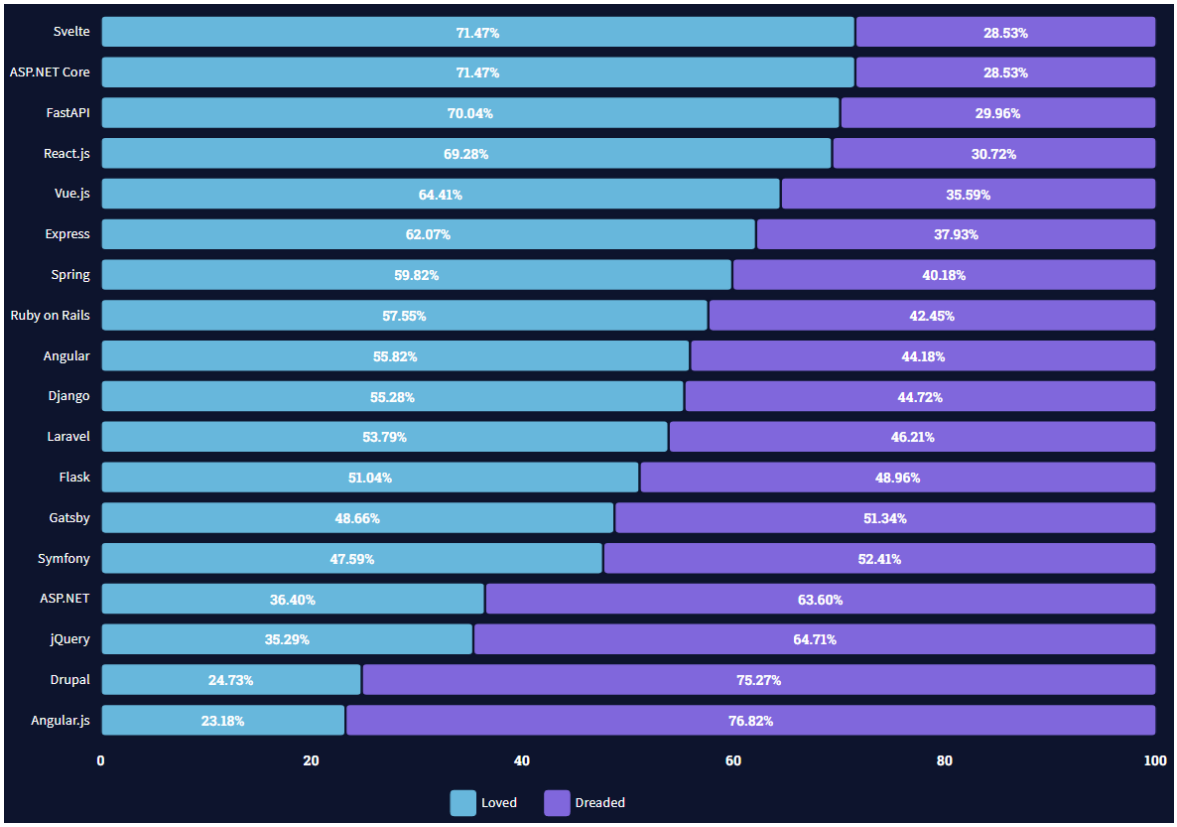
Obrázek 1 – Nejpoužívanější webové frameworky roku 2021 [1]

Téměř každá webová aplikace nyní pracuje s nějakým typem databáze. Co se týče databázových technologií, na otázku „Se kterými databázovými prostředími jste v uplynulém roce intenzivně pracovali, a ve kterých chcete pracovat v roce příštím?“ odpovědělo 73 317 programátorů a nejvíce používanou databázovou technologií za rok 2021 je stále MySQL, jak můžeme vidět na obrázku číslo 2. Oproti předešlému roku se z třetího na páté místo propadl Microsoft SQL Server (pokles o zhruba 6 %). Mezi profesionálními programátory je ale Microsoft SQL server stále populárnější než MangoDB. Dalšími velmi populárními databázovými technologiemi je například PostgreSQL nebo SQLite. [1]



Obrázek 2 – Nejpoužívanější databázové technologie roku 2021 [1]

Na obrázku číslo 3 lze vidět, že nejvíce „Loved“ webovým frameworkem je relativně nový framework Svelte, který zatím ještě není vůbec rozšířený, ale rozhodně je oblíbený, jelikož 71,47 % z 66 202 dotazovaných dalo Svelte velmi kladné hodnocení v kategorii, kde měli rozhodovat, zda jsou jednotlivé webové frameworky „Loved“ nebo „Dreaded“. O první místo se ale skoro dělí s backendovým frameworkem ASP.NET Core s totožnou oblíbeností. Dalšími komunitou oblíbenými frameworky jsou FastAPI, React.js nebo například Vue.js. Naopak nejvíce „Dreaded“ frameworkem je původní verze Angularu - Angular.js, dále Drupal nebo jQuery, který ještě minulý rok byl nejpoužívanějším frameworkem. [1]



Obrázek 3 – Nejpopulárnější webové frameworky roku 2021 [1]

1.1 Frontendové frameworky

Frontend je ta část webové aplikace, kterou uživatel vidí a přímo s ní interaguje. Označuje se také jako klientská strana aplikace. Pro vývoj frontendu se nyní využívají jen tři programovací jazyky a to HTML, CSS a JavaScript. [2] Frontendové frameworky pak tvoří platformu pro vývoj frontendu aplikace. To zahrnuje určitý způsob organizace a konstrukce souborů, přiřazování dat k prvkům DOM, stylování elementů, routování nebo posílání AJAX požadavků na server. [3]

1.1.1 React.js

React [4] je knihovna pro vývoj uživatelského rozhraní založená na programovacím jazyku JavaScript. Knihovna je vyvíjena společností Facebook a jelikož je open-source, tak i vývojářskou komunitou. React se poprvé objevil v roce 2013 a nyní je nejpoužívanějším webovým frameworkem pro vývoj frontendu webových stránek. React velmi usnadňuje tvorbu dynamických webových aplikací, protože vyžaduje méně programování a nabízí více funkcí než samotný JavaScript. Umožňuje vytvářet a znovu používat komponenty, což jsou základní stavební kameny každé React aplikace a jejich využití výrazně zkracuje čas vývoje a snižuje redundanci kódu. Využívá se zde tzv. virtuálního DOM, který porovnává předchozí stavy komponent a aktualizuje pouze ty komponenty, které byly změněny, a to v reálném čase. Díky tomuto se nemusí aktualizovat úplně všechny komponenty a webová aplikace má vyšší výkon. Dalšími výhodami je využitelnost pro vývoj mobilních aplikací pomocí frameworku React Native, který je odvozený od samotného Reactu. Společnost Facebook také vydala rozšíření pro prohlížeč Chrome, které lze využít pro snadné ladění aplikací vyvíjených v tomto frameworku. Dalším rozšířením je JSX, což je syntaktické rozšíření jazyka JavaScript, které umožňuje psát JavaScript kód i HTML kód do stejného souboru. [5]

1.1.2 jQuery

jQuery [6] je malá, lehká a rychlá JavaScript knihovna. Je multiplatformní a podporuje velkou škálu různých prohlížečů. Řídí se motem „napiš méně, udělej více“, jelikož přebírá velké množství běžných úloh, k jejichž provedení je potřeba napsat spoustu řádků v JavaScriptu, a tvoří z nich metody, které lze zavolat jediným řádkem kódu. Knihovna je také velmi užitečná pro zjednodušení mnoha dalších složitých věcí z jazyka JavaScript, jako jsou např. AJAX volání nebo manipulace s DOM. Nejdůležitějšími funkcemi jQuery jsou usnadnění HTML, DOM a CSS manipulace, výběr HTML elementů pomocí id nebo třídy, podpora animací a efektů, zjednodušená syntaxe AJAX volání, metody událostí HTML elementů, parsování JSON dat a rozšiřitelnost pomocí pluginů. [7]

1.1.3 Vue.js

Vue [8] je JavaScriptový framework pro vytváření uživatelských rozhraní a jednostránkových aplikací. Je open-source a využívá architektonický vzor model-view-viewmodel (MVVM). Framework byl vytvořen Evanem You v roce 2014 jako přímá reakce na práci s frameworkem AngularJS. Cílem bylo vzít z AngularJS části, které se mu líbily, a vytvořit zcela nový a lehký framework. Architektura MVVM umožňuje odlišit obchodní logiku od grafického uživatelského rozhraní. Vue narozdíl od jiných frameworků je velmi jednoduchý a úplně nepřetéká funkcionalitou. Naopak poskytuje téměř úplné minimum, které bychom od JavaScriptového frameworku očekávali. Obsahuje základní knihovnu, která se zaměřuje pouze na vrstvu view, jakékoliv další funkce je třeba získat pomocí podpůrných knihoven. Nejvýraznější vlastností je možnost přidávání vlastních HTML direktiv. Direktivy jsou atributy programovacího jazyka HTML, které pak vývojářům umožňují rozšiřovat funkcionalitu HTML elementů. Výhodou je, že je s frameworkem velmi snadné začít, ale navýšení komplexnosti aplikace v budoucnu není problém. Je velmi lehký, velikost balíčku je jen 21 kB a stejně jako React využívá virtuálního DOM. To pak výrazně zrychlí vykreslování webových aplikací vytvořených ve Vue. Nevýhodou Vue pak je to, že je velmi nový, a tím pádem má stále malou vývojářskou komunitu a neobsahuje tolik funkcionalit, pluginů, komponent a rozšíření jako další webové frameworky. I přes to má framework uplatnění např. při tvorbě prototypů webového designu nebo pro tvorbu jednostránkových aplikací, které jsou jednoduché a nepotřebují tak robustní framework, který by je zastřešoval. [9]

1.2 Backendové frameworky

Backend je označení pro serverovou část webové aplikace. Ukládá a uspořádává data, obsahuje veškerou business logiku aplikace a uživatel jej nevidí a přímo s ním neinteraguje jako s frontendem. Klientská strana aplikace s backendem nejčastěji komunikuje pomocí rozhraní API. Pro vývoj backendu se může využít téměř jakýkoliv programovací jazyk, např. PHP, C++, Java, Python nebo C#. [2] Backend framework pak stejně jako frontend framework poskytuje způsob organizace souborů aplikace, dále také obsahuje knihovny funkcí, které usnadňují vývoj backend strany aplikace.

1.2.1 Express

Express [10] je nejoblíbenějším webovým frameworkem pro Node.js. Je také základní knihovnou pro řadu dalších webových frameworků využívajících Node.js. Je open-source a určen primárně k tvorbě backendu webových aplikací, a hlavně API rozhraní. Jelikož využívá programovacího jazyka JavaScript, je velmi jednoduché s ním začít, protože se vývojář nemusí učit 2 programovací jazyky, jeden pro frontend aplikace a druhý pro backend. Express je také součástí tzv. MEAN stacku, což je soubor technologií, díky kterým lze tvořit webové aplikace čistě za pomocí jazyka JavaScript. Jedná se o technologie MangoDB (databáze), Express (backend framework), Angular (frontend framework) a Node.js (hlavní webový server). [11] V základu Express poskytuje zpracování HTTP požadavků v kombinaci s různými URL cestami, umožňuje měnit nastavení webové aplikace, jako je port, který se používá pro připojení nebo umístění šablon, které se používají pro vykreslování odpovědi na HTTP požadavky. Obsahuje také integrovaný renderovací view engine, který při odpovědi na HTTP požadavky vkládá data do HTML šablon a vykresluje je. I přes to, že je Express poměrně jednoduchý, vývojáři vytvořili velké množství kompatibilních middleware balíčků, které řeší prakticky jakýkoliv problém, který může nastat při vývoji webové aplikace. To zahrnuje práci se soubory cookie, relacemi, přihlašování a registrace uživatelů, využívání URL parametrů, bezpečnostní hlavičky a další. [12]

1.2.2 ASP.NET Core

Framework ASP.NET [13] se k vývoji webových aplikací využívá již dlouhou řadu let, od jeho počátků prošel framework velkým vývojem, který stále pokračuje, a nakonec vedl i ke vzniku jeho nejnovějšímu potomkovi ASP.NET Core 1.0. ASP.NET Core není pokračováním ASP.NET verze 4.6, ale je zcela novým frameworkem. Je vytvořen společností Microsoft a jedná se o kompletní přepracování ASP.NET frameworku, tak aby byl rychlejší, více flexibilní, moderní a multiplatformní, což znamená, že webová aplikace může běžet na operačním systému Windows, Linux i Mac. Je open-source a je založen na architektuře MVC (Model-View-Controller) a Web API, které kombinuje do jednoho webového frameworku. [14] Architektura MVC rozděluje webovou aplikaci na tři hlavní logické prvky: model, který obsahuje veškerou business logiku a reprezentuje data. View, což je část, která obsahuje uživatelské rozhraní a s tím spojenou logiku. A nakonec Controller, který se chová jako prostředník mezi modelem a view. Zpracovává příchozí HTTP požadavky, manipuluje s daty a předává výsledky do konkrétních view. [15] ASP.NET Core je navržen tak, aby poskytoval optimalizovanou podporu pro aplikace, které jsou pak nasazeny na cloudu, a je založen na sadě modulárních a specializovaných balíčků NuGet, což umožňuje optimalizovat webovou aplikaci tak, aby obsahovala jen ty balíčky, které jsou třeba. Má vestavěnou podporu pro dependency injection, obsahuje tag helpery a umožňuje vývojáři hostovat webovou aplikaci na Windows IIS serveru. [14]

1.2.3 Flask

Flask [16] je framework pro vývoj webových aplikací vyvinutý v jazyce Python. Je velmi jednoduchý na počáteční naučení a používání, a tedy ideální pro začátečníky, jelikož neobsahuje tzv. boilerplate kód nebo závislosti, které by odváděly pozornost od vývoje primárních funkcí aplikace. Framework vzniknul v roce 2004 jako aprílový žert vývojáře Armina Ronachera. Open-source programátorská komunita si jej ale rychle oblíbila a projekt začala rozvíjet a programátorskou komunitou je udržován do dnes. Oproti full-stack frameworkům, které by v základu nabízely mimo jiné i autentizaci, autorizaci, databázový ORM, validaci vstupů a další funkce, je Flask označován jako mikro-framework, protože je lehký a poskytuje pouze nezbytné komponenty pro vývoj webové aplikace, jako jsou routing, zpracování HTTP požadavků nebo práce s relacemi. Flask také poskytuje v základu vývojový server a ladící program, má integrovanou podporu pro jednotkové testování a taky obsahuje mnoho podpůrných knihoven, kterými lze rozšířit jeho funkce. [17] V neposlední řadě využívá

populární šablonovací systém pro Python – jinja2, který umožňuje kombinovat HTML šablony s konkrétním zdrojem dat, a je tak možné zapisovat Python proměnné přímo do HTML šablon a tvořit tak dynamické webové aplikace. V poslední řadě také podporuje rozhraní WSGI, které se používá jako standard pro vývoj webových aplikací v programovacím jazyce Python. [18]

1.3 Databázové technologie

Databáze je uspořádaný soubor strukturovaných informací nebo dat, obvykle uložených elektronicky v počítačovém systému. Databáze je často řízena systémem pro správu databází (DBMS). Data v nejběžnějších typech databázových systémů jsou obvykle strukturována do řádků a sloupců propojených tabulek, aby bylo možné efektivně data zpracovávat a provádět nad nimi různé dotazy. K datům lze pak snadno přistupovat, upravovat nebo je mazat. Většina databází používá pro zápis a dotazování strukturovaný dotazovací jazyk SQL. [19]

1.3.1 MySQL

MySQL [20] bylo původně vyvinuto v roce 1994 Švédskou společností MySQL AB. Společnost byla pak koupena v roce 2008 americkou společností Sun Microsystems, kterou dva roky na to odkoupila společnost Oracle. Od roku 2010 je tedy MySQL pod vlastnictvím společnosti Oracle. Definičně je MySQL open-source systém pro správu relačních databází (RDBMS) s modelem klient-server. RDBMS je software používaný pro vytváření a správu databází na základě relačního modelu. Počítače, které mají nainstalovaný RDBMS software, jsou označovány jako klient a v případě, že chtějí přistoupit k datům, tak se připojují na RDBMS server. Vysokou popularitu pak lze vysvětlit např. díky flexibilitě a jednoduchosti použití. Jelikož jde o open-source projekt, lze upravit zdrojový kód pro specifické potřeby aplikace. Dále díky vysokému výkonu i při velkých objemech dat a silnému zabezpečení skrz verifikaci a šifrování hesel. [21]

1.3.2 PostgreSQL

PostgreSQL [22] je open-source objektově-relační systém pro správu databází s velkým důrazem na rozšiřitelnost a dodržování standardů. Jde po MySQL o druhý nejpoužívanější systém. První verze vznikla roku 1981 s názvem Ingres na Kalifornské univerzitě, později byl projekt přejmenován na Postgres a po přidání podpory jazyka SQL v roce 1996 byl znovu přejmenován do nynější podoby PostgreSQL. Je zdarma a vyvíjí jej Global Development Group, což je skupina různých společností a také tisíce individuálních programátorů. Systém je schopen běžet na všech hlavních operačních systémech včetně Linuxu, variant UNIXu a Windowsu. Je plně kompatibilní s databázovým standardem ACID [23], který definuje vlastnosti, které by databáze měla mít, aby došlo k spolehlivému zpracování databázových transakcí (atomicita, konzistence, izolovanost a trvalost). Podporuje také mezinárodní znakové sady, vícebajtové kódování znaků, Unicode znaky, rozlišování velkých a malých písmen nebo formátování. [24] Nevýhodou oproti MySQL je pak nutnost provedení většího množství práce, pokud chceme dosahovat větších rychlostí, a také fakt, že větší počet aplikací podporuje MySQL než podporuje PostgreSQL. [25]

1.3.3 Microsoft SQL Server

Microsoft SQL Server [26] je stejně jako MySQL a PostgreSQL systém pro správu relačních databází (RDBMS). Úplně první verze se SQL Server dočkal roku 1989 skrz spolupráci společností Microsoft a Sybase. Partnerství společností se ale rozpadlo a SQL Server převzal Microsoft. [27] Stejně jako všechny hlavní systémy RBDMS podporuje jazyk SQL, ale obsahuje také vlastní implementaci SQL jazyka, tzv. T-SQL (Transact-SQL), který rozšiřuje SQL jazyk o další funkcionality, nová klíčová slova a narozdíl od SQL příkazů se T-SQL příkazy provádí procedurálně, což znamená, že kód se zpracovává jako blok, logicky a ve strukturovaném pořadí. [28] Hlavním uživatelským rozhraním pro přístup k SQL Serveru je aplikace SQL Server Management Studio (SSMS). SQL Server není open-source a ve své plné verzi není zdarma. Je rozdělen do několika edicí podle potřeby užití od edice Express, která je zdarma, ale neobsahuje pokročilejší funkcionality, přes Developer edici určenou pro vývojáře, web edici, Standard edici a nakonec edici Enterprise (cena až 300 000 Kč), což je prémiová verze SQL serveru, která obsahuje všechny nabízené funkcionality, obrovský výkon a maximální kapacity. [29] [30]

1.4 Zabezpečení

Téměř každá webová aplikace je do jisté míry zranitelná vůči bezpečnostním útokům. Hrozby se mohou lišit, od lidských pochybení až po koordinované a velmi sofistikované útoky využívající softwarových chyb v systému. Dle studie společnosti Verizon [31] je z velké části motivací útočníků finanční výdělek, jelikož ať už provozujeme sofistikovanou webovou aplikaci, elektronický obchod, nebo malou firemní stránku, téměř vždy je napojená na databázi často obsahující citlivé údaje, které se dají využít nebo prodat. Níže můžeme vidět 8 nejčastějších útoků na zabezpečení webových aplikací a také, jak je možné se proti nim bránit.

1.4.1 Cross-site scripting (XSS)

XSS je dle studie od Precise Security [32] nejčastějším kybernetickým útokem na webové stránky. Podle studie se jedná o téměř 40 % veškerých kybernetických útoků. Jedná se sice o nejčastější typ napadení webových aplikací, ale většina těchto napadení není příliš sofistikovaná a provádějí je amatéři pomocí skriptů, které vytvořil někdo jiný. XSS je zranitelnost webu, která umožňuje útočníkovi vložit do zranitelné aplikace škodlivé skripty napsané v jazyce JavaScript. Když pak uživatel přijde na webovou stránku a vyžádá si od ní data, může mu být webovou stránkou zaslán tento škodlivý skript a webovým prohlížečem spuštěn. Nejjednodušší verzí tohoto útoku je poskytnutí odkazu, v němž se nachází v parametrech škodlivý skript, uživatel na něj klikne a aplikace bez jakékoliv kontroly využije data z parametrů. Často se pak mohou útočníci takto vydávat za napadené uživatele a využívat všech jejich privilegií na dané webové stránce. Bránit webovou stránku proti tomuto typu útoku je možné hned několika způsoby. Striktně filtrovat uživatelské vstupy, tak aby obsahovaly jen to, co je očekávaný vstup, a odstranit např. veškeré html tagy. Pokud naše webová aplikace zasílá v HTTP odpovědích uživateli data řízená uživatelem, tak daná data kódovat, tak aby nemohla být interpretována webovým prohlížečem jako aktivní obsah stránky. Další možností je využít nastavení HTTP hlaviček, tak aby HTTP odpověď webové aplikace neobsahovala žádný HTML a JavaScript kód. K tomu se dají využít hlavičky Content-Type nebo X-Content-Type-Options. V poslední řadě se dá využít tzv. Content Secure Policy (CSP), který informuje webový prohlížeč o platných zdrojích pro právě načítaný obsah, které akce lze provést nebo které pluginy lze načíst. [33]

1.4.2 Brute force

Útok hrubou silou je jedním z nejjednodušších útoků v tomto seznamu. V oblasti webových aplikací se nejčastěji využívá k prolomení přihlašovacích údajů. Útok spočívá v tom, že se útočník snaží uhádnout kombinaci přihlašovacího jména nebo hesla, a to nejčastěji za pomoci nějakého programu, který co nejrychleji zkouší kombinace všech možných znaků nebo také jen zkoušení nejčastějších jednoduchých hesel a jejich kombinací. Proti tomuto útoku se dá na aplikační úrovni bránit např. omezením počtu pokusů o přihlášení. Ale nejúčinnější ochranou proti tomuto útoku je silné heslo, které by mělo obsahovat malá i velká písmena, čísla a alespoň jeden speciální znak. Při využití všech možných znaků a délce hesla alespoň 12 znaků by pak jednomu počítači trvalo prolomit heslo několik milionů let. Velmi rozšířenou funkcí aplikací je dvoufázové ověření, které také téměř eliminuje hrozbu útoku hrubou silou, tak že při přihlášení z nového zařízení nebo při přihlášení po uplynutí specifikované časové periody požaduje po uživateli ověření pomocí kódu, který se mu zobrazí na mobilním zařízení. Pro největší bezpečnost je doporučované vyžadovat po uživateli silné heslo i nastavení dvoufázového ověření.

1.4.3 SQL injection

Útok SQL injection je nejčastějším útokem svého typu (injektování kódu nebo příkazů do zranitelné aplikace). Útok umožňuje útočníkovi zasahovat do dotazů, které aplikace provádí nad vlastní databází. Nejčastěji umožňuje útočníkovi zobrazovat data, ke kterým má přístup samotná aplikace a ke kterým by se za běžného chodu aplikace nedostal. Ve velkém množství případů může útočník data i upravovat nebo mazat, čímž může trvale změnit či poškodit chování aplikace. V ojedinělých případech lze tímto útokem eskalovat i na útok typu odepření služby (DoS). Zranitelnost vůči tomuto typu útoku vzniká neošetřenými vstupy nebo špatně napsanými SQL dotazy, které využívají vstupy od uživatele tak, jak jsou jim předány, a není nad nimi provedena žádná validace. [34] Ochranou číslo 1 proti tomuto typu útoku je vždy používat parametrizace při skládání SQL dotazů. K parametrizaci SQL dotazů slouží velké množství knihoven pro různé frameworky. Parametrizace zaručí, že se s parametry SQL dotazu zachází bezpečným způsobem. Pokud není možné využít žádné knihovny, která by za nás sestavila parametrizované SQL dotazy, je další dobrou ochranou tzv. „escapovat“ speciální znaky označující řetězec, jelikož útočník často spoléhá na to, že vytvoří vstup, který předčasně uzavře řetězec argumentů. Toto řešení ale není tak efektivní jako parametrizace. Často se v praxi využívá tzv. ORM neboli objektově relační mapování. Což jsou

knihovny, které umožňují psát dotazy na databázi v jazyce frameworku, a využívat tak objekty. ORM často umožňují psát SQL příkazy přímo a spustit je nad databází, což neochraňuje proti SQL injection útoku, ale pokud je ORM využíváno jen skrz poskytnuté metody knihovny, tedy není využito přímo jazyka SQL, tak se využívá opět parametrizace a jedná se o velmi dobrý způsob, jak se chránit proti tomuto typu útoku. [35]

1.4.4 OS command injection

Tento typ útoku využívá špatného zabezpečení webové aplikace, která poté umožní útočníkovi provádět příkazy operačního systému přímo na serveru hostujícím webovou aplikaci. Od SQL injection se liší tím, že se příkazy zadávají ze strany serveru a ne ze strany aplikace. Zranitelnost vůči tomuto útoku vzniká, když webová aplikace odesílá ke spuštění nesanitizované a nefiltrované příkazy operačního systému, na kterém běží. Kvůli nedostatečné validaci vstupů může útočník injektovat vlastní příkazy, které budou spuštěny v rámci instance právě běžící aplikace. Příkazy může injektovat např. skrz soubory cookie, skrz formuláře nebo v hlavičkách HTTP požadavků. Jedná se o obzvláště nebezpečný útok, jelikož útočníkem spouštěné příkazy jsou často spouštěny s právy aplikace, která na systému běží. Což mohou být i práva správce systému a útočník tak může získat úplnou kontrolu nad hostitelským operačním systémem, čímž ohrozí aplikaci i veškeré její data. Takto napadená aplikace může být využita i ke kompromitaci dalších částí firemní sítě, což může vést k dalším útokům v rámci firmy. Nejjednodušším způsobem, jak zabránit zranitelnosti aplikace vůči tomuto útoku, je vůbec nevolat příkazy operačního systému z kódu aplikace. Prakticky jakýkoliv příkaz operačního systému lze provést skrz funkce poskytované v rámci frameworku nebo vestavěných knihoven. Pokud je ale volání příkazů operačního systému nutné, je potřeba zavést silnou validaci a sanitizaci vstupů, to znamená např. omezení vstupů podle jeho typu jen na čísla či písmena. [36]

1.4.5 Distributed Denial-of-Service (DDoS)

Útok typu odepření služby je útok, kdy se útočník snaží zahltit webovou stránku obrovským množstvím dat. Narozdíl od jiných útoků na webové stránky, tento typ útoku se nesnaží prolomit bezpečnost webové aplikace, ale snaží se vyčerpat serverové zdroje, na kterých webová stránka běží (RAM, CPU), a narušit tak nebo úplně zastavit provoz daného serveru. Může být využit také jako zástěrka pro jiné škodlivé útoky, jelikož mohou být tímto útokem zacíleny i servery s bezpečnostním softwarem, a DDoS útokem se toto zabezpečení naruší. Tento typ útoku také využívají často aktivisté, dále vyděrači, kteří za poplatek s útokem

přestanou, nebo lidé, které chtějí shozením serverů získat pozornost. Útoky mohou probíhat v krátkých intervalech nebo opakovaně, mohou trvat dny, týdny i měsíce, a pokud společnost spoléhá na chod webových stránek nebo jejich serverů, tak to pro ni může být velmi nákladné, jelikož při nefunkčnosti dochází ke ztrátě příjmů nebo např. k podkopání důvěry od uživatelů. DDoS útoky patří pod útok typu DoS (Denial of Service). Rozdíl mezi těmito dvěma útoky je v jejich provedení a škále. Při DoS útoku je využito jen jednoho počítače nebo skupiny počítačů připojených k jednomu internetovému připojení. Oproti tomu DDoS využívá obrovské množství počítačů různě po světě. Útok DoS se pak dá odrazit např. zablokováním jakékoliv komunikace serveru s IP adresou, ze které přichází právě to obrovské množství dat. To je při DDoS útoku obtížnější, jelikož se jedná o velké množství zařízení. DDoS útokům se nedá zabránit, ale dá se proti nim chránit. Je důležité monitorovat provoz na serveru a hlásit jakékoliv velké výkyvy v přijímaných datech nebo podezřelé připojení. Další dobrou praktikou je mít obsah distribuován na několika redundantních serverech, při útoku pak provoz převezmou nezasažené servery nebo využít tzv. Content delivery network (CDN), což je systém, který dokáže rozprostřít zátěž z DDoS útoku mezi všechny servery rovnoměrně. [37] V poslední řadě je možné využít služeb zaměřených právě na ochranu proti tomuto typu útoku, ale i jiným kybernetickým útokům, jako je např. CloudFlare. [38]

1.4.6 Man-In-The-Middle

Útoky typu Man-In-The-Middle jsou využívány hlavně u webů, které nešifrují data, která se zasílají od uživatele na server. Útočník využívá tento typ útoku ke shromažďování velmi často citlivých informací jako přihlašovací údaje, detaily účtu nebo čísla kreditních karet. Útočník se zapojí do komunikace uživatele a serveru a dělá tzv. prostředníka mezi nimi. Může tak odposlouchávat a zachytávat data při jejich přenosu mezi uživatelem a serverem, a pokud data nejsou šifrována, tak je může útočník snadno přečíst. Může se také přímo vydávat za jednu ze stran (uživatele i server), aby tak vyvolal zdání, že se jedná o běžnou výměnu informací. Cílem útoku jsou obvykle finanční aplikace, SaaS webové aplikace nebo elektronické obchody. Obecně stránky, které vyžadují přihlášení a obsahují citlivé finanční údaje uživatele. Nejběžnějším a nejjednodušším způsobem, jak útočníci zachytí komunikaci mezi uživatelem a serverem, je tzv. pasivní útok, při kterém útočník vytvoří Wi-Fi hotspot na veřejném místě, který je často pojmenován, tak aby odpovídal lokaci, a neobsahuje heslo. V momentě, kdy se někdo připojí na tento hotspot, je útočníkovi k dispozici veškerá výměna dat online dané osoby a může tak působit jako prostředník. Na straně webové aplikace se lze proti tomuto útoku chránit využitím zabezpečeného bezpečnostního protokolu HTTPS

nainstalováním SSL certifikátu. Tím se útoku přímo nezabrání, ale všechna přenášená data jsou pak šifrována a ověřována a útočník by musel vynaložit další úsilí, aby dále ještě prolomil šifru a mohl data přečíst. Chránit se proti tomuto útoku může také sám, uživatel tím že se bude vyhýbat sítím Wi-Fi, která neobsahují hesla. Nenavštěvovat webové stránky, které nevyužívají HTTPS protokolu (na tuto skutečnost vždy upozorní i webový prohlížeč). Nebo obecně neprovádět akce, při kterých dochází k přesunu citlivých informací na veřejných sítích. [39]

1.4.7 Fuzzing

Fuzzing útok nebo také Fuzz testování je automatizovaný proces využívaný k nalezení bezpečnostních chyb a zranitelností aplikace. Princip spočívá ve vkládání velkého množství náhodných dat neboli Fuzzu do aplikace a pozorování výsledků. Je běžně používán týmy kybernetické bezpečnosti, jelikož se jedná o způsob otestování zranitelnosti aplikace s velmi nízkými náklady, malou časovou náročností a vyžaduje jen malé nebo žádné předchozí znalosti či analýzu testované aplikace. Stejně jako využívají fuzzing testování týmy kybernetické bezpečnosti, tak jej ale mohou využít i útočníci, kteří díky fuzzing útoku zjistí zranitelnosti aplikace a následně je využijí. Fuzzing lze rozdělit dle výše znalosti o aplikaci před tím, než byl zahájen fuzzing útok. Dělí se na hloupý fuzzing a chytrý fuzzing. Hloupý fuzzing zde již byl popsán, je to levný a rychlý způsob, jak může útočník poškodit aplikaci. Neobsahuje žádnou předešlou analýzu aplikace a útočníci do aplikace akorát posílají náhodná data a čekají, zda objeví nějakou zranitelnost. Tento útok často odhalí jen jednoduché a velmi povrchní bezpečnostní chyby, které aplikaci tolik neohrozí. Chytrý fuzzing útok narozdíl od hloupého vyžaduje určité programátorské schopnosti a porozumění infrastruktury aplikace. Zaměřuje se např. na protokoly, formáty souborů a zpracování dat. Chytrý fuzzing je dražší než hloupý fuzzing a vyžaduje více času na provedení. Čas strávený analýzou aplikace a vývojem fuzzing programu ale může vyústit v odhalení citlivějších bezpečnostních chyb. Chránit se proti tomuto typu útoku lze takzvaně ohněm proti ohni, tedy provedením fuzzing útoku na vlastní aplikaci a následným opravením vzešlých bezpečnostních chyb. Nejlepším způsobem ochrany proti fuzzing útoku je pak udržovat aktuální všechny využívané aplikace a technologie, které využíváme pro vývoj aplikace. [40]

1.4.8 Zero-Day

Termín Zero-Day odkazuje na skutečnost, že dodavatel, a tedy vývojář softwaru, se o bezpečností chybě právě dozvěděl a má tak 0 dní na opravení. Jako den 1 se označuje moment, kdy je software aplikace aktualizován. Útok tedy spočívá v tom, že útočníci využijí chyby v softwaru např. webové aplikace, ale může se jednat o jakýkoliv typ softwaru, třeba i operační systémy. Existují zde dva scénáře, jak nejčastěji útočníci využívají útoku nultého dne. Prvním případem je ten, kdy útočníci získají informace o chystané aktualizaci zabezpečení softwaru, ze které mohou vyčíst, kde se nachází bezpečností chyba, a využít ji tak ještě před tím, než je chyba odstraněna. Druhý případ je moment, kdy už je chyba odstraněna, ale i přes to se zaměří na slabinu této bezpečnostní chyby a budou se ji snažit zneužít u uživatelů, kteří si software ještě neaktualizovali. Zero-Day útok je tedy velmi obecný pojem a nejedná se přímo o specifický útok, ale jen o obecné využití bezpečnostních chyb aplikace. Chránit se proti němu dá jen z uživatelského hlediska využitím firewallu, antivirových programů a hlavně aktualizací aplikací či jiného softwaru ihned po tom, co je aktualizován, zejména pokud se jedná o aplikace, které obsahují nebo spravují citlivé osobní informace. [41]

2 FRAMEWORK BLAZOR

Blazor [42] je jeden z nejnovějších frontendových frameworků pro vytváření jednostránkových webových aplikací vytvořený společností Microsoft. Je součástí open-source platformy .NET, která je zdarma a nemusí se platit žádné poplatky ani za komerční užití. Blazor umožňuje vytvářet interaktivní dynamické webové aplikace pomocí programovacího jazyku C# namísto JavaScriptu. To ale neznamená, že JavaScript a v něm napsané knihovny není možné využít. Blazor aplikace se skládají z opakovaně použitelných komponent, které jsou implementovány pomocí jazyků C#, HTML a CSS. Jelikož klientská i serverová část aplikace lze napsat v programovacím jazyku C#, je možné skrz celou aplikaci sdílet knihovny, modely a celkově jakýkoliv kód. Pomocí frameworku Blazor je možné spustit klientskou část aplikace přímo v prohlížeči klienta díky WebAssembly [43] nebo je možné mít tuto část aplikace na serveru a komunikovat s prohlížečem pomocí frameworku SignalR [44]. Podle těchto dvou možností běhu webové aplikace se také jmenují 2 modely hostování Blazor WebAssembly a Blazor Server. Blazor využívá otevřených webových standardů bez nutnosti instalovat dodatečné pluginy nebo převedení kódu do JavaScriptu. Blazor WebAssembly i Blazor Server jsou podporovány všemi hlavními prohlížeči (Apple Safari, Google Chrome, Microsoft Edge, Mozilla Firefox). [45]

2.1 Historie

Blazor prvotně vznikl jako osobní projekt Steva Sandersona – hlavního vedoucího softwarového inženýra ve společnosti Microsoft. Poprvé byl Blazor veřejnosti představen roku 2017 na konferenci NDC v Oslu. Tato první verze byla postavena na interpretovaném běhovém prostředí .NET CIL (Common Intermediate Language) s názvem DotNetAnywhere. Kvůli velkému nadšení a velkému potenciálu byl Blazor po představení v Oslu přidán jako experimentální projekt do ASP.NET GitHub organizace. To znamenalo, že společnost Microsoft začala zkoumat poptávku a řešit technické problémy, které by bylo třeba vyřešit, kdyby se Blazor stal plnohodnotným produktem nabízeným veřejnosti. Po přijetí mezi oficiální Microsoft produkty byl Blazor v roce 2018 přepsán od základu týmem spravujícím ASP.NET. DotNetAnywhere byl zahozen a nahrazen mnohem pokročilejší platformou Mono, která je součástí společnosti Microsoft již od roku 2016 a je oficiálním běhovým prostředím pro klientské frameworky, jako je např. Xamarin nebo Unity. Obsahuje také daleko více knihoven a funkcí v rámci .NET CIL. Oficiální preview bylo zveřejněno v dubnu roku 2019 a prvotní produkční verze byla vydána v květnu roku 2020. Momentálně

nejnovější verzi Blazoru je možné najít v rámci ASP.NET Core 6 frameworku zveřejněném v roce 2021. [46]

2.2 Blazor Server

Jak již z názvu vypovídá, při použití hostovacího modelu Blazor Server se webová aplikace spouští na serveru v rámci ASP.NET Core aplikace. Jakákoliv aktualizace uživatelského rozhraní probíhá prostřednictvím SignalR, což je open-source knihovna pro aplikace založené na ASP.NET, která slouží pro asynchronní obousměrnou komunikaci mezi serverovou a klientskou částí webové aplikace. Výhodou využití Blazor serveru je rychlé načtení a renderování uživatelského rozhraní, jelikož všechny knihovny jsou uloženy na straně serveru a nemusí se čekat na jejich stažení do prohlížeče. Jelikož je všechen kód na straně serveru, je možné plně využít funkcionalit ASP.NET frameworku jako např. ladění. Nevýhodou pak může být vyšší latence, jelikož každá uživatelská interakce s aplikací potřebuje využít sítě ke komunikaci se serverem. Aplikace nedokáže fungovat v offline režimu, tedy pokud uživateli selže připojení, tak aplikace přestane úplně fungovat. Problémem je také škálovatelnost, s velkým množstvím uživatelů bude využíváno hodně serverových zdrojů. [47]

2.3 Blazor WebAssembly

Daleko populárnějším hostovacím modelem, díky kterému je Blazor konkurenceschopný, je Blazor WebAssembly. Webové aplikace využívající tohoto hostovacího modelu se spouští v prohlížeči na straně uživatele v prostředí .NET založeném na WebAssembly. WebAssembly [43] je nový standard popisující přenositelný formát binárního kódu optimalizovaný pro spuštění v prohlížečích pro klientské i serverové aplikace. Snaží se podporovat jakýkoliv programovací jazyk na jakémkoliv operačním systému. Při prvotním navštívení webu je aplikace Blazor, všechny její závislosti i .NET runtime stažen do prohlížeče uživatele. Webové aplikace se spouští přímo ve vlákně uživatelského rozhraní prohlížeče a v rámci tohoto samého vlákna se také zpracovávají veškeré události a aktualizace uživatelského rozhraní. Aplikační prostředky jsou pak uloženy jako statické soubory dle druhu nasazení buď na serveru, nebo ve službě, která je schopná uživateli statický obsah poskytnout. Webová aplikace Blazor WebAssembly může být nasazena tzv. standalone neboli samostatným stylem, což znamená, že neobsahuje serverovou část aplikace, která by poskytovala služby a soubory uživatelské části. Naopak aplikace může být nasazena jako hosted neboli hostovaná, která serverovou část aplikace ASP.NET Core obsahuje a komunikuje s klientskou částí buď

pomocí knihovny SignalR, nebo voláním webového API. Aby aplikace mohla fungovat, musí ji ale podporovat prohlížeč uživatele, její rychlost také do jisté míry závisí na uživatelském zařízení a prvotní stažení aplikace do prohlížeče může trvat delší dobu. Naopak není zde závislost na .NET serveru jako u Blazor Server hostovacího modelu, takže aplikace může běžet i když je server offline a server není tolik zatížen, jelikož aplikace využívá prostředky uživatele pro fungování. [47]

2.4 Funkcionality

- Razor pages – Blazor využívá Razor syntaxi ve svých komponentách pro tvorbu dynamických uživatelských rozhraní, díky tomu je možné využívat C# kód v kombinaci s HTML a CSS pro vývoj uživatelského rozhraní.
- Hot reload – nově přidaná funkcionalita v rámci .NET 6, která dovoluje programátorům provádět změny uživatelského rozhraní za běhu aplikace bez ztráty stavu aplikace.
- Lazy loading podpora – technika pro načítání knihoven a zdrojů na vyžádání. Zkracuje počáteční dobu načítání aplikace tím, že aplikace odkládá požadavek na stažení daných zdrojů až na moment, kdy jsou potřeba. V základu tato funkcionalita v Blazoru není zapnutá a Blazor aplikace musí být instruována, kdy má zdroje načíst.
- CSS izolace – technika umožňující stylovat komponenty jednotlivě narozdíl od jednoho souboru obsahujícího všechny styly. Kód je přehlednější a je možné znovupoužít názvy CSS tříd a id mezi jednotlivými komponentami.
- AoT kompilace – ahead-of-time kompilace (kompilace v předstihu) umožňuje kompilovat .NET kód přímo do WebAssembly, což vede ke zlepšení výkonu za běhu na úkor větší velikosti aplikace. Během převodu se také odstraňuje nepoužitý .NET WebAssembly runtime kód pro zmenšení velikosti aplikace.
- Podpora virtualizace – technika, která omezuje vykreslování uživatelského rozhraní tak, aby se zobrazovaly pouze části aktuálně viditelné uživatelem.

3 FRAMEWORK ANGULAR

Angular je frontendový framework zaměřený na tvorbu komplexních a efektivních jednostránkových webových aplikací. Je open-source a byl vyvinut společností Google. Je postaven na programovacím jazyku TypeScript, což je nadstavba jazyka JavaScript, která JavaScript podstatně zjednodušuje, usnadňuje jeho čtení a ladění a podporuje základní datové typy, ale i typy jako třídy nebo rozhraní. Angular je založen na znovupoužitelných komponentách rozdělených na HTML, CSS a TypeScript část, tedy každá komponenta se skládá ze 3 souborů, první obsahuje kód, druhý je tvořen jazykem HTML a třetí obsahuje stylování dané komponenty. Framework obsahuje také celou řadu užitečných nástrojů, jako je třeba ng-bootstrap, Angular material, což je knihovna již hotových komponent pro uživatelské rozhraní dle Google specifikací [48]. A hlavně obsahuje vlastní CLI (Command-line interface), díky kterému je velmi jednoduché vytvořit nebo nastavit Angular projekt nebo generovat komponenty, což eliminuje psaní boilerplate kódu. Angular CLI obsahuje také server, který sestaví aplikaci a lokálně ji spustí v prohlížeči. Velmi používaným návrhovým vzorem v Angularu je reaktivní programování, které je implementováno díky RxJS knihovně, která umožňuje implementaci asynchronního načítání dat.

3.1 Historie

Angular byl původně vytvořen zaměstnancem společnosti Google Miškem Hevery jako vedlejší projekt pro jednodušší tvorbu dalších projektů, na kterých v té době pracoval. První verze byla vydána v roce 2010 jako open-source projekt s názvem AngularJS. Projekt rychle nabral na popularitě, ale objevily se nové pokroky a standardy v JavaScriptu a AngularJS začínal být pozadu, také kvůli kapacitní vytíženosti vývojářského týmu. V roce 2016 přichází nová verze s názvem Angular 2, což byl kompletní přepis celého AngularJS frameworku, a bohužel neexistovala jednoduchá cesta, jak přejít z AngularJS na Angular 2, programátorské týmy tedy musely často projekty napsané v AngularJS téměř celé přepsat od začátku v novém frameworku. Tato změna snesla velkou vlnu kritiky. Angular se snaží zveřejnit novou verzi každých 6 měsíců, každá nová verze je nyní již zpětně kompatibilní. Postupem času byla vydána verze 4 s vylepšeným view-enginem a knihovnou pro jednoduché tvoření HTTP požadavků. Verze 5 přidala například podporu pro progresivní webové aplikace. Verze 7 byla primárně zaměřena na vylepšení výkonu. Nejnovější verzí je Angular 13 zveřejněná v listopadu roku 2021. Tato verze už nepodporuje zastaralý view-engine, ale využívá Angular Ivy pro generování obsahu. Podporuje také novější verze RxJS knihovny a

jazyka TypeScript. Obsahuje také nové API pro tvorbu komponent nebo vylepšené CLI a spoustu dalších pokroků. [49]

3.2 Architektura

Každá webové aplikace, která využívá frameworku Angular, se opírá o základní stavební kameny, kterým se říká komponenty. Komponenty jsou pak uspořádány do tzv. NgModules neboli modulů, které seskupují související kód do funkčních sad. Každá webová aplikace je pak tvořena sadou modulů. Pro základní fungování musí mít Angular aplikace vždy alespoň jeden kořenový modul, který umožňuje bootstrapping, ale obvykle má i mnoho dalších modulů s různými funkcionalitami. Angular je tedy postaven na komponentové architektuře, kde každá komponenta se skládá z view, které reprezentuje uživatelské rozhraní a prezentuje uživatelům data. Dále se skládá z komponentové logiky, která mimo jiné využívá modely a zprostředkovává komunikaci mezi view a modelem. Modely udržují data a představují business logiku aplikace. Angular také umožňuje obousměrnou datovou vazbu (Two-way data binding), která zajišťuje synchronizaci dat mezi modelem a view. Jakmile se změní data v modelu, tak je změna ihned reflektována ve view, stejně tak to funguje i naopak, pokud jsou data změněna uživatelem ve view, tak jsou data upravena v modelu. [50]

3.2.1 Komponenty

Každá webová aplikace Angular má alespoň jednu komponentu, a to komponentu kořenovou, která spojuje hierarchii komponent s objektovým modelem dokumentu stránky (DOM). Každá komponenta pak definuje třídu, která obsahuje data a logiku, a je spojena s HTML šablonou, která definuje díky HTML tagům to, jak bude strukturované uživatelské rozhraní. Komponenty také využívají služby, které poskytují specifickou funkcionalitu, která úplně nemusí souviset s danou komponentou. Poskytovatele služeb lze do komponent injektovat jako závislosti, díky čemuž je pak kód modulární, znovupoužitelný a efektivní. [50]

3.2.2 Moduly

Angular moduly se od JavaScript modulů liší a doplňují je. Angular modul deklaruje kontext kompilace pro sadu komponent, které obsahují úzce související sadu funkcionalit. Modul může své komponenty spojovat se souvisejícím kódem, například službami, a vytvářet tak funkční jednotky. Každá webová aplikace Angular má kořenový modul pojmenovaný AppModule, který poskytuje zaváděcí mechanizmus (bootstrapping), který spouští aplikaci. Modul může importovat funkcionality z jiných modulů a umožnit exportování své vlastní

funkcionality do jiných modulů. Uspořádání kódu do funkčních modulů podporuje znovupoužitelnost a pomáhá při řízení vývoje složitých aplikací. Je podporována také technika lazy-loading a moduly lze načítat na vyžádání. [50]

3.2.3 Routing

V rámci Angular frameworku je zabudovaný v základu routing modul, což je služba, která umožňuje definovat navigační cesty mezi různými stavy aplikace. Modul je vytvořen dle známých prohlížečových konvencí:

- Uživatel zadá URL adresu do adresního řádku a bude přesměrován na odpovídající stránku.
- Uživatel klikne na odkaz na stránce a bude přesměrován na odpovídající stránku.
- Uživatel klikne na tlačítko zpět nebo vpřed a uživatel bude přesměrován zpět nebo vpřed dle historie zobrazených stránek.

Routing tedy mapuje cesty podobné URL na komponenty místo stránek. Když uživatel provede akci, která by v prohlížeči načetla novou stránku, routing modul zachytí chování prohlížeče a přesměruje uživatele dle aplikační logiky. Pokud by měla být načtena komponenta, která využívá funkci modulu, který ještě nebyl načten, je možné modul načíst díky lazy-loading technice. Pro definování navigačních pravidel, které upravují, které komponenty se za jakých podmínek mohou uživatelům zobrazit nebo skrýt, stačí přiřadit v routing modulu navigační cestu ke komponentám a k nim navigační logiku. [50]

3.3 Funkcionality

- SCSS a SASS – Angular podporuje stylování skrz SCSS i SASS, což jsou vylepšené verze základního programovacího jazyka pro stylování CSS. Tyto vylepšené verze obsahují plno funkcionalit, které CSS nemá jako např. referencování elementů, proměnné, interpolace, placeholdery, funkce a další.
- Direktivy – pomocí direktiv je možné vytvářet vlastní HTML třídy, které přidávají další chování prvkům, které nimi byly dekorovány. Direktivy lze tvořit vlastní, ale Angular obsahuje i vestavěné direktivy jako např ngClass.
- RxJS – Angular podporuje reaktivní programování a těžce při tom spoléhá na knihovnu RxJS, která poskytuje implementaci typu observable a usnadňuje tak sestavování asynchronního kódu nebo kódu založeném na zpětných voláních. Knihovna také poskytuje obslužné funkce pro práci s observable objekty.

- Angular material – knihovna komponent uživatelského rozhraní, které lze používat v Angular aplikacích. Řídí se specifikacemi Material Design od společnosti Google, ale jejich vzhled lze i přizpůsobit.
- Ng-bootstrap – open-source repositář, který poskytuje nativní widgety a komponenty tvořené pomocí Bootstrap 5 CSS.
- HttpInterceptor – rozhraní, které umožňuje zachycovat veškeré HTTP požadavky a odpovědi serveru v rámci Angular aplikace. Během zachycení lze HTTP požadavky a odpovědi ověřovat, ukládat do cache paměti, modifikovat jejich hlavičky nebo na základě jejich obsahu a atributů spouštět vlastní logiku.
- Ivy – kompilační a renderovací engine nové generace, který pomáhá převést kód napsaný v jazyce TypeScript a HTML do standardních instrukcí jazyka JavaScript a pomáhá tak prohlížeči s interpretací aplikace. Ivy navíc odstraňuje nepoužívaný kód, čímž zmenšuje velikost aplikace a zvyšuje její rychlost.

II. PRAKTICKÁ ČÁST

4 NÁVRH WEBOVÉ APLIKACE

Webová aplikace je koncipována jako samosjednávač cestovního pojištění a byla inspirována již existujícími samosjednávacími kalkulačkami pojišťoven. Aplikace je jednostránková (single-page) a neobsahuje tedy více stránek, ale je rozdělena do 5 kroků, kterými uživatel postupně prochází, a na základě uživatelských vstupů se dynamicky mění obsah a stav stránky. Hlavní funkcí aplikace je vytvořit uživateli nabídku 3 pojišťovacích balíčků na základě uživatelem zvolených parametrů a následné sjednání smlouvy jednoho z vybraných balíčků. Jedná se o demo sloužící k porovnání implementací základních funkcí frameworků Blazor a Angular pro implementaci jednostránkové aplikace z oblasti pojišťovnictví. Z tohoto důvodu aplikace neřeší např. možnost smlouvu mazat ze systému nebo obecná nařízení o ochraně osobních údajů (GDPR). V produkční verzi aplikace by uživatel tuto možnost měl např. zasláním emailu a musel by také souhlasit se zpracováním osobních údajů. Návrh aplikace začal vytvořením funkčních a nefunkčních požadavků, následně vytvořením scénářů případů užití a návrhem zabezpečení.

4.1 Funkční požadavky

Funkční požadavky definují funkce, které musí být aplikace schopna vykonat. Níže je uveden seznam funkčních požadavků navržené aplikace:

- R1: Systém musí umožnit výběr typu pojištění. Na výběr musí být jedna cesta, více cest za rok a roční pobyt.
- R2: Systém musí umožnit výběr typu destinace. Na výběr musí být Česká republika, Evropa a celý svět.
- R3: Systém musí umožnit zvolit datum odjezdu a návratu.
- R4: Systém musí umožnit vybrat typ pojistníka mezi osobu fyzickou, právnickou nebo osobu samostatně výdělečně činnou a na základě toho změnit požadované údaje o pojistníkovi.
- R5: Systém musí umožnit pojistit maximálně 10 dalších pojišťovaných osob.
- R6: Systém musí umožnit výběr jednotlivých připojištění.
- R7: Systém musí generovat tři typy balíčků dle definovaných limitů plnění a zahrnovat zvolená připojištění uživatelem.
- R8: Systém musí vypočítat cenu každého balíčku dle zvolených parametrů uživatelem.

- R9: Systém musí umožnit uživateli výběr jednoho z pojišťovacích balíčků.
- R10: Systém musí umožnit uživateli sjednat pojistnou smlouvu cestovního pojištění dle uživatelem zadaných a vybraných specifikací.
- R11: Systém musí generovat smlouvu ve formátu pdf.
- R12: Systém musí po vytvoření smlouvy zaslat email se smlouvou na uživatelem zadaný email.
- R13: Systém musí evidovat vygenerované smlouvy do databáze.
- R14: Systém musí umožnit uživateli průchod mezi jednotlivými stavy aplikace dopředu i zpět.
- R15: Systém musí zachovávat uživatelem vyplněná data skrz jednotlivé stavy aplikace.

4.2 Nefunkční požadavky

Nefunkční požadavky definují vlastnosti aplikace jako celku a její omezení. Níže je uveden seznam nefunkčních požadavků navržené aplikace:

- NF1: Serverová část webové aplikace (backend) musí podporovat operační systémy Linux, Windows a macOS.
- NF2: Klientská část webové aplikace (frontend) musí podporovat prohlížeče Google Chrome, Mozilla Firefox, Microsoft Edge a Safari.
- NF3: Systém musí být dostupný v českém jazyce.
- NF4: Systém musí používat Korunu českou jako základní měnu.
- NF5: Systém musí být rozšiřitelný a modifikovatelný.

4.3 Scénáře případů užití

Případy užití popisují činnost, kterou určitý aktér vykonává v rámci aplikace. Jsou psány z pohledu primárního aktéra a obsahují hlavní scénáře a alternativní scénáře. Scénáře jsou pak posloupnosti kroků, které musí aktér a systém vykonat pro splnění dané činnosti. Hlavní scénář popisuje průběh bez výskytu výjimek a alternativní scénáře tyto výjimky zachycují. Níže je uveden seznam scénářů případů užití navržené aplikace.

Tabulka 1 - Scénář případu užití: Vygenerování nabídek pojištění

Název: Vygenerování nabídek pojištění		
ID: UC1		
Charakteristika: Vygenerování nabídek pojištění na základě uživatelem specifikovaných hodnot.		
Primární aktér: Uživatel		
Vedlejší aktéři: Žádní		
Vstupní podmínky: Uživatel se nachází na úvodní stránce webové aplikace.		
Výstupní podmínky: Uživateli budou prezentovány 3 balíčky pojištění s rozdílnými cenami na základě uživatelem specifikovaných hodnot.		
Hlavní scénář:		
Krok	**Aktér/Systém**	**Popis**
1.	Uživatel	Uživatel vyplní základní informace o cestě a o pojistníkovi (podle vybraného typu osoby).
2.	Uživatel	Uživatel klikne na tlačítko „Pokračovat“.
3.	Systém	Systém zvaliduje uživatelem vyplněné údaje.
4.	Systém	Systém zobrazí uživateli nabídku všech připojištění.
5.	Uživatel	Uživatel zaškrtne checkbox u připojištění, která chce v rámci pojištění mít zahrnuta.
6.	Uživatel	Uživatel klikne na tlačítko „Vytvořit nabídku“.

7.	Systém	Systém zobrazí uživateli 3 balíčky, jejich cenu a limity plnění jednotlivých připojištění.
Alternativní scénáře: UC1_1a – Přidání další pojištěné osoby UC1_1b – Odstranění přidané pojištěné osoby UC1_3a – Nevyplněné nebo špatně vyplněné údaje		

Tabulka 2 – Alternativní scénář případu užití: Přidání další pojištěné osoby

Název – alternativní scénář: Vygenerování nabídek pojištění – Přidání další pojištěné osoby		
ID: UC1_1a		
Charakteristika: Přidání další pojištěné osoby.		
Alternativní scénář:		
Krok	**Aktér/Systém**	**Popis**
1.	Uživatel	Uživatel klikne na tlačítko „Přidat osobu“.
2.	Systém	Systém zobrazí pole k vyplnění informací další pojišťované osoby.
3.	Uživatel	Uživatel vyplní údaje o další pojišťované osobě.

Tabulka 3 – Alternativní scénář případu užití: odstranění přidané pojištěné osoby

Název – alternativní scénář: Vygenerování nabídek pojištění – Odstranění přidané pojištěné osoby
ID: UC1_1b
Charakteristika: Odstranění přidané pojišťované osoby.
Alternativní scénář:

Krok	**Aktér/Systém**	**Popis**
1.	Uživatel	Uživatel klikne na tlačítko „Odstranit“ vedle pojišťované osoby, kterou chce odstranit.
2.	Systém	Systém odstraní uživatelem specifikovanou osobu ze seznamu dalších pojišťovaných osob.

Tabulka 4 - Alternativní scénář případu užití: Nevyplněné nebo špatně vyplněné údaje

Název – alternativní scénář: Vygenerování nabídek pojištění – Nevyplněné nebo špatně vyplněné údaje		
ID: UC1_3a		
Charakteristika: Validace špatně zadaných údajů.		
Alternativní scénář:		
Krok	**Aktér/Systém**	**Popis**
1.	Systém	Systém označí červeně pole, ve kterých jsou špatně zadané údaje.
2.	Systém	Systém zobrazí validační hlášku vedle polí, které obsahují špatně zadané údaje.

Tabulka 5 - Scénář případu užití: Sjednání smlouvy konkrétního pojišťovacího balíčku

Název: Sjednání smlouvy konkrétního pojišťovacího balíčku
ID: UC2
Charakteristika: Vytvoření smlouvy cestovního pojištění na základě uživatelem zvoleného pojišťovacího balíčku.
Primární aktér: Uživatel
Vedlejší aktéři: Žádní

Vstupní podmínky: Uživatel se nachází na obrazovce s výběrem pojišťovacích balíčků.		
Výstupní podmínky: Bude vytvořena smlouva na základě uživatelem zvoleného pojišťovacího balíčku, která mu bude společně s platebními pokyny zaslána na email.		
Hlavní scénář:		
Krok	**Aktér/Systém**	**Popis**
1.	Uživatel	Uživatel klikne na tlačítko „Vybrat balíček“ pod balíčkem, který chce sjednat.
2.	Systém	Systém zobrazí uživateli formulář pro vyplnění kontaktních údajů.
3.	Uživatel	Uživatel vyplní kontaktní údaje.
4.	Uživatel	Uživatel klikne na tlačítko „Sjednat smlouvu“.
5.	Systém	Systém zvaliduje uživatelem vyplněné údaje.
6.	Systém	Systém zobrazí uživateli potvrzení o vytvoření smlouvy.
7.	Systém	Systém vygeneruje smlouvu ve formátu pdf a zašle ji na email zadaný uživatelem.
8.	Systém	Systém uloží smlouvu do databáze.
Alternativní scénáře: UC2_5a – Nevyplněné nebo špatně vyplněné údaje		

Tabulka 6 - Alternativní scénář případu užití: Nevyplněné nebo špatně vyplněné údaje

Název – alternativní scénář: Sjednání smlouvy konkrétního pojišťovacího balíčku – Nevyplněné nebo špatně vyplněné údaje
ID: UC2_5a
Charakteristika: Validace špatně zadaných údajů.

Alternativní scénář:		
Krok	**Aktér/Systém**	**Popis**
1.	Systém	Systém označí červeně pole, ve kterých jsou špatně zadané údaje.
2.	Systém	Systém zobrazí validační hlášku vedle polí, které obsahují špatně zadané údaje.

4.4 Zabezpečení

Webová aplikace je zabezpečena hned několika způsoby. Aplikace není zranitelná vůči útoku typu cross-site scripting, jelikož v žádné části aplikace se uživatelské vstupy nevkládají do HTML šablony nebo do JavaScript kódu. Navíc frameworky Blazor i Angular automaticky kódují všechen zobrazovaný text a škodlivý kód tedy není vůbec interpretován jako kód. [51][52] Aplikace je také ze své podstaty odolná vůči útoku brute force, jelikož se jedná o samosjednávač a není nutná registrace a přihlášení uživatele. Uživatel je tedy anonymní a nemá v aplikaci uložené žádné údaje, které by se daly brute force útokem prolomit. Zranitelnost vůči útoku OS command injection také neexistuje, jelikož aplikace v žádném bodu jejího běhu nevolá příkazy operačního systému. Útočník tedy nemá šanci při zadání škodlivých dat spustit jakýkoliv příkaz operačního systému. Proti útoku typu SQL injection je aplikace chráněna využitím knihovny Entity Framework. Ta sama o sobě nebrání tomuto typu útoku, stále je možné i s využitím této knihovny napsat SQL dotazy, které budou zranitelné vůči tomuto útoku. V rámci implementace aplikace byly ale zvoleny pro práci s databází metody, které jsou parametrizované a s databází je tak komunikováno bezpečným způsobem. Proti útoku man-in-the-middle je aplikace chráněna využitím šifrované komunikace mezi klientem a serverem. Je využito podporovaných funkcí frameworku ASP.NET Core pro přesměrování všech HTTP požadavků na HTTPS požadavky. Zároveň je využito protokolu HSTS (HTTP Strict Transport Security) s nastavenou hodnotou max-age na 1 rok. Tento protokol způsobuje to, že si prohlížeč zapamatuje webovou stránku a spojí se s ní po specifikovanou dobu (1 rok) jen skrz protokol HTTPS. Šifrovaná komunikace skrz SSL je také zajištěna při zasílání emailu se smlouvou. Aplikace také využívá serverových i klientských validací pro omezení vstupů jen na datové typy, které jsou od vstupu očekávané.

5 KLÍČOVÉ ČÁSTÍ WEBOVÉ APLIKACE

Při tvorbě aplikace byly vytvořeny celkem 4 projekty. 2 projekty klientské části aplikace, jeden ve frameworku Blazor WebAssembly a druhý ve frameworku Angular. Dále projekt serverové části aplikace ve frameworku ASP.NET Core a sdílený projekt Shared, který slouží v rámci Blazor frameworku pro sdílení kódu mezi klientskou a serverovou částí aplikace. Klíčovými prvky serverové části aplikace jsou endpointy, služby (hlavní pojišťovací služba, zasílání emailu, tvorba smlouvy ve formátu pdf) a propojení s databází. V klientských projektech Blazor WebAssembly a Angular jsou klíčovými prvky jejich komponenty pro navigaci, zadání základních informací, přidání dalších pojišťovaných osob, výběr připojištění, zobrazení a výběr balíčku, zadání kontaktních údajů, potvrzení sjednání smlouvy a nakonec také jejich modely a validační pravidla aplikace.

5.1 Endpointy

Aplikace obsahuje jeden controller s názvem InsuranceController, který slouží pro komunikaci s klientskou částí aplikace. Cesta pro komunikaci s tímto controllerem je definována v Route atributu a názvy endpointů jsou definovány v atributech HttpGet a HttpPost, které specifikují, o jaký typ metody se jedná. Cesta pro komunikaci s controllerem a jeho metodami je tedy {URL adresa webu}/InsuranceController/{Endpoint}. Controller využívá služby InsuranceService, která obsahuje veškerou pojišťovací logiku a data. [53] Controller obsahuje 3 endpointy:

- *GetInsuranceItems()* – metoda typu GET, která vrátí zpět seznam připojištění se základními daty.
- *CalculateAllPackages()* – metoda typu POST, která přijímá model InsuranceData, obsahující veškerá uživatelem vyplněná data a vrátí zpět seznam 3 balíčků s vypočítanou cenou.
- *CreateContract()* - metoda typu POST, která přijímá model InsuranceData, obsahující veškerá uživatelem vyplněná data a vybraný pojišťovací balíček. Nevrací zpět žádnou informaci, ale jen spouští logiku pro vytvoření smlouvy. [53]

```
[ApiController]
[Route("InsuranceController")]
public class InsuranceController : ControllerBase
{
	private readonly IInsuranceService _insuranceService;

	public InsuranceController(IInsuranceService insuranceService)
	{
		_insuranceService = insuranceService;
	}

	[HttpGet("GetInsuranceItems")]
	public List<InsuranceItem> GetInsuranceItems()
	{
		return _insuranceService.GetInsuranceItems();
	}

	[HttpPost("CalculateAllPackages")]
	public List<Package> CalculateAllPackages([FromBody] InsuranceData insuranceData)
	{
		return _insuranceService.CalculateAllPackages(insuranceData);
	}

	[HttpPost("CreateContract")]
	public void CreateContract([FromBody] InsuranceData insuranceData)
	{
		_insuranceService.CreateContract(insuranceData);
	}
}
```

5.2 Služby (services)

Aplikace je tvořena celkem 3 službami. Služba InsuranceService obsahuje hlavní pojišťovací logiku a data o cenách a limitech plnění pojištění. Služba EmailService se stará o zasílání emailů na uživatele a nakonec služba PdfService, která se stará o generování pdf souborů. Všechny služby jsou implementací rozhraní, které pak díky dependency injection můžeme využívat kdekoliv v aplikaci. Aby toto bylo možné, tak se musí služby zaregistrovat v metodě *ConfigureServices()* v třídě Startup.

```
services.AddTransient<IInsuranceService, InsuranceService>();
services.AddSingleton<IEmailService, EmailService>();
services.AddSingleton<IPdfService, PdfService>();
```

5.2.1 Hlavní pojišťovací služba (InsuranceService)

Metoda *GetInsuranceItems()* vytvoří list objektů InsuranceItem a následně daný list naplní hodnotami za pomoci privátní metody *SetInsuranceItem()*, která zjednodušuje zápis pro přidávání připojištění do listu.

```
public List<InsuranceItem> GetInsuranceItems()
{
	var insuranceItems = new List<InsuranceItem>();

	SetInsuranceItem(insuranceItems, InsuranceItemType.MedicalExpenses, true, true,
false, 0);
	SetInsuranceItem(insuranceItems, InsuranceItemType.RecreationalSports, true,
true, true, 0);
	SetInsuranceItem(insuranceItems, InsuranceItemType.TravelAssistance, true, true,
true, 0);
	SetInsuranceItem(insuranceItems, InsuranceItemType.LegalAssistance, false, false,
true, 10);
	SetInsuranceItem(insuranceItems, InsuranceItemType.CarAssistance, false, false,
true, 15);
	SetInsuranceItem(insuranceItems, InsuranceItemType.Covid, false, false, true,
20);
	SetInsuranceItem(insuranceItems, InsuranceItemType.RiskySports, false, false,
false, 20);
	SetInsuranceItem(insuranceItems, InsuranceItemType.WorkAndStudy, false, false,
false, 25);
	SetInsuranceItem(insuranceItems, InsuranceItemType.Responsibility, false, false,
false, 15);
	SetInsuranceItem(insuranceItems, InsuranceItemType.Accident, false, false, false,
30);
	SetInsuranceItem(insuranceItems, InsuranceItemType.RentalCar, false, false,
false, 10);
	SetInsuranceItem(insuranceItems, InsuranceItemType.Luggage, false, false, false,
15);
	SetInsuranceItem(insuranceItems, InsuranceItemType.FlightDelay, false, false,
false, 5);

	return insuranceItems;
}

private void SetInsuranceItem(
	List<InsuranceItem> insuredItems,
	InsuranceItemType falše,
	bool isSelected,
	bool isIncluded,
	bool isBoolType,
	int pricePerPerson
	)
{
	insuredItems.Add(new InsuranceItem()
	{
		InsuranceItemType = insuranceItemType,
		IsSelected = isSelected,
		IsIncluded = isIncluded,
		IsBoolType = isBoolType,
		PricePerPerson = pricePerPerson
	});
}
```

Metoda *CalculateAllPackages()* přijímá objekt typu InsuranceData, který použije pro výpočet ceny balíčku. Metoda pro každý balíček nejdříve překopíruje vybrané připojištění do každého balíčku a pro každé takto překopírované připojištění získá ze statických dat, pomocí privátní metody *GetInsuredItemInsuredSum()*, limit plnění daného připojištění pro daný typ balíčku. Následně je pomocí další privátní metody *CalculatePackagePrice()* spočítána cena jednotlivých balíčků na základě vybraných připojištění, typu balíčku, typu destinace, typu pojištění a počtu pojišťovaných osob.

```
public List<Package> CalculateAllPackages(InsuranceData insuranceData)
{
	foreach (var package in Packages)
	{
		foreach (var insuranceItem in insuranceData.InsuranceItems.Where(item =>
item.IsSelected))
		{
			var item = new InsuranceItem()
			{
				InsuranceItemType = insuranceItem.InsuranceItemType,
				PricePerPerson = insuranceItem.PricePerPerson,
				InsuredSum = GetInsuredItemInsuredSum(insuranceItem.Insuran-
ceItemType, package.PackageType),
				IsSelected = insuranceItem.IsSelected,
				IsIncluded = insuranceItem.IsIncluded,
				IsBoolType = insuranceItem.IsBoolType,
			};

			package.InsuranceItems.Add(item);
		}

		package.Price = CalculatePackagePrice(insuranceData, package);
	}

	return Packages;
}

private int CalculatePackagePrice(InsuranceData insuranceData, Package package)
{
	var packagePrice = package.Price;

	packagePrice += package.InsuranceItems.Sum(item => item.PricePerPerson) * in-
suranceData.NumberOfInsuredPersons();
	packagePrice += InsuranceTypePrices.GetValueOrDefault(insuranceData.Insurance-
Type);
	packagePrice += DestinationPrices.GetValueOrDefault(insuranceData.Destination);

	if (insuranceData.InsuranceType == InsuranceType.SingleTravel)
	{
		packagePrice += insuranceData.GetInsuranceNumberOfDays() * 2;
	}
	return packagePrice;
}

private int GetInsuredItemInsuredSum(InsuranceItemType insuranceItemType, PackageType
packageType)
{
	return InsuredSums.GetValueOrDefault(insuranceItemType)?.GetValueOrDefault(pac-
kageType) ?? 0;
}
```

Metoda *CreateContract()* přijímá objekt typu InsuranceData, který použije pro vygenerování pdf smlouvy a zaslání emailu na klienta. Metoda nejdříve využije injektované služby PdfService a její metody *CreateContractFile()* pro získání pdf souboru, který následně předá do metody *SendContractEmail()* služby EmailService. Metoda *SendContractEmail()* následně vrátí, zda se povedlo email odeslat, a uloží hodnotu do proměnné *wasEmailSentToClient*. V poslední řadě je využito metody *SaveContractToDatabase()*, které je předán objekt InsuranceData a proměnná *wasEmailSentToClient*. Tato metoda pak vytvoří instanci třídy ContractDto, kterou následně uloží do databáze skrz EntityFramework a jím poskytované metody *Add()* a *SaveChanges()*.

```
public void CreateContract(InsuranceData insuranceData)
{
	var contractFile = _pdfService.CreateContractFile(insuranceData);
	var wasEmailSentToClient = _emailService.SendContractEmail(insuranceData, con-
tractFile);
	SaveContractToDatabase(insuranceData, wasEmailSentToClient);
}

private void SaveContractToDatabase(InsuranceData insuranceData, bool wasEmailSent-
ToClient)
{
	var jsonContractDataDto = new JsonContractDataDto()
	{
		InsuranceItems = insuranceData.InsuranceItems,
		MainInsuredPerson = insuranceData.MainInsuredPerson,
		InsuredPersons = insuranceData.InsuredPersons,
		PickedPackage = insuranceData.PickedPackage,
	};

	var contractDto = new ContractDto()
	{
		InsuranceType = insuranceData.InsuranceType,
		DateCreated = DateTime.Now,
		Destination = insuranceData.Destination,
		StartDate = insuranceData.StartDate,
		EndDate = insuranceData.EndDate,
		InsuranceNumberOfDays = insuranceData.GetInsuranceNumberOfDays(),
		NumberOfInsuredPersons = insuranceData.NumberOfInsuredPersons(),
		JsonContractData = JsonSerializer.Serialize(jsonContractDataDto),
		WasEmailSentToClient = wasEmailSentToClient,
	};

	_databaseContext.Contracts.Add(contractDto);
	_databaseContext.SaveChanges();
}
```

Statická data jsou ve službě uložena v kolekci klíčů a hodnot známé také jako Dictionary.

```
private Dictionary<InsuranceType, int> InsuranceTypePrices { get; set; } = new Dictio-
nary<InsuranceType, int>
{
	{InsuranceType.SingleTravel, 10 },
	{InsuranceType.MultipleTravelsInYear, 150 },
	{InsuranceType.AnnualStay, 250 },
};
```

5.2.2 Zasílání emailu (EmailService)

V službě EmailService je pro zasílání emailu definována metoda *SendContractEmail()*, která je obalena blokem try-catch pro zachycení jakéhokoliv chybového stavu, který by znemožnil odeslání emailu. Pokud k tomuto chybovému stavu dojde, tak metoda vrátí hodnotu false indikující, že se email nepodařilo odeslat. V rámci metody je vytvořena instance třídy MailMessage, u které je nastaven odesílatel, předmět emailu, příjemce emailu, do přílohy je vložena smlouva a tělo emailu je vyplněno zavoláním metody *GetContractEmailBody()*, která vrací zformátovaný text typu string s cenou smlouvy. Následně je vytvořena instance třídy SmtpClient za použití hodnot uložených v souboru appsettings.json a je specifikováno, aby zaslání emailu bylo zašifrováno pomocí SSL. Hostem smtp klienta je smtp.gmail.com, je využito portu 587 a emailu jakub.horak.development@gmail.com, který byl vytvořen specificky pro tuto bakalářskou práci.

```
public bool SendContractEmail(InsuranceData insuranceData, byte[] contractFile)
{
	var wasEmailSentToClient = false;

	try
	{
		var email = new MailMessage()
		{
			From = new MailAddress(_configuration.GetVa-
lue<string>("Smtp:email")),
			Subject = "Cestovní pojištění - Smlouva",
			Body = GetContractEmailBody(insuranceData.PickedPackage.Price),
		};
		email.To.Add(insuranceData.MainInsuredPerson.Email);
		email.Attachments.Add(new Attachment(new MemoryStream(contractFile),
"Smlouva.pdf"));

		using (SmtpClient smtp = new SmtpClient(_configuration.GetVa-
lue<string>("Smtp:host"), _configuration.GetValue<int>("Smtp:port")))
		{
			smtp.EnableSsl = true;
			smtp.Credentials = new System.Net.NetworkCredential(_configura-
tion.GetValue<string>("Smtp:email"), _configuration.GetValue<string>("Smtp:password"));
			smtp.Send(email);
		}

		wasEmailSentToClient = true;
	}
	catch {}

	return wasEmailSentToClient;
}
```

5.2.3 Tvorba smlouvy ve formátu pdf (PdfService)

Pro vytvoření smlouvy ve formátu pdf je využito knihovny iText 7 [54], která umožňuje jednoduše tvořit pdf soubor skrz přidávání objektů s daty do instance dokumentu. Nejdříve bylo nutné vytvořit instanci třídy Document. Následně byla dokumentu nastavena znaková sada Cp1250, která mimo jiné také definuje všechny potřebné české znaky. Bez tohoto kroku by některé znaky v pdf dokumentu chyběly. Po nastavení dokumentu již následuje samotné plnění dokumentu objekty. Pro přidání objektu do dokumentu stačí nad instancí dokumentu zavolat metodu *Add()* a předat do ní instanci objektu. Jakmile je dokument naplněn objekty s daty, je uzavřen metodou *Close()* a jeho MemoryStream zapsán do pole bytů, které je pak vráceno jako soubor smlouvy. Následuje ukázka přidání paragrafu „SMLOUVA CESTOVNÍHO POJIŠTĚNÍ“, který je zarovnán na střed, velikost fontu nastavena na 20 jednotek a spodní odsazení je nastaveno na 50 jednotek. Zbytek dat je do pdf dokumentu zapsáno obdobným způsobem.

```
var document = new Document(pdf);
document.SetFont(PdfFontFactory.CreateFont(FontProgramFactory.CreateFont(), "Cp1250" ));

document.Add(new Paragraph("SMLOUVA CESTOVNÍHO
POJIŠTĚNÍ").SetTextAlignment(TextAlignment.CENTER).SetFontSize(20).SetMarginBottom(50).S
etBold());
```

5.3 Databáze

Pro propojení s databází je využito knihovny Entity Framework Core [55]. Pro komunikaci s databází je nutné vytvořit třídu databázového kontextu, která bude dědit od třídy DbContext, a v konstruktoru třídy předat nastavení, které je nakonfigurováno v metodě *ConfigureServices()* třídy Startup.

```
public class TravelInsuranceDbContext : DbContext
{
       public TravelInsuranceDbContext(DbContextOptions<TravelInsuranceDbContext> opti-
ons) : base(options) {}
  public DbSet<ContractDto> Contracts { get; set; }
}

services.AddDbContext<TravelInsuranceDbContext>(options => options.UseSqlServer(Configu-
ration.GetConnectionString("db")));
```

V konfiguraci databázového kontextu je nastaveno, aby se použil Microsoft SQL Server s connection stringem specifikovaným v souboru appsettings.json.

```
"ConnectionStrings": {
    "db": "Data Source=DESKTOP-6QMNM2R; Initial Catalog=TravelInsurance; Integrated
Security=true"
  },
```

Databáze je tvořena stylem Code-first, což znamená, že jsou nejdříve vytvořeny modely, které jsou následně převedeny do tabulek. V aplikaci je tímto modelem ContractDto. Pro omezení a nastavení, jak mají být namapovány jednotlivé hodnoty modelu, je nutné využít třídy DataAnnotations. V modelu ContractDto je využito této třídy pro označení primárního klíče tabulky atributem *[Key]* a také pro specifikování datového typu atributem *[DataType]* u hodnot StartDate a EndDate na typ Date.

```
public class ContractDto
{
        [Key]
        public int ContractId { get; set; }

        public DateTime DateCreated { get; set; }

        public InsuranceType InsuranceType { get; set; }

        public Destination Destination { get; set; }

        [DataType(DataType.Date)]
        public DateTime? StartDate { get; set; }

        [DataType(DataType.Date)]
        public DateTime? EndDate { get; set; }

        public int InsuranceNumberOfDays { get; set; }

        public int NumberOfInsuredPersons { get; set; }

        public string JsonContractData { get; set; }

        public bool WasEmailSentToClient { get; set; }
}
```

V kontextu databáze je také definována properta Contracts, která odpovídá tabulce v databázi, a skrz tuto propertu jsou pak data do tabulky zapisována.

5.4 Komponenty

Komponenty jsou základními stavebními bloky frameworků Blazor i Angular, zajišťují modularitu, znovupoužitelnost a rozšiřitelnost aplikace. Uživatelská část aplikace je rozdělena do 5 kroků a každý z nich je tvořen jednou komponentou. Všechny tyto komponenty jsou pak zaobaleny do komponenty navigace, která se stará o přepínání stavu (zobrazování a schovávání jednotlivých komponent) a předávání dat mezi komponentami. Zvláštní komponentou je komponenta pro přidání dalších pojišťovaných osob, která zajišťuje dynamické generování dalších polí, pokud si uživatel přeje připojistit další pojišťované osoby. Aplikace obsahuje následující komponenty: Navigation, DataInputStep, InsuredPersonInput, InsuranceInputStep, CalculationResultStep, AdditionalInformationStep a ContractResultStep.

Následuje popis komponenty Navigation a DataInputStep ve frameworcích Blazor a Angular. Ostatní komponenty jsou vytvořeny velmi podobně, a proto nebudou zmíněny.

5.4.1 Komponenta navigace (Navigation) – Blazor

U tlačítek je využito atributu *@onclick*, který naslouchá události stisknutí tlačítka, a v případě, že se tak stane, spustí specifikovanou metodu. Při kliknutí na tlačítko „Zpět“ se zavolá metoda *PreviousStep()* a z property *Step* je odečtena hodnota 1. Při kliknutí na tlačítko „Pokračovat“ je naopak hodnota 1 do property *Step* přičtena.

```
<div class="navigation-buttons">
        <button class="btn btn-outline-primary" type="button" @onclick="Previ-
ousStep">Zpět</button>
        <button class="btn btn-outline-primary" type="button" @onclick="NextStep" disa-
bled="@NextStepDisabled()">@GetNextButtonText()</button>
</div>
```

Jednotlivé komponenty jsou obaleny podmínkou *if*, která kontroluje, jakou hodnotu právě properta *Step* má, a podle toho se komponenty zobrazí nebo schovají. Celé tělo a všechny komponenty jsou pak obaleny do formuláře EditForm, který je využit pro validaci všech obsažených polí. V EditFormu je definován EditContext, který je použit jako reference na EditForm a je v aplikaci použit pro manuální spouštění validace při stisknutí tlačítka „Pokračovat“. Uvnitř EditFormu se nachází také komponenta FluentValidationValidator, která specifikuje, že EditForm má pro validaci použít Fluent validátory. Každá z komponent má na sobě atribut *@ref* specifikující referenci na danou komponentu, což je následně využito v metodě *NextStep()* pro zavolání metody *UpdateData()* potomků, která předá aktuální data navigační komponentě, a následně je spuštěna validace prostřednictvím *editContext.Validate()*. Každá z komponent obsahuje atribut insuranceData pro vstup dat a atribut onChange typu eventCallback, který je spuštěn v moment kliknutí na tlačítko „Pokračovat“. V části komponenty obsahující kód je metoda *OnInitialized(),* která je spuštěna v moment, kdy je komponenta připravena k použití. Během této metody se tvoří instance objektu InsuranceData, který je pak předáván mezi komponentami, a data jsou tak skrz aplikaci zachována.

```
<EditForm EditContext="@editContext">
        <FluentValidationValidator />

        @if (InsuranceData.Step == InsuranceStep.DataInputStep)
        {
                <DataInputStep @ref="dataInputStep" insuranceData="InsuranceData" on-
Change="UpdateData"></DataInputStep>
        }
        else if (InsuranceData.Step == InsuranceStep.InsuranceInputStep)
        {
                <InsuranceInputStep @ref="insuranceInputStep" insuranceData="Insurance-
Data" onChange="UpdateData"></InsuranceInputStep>
        }
        else if (InsuranceData.Step == InsuranceStep.CalculationResultStep)
        {
                <CalculationResultStep @ref="calculationResultStep" insuranceData="In-
suranceData" onChange="UpdateData"></CalculationResultStep>
        }
        else if (InsuranceData.Step == InsuranceStep.AdditionalInformationStep)
        {
                <AdditionalInformationStep @ref="additionalInformationStep" insurance-
Data="InsuranceData" onChange="UpdateData"></AdditionalInformationStep>
        }
        else if (InsuranceData.Step == InsuranceStep.ContractResultStep)
        {
                <ContractResultStep insuranceData="InsuranceData"></ContractResultStep>
        }
</EditForm>

@code {
        public InsuranceData InsuranceData { get; set; }
        private EditContext editContext { get; set; }

        private DataInputStep dataInputStep;
        private InsuranceInputStep insuranceInputStep;
        private CalculationResultStep calculationResultStep;
        private AdditionalInformationStep additionalInformationStep;

        protected override void OnInitialized()
        {
                InsuranceData = new InsuranceData();
                editContext = new EditContext(InsuranceData);
                base.OnInitialized();
        }

        private void UpdateData(InsuranceData insuranceData)
        {
                InsuranceData = insuranceData;

                if (insuranceData.Step == InsuranceStep.CalculationResultStep)
                {
                        InsuranceData.Step++;
                }
        }
```

```
private void NextStep()
{
        switch (InsuranceData.Step)
        {
                case InsuranceStep.DataInputStep:
                        dataInputStep.UpdateData();
                        break;
                case InsuranceStep.InsuranceInputStep:
                        insuranceInputStep.UpdateData();
                        break;
                case InsuranceStep.AdditionalInformationStep:
                        additionalInformationStep.UpdateData();
                        break;
                default:
                        break;
        }

        if (editContext.Validate())
        {
                if (InsuranceData.Step != InsuranceStep.ContractResultStep)
                {
                        InsuranceData.Step++;
                }
        }
}
```

5.4.2 Komponenta navigace (Navigation) – Angular

U tlačítek je využito obdobně jako v Blazoru atributu *(click)*, který naslouchá události stisknutí tlačítka, a v případě, že se tak stane, spustí specifikovanou metodu. Tělo navigační komponenty pak tvoří posloupnost dalších komponent, které jsou omezeny direktivou **ngIf* sloužící pro zobrazení nebo schování komponenty na základě podmínky. Při kliknutí na tlačítko „Zpět“ se zavolá metoda *PreviousStep()* a z property *Step* je odečtena hodnota 1. Při kliknutí na tlačítko „Pokračovat“ je naopak hodnota 1 do property *Step* přičtena. Dle toho jsou pak díky direktivě **ngIf*, která kontroluje, jakou hodnotu má properta *Step*, zobrazovány nebo skrývány jednotlivé komponenty. Všechny komponenty také obsahují obousměrný atribut označený hranatými a kulatými závorkami *[(insuranceData)]* pro předání objektu mezi rodičovskou komponentou a potomkem. Komponenta contract-result-step pak má jen jednosměrný atribut označený jen hranatými závorkami *[insuranceData]* pro předání dat směrem od rodiče k potomkovi.

```
<div class="navigation-buttons">
 <button
  class="btn btn-outline-primary"
  type="button"
  (click)="PreviousStep()"
 >
  Zpět
 </button>
```

```
 <button
  class="btn btn-outline-primary"
  type="button"
  (click)="NextStep()"
  [disabled]="NextStepDisabled()"
 >
  {{ GetNextButtonText() }}
 </button>
</div>

<data-input-step
 *ngIf="insuranceData.step === InsuranceSteps.DataInputStep"
 [(insuranceData)]="insuranceData"
></data-input-step>
<insurance-input-step
 *ngIf="insuranceData.step === InsuranceSteps.InsuranceInputStep"
 [(insuranceData)]="insuranceData"
></insurance-input-step>
<calculation-result-step
 *ngIf="insuranceData.step === InsuranceSteps.CalculationResultStep"
 [(insuranceData)]="insuranceData"
 (onChange)="NextStep()"
></calculation-result-step>
<additional-information-step
 *ngIf="insuranceData.step === InsuranceSteps.AdditionalInformationStep"
 [(insuranceData)]="insuranceData"
></additional-information-step>
<contract-result-step
 *ngIf="insuranceData.step === InsuranceSteps.ContractResultStep"
 [insuranceData]="insuranceData"
></contract-result-step>
```

Každá komponenta obsahuje v jejím TypeScript souboru definici pro její selector, tedy jak se bude komponenta zapisovat v HTML části kódu, templateUrl odkazující na HTML část dané komponenty a styleUrls odkazující na CSS styly dané komponenty. Komponenta obsahuje metodu *ngOnInit()*, která je spuštěna v moment, kdy je komponenta připravena k použití. V rámci této metody se volá funkce *InitModelWithDefaultValues(),* která do objektu insuranceData předá základní hodnoty. Je zde využit také *@ViewChild* dekorátor, který umožňuje vytvořit referenci na potomka. Reference na potomka je pak využita pro zavolání metody potomka *ValidateAllInputs()*.

```
@Component({
 selector: 'navigation',
 templateUrl: './navigation.component.html',
 styleUrls: ['./navigation.component.css']
})
export class NavigationComponent implements OnInit {

  insuranceData: InsuranceData = new InsuranceData();
  InsuranceSteps = InsuranceStep;

  @ViewChild(DataInputStepComponent) dataInputStepComponent!: DataInputStepComponent;
  @ViewChild(AdditionalInformationStepComponent) additionalInformationStepComponent!:
AdditionalInformationStepComponent;

  isDataInputStepValid = false;
  isAdditionalInformationStepValid = false;

  constructor() { }

  ngOnInit(): void {
   this.InitModelWithDefaultValues();
  }

  NextStepDisabled(): boolean {
   return this.insuranceData.step === InsuranceStep.CalculationResultStep || this.in-
suranceData.step === InsuranceStep.ContractResultStep;
  }

  NextStep() {
   const validation =
     ((this.insuranceData.step === InsuranceStep.DataInputStep && this.dataInputStepCom-
ponent.ValidateAllInputs()) ||
     (this.insuranceData.step === InsuranceStep.AdditionalInformationStep && this.ad-
ditionalInformationStepComponent.ValidateAllInputs())) ||
     this.insuranceData.step === InsuranceStep.InsuranceInputStep ||
     this.insuranceData.step === InsuranceStep.CalculationResultStep ||
     this.insuranceData.step === InsuranceStep.ContractResultStep

   if (validation)
   {
     if (this.insuranceData.step != InsuranceStep.ContractResultStep)
     {
      this.insuranceData.step++;
     }
   }
  }

  PreviousStep() {
```

```
    if (this.insuranceData.step != InsuranceStep.DataInputStep)
    {
      this.insuranceData.step--;
    }
  }

  InitModelWithDefaultValues() {
    const today = new Date();
    this.insuranceData.step = InsuranceStep.DataInputStep;
    this.insuranceData.insuranceType = InsuranceType.SingleTravel;
    this.insuranceData.destination = Destination.CzechRepublic;
    this.insuranceData.startDate = (moment(new Date())).format('YYYY-MM-DD');
    this.insuranceData.endDate = (moment(new Date())).format('YYYY-MM-DD');
    this.insuranceData.pickedPackage = new Package();
    this.insuranceData.insuranceItems = [];
    this.insuranceData.insuredPersons = [];
    this.insuranceData.mainInsuredPerson = new InsuredPerson();
    this.insuranceData.mainInsuredPerson.address = new Address();
    this.insuranceData.mainInsuredPerson.insuredPersonType = InsuredPersonType.Physical-
Person;
    this.insuranceData.mainInsuredPerson.birthDate = (moment(new Date().set-
FullYear(today.getFullYear() - 18))).format('YYYY-MM-DD');
  }
}
```

5.4.3 Komponenta se základními informacemi (DataInputStep) – Blazor

Komponenta se skládá z velkého množství vstupních polí. Pro obousměrný průpis dat mezi propertami objektu a vstupním polem je využito atributu *@bind-Value*. U rozbalovacího menu InputSelect je využito smyčky foreach, která vygeneruje možnost výběru pro každou hodnotu výčtového typu enum. Popis hodnoty je získán metodou *GetDescription(),* která získá popis z atributu *[Description]* definovaného nad každou hodnotou výčtového typu enum. Pod vstupními poli, které potřebují validaci, se nachází komponenta Validation-Message, která zobrazí validační hlášku pro specifikovanou propertu. Některé vstupní pole jsou obaleny podmínkou if a zobrazují se nebo schovávají na základě typu pojistníka. Ve spodní části komponenty se nachází tlačítko „Přidat osobu“, které po zmáčknutí přidá do listu *insuranceData.InsuredPersons* novou instanci třídy OtherInsuredPerson. Tato změna je okamžitě reflektována ve smyčce foreach, která pro každou položku v poli *insurance-Data.InsuredPersons* vygeneruje novou komponentu InsuredPersonInput.

```
<div class="input-card">
        <div class="input-card-title">Základní informace</div>

        <div class="input-row">
                <div class="input-title">Typ pojištění</div>
                <InputSelect class="form-control not-required" @bind-Value="insurance-
Data.InsuranceType">
                        @foreach (Enum option in Enum.GetValues(typeof(InsuranceType)))
                        {
                                <option value="@option">@GetDescription(option)</option>
                        }
                </InputSelect>
        </div>

        <div class="input-row">
                <div class="input-title">Destinace</div>
                <InputSelect class="form-control not-required" @bind-Value="insurance-
Data.Destination">
                        @foreach (Enum option in Enum.GetValues(typeof(Destination)))
                        {
                                <option value="@option">@GetDescription(option)</option>
                        }
                </InputSelect>
        </div>

        <div class="input-row">
                <div class="input-title">Počátek cesty</div>
                <InputDate class="form-control" @bind-Value="insuranceData.StartDate" @on-
change="UpdateData" min="@minDate"/>
                <ValidationMessage For="@(() => insuranceData.StartDate)" />
        </div>

        <div class="input-row">
                <div class="input-title">Konec cesty</div>
                <InputDate class="form-control" @bind-Value="insuranceData.EndDate"
min="@minDate"/>
                <ValidationMessage For="@(() => insuranceData.EndDate)" />
        </div>
                                        .
                                        .
                                        .

        <div class="person-list-title">
                <div>Další pojištěné osoby</div>
                <button class="btn btn-outline-primary" type="button" @onclick="AddIn-
suredPerson" disabled="@AddInsuredPersonDisabled()">Přidat osobu</button>
        </div>

        @if (!insuranceData.InsuredPersons.Any())
        {
                <div class="no-result">Žádné další pojištěné osoby</div>
        }

        @foreach (var insuredPerson in insuranceData.InsuredPersons)
        {
                <InsuredPersonInput insuredPerson="insuredPerson" insuranceData="insuran-
ceData" onChange="UpdateData"></InsuredPersonInput>
        }

</div>
```

Komponenta obsahuje propertu insuranceData označenou atributem *[Parameter]*, to specifikuje danou propertu jako parametr a je možné do ní předávat data z rodičovské komponenty. Dalším parametrem komponenty je onChange typu EventCallback. Tento typ parametru lze využít pro vyvolání události v rodičovské komponentě, která tomuto parametru naslouchá. Vyvolání události se provádí metodou *InvokeAsync()*.

```
@code
{
        [Parameter] public InsuranceData insuranceData { get; set; }
        [Parameter] public EventCallback<InsuranceData> onChange { get; set; }
        private string minDate { get; set; } = DateTime.Now.ToString("yyyy-MM-dd");
        private string maxMainInsuredPersonDate { get; set; } = DateTime.Now.AddYears(-
18).ToString("yyyy-MM-dd");

        public string GetDescription(Enum value)
        {
                var fieldInfo = value.GetType().GetField(value.ToString());
                DescriptionAttribute descriptionAttributes = (DescriptionAttribute)fiel-
dInfo.GetCustomAttributes(typeof(DescriptionAttribute), false).SingleOrDefault();
                return descriptionAttributes == null ? value.ToString() : descriptionA-
ttributes.Description;
        }



        private bool AddInsuredPersonDisabled()
        {
                return this.insuranceData.InsuredPersons.Count() == 10;
        }

        public void AddInsuredPerson()
        {
                insuranceData.InsuredPersons.Add(new OtherInsuredPerson());
        }

        public void UpdateData()
        {
                onChange.InvokeAsync(insuranceData);
        }
}
```

5.4.4 Komponenta se základními informacemi (DataInputStep) – Angular

Komponenta se skládá z velkého množství vstupních polí. Jelikož je zde potřeba validovat hodnoty jednotlivých polí, tak jsou vstupní pole obaleny v elementu *ng-container* s parametrem *[FormGroup],* kterému je předána instance třídy FormGroup s validacemi pro jednotlivé vstupní pole. Pokud pole nechceme validovat, je v parametru *[ngModelOptions]* nastavena hodnota standalone na true. Pro obousměrné předávání dat mezi vstupními poli a propertami objektu insuranceData je využito direktivy *[(ngModel)],* která také předává hodnoty polí do definované instance FormGroup, a dochází tak k jejich validaci. Validační hlášky jsou definovány jako div element obsahující příslušnou validační hlášku. Tento element je omezen direktivou **ngIf*, která kontroluje, zda se nachází v příslušné validační propertě chybová hláška. Všechny vstupní pole také využívají události *(change)* a při změně hodnoty pole je volána metoda *modelChanged(),* která emituje nově upravená data skrz obousměrný parametr *[(insuranceData)]* do rodičovské komponenty. Ve spodní části komponenty se nachází tlačítko „Přidat osobu“, které po zmáčknutí přidá do listu *insuranceData.InsuredPersons* novou instanci třídy OtherInsuredPerson. Tato změna je okamžitě reflektována ve smyčce **ngFor*, která pro každou položku v poli *insuranceData.InsuredPersons* vygeneruje novou komponentu insured-person-input.

```
<div class="input-card">
 <div class="input-card-title">Základní informace</div>

 <ng-container [formGroup]="validations">
  <div class="input-row">
   <div class="input-title">Typ pojištění</div>
   <select
    class="form-control not-required"
    [(ngModel)]="insuranceData.insuranceType"
    [ngModelOptions]="{ standalone: true }"
    (change)="modelChanged()"
   >
    <option *ngFor="let type of insuranceTypes" [value]="type">
     {{ insuranceTypeDescriptions[type] }}
    </option>
   </select>
  </div>

  <div class="input-row">
   <div class="input-title">Destinace</div>
   <select
    class="form-control not-required"
    [(ngModel)]="insuranceData.destination"
```

```
   [ngModelOptions]="{ standalone: true }"
   (change)="modelChanged()"
  >
   <option *ngFor="let type of destinations" [value]="type">
    {{ destinationDescriptions[type] }}
   </option>
  </select>
</div>

<div class="input-row">
 <div class="input-title">Počátek cesty</div>
 <input
  type="date"
  class="form-control"
  [(ngModel)]="insuranceData.startDate"
  formControlName="startDate"
  (change)="modelChanged()"
  min="{{ minDate }}"
 />
 <div
  *ngIf="startDate?.invalid && startDate?.touched"
  class="validation-message"
 >
  <div *ngIf="startDate?.errors?.dateValidation">
   Vyplňte datum počátku cesty.
  </div>
  <div *ngIf="startDate?.errors?.dateDiffValidation">
   Datum počátku cesty nesmí být před datem konce cesty.
  </div>
 </div>
</div>
<div class="input-row">
 <div class="input-title">Konec cesty</div>
 <input
  type="date"
  class="form-control"
  [(ngModel)]="insuranceData.endDate"
  formControlName="endDate"
  (change)="modelChanged()"
  min="{{ minDate }}"
 />
 <div
  *ngIf="endDate?.invalid && endDate?.touched"
  class="validation-message"
 >
  Vyplňte datum konce cesty.
 </div>
</div>
```

```
.
.
.
<div *ngIf="insuranceData.insuredPersons.length === 0" class="no-result">
 Žádné další pojištěné osoby
</div>

<ng-container
 *ngFor="let insuredPerson of insuranceData.insuredPersons; let i = index"
>
 <insured-person-input
  [index]="i"
  [(insuredPerson)]="insuranceData.insuredPersons[i]"
  [(insuranceData)]="insuranceData"
 ></insured-person-input>
</ng-container>
</div>
```

Vstupní parametr komponenty se definuje dekorátorem *@Input()* a výstupní parametr se definuje dekorátorem *@Output()*. Při zvolení správného názvu vstupního a výstupního parametru je vytvořen parametr obousměrný. Názvy se musí shodovat s tím rozdílem, že výstupní parametr musí končit slovem „Change“. Pro tvoření popisu hodnot v rozbalovacím menu select je nutné využít namapování hodnot výčtového typu enum na proměnnou typu Record příslušného výčtového typu, která obsahuje popisy jednotlivých hodnot. Při inicializování komponenty v rámci metody *initValidation()* jsou zaregistrována jednotlivá validační pravidla pro jednotlivé hodnoty v rámci třídy FormGroup. Pro jednodušší zobrazování validačních hlášek jsou také vytvořeny get metody, které vrátí vlastnosti dané property, ve kterých se mimo jiné nachází i informace, zda je hodnota validní, nebo ne. V poslední řadě je zde metoda *ValidateAllInputs(),* která je volána z rodičovské komponenty při stisknutí tlačítka „Pokračovat“. Pro zvalidování všech hodnot je nad všemi validovanými hodnotami spuštěna metoda *markAsTouched(),* která zapříčiní to, že si vstupní pole myslí, že do něj uživatel kliknul, a zvaliduje se. Metoda nakonec rodičovské komponentě vrátí boolean hodnotu true, nebo false na základě toho, jestli jsou všechny hodnoty validní nebo ne.

```
@Component({
 selector: 'data-input-step',
 templateUrl: './data-input-step.component.html',
 styleUrls: ['./data-input-step.component.css']
})
export class DataInputStepComponent implements OnInit {

  @Input() insuranceData!: InsuranceData;
  @Output() insuranceDataChange: EventEmitter<InsuranceData> = new EventEmitter<Insuran-
ceData>();
  @ViewChildren(InsuredPersonInputComponent) insuredPersonInputComponents!: Query-
List<InsuredPersonInputComponent>;

  validations!: FormGroup;
  insuranceTypeDescriptions = InsuranceTypeDescriptions;
  destinationDescriptions = DestinationDescriptions;
  insuredPersonTypeDescriptions = InsuredPersonTypeDescriptions;
  insuranceTypes = Object.values(InsuranceType).filter(value => typeof value === 'num-
ber').map(x => x as InsuranceType);
  destinations = Object.values(Destination).filter(value => typeof value === 'num-
ber').map(x => x as Destination);
  insuredPersonTypes = Object.values(InsuredPersonType).filter(value => typeof value ===
'number').map(x => x as InsuredPersonType);

  minDate = "";
  maxMainInsuredPersonDate = "";
  areInsuredPersonsAllValid = true;

  get startDate() { return this.validations.get('startDate'); }
  get endDate() { return this.validations.get('endDate'); }
  get name() { return this.validations.get('name'); }
  get surname() { return this.validations.get('surname'); }
  get birthNumber() { return this.validations.get('birthNumber'); }
  get birthDate() { return this.validations.get('birthDate'); }
  get personalIdentificationNumber() { return this.validations.get('personalIdentifica-
tionNumber'); }
  get companyName() { return this.validations.get('companyName'); }

  constructor() {}

  ngOnInit(): void {
   this.initValidations()
   const today = new Date();
   this.minDate = (moment(new Date())).format('YYYY-MM-DD');
   this.maxMainInsuredPersonDate = (moment(new Date().setFullYear(today.getFullYear() -
18))).format('YYYY-MM-DD');
  }
```

```
  AddInsuredPersonDisabled()
  {
   return this.insuranceData.insuredPersons.length == 10;
  }

  AddInsuredPerson()
  {
   this.insuranceData.insuredPersons.push(new OtherInsuredPerson());
  }

  convertEnumsToNumber() {
   this.insuranceData.insuranceType = Number(this.insuranceData.insuranceType);
   this.insuranceData.destination = Number(this.insuranceData.destination);
   this.insuranceData.mainInsuredPerson.insuredPersonType = Number(this.insurance-
Data.mainInsuredPerson.insuredPersonType);
  }

  modelChanged() {
   this.convertEnumsToNumber();
   this.insuranceDataChange.emit(this.insuranceData);
  }

  insuredPersonTypeChanged() {
   this.initValidations();
  }

  initValidations() {
   this.convertEnumsToNumber();
   this.validations = new FormGroup({
     startDate: new FormControl(this.insuranceData.startDate, [ DateValidator(), DateDi-
ffValidator(this.insuranceData.startDate, this.insuranceData.endDate) ]),
     endDate: new FormControl(this.insuranceData.endDate, DateValidator()),
     name: new FormControl(this.insuranceData.mainInsuredPerson.name, NameValida-
tor(this.insuranceData.mainInsuredPerson.insuredPersonType)),
     surname: new FormControl(this.insuranceData.mainInsuredPerson.surname, SurnameVali-
dator(this.insuranceData.mainInsuredPerson.insuredPersonType)),
     birthNumber: new FormControl(this.insuranceData.mainInsuredPerson.birthNumber, Bir-
thNumberValidator(this.insuranceData.mainInsuredPerson.insuredPersonType)),
     birthDate: new FormControl(this.insuranceData.mainInsuredPerson.birthDate, BirthDa-
teValidator(this.insuranceData.mainInsuredPerson.insuredPersonType)),
     personalIdentificationNumber: new FormControl(this.insuranceData.mainInsuredPer-
son.personIdentificationNumber, PersonalIdentificationNumberValidator(this.insurance-
Data.mainInsuredPerson.insuredPersonType)),
     companyName: new FormControl(this.insuranceData.mainInsuredPerson.companyName, Com-
panyNameValidator(this.insuranceData.mainInsuredPerson.insuredPersonType)),
   });
  }
  ValidateAllInputs() {
```

```
    this.startDate?.markAsTouched();
    this.endDate?.markAsTouched();
    this.name?.markAsTouched();
    this.surname?.markAsTouched();
    this.birthNumber?.markAsTouched();
    this.birthDate?.markAsTouched();
    this.personalIdentificationNumber?.markAsTouched();
    this.companyName?.markAsTouched();

    this.insuredPersonInputComponents.forEach((c) => c.ValidateAllInputs());
    this.areInsuredPersonsAllValid = this.insuredPersonInputComponents.toArray().every(c
=> c.IsValid());

    return (this.validations.status === "VALID" && this.areInsuredPersonsAllValid);
  }
}
```

5.5 Modely

5.5.1 Blazor

Modely reprezentující data jsou ve frameworku Blazor uloženy ve složce Models v projektu Shared. Díky umístění v projektu Shared jsou modely použitelné v klientské i serverové části aplikace. Ve složce Models se nachází také výčtové typy enum, které jsou často využívány v rámci aplikace pro větší přehlednost kódu. Hodnoty výčtových typů enum využívají atributu *[Description]*, který specifikuje popis hodnoty a je v klientské části používán pro tvorbu rozbalovacích menu. Následuje ukázka modelu InsuranceItem a výčtového typu Destination.

```
public class InsuranceItem
{
        public InsuranceItemType InsuranceItemType { get; set; }

        public int PricePerPerson { get; set; }

        public int InsuredSum { get; set; }

        public bool IsSelected { get; set; }

        public bool IsIncluded { get; set; }

        public bool IsBoolType { get; set; }

        public string GetValue() => IsBoolType ? "Ano" : $"{InsuredSum} Kč";
}
```

```
public enum Destination
{
        [Description("Česká republika")]
        CzechRepublic = 0,

        [Description("Evropa")]
        Europe = 1,

        [Description("Celý svět")]
        World = 2,
}
```

5.5.2 Angular

Pro framework Angular bylo nutné přepsat všechny modely a výčtové typy v jazyce TypeScript. Všechny modely musí být přesné kopie těch v serverové části aplikace, jinak může dojít k chybám. Každý model je opatřen klíčovým slovem *export*, které je ekvivalentem klíčového slova public v jazyce C#. Jedná se tedy o modifikátor přístupu, který specifikuje, že se jedná o veřejnou třídu, která může být importována a použita kdekoliv v projektu. Pro specifikování popisu hodnot výčtových typů bylo využito objektu typu Record, který specifikuje sadu klíčů a hodnot. Pro každou hodnotu výčtového typu enum je v tomto objektu specifikován popis. Následuje ukázka modelu InsuranceItem, výčtového typu Destination a jeho příslušnému objektu DestinationDescriptions.

```
export class Address {
  street: string;
  city: string;
  number: string;
  postalCode: string;
}

export enum Destination {
  CzechRepublic = 0,
  Europe = 1,
  World = 2
}

export const DestinationDescriptions: Record<Destination, string> = {
  [Destination.CzechRepublic]: "Česká republika",
  [Destination.Europe]: "Evropa",
  [Destination.World]: "Celý svět",
};
```

5.6 Validace

5.6.1 Blazor

Pro validování dat byla zvolena knihovna Fluent Validation [56], jelikož odděluje validační logiku od business logiky. Umožňuje psát komplexní validační třídy za použití lambda výrazů. Je možné validovat i vnořené třídy a validátory jsou použity jak pro klientské validování vstupů, tak pro validování dat, která jsou zasílána na endpointy serverové části aplikace. Validace jsou sjednoceny na jednom místě a nemůže se tak stát, že by byly nekonzistentní mezi klientskou a serverovou částí aplikace, proto jsou také validace umístěny v projektu Shared. Pro aplikaci byly definovány celkem 3 validátory, InsuranceDataValidator, který validuje obecná data o pojištění zadaná uživatelem. InsuredPersonValidator, který validuje data pojistníka. A OtherInsuredPersonValidator, který validuje data dalších přidaných osob. Pro definování validační třídy stačí dědit třídu *AbstractValidator<T>,* které v parametru předáme třídu, jejíž property budeme validovat. V rámci konstruktoru validační třídy jsou pak definována validační pravidla díky pomocným metodám FluentValidation, jako jsou *RuleFor(), When()* nebo pokud je potřeba validovat list objektu, tak *RuleForEach().* Pro validaci vnořených tříd je specifikován validátor pro danou třídu v metodě *SetValidator().* Následuje ukázka validační třídy InsuranceDataValidator.

```
public class InsuranceDataValidator : AbstractValidator<InsuranceData>
{
        public InsuranceDataValidator()
  {
                RuleFor(data => data.StartDate).NotEmpty().WithMessage("Vyplňte datum po-
čátku cesty.");
                RuleFor(data => data.EndDate).NotEmpty().WithMessage("Vyplňte datum konce
cesty.");

                When(data => data.StartDate.Date != data.EndDate.Date, () => {
                        RuleFor(data => data.StartDate).LessThan(data => data.End-
Date).WithMessage("Datum počátku cesty nesmí být před datem konce cesty.");
                });

                RuleFor(data => data.MainInsuredPerson).SetValidator(data => new Insured-
PersonValidator(data.Step));
                RuleForEach(data => data.InsuredPersons).SetValidator(new OtherInsuredPer-
sonValidator());
  }
}
```

Aby byly validační třídy aplikací využívány, je potřeba je zaregistrovat v metodě *ConfigureServices()* třídy Startup. Pro to je využita metoda *AddFluentValidation(),* která přidá validační služby do aplikace, a uvnitř této metody je využita metoda *RegisterValidatorsFromAssemblyContaining<T>,* která zaregistruje do aplikace všechny validační třídy, které se nachází ve stejném assembly, jako je specifikovaná třída v generickém parametru metody.

```
services.AddControllersWithViews().AddFluentValidation(fv =>
{
        fv.ImplicitlyValidateChildProperties = true;
        fv.ImplicitlyValidateRootCollectionElements = true;
        fv.RegisterValidatorsFromAssemblyContaining<InsuranceDataValidator>();
});
```

Po zaregistrování validačních tříd serverová část aplikace automaticky validuje všechny příchozí data dle validátorů FluentValidation. Pro použití v klientské části aplikace je nutné vložit FluentValidationValidator komponentu do komponenty EditForm, aby bylo zaregistrováno, že má EditForm využívat validačních tříd FluentValidation, a pod validované vstupní pole je nutné přidat komponentu ValidationMessage, ve které specifikujeme, pro kterou propertu objektu má komponenta získat validační hlášku zaslanou validačními třídami.

```
<EditForm EditContext="@editContext">
        <FluentValidationValidator />

                        .
                        .
                        .

</EditForm>

<ValidationMessage For="@(() => insuranceData.MainInsuredPerson.Name)" />
```

5.6.2 Angular

Ve frameworku Angular je využito třídy FormControl, která je součástí frameworku, a je možné je využít pro validování reaktivních formulářů. [57] Vstupní pole, které mají být validovány, jsou obaleny *ng-containerem*, který definuje hranice formuláře. Na tento container je připnuta direktiva *[formGroup],* do které je následně předán objekt validations. Pro předávání hodnoty do validačního formuláře je na vstupních polích využita direktiva *[(ngModel)].* Validační hlášky jsou pak definovány jako div element obsahující text validační hlášky. Div element je omezen direktivou **ngIf*, která kontroluje, zda se nachází v příslušné validační propertě chybová hláška. Pokud ano, tak je element s chybovou hláškou zobrazen.

```
<ng-container [formGroup]="validations">
  <div class="input-row">
   <div class="input-title">Email</div>
   <input
    class="form-control"
    [(ngModel)]="insuranceData.mainInsuredPerson.email"
    formControlName="email"
    (change)="modelChanged()"
   />
   <div *ngIf="email?.invalid && email?.touched" class="validation-message">
    <div *ngIf="email?.errors?.emailValidation">Vyplňte email.</div>
    <div *ngIf="email?.errors?.email">Nevalidní email.</div>
   </div>
  </div>
                      .
                      .
                      .
</ng-container>
```

Pro zaregistrování validačních pravidel je nutné při inicializaci komponenty vytvořit instanci třídy FormGroup a té předat seznam kontrolních proměnných typu FormControl. Do třídy FormControl je vždy předána hodnota property, která se má validovat, a seznam validačních pravidel, které mohou být vlastní nebo je možné využít již hotových validačních pravidel frameworku Angular.

```
this.validations = new FormGroup({
  email: new FormControl(this.insuranceData.mainInsuredPerson.email, [ EmailEmptyValida-
tor(), Validators.email ]),
  phoneNumber: new FormControl(this.insuranceData.mainInsuredPerson.phoneNumber, Phone-
NumberValidator()),
});
```

Pro jednodušší přístup ke kontrolním hodnotám validační třídy jsou vytvořeny get metody, které vrátí vlastnosti dané property, ve kterých se mimo jiné nachází i informace, zda je hodnota validní, nebo ne. Díky těmto metodám jsou pak zobrazovány jednotlivé validační hlášky.

```
get email() { return this.validations.get('email'); }
get phoneNumber() { return this.validations.get('phoneNumber'); }
```

Pro aplikaci musely být vytvořeny vlastní validační funkce, jelikož bylo nutné specifikovat, za jakých podmínek nevalidovat některé hodnoty. Všechny vlastní validační funkce jsou umístěny v souboru validations.functions.ts. Pro definování vlastní validační funkce je potřeba, aby funkce vracela jeden z typů rozhraní ValidatorFn. Níže uvedená validační funkce přijímá hodnotu typu AbstractControl, která obsahuje validovanou hodnotu. Následuje vlastní validační logika, která určí, zda je hodnota validní, nebo ne, a dle toho je buď vrácena chybová validační hláška typu ValidationErrors, nebo hodnota null reprezentující validní stav.

```
export function EmailEmptyValidator(): ValidatorFn {
 return (control: AbstractControl): ValidationErrors | null => {
  const forbidden = control.value == null || control.value == '';
  return forbidden ? { emailValidation: { value: control.value } } : null;
 };
}
```

5.7 Komunikace se serverovou částí aplikace

5.7.1 Blazor

Pro komunikaci se serverovou částí aplikace je do komponent, které tuto komunikaci potřebují, injektována závislost na třídu HttpClient, která umožňuje zasílat HTTP požadavky na server a získávat HTTP odpovědi od serveru. Samotné injektování se provádí klíčovým slovem *@inject*. Pro zasílání dat mezi serverem a klientem byl vybrán formát json. Třída HttpClient obsahuje metody, které za nás objekty serializují do formátu json i z něj deserializují zpět do objektu. V následující ukázce se využívá metody *PostAsJsonAsync(),* jejími parametry jsou URL cesta k endpointu, na který chceme zaslat HTTP požadavek typu POST, a data, která mají být serializována a zaslána na specifikovaný endpoint. Jelikož framework Blazor úzce spolupracuje s frameworkem ASP.NET Core a je spouštěn na stejné doménové adrese, tak není třeba specifikovat doménu, ale jen cestu k danému enpointu. Následně je výsledek

požadavku uložen do proměnné response, na kterou se použije metoda *ReadFromJsonAsync<T>(),* a data, která byla zaslána ze serveru ve formátu json, jsou opět serializována do objektu specifikovaného v generickém parametru metody. [58]

```
@inject HttpClient HttpClient

@code {
        protected override async Task OnInitializedAsync()
        {
                var response = await HttpClient.PostAsJsonAsync(
                        "InsuranceController/CalculateAllPackages", insuranceData
                );
                packages = await response.Content.ReadFromJsonAsync<List<Package>>();
        }
}
```

5.7.2 Angular

Angular má v základu také vestavěnou třídu HttpClient. [59] Třída obsahuje metody pro zasílání všech typů HTTP požadavků. Metody třídy automaticky serializují a deserializují objekty do formátu json. Jelikož projekt Angular ale běží na jiné URL, je nutné specifikovat v rámci volání endpointů serveru celou cestu i s doménovou částí URL. Metody vrací typ observable, který je využíván pro asynchronní operace. Typ observable definuje metodu *subscribe(),* kterou začneme naslouchat na změny, a pokud ke změně dojde, tak jsme notifikováni, a na změnu reagujeme kódem definovaným uvnitř metody *subscribe().* V následující ukázce lze vidět volání metody typu POST zasílající model typu InsuranceData na endpoint http://localhost:40568/InsuranceController/GetInsuranceItems a následné zpracování odpovědi serveru díky metodě *subscribe()*, která obsah odpovědi zapíše do proměnné packages.

```
CalculateAllPackages(insuranceData: InsuranceData): Observable<Package[]> {
  return this.httpClient.post<Package[]>(
  `${this.apiBaseUrl}/InsuranceController/CalculateAllPackages`,
  insuranceData
  );
}


this.apiService.CalculateAllPackages(this.insuranceData).subscribe(
        (result) => this.packages = result
);
```

5.8 Rozdíly a podobnosti v implementaci

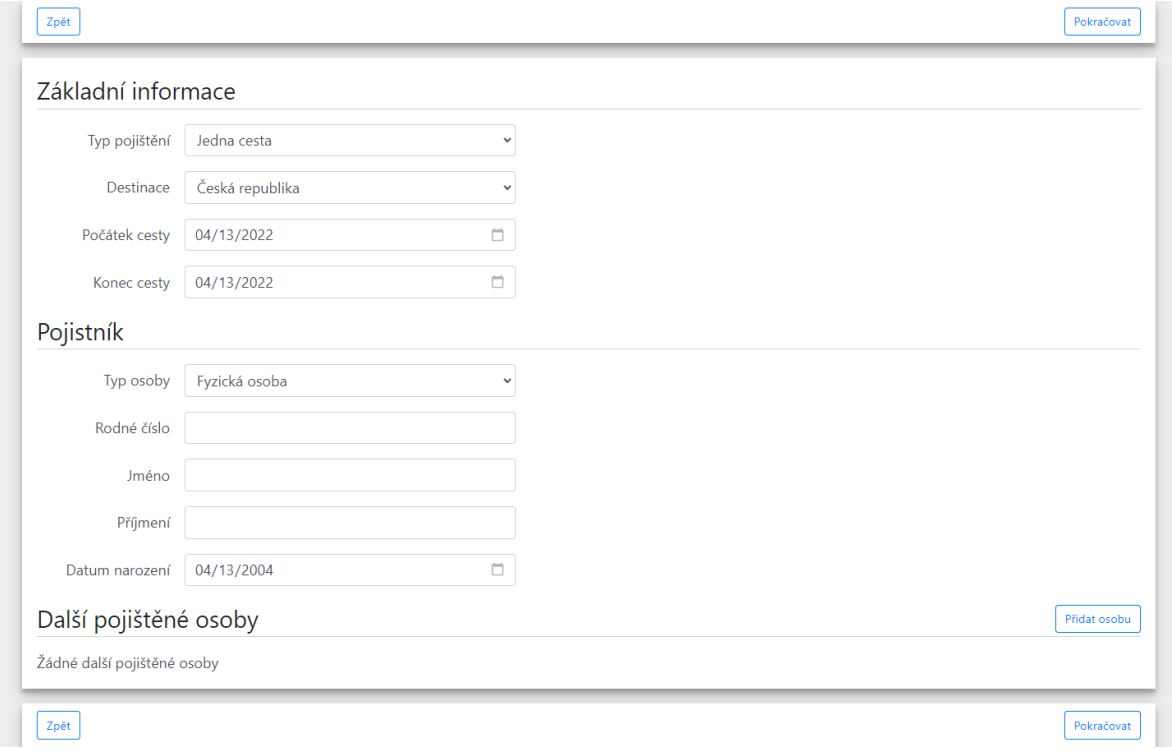
Hlavním rozdílem v implementaci jednostránkové aplikace ve frameworcích Blazor a Angular je, že Blazor obsahuje daleko méně redundantního kódu hlavně díky sdílenému projektu Shared, který sdílí mezi klientskou a serverovou částí aplikace modely a validace, ale můžou se zde sdílet i služby a další kusy kódu. V Angularu je nutné všechny modely i validace přepsat v jazyce TypeScript a dochází tak k duplikování kódu. Validace v Angularu také obsahují o dost více boilerplate kódu než v Blazoru, ve kterém pro zobrazení validační hlášky stačí jediná komponenta, a definice validačních pravidel je také jednodušší. Další velkou výhodou Blazoru je, že pro psaní logiky stačí znalost jen jednoho programovacího jazyka – C#. Při práci v Angularu je nutné znát i TypeScript, popř. JavaScript. Implementace se také liší ve strukturování komponent. Blazor v základu tvoří jen jeden soubor, ve kterém se nachází HTML šablona, stylování i logika komponenty, což působí nepřehledně, hlavně při rozsáhlejších komponentách. Ale je zde také možnost oddělit logiku i stylování do samostatných souborů. Angular naopak komponenty rozděluje v základu do několika souborů. Na soubor obsahující HTML šablonu, soubor s logikou komponenty a soubor s CSS styly komponenty. Rozdělování komponent na jednotlivé soubory pak vede sice k velkému množství souborů v rámci složitějších projektů, ale také je kód čistší a lépe se v něm orientuje. Oba frameworky jsou pak pro vývoj jednoduchých jednostránkových aplikací téměř totožné, jen s menším rozdílem v syntaxi. Oba nabízejí třídu HttpClient pro komunikaci se serverem. Je možný jednosměrný i obousměrný data binding. V Blazoru atributem *@bind-Value* a v Angularu direktivou *[(ngModel)]*. Komponenty lze zobrazovat či skrývat v Blazoru podmínkou *if* a v Angularu direktivou **ngIf*. Komponenty lze cyklicky renderovat např. s využitím foreach smyčky v Blazoru nebo direktivy **ngFor* v Angularu. Jednou z velkých výhod Angularu je jeho věk. Jedná se o již zaběhnutý framework obsahující daleko detailnější dokumentaci a velkou programátorskou komunitu. Při vývoji větších a složitějších projektů je Angular lepší volba, jelikož obsahuje daleko více nástrojů než Blazor a jedná se o již hotový produkt, který je ale i tak stále aktualizován. Blazor je stále v aktivním vývoji a není tak robustní. Pro jednodušší aplikace je ale i tak vhodný. Zejména pro začátečníky díky tomu, že využívá jen jeden programovací jazyk a obsahuje méně boilerplate kódu. Je tak jednodušší na naučení než framework Angular.

6 DEMONSTRACE APLIKACE

V následující kapitole je prezentována aplikace z pohledu uživatele. Jelikož verze aplikace v Blazoru i Angularu vypadají totožně, byla vybrána pro prezentaci aplikace vytvořena ve frameworku Blazor.

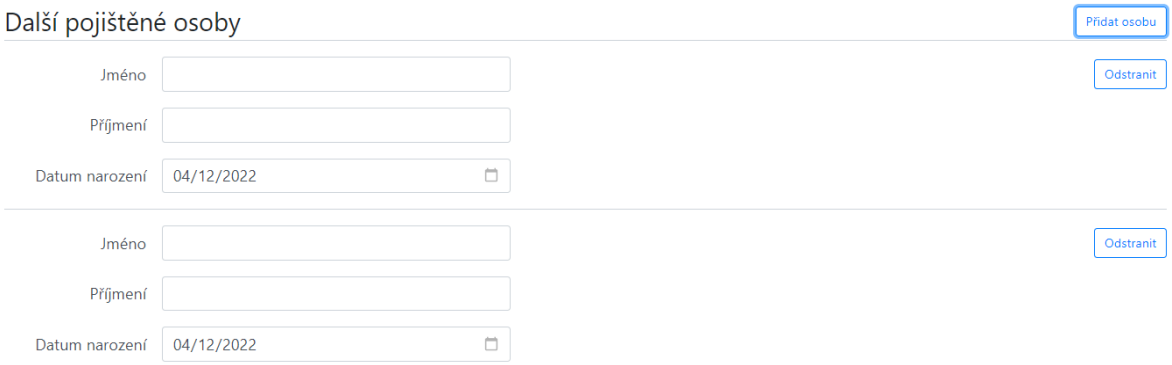
6.1 Základní informace (1. krok)

Při prvotním načtení webové aplikace se uživateli zobrazí první krok aplikace. Hlavní tělo stránky je obaleno z vrchu i ze spodu navigačním panelem s tlačítky „Pokračovat“ a „Zpět“. Po uživateli je požadováno vyplnění základních informací o cestě, což zahrnuje výběr typu pojištění (jedna cesta, více cest za rok, roční pobyt), destinace (Česká republika, Evropa, celý svět) a také zvolení datumu počátku a konce cesty. Dále musí vyplnit informace o pojistníkovi. Na základě vybraného typu osoby se zobrazí jiná povinná vstupní pole. Pro fyzickou osobu je nutné vyplnit rodné číslo, jméno, příjmení a datum narození. Pro právnickou osobu je nutné vyplnit IČO a název firmy a pro OSVČ je nutné vyplnit rodné číslo, jméno, příjmení, datum narození, IČO a název firmy. Na obrázku číslo 4 lze vidět stránku pro vyplnění základních informací o cestě a pojistníkovi.



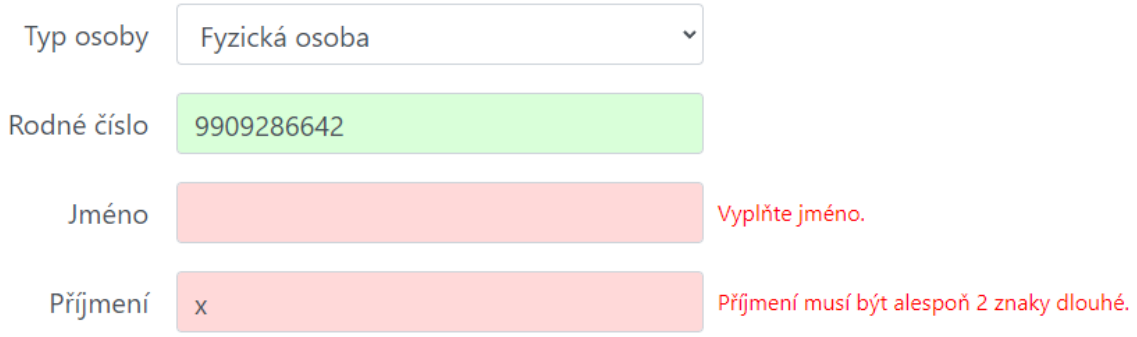
Obrázek 4 - Základní informace

Pokud si uživatel přeje pojistit více osob, může kliknout na tlačítko „Přidat osobu“. Lze pojistit maximálně 10 dalších osob. Při kliknutí na tlačítko se zobrazí nové povinné pole pro každou přidanou osobu. Jedná se o jméno, příjmení a datum narození. Vedle každé přidané osoby je také tlačítko „Odstranit“, které při kliknutí danou osobu ze seznamu odstraní. Na obrázku číslo 5 lze vidět přidání dalších pojištěných osob.



Obrázek 5 - Další pojištěné osoby

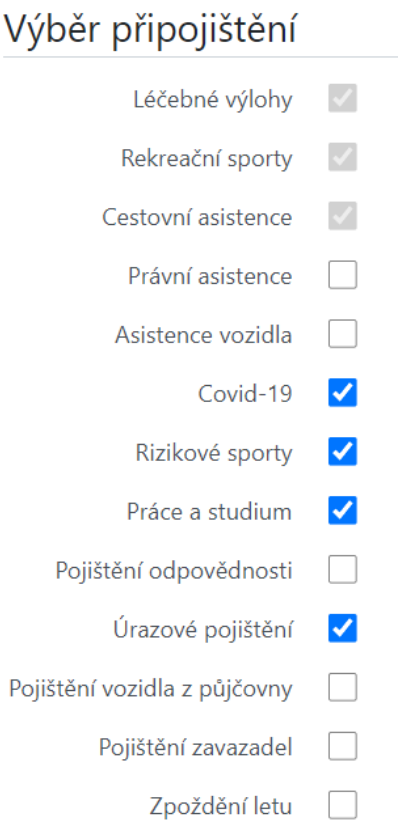
Při kliknutí na tlačítko „Pokračovat“ se spustí validace všech povinných vstupních polí, pokud jsou validní, tak se označí zelenou barvou, a pokud hodnota validní není, tak se pole označí červeně a vedle něj se zobrazí validační hláška. Pokud jsou všechna pole validní, tak je uživatel přesměrován na další krok. Na obrázku číslo 6 lze vidět funkčnost validací.



Obrázek 6 - Validace

6.2 Výběr pojištění (2. krok)

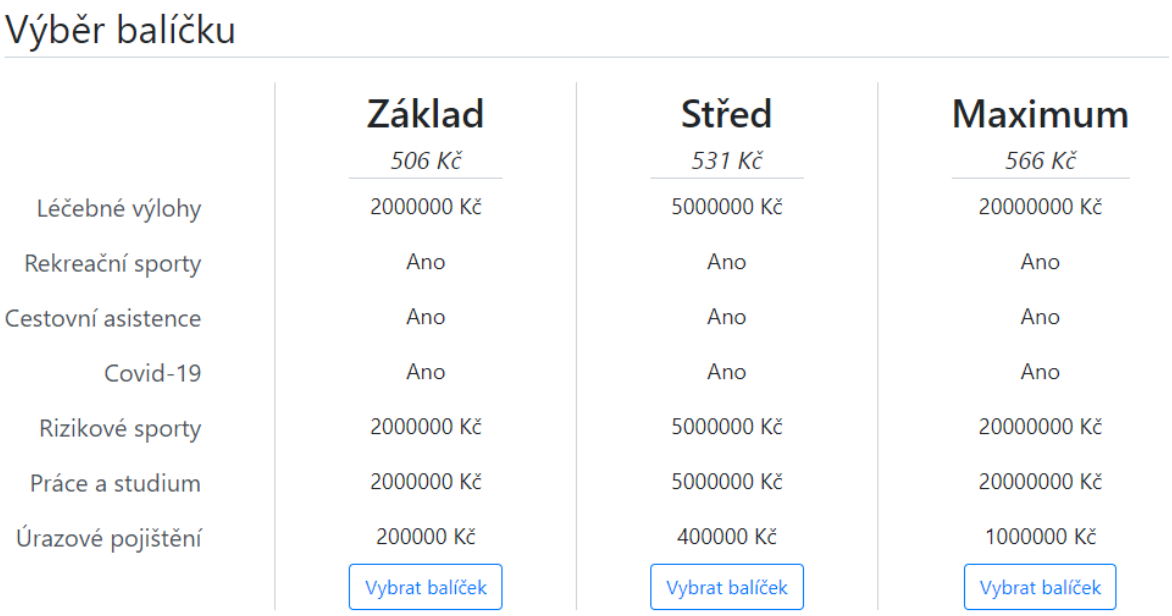
Uživateli je zobrazen seznam všech možných připojištění, které lze sjednat. Uživatel následně může zaškrtnout checkbox u všech připojištění, které chce mít zahrnuty v rámci pojištění. První 3 připojištění mají zašednutý vypnutý checkbox, jelikož jsou vždy zahrnuta v rámci ceny balíčku. Po vybrání připojištění uživatel klikne opět na tlačítko „Pokračovat“ a je přesměrován na další krok. Na obrázku číslo 7 lze vidět výběr jednotlivých připojištění.



Obrázek 7 - Výběr připojištění

6.3 Výběr balíčku (3. krok)

Uživateli je zobrazena nabídka 3 balíčků s názvy Základ, Střed a Maximum. Pod každým z balíčků se nachází jeho cena v Korunách českých, která byla vypočítána na základě dat zadaných uživatelem v předešlých 2 krocích. Pod cenou balíčku je seznam připojištění, které byly vybrány na předchozím kroku, a jejich limit plnění. Tlačítko „Pokračovat“ je v tomto bodě vypnuto a uživatel musí pokračovat kliknutím na tlačítko „Vybrat balíček“. Po kliknutí je balíček uložen a uživatel přesměrován na další krok. Na obrázku číslo 8 lze vidět výběr pojišťovacího balíčku.



Obrázek 8 - Výběr balíčku

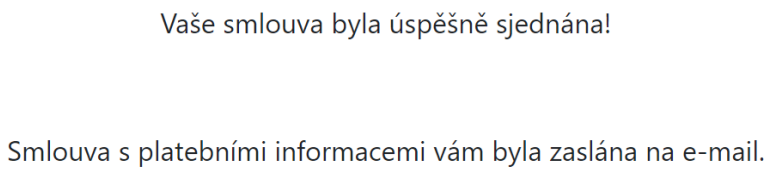
6.4 Kontaktní údaje (4. krok)

Ve 4. kroku je vyžadováno po uživateli vyplnit kontaktní údaje. Konkrétně musí vyplnit ulici, číslo popisné, město, poštovní směrovací číslo, email a telefonní číslo. Po validním vyplnění kontaktních údajů stačí již jen kliknout na tlačítko „Sjednat smlouvu“. Po kliknutí na tlačítko je uživatel přesměrován na poslední krok. Na obrázku číslo 9 lze vidět stránku pro vyplnění kontaktních údajů pojistníka.

Obrázek 9 - Kontaktní údaje

6.5 Smlouva

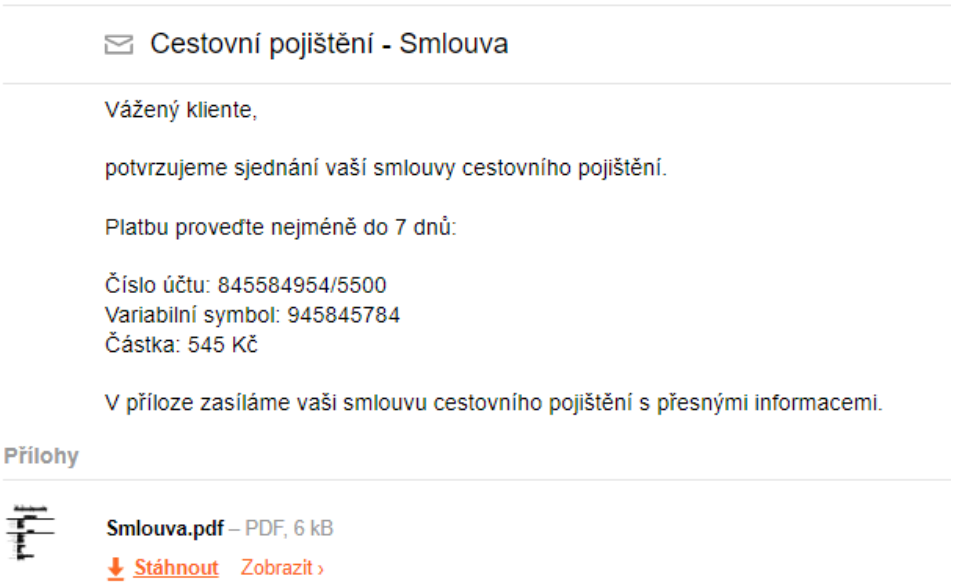
V posledním kroku je uživateli zobrazena jen potvrzovací zpráva s informací, že byla smlouva úspěšně sjednána a že mu byla zaslána na email. Na obrázku číslo 10 lze vidět stránku s potvrzením o zaslání smlouvy.



Obrázek 10 - Potvrzení o sjednání smlouvy

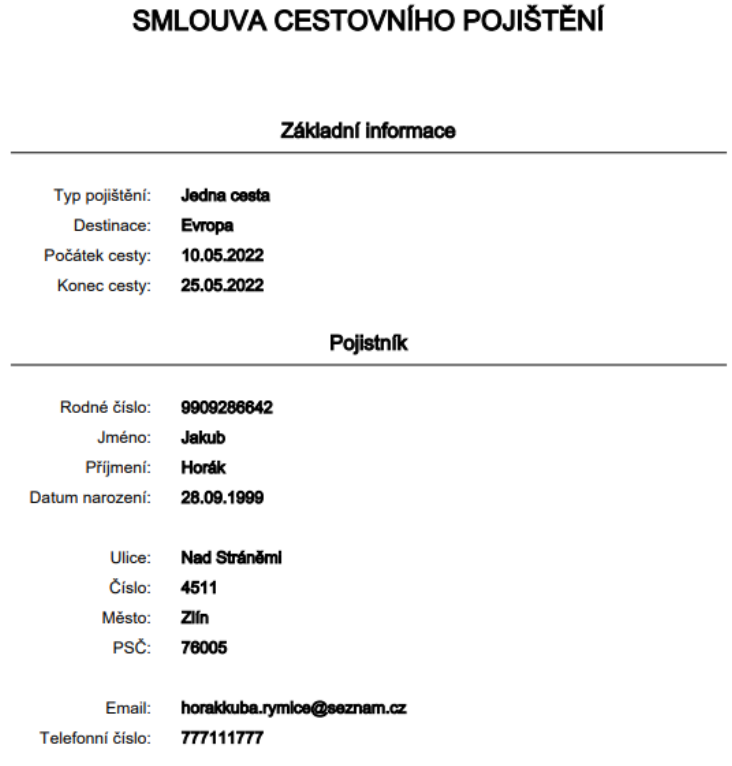
Aplikace zašle na uživatelem vyplněný email smlouvu a platební informace. Níže lze vidět formát zaslaného emailu na uživatele. Obsahuje informace pro zaplacení smlouvy, konkrétně číslo účtu, variabilní symbol a částku vybraného pojišťovacího balíčku. V příloze je

smlouva ke stažení ve formátu pdf. Na obrázku číslo 11 lze vidět email zaslaný na email pojistníka při sjednání smlouvy.



Obrázek 11 – Email

Soubor smlouvy je rozdělen do několika sekcí. Každá sekce obsahuje informace vyplněné uživatelem při sjednávání pojištění. Na 1. stránce se nachází sekce se základními informacemi o cestě a sekce s informacemi o pojistníkovi. Na 2. stránce je sekce se seznamem dalších pojištěných osob a na poslední stránce je sekce s vybraným pojišťovacím balíčkem, která obsahuje seznam vybraných připojištění a jejich limit plnění a nakonec cenu pojištění. Na obrázcích 12, 13 a 14 lze vidět vygenerovanou pojistnou smlouvu ve formátu pdf zaslanou na email pojistníka.



SMLOUVA CESTOVNÍHO POJIŠTĚNÍ

Základní informace

Typ pojištění: **Jedna cesta**
Destinace: **Evropa**
Počátek cesty: **10.05.2022**
Konec cesty: **25.05.2022**

Pojistník

Rodné číslo: **9909286642**
Jméno: **Jakub**
Příjmení: **Horák**
Datum narození: **28.09.1999**

Ulice: **Nad Stráněmi**
Číslo: **4511**
Město: **Zlín**
PSČ: **76005**

Email: **horakkuba.rymice@seznam.cz**
Telefonní číslo: **777111777**

Obrázek 12 - Smlouva (1)

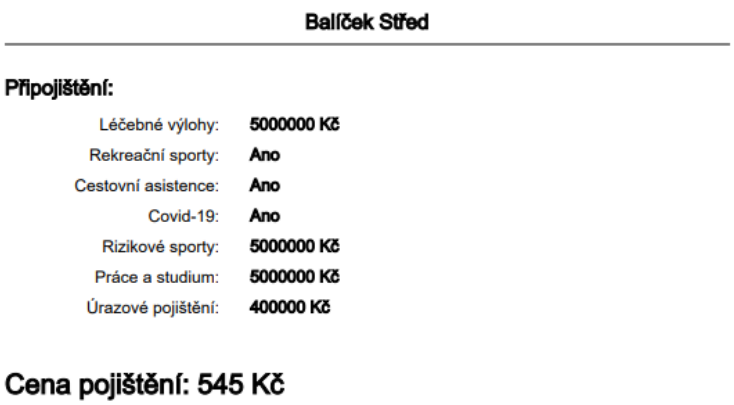
Další pojištěné osoby

Jméno: **Fiktivní**
Příjmení: **Osoba 1**
Datum narození: **15.07.2005**

Jméno: **Fiktivní**
Příjmení: **Osoba 2**
Datum narození: **12.04.1990**

Jméno: **Fiktivní**
Příjmení: **Osoba 3**
Datum narození: **24.06.2003**

Obrázek 13 - Smlouva (2)



Balíček Střed

Připojištění:

Léčebné výlohy: **5000000 Kč**
Rekreační sporty: **Ano**
Cestovní asistence: **Ano**
Covid-19: **Ano**
Rizikové sporty: **5000000 Kč**
Práce a studium: **5000000 Kč**
Úrazové pojištění: **400000 Kč**

Cena pojištění: 545 Kč

Obrázek 14 - Smlouva (3)

ZÁVĚR

Výsledkem teoretické části bakalářské práce je seznámení čtenáře s aktuálním stavem technologií pro vývoj a zabezpečení webových aplikací a s frameworky Blazor a Angular. Výsledkem praktické části bakalářské práce jsou pak dvě verze webové aplikace samosjednávače cestovního pojištění vytvořené ve frontendových frameworcích Blazor a Angular a backendovém frameworku ASP.NET Core. Čtenář by po přečtení měl být seznámen se základními koncepty vývoje webových aplikací ve frameworcích Blazor, Angular a ASP.NET Core a je seznámen s rozdíly a podobnostmi, na které může narazit při vývoji jednostránkových webových aplikací v těchto frameworcích. V teoretické části bakalářské práce byl popsán současný stav technologií pro vývoj a zabezpečení webových aplikací, v rámci kterého byly popsány tři nejpoužívanější frontendové frameworky, backendové frameworky a databázové technologie získané z dotazníku webu StackOverflow provedeného roku 2021. Dále byly detailněji popsány frameworky Blazor a Angular. U frameworku Blazor byla popsána jeho historie, hostovací modely a funkcionality. U frameworku Angular byla popsána také jeho historie, jeho architektura a funkcionality. V rámci praktické části byla webová aplikace z oblasti pojišťovnictví nejdříve navržena. Byly vytvořeny funkční a nefunkční požadavky, scénáře případu užití a také zabezpečení aplikace. Aplikace je koncipována jako jednostránková a jedná se o samosjednávač cestovního pojištění. Následně byly popsány klíčové části webové aplikace, mezi které patří endpointy, služby, propojení s databází, komponenty, modely, validace a komunikace se serverovou částí aplikace. V následující kapitole byly popsány rozdíly v implementaci webové aplikace mezi frameworky Blazor a Angular. Bylo zjištěno, že hlavním rozdílem mezi frameworky je redundance kódu. Blazor je schopen sdílet kód mezi klientskou a serverovou částí aplikace a nemusí se tak psát 2x všechny validace, modely či sdílená logika. Výhodou Blazoru je také, že využívá jen jazyka C# oproti Angularu, který vyžaduje znalost jazyka TypeScript. Angular také obsahuje oproti Blazoru více boilerplate kódu. Co se týče jednoduchých webových aplikací, tak oba frameworky nabízejí velmi podobné nástroje pro vývoj jen s rozdílnou syntaxí. Pro začátečníky nebo jednoduché aplikace je Blazor vhodnou volbou. Pro složitější projekty určené např. i ke komerčnímu použití je doporučeno využít frameworku Angular, jelikož je starší, vyspělejší a pro složitější aplikace obsahuje větší množství nástrojů. Blazor je stále poměrně nový framework, který je stále ve vývoji, a kvůli tomu se také při tvorbě webové aplikace v Blazoru vyskytlo několik problémů. Blazor nedokáže v některých případech správně určit pozici chyby v kódu, která způsobila pád aplikace. Často je schopen zobrazit jen název

souboru. Angular dokáže chyby identifikovat na řádek přesně. Dalším problémem bylo to, že technologie IntelliSense někdy nebyla schopna vidět některé komponenty a bylo nutné vývojové prostředí restartovat. Posledním problémem byly validace, které v základu využívají třídy DataAnnotations. Validace nebyly schopny validovat vnořené třídy nebo dynamicky generovaný obsah stránky. Mimo jiné i z tohoto důvodu byla pro validování vybrána knihovna FluentValidation.

SEZNAM POUŽITÉ LITERATURY

[1] Stack Overflow Developer Survey 2021. *StackOverflow* [online]. New York City, 2021 [cit. 2022-02-09]. Dostupné z: https://insights.stackoverflow.com/survey/2021

[2] Frontend vs Backend. *GeeksforGeeks* [online]. Noida, 2021 [cit. 2022-02-09]. Dostupné z: https://www.geeksforgeeks.org/frontend-vs-backend/

[3] List of 10 Best Front end Frameworks to Use For Web Development. *Monocubed* [online]. Brampton, 2021 [cit. 2022-02-09]. Dostupné z: https://www.monocubed.com/best-front-end-frameworks/

[4] React A JavaScript library for building user interfaces. *ReactJS* [online]. Menlo Park (Kalifornie), c2022 [cit. 2022-02-09]. Dostupné z: https://reactjs.org/

[5] The Best Guide to Know What Is React. *Simplilearn* [online]. 2021 [cit. 2022-02-09]. Dostupné z: https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs

[6] JQuery. *JQuery* [online]. c2022 [cit. 2022-02-09]. Dostupné z: https://jquery.com/

[7] What is jQuery. *Javatpoint* [online]. Noida, c2011-2021 [cit. 2022-02-09]. Dostupné z: https://www.javatpoint.com/what-is-jquery

[8] Vue.js - The Progressive JavaScript Framework. *Vuejs* [online]. c2014-2022 [cit. 2022-02-09]. Dostupné z: https://vuejs.org/

[9] What Is Vue.js? The Pros and Cons of Vue.js in 2022. *Trio* [online]. 2022 [cit. 2022-02-09]. Dostupné z: https://trio.dev/blog/why-use-vue-js

[10] Express - Node.js web application framework. *Express* [online]. c2017 [cit. 2022-02-09]. Dostupné z: https://expressjs.com/

[11] What is the MEAN Stack?. *MangoDB* [online]. c2021 [cit. 2022-02-09]. Dostupné z: https://www.mongodb.com/mean-stack

[12] Express/Node introduction. *MDN Web Docs* [online]. 2022 [cit. 2022-02-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

[13] ASP.NET | Open-source web framework for .NET. *Microsoft* [online]. Redmond (Washington), c2022 [cit. 2022-02-09]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet

[14] ASP.NET Core - Overview. *Tutorialspoint* [online]. Hyderabad, c2021 [cit. 2022-02-09]. Dostupné z: https://www.tutorialspoint.com/asp.net_core/asp.net_core_overview.htm

[15] MVC Framework - Introduction. *Tutorialspoint* [online]. Hyderabad, c2021 [cit. 2022-02-09]. Dostupné z: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm

[16] Welcome to Flask - Flask Documentation (2.0.x). *Flask* [online]. c2010 [cit. 2022-02-09]. Dostupné z: https://flask.palletsprojects.com/en/2.0.x/

[17] What is Flask?. *Educative* [online]. Bellevue, c2022 [cit. 2022-02-09]. Dostupné z: https://www.educative.io/courses/flask-develop-web-applications-in-python/qZWAmEGDBkR

[18] What is Flask Python. *Pythonbasics* [online]. c2021 [cit. 2022-02-09]. Dostupné z: https://pythonbasics.org/what-is-flask-python/

[19] What Is a Database?. *Oracle* [online]. Austin (Texas), c2022 [cit. 2022-02-09]. Dostupné z: https://www.oracle.com/cz/database/what-is-database/

[20] MySQL. *MySQL* [online]. Austin (Texas), c2022 [cit. 2022-02-09]. Dostupné z: https://www.mysql.com/

[21] What is MySQL: MySQL Explained For Beginners. *Hostinger* [online]. Kaunas, 2021 [cit. 2022-02-09]. Dostupné z: https://www.hostinger.com/tutorials/what-is-mysql

[22] PostgreSQL: The World's Most Advanced Open Source Relational Database. *PostgreSQL* [online]. c1996-2022 [cit. 2022-02-09]. Dostupné z: https://www.postgresql.org/

[23] [What does ACID mean in Database Systems?. *Database.Guide* [online]. 2016 [cit. 2022-02-09]. Dostupné z: https://database.guide/what-is-acid-in-databases/

[24] What is PostgreSQL?. *Postgresql* [online]. c2022 [cit. 2022-02-09]. Dostupné z: https://www.postgresql.fastware.com/what-is-postgresql

[25] What is PostgreSQL? Introduction, Advantages & Disadvantages. *Guru99* [online]. 2021 [cit. 2022-02-09]. Dostupné z: https://www.guru99.com/introduction-postgresql.html

[26] Soubory ke stažení pro SQL Server. *Microsoft* [online]. Redmond (Washington), c2022 [cit. 2022-02-09]. Dostupné z: https://www.microsoft.com/cs-cz/sql-server/sql-server-downloads

[27] SQL Server. *Techopedia* [online]. 2020 [cit. 2022-02-09]. Dostupné z: https://www.techopedia.com/definition/1243/sql-server

[28] CLARK, Daniel. SQL vs T-SQL: Understanding the Differences. *Dataquest* [online]. Somerville, 2021 [cit. 2022-02-09]. Dostupné z: https://www.dataquest.io/blog/sql-vs-t-sql/

[29] Editions and supported features of SQL Server 2017. *Microsoft docs* [online]. Washington, 2021 [cit. 2022-02-09]. Dostupné z: https://docs.microsoft.com/en-us/sql/sql-server/editions-and-components-of-sql-server-2017?view=sql-server-ver15

[30] SQL Server 2019 pricing. *Microsoft* [online]. Washington, c2022 [cit. 2022-02-09]. Dostupné z: https://www.microsoft.com/en-us/sql-server/sql-server-2019-pricing

[31] 2021 Data Breach Investigations Report. *Verizon* [online]. New York, c2022 [cit. 2022-03-20]. Dostupné z: https://www.verizon.com/business/resources/reports/dbir/

[32] Cross-Site Scripting (XSS) Makes Nearly 40% of All Cyber Attacks in 2019. *Precise security* [online]. London, 2019 [cit. 2022-03-20]. Dostupné z: https://www.precisesecurity.com/articles/cross-site-scripting-xss-makes-nearly-40-of-all-cyber-attacks-in-2019/

[33] Cross-site scripting. *Portswigger* [online]. Knutsford, c2022 [cit. 2022-03-20]. Dostupné z: https://portswigger.net/web-security/cross-site-scripting

[34] SQL injection. *Portswigger* [online]. Knutsford, c2022 [cit. 2022-03-20]. Dostupné z: https://portswigger.net/web-security/sql-injection

[35] Protecting against sql injection. *Hacksplaining* [online]. c2022 [cit. 2022-03-20]. Dostupné z: https://www.hacksplaining.com/prevention/sql-injection

[36] OS Command Injection. *Whitehatsec* [online]. San Jose, c2022 [cit. 2022-03-20]. Dostupné z: https://www.whitehatsec.com/glossary/content/os-command-injection

[37] Distributed Denial of Service (DDoS). *Imperva* [online]. c2021 [cit. 2022-03-20]. Dostupné z: https://www.imperva.com/learn/ddos/denial-of-service/

[38] Cloudflare. *Cloudflare* [online]. San Francisco, c2022 [cit. 2022-03-20]. Dostupné z: https://www.cloudflare.com/

[39] Man in the middle (MITM) attack. *Imperva* [online]. c2021 [cit. 2022-03-20]. Dostupné z: https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/

[40] Fuzzing. *Contrastsecurity* [online]. Los Altos [cit. 2022-03-20]. Dostupné z: https://www.contrastsecurity.com/knowledge-hub/glossary/fuzzing

[41] What is a Zero-day Attack? - Definition and Explanation. *Kaspersky* [online]. Moskva, c2022 [cit. 2022-03-20]. Dostupné z: https://www.kaspersky.com/resource-center/definitions/zero-day-exploit

[42] Blazor Build client web apps with C#. *Microsoft* [online]. Redmond (Washington), c2022 [cit. 2022-03-20]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor

[43] WebAssembly. *Webassembly* [online]. [cit. 2022-03-20]. Dostupné z: https://webassembly.org/

[44] Overview of ASP.NET Core SignalR. *Microsoft docs* [online]. Redmond (Washington), c2022 [cit. 2022-03-20]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/signalr/introduction?view=aspnetcore-6.0

[45] ASP.NET Core Blazor supported platforms. *Microsoft docs* [online]. Redmond (Washington), c2022 [cit. 2022-03-20]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/blazor/supported-platforms?view=aspnetcore-6.0

[46] What is Blazor and why is it so exciting?. *Chrissainty* [online]. 2019 [cit. 2022-03-20]. Dostupné z: https://chrissainty.com/what-is-blazor-and-why-is-it-so-exciting/

[47] ASP.NET Core Blazor hosting models. *Microsoft docs* [online]. Redmond (Washington), c2022 [cit. 2022-03-20]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-6.0

[48] Introduction - Material Design. *Material* [online]. [cit. 2022-03-20]. Dostupné z: https://material.io/archive/guidelines/#

[49] The History of Angular. *Medium* [online]. San Francisco, 2018 [cit. 2022-03-20]. Dostupné z: https://medium.com/the-startup-lab-blog/the-history-of-angular-3e36f7e828c7

[50] Introduction to Angular concepts. *Angular* [online]. Mountain View, c2010-2022 [cit. 2022-03-20]. Dostupné z: https://angular.io/guide/architecture

[51] Prevent Cross-Site Scripting (XSS) in ASP.NET Core. *Microsoft docs* [online]. Redmon (Washington), c2022 [cit. 2022-04-30]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/security/cross-site-scripting?view=aspnetcore-6.0

[52] Security. *Angular* [online]. Mountain View, c2010-2022 [cit. 2022-04-30]. Dostupné z: https://angular.io/guide/security

[53] FREEMAN, Adam. Pro ASP.NET Core MVC: Develop cloud-ready web applications using Microsoft's latest framework, ASP.NET Core MVC. 6th ed. New York: Springer Science+Business Media, 2016. ISBN 9781484203972.

[54] The Leading PDF Library for Developers | iText. *Itextpdf* [online]. c2022 [cit. 2022-04-14]. Dostupné z: https://itextpdf.com/en

[55] Entity Framework Core. *Microsoft* [online]. Redmond: Microsoft, ©2022 [cit. 2022-04-14]. Dostupné z: https://docs.microsoft.com/en-us/ef/core/

[56] Fluent Validation. *FluentValidation* [online]. c2021 [cit. 2022-04-14]. Dostupné z: https://fluentvalidation.net/

[57] MURRAY, Nate, Felipe COURY, Eri LERNER a Carlos TABORDA. Ng-book: The Complete Guide to Angular. Revision 77. San Francisco: Fullstack.io, 2020.

[58] HIMSCHOOT, Peter. Microsoft Blazor: Building Web Applications in .NET. 2nd ed. New York: Springer Science+Business Media, 2020. ISBN 9781484259283.

Communicating with backend services using http. *Angular* [online]. Mountain View, ©2010-2022 [cit. 2022-04-14]. Dostupné z: https://angular.io/guide/http

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

HTML	HyperText Markup Language
CSS	Cascading Style Sheets
DOM	Document Object Model
AJAX	Asynchronous Javascript and XML
JSX	JavaScript XML
MVVM	Model View Viewmodel
KB	KiloByte
API	Application Programming Interface
MEAN	MongoDB, ExpressJS, AngularJS, Node.js
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
MVC	Model-View-Controller
IIS	Internet Information Services
ORM	Object Relational Mapping
WSGI	Web Server Gateway Interface
DBMS	Database Management System
SQL	Structured Query Language
RDBMS	Relational DataBase Management System
ACID	Atomic, Consistent, Isolated, Durable
T-SQL	Transact - Structured Query Language
SSMS	SQL Server Management Studio
XSS	Cross-site Scripting
CSP	Content Security Policy
DoS	Denial of Service
ORM	Object–Relational Mapping

DDoS	Distributed denial of service
RAM	Random Access Memory
CPU	Central Processing Unit
IP	Internet Protocol
CDN	Content Delivery Network
SaaS	Software-as-a-Service
HTTPS	Hypertext Transfer Protocol Secure
SSL	Secure Sockets Layer
NDC	Norwegian Developers Conference
CIL	Common Intermediate Language
AoT	Ahead-of-Time
CLI	Command-Line Interface
RxJS	Reactive Extensions for JavaScript
URL	Uniform Resource Locator
SCSS	Sassy Cascading Style Sheets
SASS	Syntactically Awesome Style Sheets
GDPR	General Data Protection Regulation
IČO	Identifikační číslo osoby
OSVČ	Osoba samostatně výdělečně činná
HSTS	HTTP Strict Transport Security

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P 1: CD DISK

Přiložené CD obsahuje soubor priloha.zip se zdrojovým kódem aplikace a bakalářskou práci ve formátu pdf.